Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3255

Splitting BPEL Processes

Daojun Cui

Course of Study:	Computer Science
Examiner:	Prof. Dr. F. Leymann
Supervisor:	Dipl.-Inf. O. Kopp, Dipl.-Inf. S. Wagner
Commenced:	November 03, 2011
Completed:	May 04, 2012
CR-Classification:	H.4.1, K.1

Abstract

Khalaf presents a concept to split BPEL processes. This thesis illustrates how to extend the work of Khalaf to go from a BPEL process to a BPEL4Chor choreography. First, the main BPEL process given is split into fragment BPEL processes, in a way that the operational semantic of the main BPEL process is preserved in the collective behavior of the fragmented BPEL processes. The dataflow depependencies of the given BPEL process are analyzed and reflected in the fragmented BPEL processes. Based on the results of the splitting algorithm, a BPEL4Chor choreography is generated: The fragmented BPEL processes are converted into participants in the generated BPEL4Chor choreography.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

BPEL	Web Services Business Process Execution Language
BPEL4Chor	BPEL for Choreography
BPMN	Business Process Model and Notion
BPR	Business Process Re-engineering
CIP	Continuous Improvement Process
CPS	Cross Partner Scope
DOM	Document Object Model
DPE	Death Path Elimination
EPR	Endpoint Reference
PBD	Participant Behavior Description
PWDG	Partitioned Writer Dependency Graph
SOA	Service-Oriented Architecture
StAX	Streaming API for XML
WDG	Writer Dependency Graph
WS-CDL	Web Service Choreography Description Language
WSDL	Web Service Description Language

1 Introduction

Nowadays globally integrated enterprises are demanding more and more agility in the business. They pursue the ways to reinvent the business process rapidly, such as business process re-engineering (BPR) [DS90, Off97] and continuous improvement process (CIP) [Ima86]. The enterprises that embrace CIP improve their business process via modification of the non-competitive part. If improvement goal in the non-competitive business cannot be reached, the out-sourcing or off-shore of the business process will usually be carried out in order to keep the company's portfolio profitable.

To specify business process behavior based on Web services, the Business Process Execution Language (BPEL) has been introduced into industry in recent years. Companies applying BPEL can improve their business process by doing process modification. In the case of out-sourcing, the non-competitive part will be cut-out. The part can be regarded as a cut-out sub-process, which should be run by the third party companies. This cut-out sub-process is called "process fragment" [KKL08a]. The number of the process fragments depends on how the original process is "cut". The challenge is how to do the process fragmentation so that the collective behavior of the process fragments preserve the operational semantic of the original process.

An approach is proposed in [KL06] to decompose the process. In that approach, a BPEL process is firstly transformed into an intermediate form i.e. BPEL-D process in which the data flow is represented by explicit data-links. The transformation is aided by a GUI process editor like Eclipse BPEL Designer[1]. Then the control link and data link are split in the same way that sending block and receiving block are created and the control (true or false) and data (value) are passed by messaging between the two blocks. At the end, the result is a BPEL process per participant, the corresponding WSDL definition per BPEL process, and a global wiring file.

Although the BPEL-D process presents the data flow explicitly and can easily be split, it is not sufficient to split the data dependency in a BPEL process while keeping the operational semantic of that original process, due to the parallelism and Death-Path-Elimination (DPE) in BPEL process. Therefore, a more BPEL compliant approach for splitting data dependency of a BPEL process is introduced in [KKL08a]. The mechanism of splitting data dependency in that approach differs from the explicit data-links in BPEL-D in a way that the data dependencies across the BPEL process fragments are maintained in an implicit manner.

[1]Eclipse BPEL Designer: http://www.eclipse.org/bpel

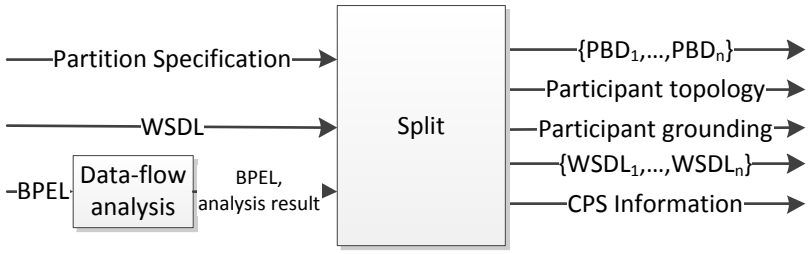

Figure 1.1: In- and Output of Splitting, the *Input* includes (1) partition specification, (2) BPEL, (3) WSDL, (4) and data-flow analysis result , the *Output* includes (1) a Participant Behavior Description (PBD) per fragment, (2) Participant Topology, (3) Participant Grounding, (4) a WSDL definition per fragment and (5) Cross-Partner-Scope (CPS) information .

The work in this thesis is based on the above mentioned approach. Nevertheless, we lack in [KKL08a] a specific definition or description of the out-coming wiring file after the process fragmentation. Thus, we choose BPEL4Chor presented in [DKLW07] as the out-coming wiring specification. BPEL4Chor is a BPEL extension for defining choreographies and is suitable for the global wiring information.

This thesis aims to extend the works of Khalaf et al. [KL06, KKL08a, Kha08] and to demonstrate how one can go from a BPEL process to a BPEL4Chor choreography. In other words, we show how to accept a BPEL process, its associated WSDL definition and the partition specification as input, how to split the BPEL process into process fragments, and eventually how to output the corresponding BPEL4Chor artifacts. Figure 1.1 illustrates the input and output of the splitting process that is the main focus in this thesis.

In this thesis, the control dependency is split as the concept introduced in [KL06]. On the other hand, the splitting approach against data dependency can be stated as follows: assume there is a BPEL process, a partition specification and the result of data-flow analysis on that BPEL process we fragment the BPEL process into smaller ones first, and each process fragment contains a subset of the activities from the non-split process. As second step for every reader of a variable, the writers of the variable will be retrieved by means of the data-flow analysis. Appropriate BPEL constructs will then be created to collect the information from the writers either in the same participant or in the different participants. The information includes variable value and whether the writer succeeded. The participant's process, which the reader resides in, receives the information and assembles the value of the variable regarding the race order of the writers. The principle of dealing with writers conflict is that *last writer wins*. Consequently, the writers' race condition in the non-split process is recreated [KKL08a].

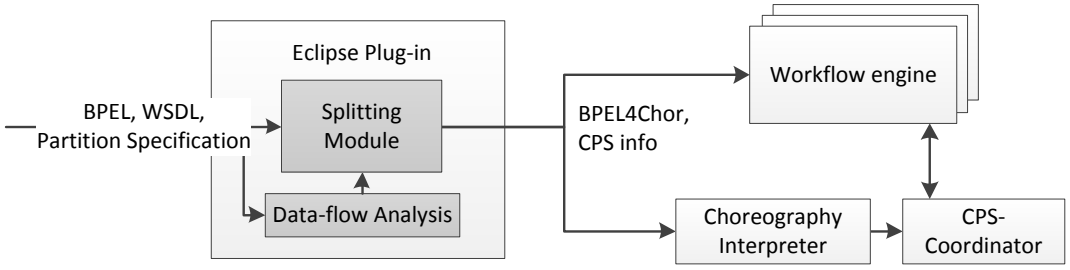

Figure 1.2: Architecture of the Process Fragmentation, the consumer of the output is workflow engines, Choreography Interpreter and CPS-Coordinator [Bor10, Abbildung 4.2].

Architecturally, the split module (cf. Figure 1.2) functions as an Eclipse Plug-in in the Eclipse BPEL designer. It takes a BPEL, an associated WSDL, the provided partition specification along with the data-flow analysis on that BPEL process as input and outputs the BPEL4Chor as well as split loop/scope information. The outputs can be consumed by Choreography interpreter and CPS-Coordinator in the workflow engine.

The details of the thesis are structured as follows:

Chapter 2 – Background and Related Works: The background knowledge that is necessary to read this thesis and the related works are introduced.

Chapter 3 – Process Fragmentation: The main process definition, the partition specification, and splitting the main process into smaller fragment processes with the partition specification are elaborated.

Chapter 4 – Control Link Fragmentation: The concept of fragmenting explicit control dependency, the issue of variable initialization and its resolution when one realizes the concept, and what should be prepared for fragmenting control dependency are introduced in this chapter.

Chapter 5 – Data Dependency Fragmentation: The third part of the splitting procedure is fragmenting implicit data dependency. It includes parsing the result of data-flow analysis of the main process, construction of Writer Dependency Graph (WDG) and Partitioned Writer Dependency Graph (PWDG), creation of Local Resolver and creation of Receiving Flow. Each of them will be addressed.

Chapter 6 – Output in BPEL4Chor Choreography: Outputting the BPEL4Chor artifacts from the results of splitting process will be specified.

Chapter 7 – Architecture and Implementation: The architecture and implementation of the application for splitting process are elaborated in the chapter.

Chapter 8 – Summary and Future Work: The conclusion of this thesis and the outlook for the future will be given.

2 Background and Related Works

In this chapter the background knowledge needed for reading this thesis is briefly introduced then the related works are presented.

2.1 Background

Web Service Description Language (WSDL)

Web Service Description Language (WSDL) is an XML standard for describing Web Services. A WSDL document is used to describe what a service does and how to consume the service. Typically a WSDL document consists of a reusable abstract part that describes "what" the service does and a concrete part that describes "how" to interact with the service. The abstract part refers to the `<types>`, `<message>`, and `<portType>` element, and the concrete part refers to the `<binding>` and `<service>` element. The up-to-date version of WSDL is 2.0, however, we use WSDL 1.1 in this thesis due to the BPEL being based on that WSDL version.

Web Service Business Process Language (BPEL)

Web Service Business Process Execution Language (BPEL for short) [OAS07] is "an extensible work-flow-based language that aggregates services by choreographing service interactions" [WCL+05], in other words BPEL is a work-flow language to operate in the service-oriented architecture (SOA) environment. BPEL provides a set of (basic and structured) activities to model the service interactions. The control flow in a BPEL process is explicitly modeled by the control links between the activities, while the data flow in a BPEL process is implicitly realized by sharing the global or local variables. In a BPEL process, one can access and manipulate the data that belongs to the BPEL process using XPath [W3C99]. Additionally, the parallelism and Death-Path-Elimination (DPE) [CKLW03] are supported by BPEL.

A BPEL process is usually an executable process that can be run in a target work-flow engine. If the intention of the process is to describe the business contract and therefore some concrete operation detail should be hidden, then a partially specified (abstract) process should be created. Such a process is called abstract process and must be explicitly declared as 'abstract'.

BPEL-D

BPEL-D is a variant of BPEL. Unlike the implicit data flow in BPEL, BPEL-D has the explicit data flow. BPEL-D covers all the constructs in BPEL 1.1, and provides the data-links that facilitate the explicit data flow. Each activity in BPEL-D contains a set of containers, and each container of the set is used to store the incoming or outgoing data. The container for incoming data is called input container and similarly the one for outgoing data is called output container. A data-link in BPEL-D contains a set of maps and defines a transition of data between the source activity and the target activity by mapping the output container of the source activity to the input container of the target activity. Interested readers are referred to [Kha07, Fer07, Kha08] for the detail about BPEL-D.

Based on the data-links in BPEL-D, the data dependency can be split in the same way as the control dependency is split. However, this approach is not sufficient to split the data dependency while maintaining operational semantic of the original BPEL process. The reason is that the data-links in BPEL-D can not deal with the parallelism and the Death-Path-Elimination in the BPEL process.

Data-Flow Analysis

Data-Flow Analysis is an approach to "derive explicit data links between the activities of a BPEL process" [KKL08b]. In that approach, the current state of the writing to a given variable element is given by the function "writes$_\circ$" and "writes$_\bullet$". Given a variable element and a reader activity, the former function returns the tuples which contain the possible writers, the invalid writers, the disabled writers and the "may be dead" (boolean) value. The latter function does the same thing. Although the tuples returned by the writes$_\circ$ reflect the state before the reader activity is executed, and the tuples returned by the writes$_\bullet$ does the state after the reader activity is executed.

BPEL4Chor Choreography

BPEL4Chor is an extension of BPEL that defines web service choreographies. In contrast to the top-down approach in the Web Service Choreography Description Language (WS-CDL, [W3C05]), BPEL4Chor provides a bottom-up approach to specify choreography. Furthermore BPEL4Chor is based on the *interconnected interface behavior models*, which can also be expressed in Business Process Modeling Notion (BPMN, cf. [OMG11]). More specifically, BPEL4Chor provides the interconnected interface behavior descriptions by utilizing the *Abstract Process Profile for Observable Behavior of BPEL [OAS07]* and by adding an interconnection layer on top of the abstract BPEL process. As in Figure 2.1, the artifacts of BPEL4Chor are explained as follows [DKLW07]:

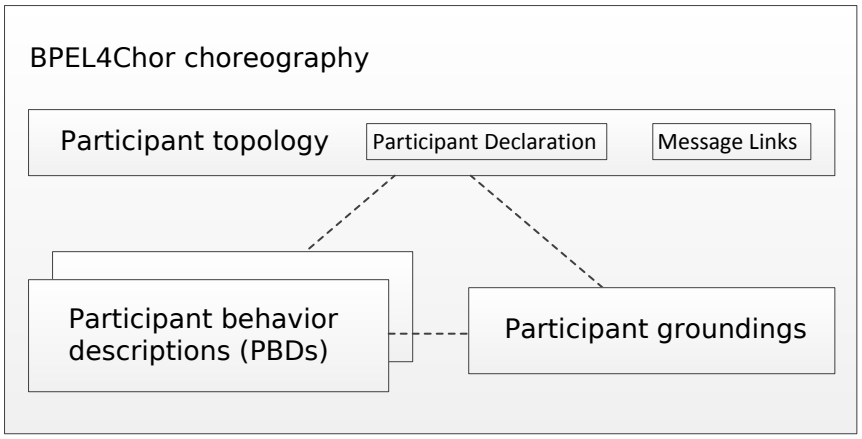

Figure 2.1: BPEL4Chor Artifacts, derived and simplified from [DKLW07, Fig. 2]

1. Participant Behavior Description (PBD)

   A PBD is an abstract BPEL process which is adapted to provide the communication activities, their behavioral dependencies, and their interconnections. In order to separate the technical configuration such as the WSDL portTypes that are referred to in the communication activities, the attribute *partnerLink*, *portType*, and *operation* in the communication activity are to be omitted. Additionally, a new attribute *wsu:id* which is of type "xsd:id" is introduced into each communication activity specially the "*onMessage*" branches in activity `<pick>`. The *wsu:id* serves as an identifier of the associated sending activity and receiving activity in different participants.

2. Participant Topology

   The participant topology gives the description of the choreographic structure. The notions of *participantType*, *participant*, and *messageLink* are used in a participant topology. Each *participant* is of certain *participantType*, and there may only be one participant of certain *participantType* in one conversation. The *messageLink* represents a communication between two participants.

3. Participant Grounding

   The web service specific configuration, which has been avoided in PBD and participant topology, is presented in the participant grounding. The notions of *participantRef* and *messageLink* are used in the participant grounding. A *messageLink* in a grounding differentiates from the *messageLink* in a topology by providing the WSDL portType and operation.

2.2 Related Work

Multiple places in this thesis have mentioned Dead Path Elimination (DPE). The understanding of how a BPEL process with the attribute "`suppressJoinFailure=yes`" (DPE on) behaves when exception is thrown is critical. A comprehensive introduction of DPE is presented in [CKLW03].

An approach for data-flow analysis of BPEL process is proposed in [KKL08b] by Kopp et al. Unlike the mainstream data-flow analysis, the data-flow analysis of BPEL process deals with the parallelism in BPEL, and is simultaneously aware of the Dead Path Elimination (DPE). The algorithms of that approach are extended by Breier [Bre08] and implemented by Gao [Gao10]. The analysis result can be consumed in the splitting step of this thesis, where the data dependencies are fragmented (cf. Chapter 5).

Khalaf's [KL06] introduces an approach to decompose the BPEL process. In that approach the control dependency is split by exchanging messages between the fragment processes. Furthermore, a BPEL process is transformed into an intermediate representation of BPEL i.e. BPEL-D in which the explicit data-link is used, then the data dependency is split in the same manner as the control dependency. The BPEL-D approach is also illustrated in [Kha07].

Due to the intricacies of the dealing with parallelism and Death Path Elimination in BPEL, a more comprehensive mechanism compared to [KL06] is presented in [KKL08a] for splitting BPEL process while maintaining the data dependencies. In that approach from [KKL08a], the fragmentation of data dependency is directly run on the BPEL process instead of BPEL-D process, i.e. the implicit data dependency in BPEL process is split without using any other intermediate form or framework. Consequently, messages are sent between the 'local resolver' and 'receiving flow' instead of between 'sending block' and 'receiving block' for passing data.

[Kha08] summarizes the approaches to split business process by using BPEL and without losing the operational semantic. Fernandez [Fer07] illustrates the approach to create a GUI to help splitting BPEL process with BPEL-D. It is the most relevant work for this thesis. Nevertheless, the approach to split BPEL process without BPEL-D is the focus of this thesis.

BPEL4Chor, which is used as the target format of the splitting output in this thesis, is introduced in [DKLW07] by Decker el at. A more concrete elaboration of BPEL4Chor is given in [DKLW09].

3 Process Fragmentation

Process fragmentation is the beginning of the splitting procedure, therefore, it is the part where the input of the split procedure is taken care of. After the input, a main BPEL process is created, and it is about to be fragmented into smaller ones upon the partition specification given.

In this chapter, the specification of the main process is presented in section 3.1, then the partition specification will be introduced, after that creation of the fragment BPEL processes and the associated WSDL definitions will be addressed in detail. At the end of this chapter, the results of process fragmentation will be reviewed.

3.1 Main Process Specification

The Figure 3.1 shows that one of the input for the splitting procedure is a BPEL process, which one can call main BPEL process or original BPEL process, since it will be split into multiple smaller fragment processes. Input that is associated to the BPEL process is the WSDL definition. For the split, it provides the message type referred by variable in BPEL process, the PortType referred by in-bound and out-bound activity, and the PartnerLinkType referred by PartnerLink

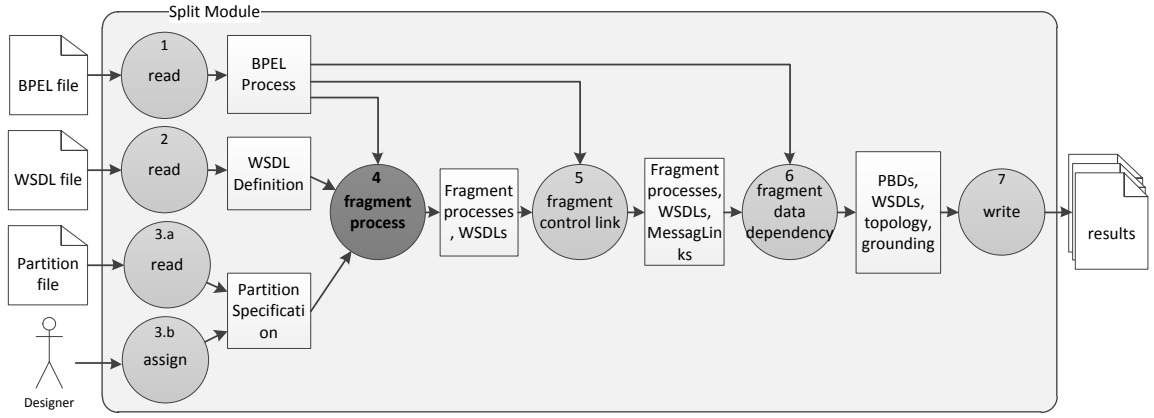

Figure 3.1: The Overview of the Split Module with Process Fragmentation (step 4) emphasized

in the BPEL process. The deployment information of the main process is absent since it is not deployed in this thesis, it is split instead. Further more, the main process is monitored in both source code mode and graph mode using GUI, e.g. Eclipse BPEL Designer.

Due to the complexity of supporting all BPEL activities, there is a need to set up a subset so that the task can be achieved in this thesis. Since the splitting concept and algorithms are across the [KL06], [Kha08], and [KKL08a], and they set up some restrictions for main process, therefore the subset of BPEL in this thesis also agrees with them and is defined as follows:

1. Process with "`suppressJoinFailure=yes`"(DPE [CKLW03] on) [KL06, KKL08a]
2. Only global variables are used for data handling in the process [KL06, KKL08a]
   a) The variable is of message type, which is usually defined in WSDL definition.
   b) The message consists of parts that are of simple type in the XML Schema.
   c) In [KL06] variables are not constrained in global scope or local scope.
3. Exactly one correlation set for the inter-routing between fragment processes [KL06, KKL08a]
   a) The correlation set contains a property named "correlProperty" of type `string` and an associated `propertyAlias` is given in WSDL definition.
   b) The type of the property is explicitly pre-defined in this thesis.
4. PartnerLinks [KL06, KKL08a]
5. A `<flow>` activity is used as top level activity of the process. In other cases, `<flow>` activity, as well as the other structured activities, is not supported. Note that in this point the usage of the structured activities is not explicitly stated in [KL06, KKL08a].
6. All basic activities except `<terminate>`, `<throw>`, `<compensate>`, and `<assign>` activity that copies to a process's endpoint references(EPR) [KL06, KKL08a].
7. A `<receive>` and its combined `<reply>` are *not* to be separated in different participants [KL06, KKL08a]. Interested readers can find more information about this point in [Kha08, Section 5.7.1].

3.2 Partition Specification

Partition specification is one part of the input for splitting procedure besides the BPEL and WSDL. It informs the splitting procedure which activity in the main process is assigned to which participant. The participants together constitute the partition of the BPEL process. The term 'participant' indicates a fragment of the main process, and it has one or more activities in the main process.

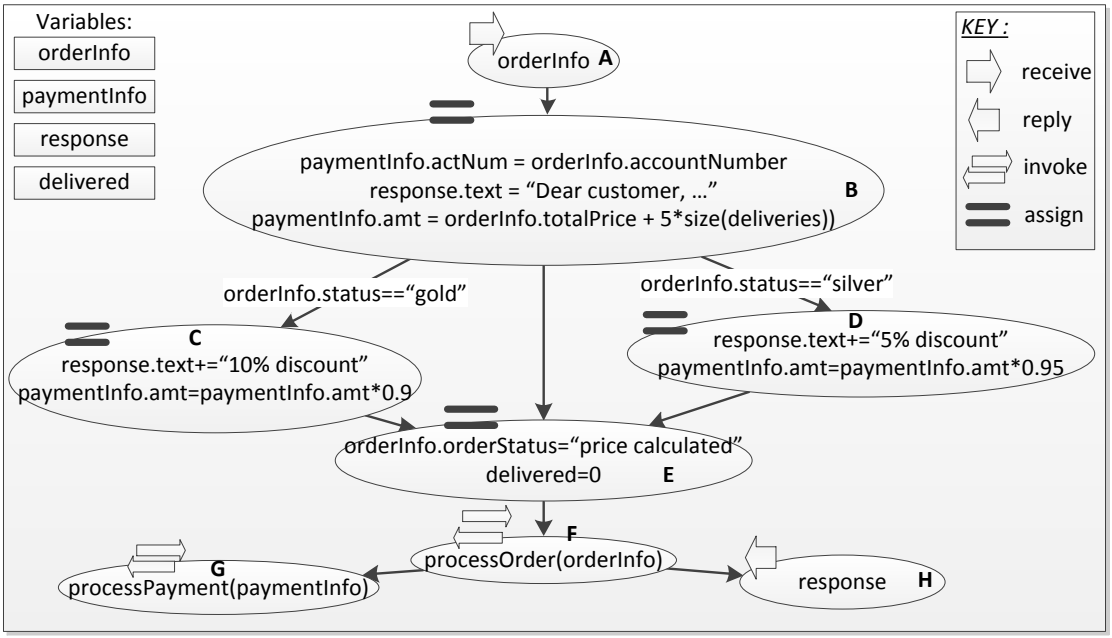

Figure 3.2: Ordering process that provides discounts to Gold and Silver customers [KKL08a, Fig. 1]

The concept to describe the partition of a process with formal definition is firstly introduced in [KL06, Section 4.1]. The main idea is that one divides the activities in different sets, and each set is regarded as a participant.

The concept is described here in more detail. First of all, we introduce the function $\pi : (x_1, x_2, \dots, x_i, \dots) \times i \mapsto x_i$, which is also used in the other equations and algorithms in this thesis. Let $x = (x_1, x_2)$ be a tuple, then $\pi_1(x) = x_1$ denotes the projection onto the first coordinate of x, and $\pi_2(x) = x_2$ denotes the projection onto the second coordinate of x. Let $\mathbb{N}$ be the natural numbers i.e. $\{1, 2, 3, \dots\}$. Let A be the set of all basic activities in the process, p_i be one participant, and P be the set of participants. A participant p_i consists of a name s_i, which is in $\{s_1, s_2, \dots\}$, and a set of activities $M_i \subseteq A$, which is in $\{M_1, M_2, \dots\}$ and holds one or more activities. Furthermore, each basic activity must be assigned to exactly one set M_i, i.e. $\bigcup_{i \in \mathbb{N}} M_i = A$. The formal definition of the set of participants is as follows [KL06]:

$$P := \{p_i \mid \forall i \in \mathbb{N} : p_i = (s_i, M_i)\} \tag{3.1}$$

There are several general conditions that a partition defined per equation (3.1) must satisfy. The conditions are initially stated in [KL06] and revised here:

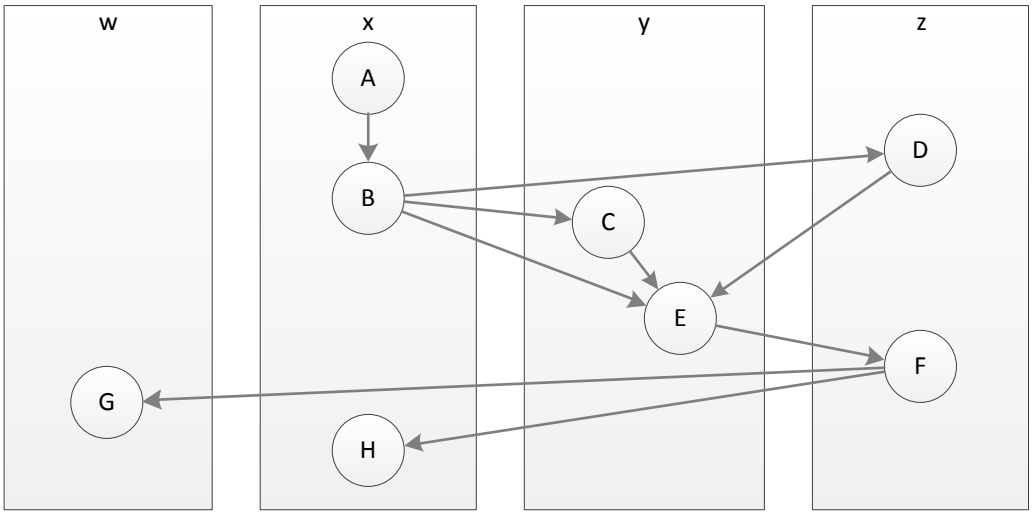

Figure 3.3: A Partition $P1$ derived from [KKL08a, Fig. 2] regarding the Ordering process in Figure 3.2. $P1 = \{p_1, p_2, p_3, p_4\}$, $p_1 = (w, \{G\})$, $p_2 = (x, \{A, B, H\})$, $p_3 = (y, \{C, E\})$, $p_4 = (z, \{D, F\})$

1. $\forall p_i \in P$: $|\pi_2(p_i)| \geqslant 1$ where $\pi_2(p_i) = M_i$ as defined above, i.e. a participant p_i must have at least one activity.
2. $\forall p_i, p_j \in P$: $p_i \neq p_j \;\Rightarrow\; \pi_1(p_i) \neq \pi_1(p_j) \;\wedge\; \pi_2(p_i) \cap \pi_2(p_j) = \emptyset$, i.e. two participants do *not* share a same name or an activity.
3. $\bigcup_{p_i \in P} \pi_2(p_i) = \bigcup_{i \in \mathbb{N}} M_i = A$, where A is the set of all basic activities in the process.

When we combine the second point with the third point we can also conclude that each basic activity must be assigned to exactly one participant, and a designer assigns only basic activities (*no* scope or loop) to a participant.

In Figure 3.3, a partition specification *P1* for the sample process in Figure 3.2 is shown. Each swim-lane in the figure represents a participant. Each of the symbols w, x, y, and z is the name of one participant. The round circles are basic activities, which are named by the letter in the middle of the circle. Each basic activity is assigned to one participant. To illustrate how one reads the information from the formal definition of a participant, we take the participant $p_2 = (x, \{A, B, H\})$ in partition *P1* for instance:

- the activities A, B, and H are in participant p_2, i.e. $\pi_2(p_2) = \{A, B, H\}$,
- and the name of participant p_2 is x, i.e. $\pi_1(p_2) = x$.

```
1 <partitionSpecification>
2       <participant name="NCName">+
3             <activity path="String"/>+
4       </participant>
5 </partitionSpecification>
```

Listing 3.1: Partition File Syntax

Recall that the 'step 4 - fragment process' in Figure 3.1 needs a partition specification. There are two alternative inputs for partition specification. First is the partition file, second is the designer assignment aided by GUI.

The Listing 3.1 is the syntax for partition file. We use the partition file to store the partition specification as described in the equation (3.1). A partition file serves also as input of the split procedure in this thesis. The partition file syntax can be interpreted as follows:

1. The `<partitionSpecification>` element (line 1) has one or multiple `<participant>` elements.
2. The `<participant>` element (line 2) contains an attribute `name` and has one or multiple `<activity>` elements (line 3).
3. The attribute `path` in the `<activity>` element (line 3) is XPath string.

Note that the Listing 3.1 uses the same informal syntax as in BPEL specification [OAS07] to describe the XML grammar. It means that the "+" appended to the elements stands for "one or more".

Upon the partition file input, it is assumed that the BPEL process is loaded, but given the circumstance that one does not have direct access to specific BPEL activity in memory. The process is in the memory, but not visible for us. To start the splitting, the split procedure must be told where to find the activity in the BPEL process and which activity is in which participant. A concept to provide the information can be stated as follows: (1) the position of activity is given via XPath stored in the partition file, (2) information about participant is given by grouping the activity XPaths, and (3) the group is labeled with the participant name. This concept is reflected in the Listing 3.1, which is used in this thesis.

The operation pattern upon the partition file input is that the partition file is parsed at the starting time, then the activity XPaths are retrieved, finally the activities in the BPEL process (in memory) are located with help of the XPath, and put into the corresponding participant.

The example partition file shown in Listing 3.2 is given to illustrate how one defines a partition specification for a BPEL process using the syntax in Listing 3.1.

As for the GUI input, normally one expects that the BPEL process has been represented in the GUI. It means the user or designer just needs several mouse clicks to assign the basic

```
<!-- Partition Specification for OrderingProcess -->
<partitionSpecification>
      <participant name="w">
            <activity path="/bpel:process/bpel:flow/bpel:invoke[@name='G']"/>
      </participant>
      <participant name="x">
            <activity path="/bpel:process/bpel:flow/bpel:receive[@name='A']"/>
            <activity path="/bpel:process/bpel:flow/bpel:assign[@name='B']"/>
            <activity path="/bpel:process/bpel:flow/bpel:reply[@name='H']"/>
      </participant>
      <participant name="y">
            <activity path="/bpel:process/bpel:flow/bpel:assign[@name='C']"/>
            <activity path="/bpel:process/bpel:flow/bpel:assign[@name='E']"/>
      </participant>
      <participant name="z">
            <activity path="/bpel:process/bpel:flow/bpel:assign[@name='D']"/>
            <activity path="/bpel:process/bpel:flow/bpel:invoke[@name='F']"/>
      </participant>
</partitionSpecification>
```

Listing 3.2: Partition File upon Partition P1 (cf. Figure 3.3)

activities to the corresponding participant. At the moment there are already tools like Eclipse BPEL Designer which can load a BPEL process. Nevertheless, a graphic representation of the participant and the logic to assign activities to the participant still needs to be done.

3.3 Creating WSDL Definitions and Fragment Processes

Upon the partition specification, the BPEL process can now be fragmented into several smaller fragment processes (in this thesis, fragment process is used as synonym for participant). Consequently, for each participant a WSDL definition and a BPEL skeleton is created. The corresponding artifacts are copied from the original BPEL process/WSDL definition in the fragment processes/WSDL definitions.

3.3.1 Creating WSDL Definitions

A WSDL definition will be created for each fragment process. In this thesis, generally, the WSDL definition's `name` attribute will be the same as the participant's name. Its `targetNamespace` attribute will come from the original process's WSDL definition.

It is explained in [KL06, Section 4.2] and [Kha08, Section 5.7.1] that (1) to support the inter-communications between the fragment processes some new artifacts are created in their WSDL definitions, and that (2) some artifacts are copied from the WSDL definition of original

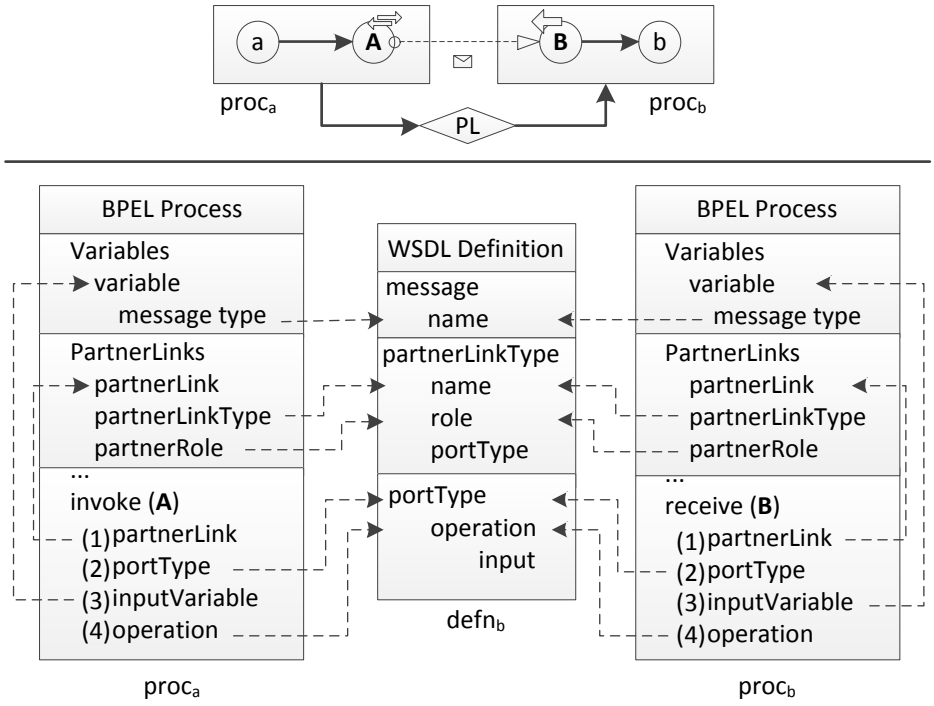

Figure 3.4: Conceptual Inter-Communication (top) and References in Processes (bottom)

process in the proper fragment process to enable the communication between the clients and the fragment processes.

First of all, we go into detail about what are to be created in the WSDL definitions of the fragment processes in regard to the inter-communication between the fragment processes, using the partition P1 (cf. Figure 3.3) as example.

Assume that the participant's activities are already placed in corresponding fragment process upon the partition P1. Consider the scenario that two activities that were directly connected by one link in original process, are now separated in two fragment processes. For instance, the activity B and D were directly connected and are in two participants in partition P1. More specifically, activity B is in participant p_2=$(x, \{A, B, H\})$ and activity D is in participant p_4=$(z, \{D, F\})$. A side-effect of the separation is that the participant p_2 and participant p_4 are logically connected due to the link from activity B to activity D, therefore there will be inter-communication between them. It may be interpreted as p_2 invokes p_4. In this scenario, necessary artifacts are to be created in the WSDL definition of the invoked participant p_4 to facilitate this communication.

Figure 3.4 is an observation of the inter-communication between two simple processes i.e. $proc_a$ and $proc_b$. Upside of that figure where the <invoke> activity *A* in $proc_a$ sends a message and the <receive> activity *B* in $proc_b$ receives the message, is the conceptual communication. The technical configuration of the activity *A* and the activity *B* is illustrated in the downside of that figure. The $defn_b$ is WSDL definition of process $proc_b$. Some structures of the process $proc_a$, $proc_b$, and definition $defn_b$ are left out, since our focus is the artifacts in the WSDL definition that are referred in the <invoke> and <receive> activity.

The arrows with dash line below the middle line of the Figure 3.4 indicate the references from the BPEL processes to the WSDL definition. The common point in the references of the Invoke and the Receive is that all of the references (partnerLink, portType, operation, inputVariable) trace back to the WSDL definition $defn_b$ of the process $proc_b$. Now, it is obvious that to facilitate the inter-communication the artifacts for WSDL are created in the party which is invoked or requested.

Based on that observation, we are able to tell what are to be created for the inter-communication. The WSDL artifacts to be created for the link (B->D) in Figure 3.3 include:

1. WSDL messages and uniquely named WSDL operations to transmit the control- and data dependency as introduced in chapter 4 and chapter 5 separately
   a) They are created in the WSDL definition of p_4=(z, $\{D, F\}$).
2. A WSDL portType for p_2=(x, $\{A, B, H\}$) to send communication via <invoke> activity and for p_4 to receive communication via <receive> activity
   a) It is created in the WSDL definition of p_4
3. A WSDL partnerLinkType to characterize the conversational relationship between p_2 and p_4
   a) In the parternLinkType, roles should be created too, in this scenario, there is only one role because it is one way conversation (B->D).
   b) The role that gets created is referred in the `myRole` attribute of the partnerLink element of the p_4's fragment process and in `partnerRole` for the p_2.
   c) Usually the partnerLinkType is created in WSDL definition of p_4 as `extensibilityElement`, which is WSDL extension for BPEL specification.

The Listing 3.3 is an example snippet that gets created for link (B->D) in the the WSDL definition of p_4. A portType named "pxpzPT" (line 10-17) and a partnerLinkType named "pxpzPLT" (line 3-5) are created for the communication from participant p_2 to participant p_4. Inside the portType "pxpzPT" are two operations "B2DOperation" (line 11-13) and "xzVarNameOp" (line 14-16) created. Both operations have only input message. One refers to message "control-LinkMessage" (line 7), and the other refers to message "statusAndDataXZVarNameMessage" (line 8). Note that some details in the messages such as the 'parts' are hidden here. Interested

```
1  <wsdl:definitions name="pz" targetNamespace="www.bpel4chor.org" ...>
2        ...
3        <plnk:partnerLinkType name="pxpzPLT">
4              <plnk:role name="pxpzRole" portType="tns:pxpzPT" />
5        </plnk:partnerLinkType>
6        ...
7        <message name="controlLinkMessage"/>
8        <message name="statusAndDataXZVarNameMessage"/>
9        ...
10       <portType name="pxpzPT">
11             <operation name="B2DOperation">
12                   <input message="tns:controlLinkMessage"/>
13             </operation>
14             <operation name="xzVarNameOP">
15                   <input message="tns:statusAndDataXZVarNameMessage"/>
16             </operation>
17       </portType>
18       ...
19 </wsdl:definitions>
```

Listing 3.3: Example Snippet for Link (B->D)

reader is referred to Listing 4.1 in Section 4.2.2 for message “controlLinkMessage”, and to Listing 5.1 and 5.2 in Section 5.4.1 for message “statusAndDataXZVarNameMessage”.

Since a link that crosses the fragment processes and gets split, requires the communication between the source and target fragment process, and therefore new artifacts to be created on the fragment processes and their WSDL definitions. The more links crossing the fragment processes there are, the more artifacts are to be created. So there is a need to assess which artifact, once created, can be reused, and which artifact should be uniquely created each time.

Table 3.1 is general analysis about which artifact can be reused and how often it can be reused. It shows two points: (1) the PartnerLinkType, Role, PortType can be reused between one pair of the invoking- and invoked participant, (2) the Message and Operation are not reusable, with the exception that the message for control link can be reused, as illustrated in the Chapter 4 – Control Link Fragmentation.

As one may realize, if we generate all the facilities in the early stage in the WSDL definitions, which enable the communication between the fragment processes, it requires an expensive analysis through all pairs of participants that interact with each other. Therefore, in this thesis, the creation of the artifacts that are necessary for the inter-communication between fragment processes is delayed until the link is being split and the constructs sending message and receiving message are being created. That way, the artifacts for fragment processes and WSDL definitions will be created together and one does not need to do the creation of artifacts in WSDL definitions at the beginning.

Artifact	Reusable	Multiplicity
Message for control	Y	Only one such message is to be created (cf. Chapter 4 – Control Link Fragmentation)
Message for data	N	One for each local resolver and receiving flow (cf. Chapter 5 – Data Dependency Fragmentation)
Operation	N	Multiple uniquely named operations are to be created for each link respecting control and data
PortType	Y	One for each pair of invoking- and invoked participant
Role	Y	One for each pair of invoking- and invoked participant
PartnerLinkType	Y	One for each pair of invoking- and invoked participant

Table 3.1: WSDL Artifacts Re-usability and Multiplicity

One then needs to do the creation of constructs for inter-communication, which refers to control links and data dependencies, between fragment processes extra in next stages. When we do these separately, one must consider how to propagate the necessary information e.g. portType name to the next stage, when the constructs , which send or receive message and thus need e.g. portType name in the attribute `portType`, for fragment processes are being created. Based on that consideration this part will be dealt with in the succeeding steps, i.e. Chapter 4 – Control Link Fragmentation, and Chapter 5 – Data Dependency Fragmentation.

Until now we have discussed the communication between the fragment processes. Some artifacts are to be created, and therefore the fragment process will not look like the part in the original process. However, from the perspective of the clients, which the service of the original process is exposed to, the collective behaviors of the split processes shouldn't be different from the original process, except that the ERP[1] of the fragment process, where the request from the client is received, must be informed.

To maintain the communication between the clients and the original process while splitting the process, the artifacts that serve the communication with clients should be located and copied from the original process's WSDL definition to that of the corresponding fragment. The looking up and copying of artifacts should happen along with copying activity from original process into fragment process as shown in Section 3.3.2, because the communication activity will be referencing the underlying WSDL artifacts. E.g. a normal `<receive>` activity will set the `partnerLink`, `portType`, `operation`, and `variable`. The `portType` and `operation` are direct references in the WSDL definition, while `partnerLink` and `variable` indirectly associate to the `PartnerLinkType` and `MessageType` in the WSDL definition.

A special case is that a `<reply>` activity is replying to a message that is received by an inbound message activity e.g. `<receive>` [OAS07, Page 25 of 264]. Their combination forms a two-

[1] ERP - Endpoint Reference

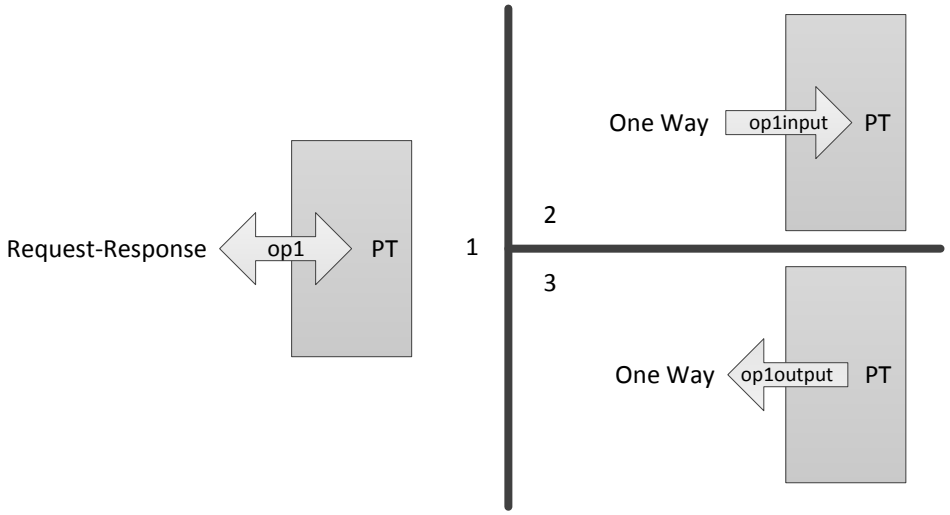

Figure 3.5: Scenario Split PortType, (1) PortType in non-split process's WSDL definition (2) input part of the PortType in one participant's WSDL definition (3) output part of the PortType in another participant's WSDL definition

way (request-reply) operation on the WSDL `portType` of the original process. If we place the `<receive>` and `<reply>` activity in different participants, it means that we must copy the operation separately from the `portType` to each of the WSDL definition of the fragment process, where each of the `<receive>` and the `<reply>` presents. In other words, we split the `portType` by assigning the `<receive>` and `<reply>` to different participant. The impact on the client is that the client is enforced to update its communication activities so that they know the correct port they should point to, since the original `portType`, after splitting, is in two participants, one for receiving, one for replying. Figure 3.5 illustrates that scenario we discussed.

Recall that we disallow the `<receive>` and `<reply>` pair for synchronous operation to be assigned in two different participants. Its intention is to minimize the effect on the client. Since the both activities of the two-way conversation are in the same participant, then the `portType` will not be split. Thus, nothing needs to be changed at the client.

Wider perspective on this topic can be found in [Kha08, Section 5.7.1].

3.3.2 Creating Fragment Processes

A fragment process will be created for each participant. The newly created process is an empty process. In the run-time, necessary artifacts will be copied into the fragment process. What is copied into the fragment process depends on the participant associated.

The new fragment process will be named after the participant. The `targetNamespace` will be inherited from the original process. And as mentioned before the "`suppressJoinFailure`" attribute in the `<process>` must be set as '`yes`' (turn on DPE), since the default value of the attribute is '`no`'.

Upon the participant, the artifacts that are copied into the fragment process include:

- partnerLink referred by the activity in the participant
- variable referred by the activity in the participant
- correlationSet
- activities assigned in the participant, and the activities' parents.

The last bullet is particularly to be noticed, since the parent of the activity assigned in the participant is usually a structured activity. The third condition of the participant specification in Section 3.2 Partition Specification implies that all activities in the participant are basic activities. It means that if one basic activity is assigned in a participant, then its parent (structured activity) should also be added into the fragment process. The addition of the parent activity is due to the *rubber-band effect* [Kha08, Section 5.4], in which it is stated that violating the *rubber-band effect* leads to an inconsistent scope state.

To illustrate the rubber-band effect. Let's assume that in a original process `p`, a basic activity ,'`a`', is in a `<flow>` activity, '`flow`', which contains a fault handler, '`fh`', if error happens in '`a`', an exception would be thrown by '`a`' and be caught by the '`fh`', and be handled properly, so that the process would not crash. While creating the fragment process, if we do not copy the parent of the '`a`' in its fragment process, there is no proper `faultHandler` to handle the error exception that is thrown by '`a`', the exception will be passed on further to the ascendant, where it will be unexpected exception, and will very likely result the crash of the process. The violating scenario is shown in [Kha08, Figure 28].

In this thesis, the algorithms in [Kha08, Section 5.7.2] are adapted to handle the creation of the process fragment. The function PROCESS_CHILD in that chapter is the main algorithm to add activity in the fragment process. The main idea is that given the main process, the participant, and the fragment process, we iterate through all activities in the main process in a top-down manner, in each iteration an activity will be handled. In an iteration, the activity '`a`' will be added (copied) into the fragment process under either of the two conditions: (1) the activity is basic activity and is assigned in the participant; (2) the activity is not basic activity, however it has child activity that is assigned in the participant. Any activity that does not

satisfy either of the conditions and does not belong to the given fragment process will therefore simply be skipped. After an activity is handled, the function iterates recursively further in its children, until there are no children anymore.

So far, the placement of the links that connect the basic activities is not mentioned. In fact, they are handled while the activities are being added into the fragment process, because they are member of the standard elements in an activity. From the splitting aspect, the links can be categorized based on whether they cross the boundary. But at this stage they are still regarded as the same and are not handled differently.

Recall that in the BPEL model, a '`link`' has two members: one is its '`source`'; the other is its '`target`'. The '`source`'/'`target`' has a member '`activity`', which is the source/target activity of the link. It means that manipulating a '`link`' involves 4 other objects: (1) '`source`', (2) '`source-activity`', (3) '`target`', and (4) '`target-activity`' .

In this stage, the links are copied, while their associated activities are being copied in the fragment processes. More particularly, if an activity is source or target of one link, then, copying the activity means also copying the link. Recall that the activity gets copied in one fragment process if it is in the process fragment, or contains a child that is in the process fragment. Note that while copying a link, one end of it is the activity that is being copied, the other end (which is also an activity) of the link is in this moment *not* the subject to be handled.

The following is how the activity including its links is copied:

1. a new empty activity regarding the original activity type will be created at first, then configuration of the original activity will be copied respectively.
2. if the original activity is source of any link, then copy the link for the new activity too. Copying the link means that a new link is generated, its source is set as the newly created activity, and its name is set as same as the original link's. The new link's '*target*' is left empty.
3. if the original activity is target of any link, then copy the link for the new activity too. The copying operation is similar to the last point, the difference is that the new link's '*source*' is left empty.

It is based on the consideration that the main task of this stage is to place the activities in the fragment process, the task is *not* to determine whether a link crosses the boundary and when yes, to split it. It is the task of next chapter, i.e. Chapter 4 – Control Link Fragmentation. As such, we just need to leave enough information for the next stage, so that the original link can be precisely located in the original process and be properly handled.

Figure 3.6 illustrates the results after the fragment processes are created and activities are copied. Given the Partition P2, the main process `p` is fragmented in 2 fragment processes, p_1 and p_2. We can realize that (1) the parent of the basic activities `<flow>` is copied in both fragment processes due to the *rubber-band effect*, (2) and there are 2 copies for each link out of

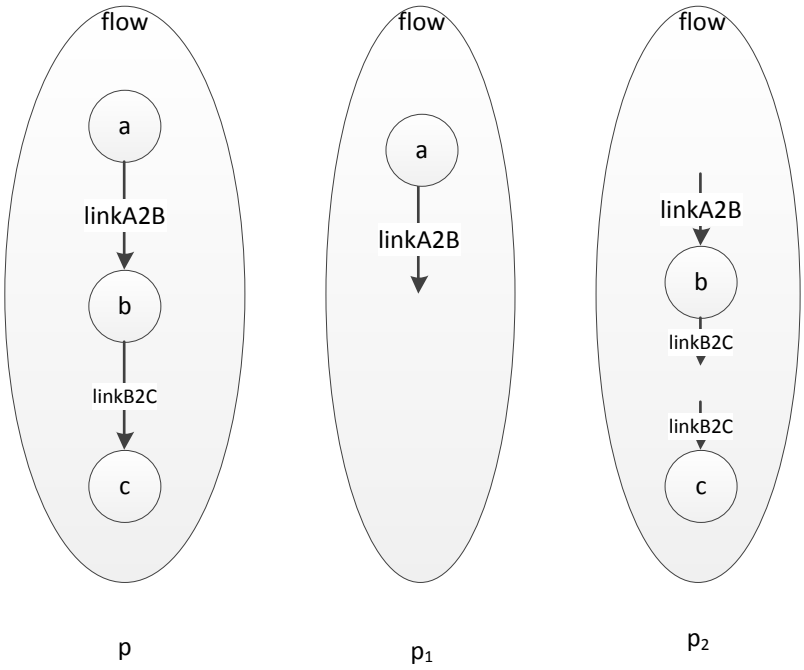

Figure 3.6: Result after Activities are copied, p is the main process, p_1 and p_2 are fragment processes upon the Partition $P2 = \{p_1, p_2\}$, $p_1 = (x, \{a\})$, $p_2 = (y, \{b, c\})$.

the original process, both having the same name, although, one end of either link is bound to an activity, and the other end is empty. We take the `linkA2B` as an example to explain how are they generated. As the activity 'a' is being copied in the fragment process p_1, it is detected that 'a' is source of the link `linkA2B`, so the link is copied too, but, its target is left empty. As the activity 'b' is being copied in the fragment process p_2, it is detected that 'b' is target of `linkA2B`, then the link is copied as well. However, the source of the link is left empty. As a result, there are two copies of the `linkA2B`, and each has only one end that is bound to an activity.

3.4 Collecting Information for BPEL4Chor

The ultimate aim of the splitting process in this thesis is to output the BPEL4Chor Choreography on the process upon the partition specification given. The idea is to prepare the output so that we collect the information pieces available in each stage of the splitting procedure, store the them in intermediate data, at the end assemble all pieces together for output.

After Process Fragmentation, the information useful for BPEL4Chor in the run-time is:

- For <topology>
  - <participantTypes> can be created, since we now know about the BPEL fragment processes, for each <participantType> we can get the information for attribute participantBehaviorDescription.
  - <participants> can be created, since we already know the participantType, for each fragment process there is a <participant>. We can get information for the attribute name and type, but the attribute selects will not be available until next stage Chapter 4 – Control Link Fragmentation.
  - <messageLinks> will not be available until next stage Chapter 4 – Control Link Fragmentation.
- For <grounding>, so far, there is nothing to collect, because the handling of the message link comes in Chapter 4 – Control Link Fragmentation and Chapter 5 – Data Dependency Fragmentation.

3.5 Summary

In this chapter, the results can be summarized as follows: (1) one WSDL definition per fragment process is created. In addition, artifacts for maintaining the communication between clients and the fragment processes are copied from original process's WSDL definition into the corresponding WSDL definition, (2) one fragment process, which sets the global DPE as on ("suppressJoinFailure=yes"), is created for each participant defined by partition specification file or designer, and (3) available information for BPEL4Chor is also collected.

4 Control Link Fragmentation

Fragmenting Control Link is to split the control link on the base of whether or not the control link crosses over processes given a specific partition specification. In the former case the control flow is transmitted by exchanging message between the processes. In the step 5 of the Figure 4.1, we can see that in this stage the control link is to be fragmented. The results of step 4 (cf. Chapter 3 – Process Fragmentation) are taken as input, and “step 5 - fragmenting control link” will be run, output of the procedure will be the modified fragment processes, WSDL definitions, and BPEL4Chor artifacts i.e. message links. In this chapter the details of fragmenting control link will be explained.

The concept to fragment control link is presented at the beginning, then we dive into the topic fragmenting the control link in BPEL, at the end a summary is given.

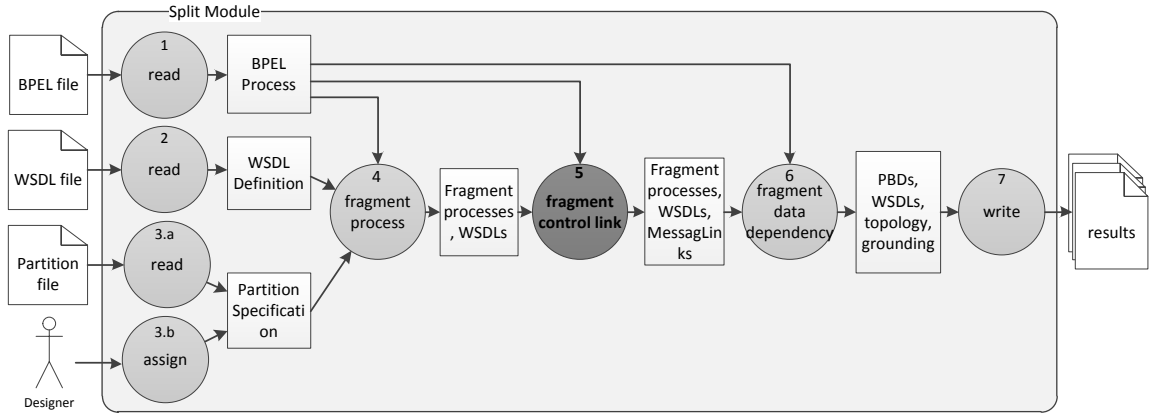

Figure 4.1: The Overview of the Split Module with Fragment Control Link (Step 5) emphasized

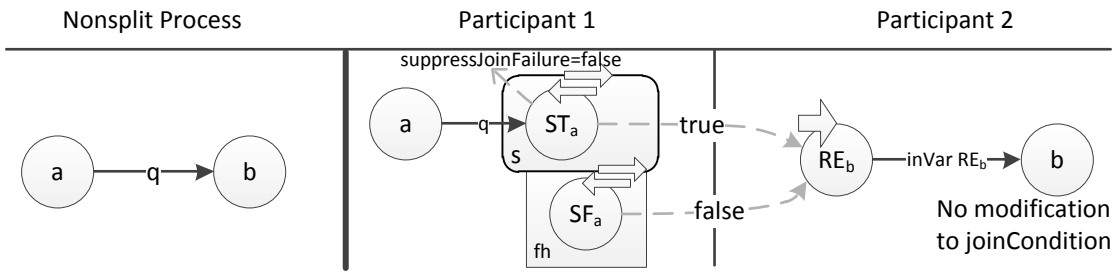

Figure 4.2: Concept for Splitting Control Link across Processes [Kha08], left: Non-Split Process, right: Participant 1 and 2

4.1 Concept to Fragment Control Link

The fragmenting control link procedure in this thesis follows the concept introduced in [KL06, Section 4.3] and [Kha08, Section 5.8] with some adjustment due to the variable initialization as shown in next section.

The concept is introduced as follows. Figure 4.2 illustrates the concept of fragmenting control link. The non-split process (left) in the figure contains two activities, *a* and *b*, with is connected by a link *l(a, b, q)*. The notion *l(a, b, q)* is used to indicate the link from activity *a* to activity *b* with *q*, which is a Boolean expression and can be omitted if it is not specified. The activity *a* is placed in Participant 1 (right) after the fragmentation, and the activity *b* in Participant 2 (right). Besides the placement of *a* and *b*, some constructs are created in Participant 1 and 2. In Participant 1, the constructs include a `<scope>` activity *s* which contains an `<invoke>` activity named ST_a and a `<faultHandler>` named *fh*, which contains an `<invoke>` activity named SF_a. The activity *a* is routed to activity ST_a via link *l(a, ST_a, q)*. In Participant 2, the new construct is a `<receive>` activity named RE_b which goes to activity *b*, via a link *l(RE_b, b, inVar)*. The *inVar* is inputVariable of `<receive>` RE_b. The `<invoke>` activity ST_a sends the message, whose value contains the evaluation result (`true` or `false`) of the transition condition *q* to the `<receive>` activity RE_b in Participant 2 while the `<invoke>` activity SF_a sends a `false` in case of exception is thrown. To note is that the `<invoke>` activity ST_a turns the DPE off by setting "`suppressJoinFalure=false`", so a *joinFailure* Exception will be thrown and caught by `<faultHandler>` *fh*, once activity ST_a's *joinCondition* evaluates to `false` and consequently the activity is not executed.

The original link *l(a, b, q)* is split across the participants. We regard the newly created constructs in Participant 1 as *sending block*, and the one in Participant 2 as *receiving block*. The

a	**transitionCondition of *l(a, b, q)***	**joinCondition of *b***
succeed	true	true
fail	false	false

Table 4.1: Status Evaluation by Original Link *l(a, b, q)*

a	**transitionCondition of *l(a,* ST_a*, q)***	**joinCondition of** ST_a	**transitionCondition of *l(*RE_b*, b, inVar)***	**joinCondition of *b***
succeed	true	true	true	true
fail	false	false	false	false

Table 4.2: Status Evaluation by Exchanging Message

control is propagated from activity *a* to activity *b*, via message exchange between sending block and receiving block. The message encodes whether the control is in valid- or faulty status.

Normally, after the activity *a* in Participant 1 is run, the transition condition of the link *l(a,* ST_a*, q)* evaluates to `true`. Then `<invoke>` activity ST_a is executed and invokes the process of Participant 2 by sending message containing the value `true`. The `<receive>` activity RE_b receives the message and fires the link *l(*RE_b*, b, inVar* with value from the message. As the join condition of activity *b* is not changed, the control flow arrives at activity *b*. That way, the behavior of the original link in the non-split process is reproduced in the fragment processes.

In scenarios that the activity *a* in Participant 1 fails, the failure is suppressed in the successive activity, since the global DPE is turned on in the non-split process, and the fragment processes inherit it along with the fragmentation. The `joinCondition` of `<invoke>` activity ST_a will evaluate to `false`. Recall that `<invoke>` activity ST_a overrides the global DPE by setting "`suppressJoinFailure=false`" for itself, which literally means that a `bpel:joinFailure` will be thrown due to the value of `joinCondition` of activity ST_a evaluating to `false`. Because the failure was thrown the activity ST_a will not be performed. On the other hand, the `<faultHandler>` activity *fh* at that moment is triggered by the fault, i.e. it contains a corresponding `<catch>` construct. After the fault is caught, the `<invoke>` activity SF_a is executed and sends the message containing `false` to Participant 2. The `<receive>` activity RE_b receives the message and fires the link *l(*RE_b*, b, inVar)* with value of the incoming message. The value of *inVar* therefore is `false`. Consequently, `joinCondition` of activity *b* fails. This behavior is also analogous to the original link.

Table 4.1 illustrates the evaluation of the `transitionCondition` of link *l(a, b, q)*, and the `joinCondition` of activity *b*, while Table 4.2 illustrates the evaluation of `transitionCondition` of link *l(a,* ST_a*, q)*, `transitionCondition` of link *l(*RE_b*, b, inVar)*, `joinCondition` of activity ST_a, and `joinCondition` of activity *b*. In both tables, which use whether activity *a* succeeds as beginning condition, the evaluation result of each column is listed. In fact, the evaluation of

the transition condition of the link depends on both the success of the activity *a* and the value of the *q*, for the calculation in both tables we simply assume that *q* evaluates to *true*.

In *both* Table 4.1 and Table 4.2, if activity *a* succeeds, then `joinCondition` of activity *b* evaluates to `true`, otherwise to `false`. In other words, in this concept, the operational semantic of the original link *l(a, b, q)* is kept by exchanging message between Participant 1 and 2 (cf. Figure 4.2).

4.2 Fragmenting Control Link in BPEL

In the last section, the conceptual fragmenting control link is introduced. A simplified scenario where a control link across processes is given, and the solution was that one re-produces the control flow of the original link by creating the *sending block* and *receiving block* each in the corresponding fragment process. In this section we focus on how one fragments the control link in BPEL.

Consider a more general scenario, based on the previous chapter, where we get fragment processes by splitting the main process with the help of partition specification. As shown in Figure 3.6 the control links in the fragment processes are still not handled. The question is how to apply the concept for fragmenting control link on those control links that are split, and how to handle the links that are not split.

Besides the link, we will encounter an issue that relates to the variables referred in the *sending block*. The variable in BPEL has to be initialized before it is used. It leads to the question of how should the variables referred in *sending block* should get initialized before the *sending block* sends the message. The following subsections will cover the questions here mentioned.

4.2.1 Algorithm for Control Link Fragmentation

Procedure SPLIT-CONTROL-LINK in Algorithm 4.1 splits the control links based on the fragment processes ($\mathcal{S}$) and their WSDL definitions ($\mathcal{D}$) that are given as input parameters.

First we introduce several functions used in the procedure SPLIT-CONTROL-LINK. The function *basicActivities(proc)* at line 3 is for getting all the basic activities of a process given, i.e. the '*proc*', while the function *sources(a)* at line 4 gets all associated *sources* of the activity '*a*'. And the function *link(s)* at line 5 returns the link associated to the source '*s*'. The function *definition(proc,*$\mathcal{D}$*)* at line 8 and analogous at line 9 looks up corresponding WSDL definition of the process '*proc*', in the set of WSDL definition '$\mathcal{D}$'.

The procedure SPLIT-CONTROL-LINK goes through all the fragment processes (line 2) and checks for each basic activity of the current fragment process (line 3) whether each of its outgoing control links (line 4) cross over processes: (i) If the expression in line 7 evaluates to

Algorithm 4.1 Splitting Control Link

```
1: procedure SPLIT-CONTROL-LINK(S, D)        // S: set of processes, D: set of definitions
2:     for all proc ∈ S do
3:         for all a ∈ basicActivities(proc) do
4:             for all s ∈ sources(a) do
5:                 l ← link(s)
6:                 proc_t ← GET-TARGET-FRAGMENT-PROCESS(l, S)
7:                 if proc ≠ proc_t then          // whether link l crosses over processes
8:                     dfn ← definition(proc, D)
9:                     dfn_t ← definition(proc_t, D)
10:                    CONSTRUCT-PREREQUISITE(dfn, dfn_t)
11:                    CREATE-SENDING-BLOCK(proc, dfn, l)
12:                    CREATE-RECEIVING-BLOCK(proc_t, dfn_t, l)
13:                else
14:                    COMBINE-NONSPLIT-LINK(proc, l)
15:                end if
16:            end for
17:        end for
18:    end for
19: end procedure
```

`true`, i.e. the link does crosses over processes, then split the explicit control dependency by creating sending- and receiving block (line 8-12) in the source- and target process. (ii) If the expression in line 7 evaluates to `false`, which means the `If`-branch in line 13 will be run, then combine the link l's source and target together (line 14).

The GET-TARGET-FRAGMENT-PROCESS(l, $\mathcal{S}$) at line 6 gets the fragment process targeted by the link given. Note that behind the lines (5-7) is the idea of how one determines whether the link crosses over processes.

We examine the lines (5-7) with the example in Figure 3.6. Assume that l in line 5 is currently *linkA2B* and the $proc$ is p_1, then the result of the line 6 is that the value of $proc_t$ is the target fragment process of *linkA2B*, p_2. Since $p_1 \neq p_2$, the expression $proc \neq proc_t$ in line 7 evaluates to `true`. Now we know that the *linkA2B* crosses process boundaries. On the other hand, if we take the *linkB2C* as l (line 5) and the $proc = p_2$, after GET-TARGET-FRAGMENT-PROCESS(l, $\mathcal{S}$) at line 6 is executed, we get $proc_t = p_2$. Given values of $proc$ and $proc_t$ the expression $proc \neq proc_t$ will evaluate to `false`, which means that the *linkB2C* does not cross processes.

In the CONSTRUCT-PREREQUISITE(dfn, dfn_t) at line 10, the prerequisites, will be created in the definition dfn and dfn_t. As mentioned in Table 3.1, the underlying WSDL artifacts (prerequisites) such as message and portType will be created there. That procedure will be introduced in section 4.2.2–Constructing Prerequisite. The CONSTRUCT-PREREQUISITE(dfn, dfn_t)

```
1   <wsdl:message name="ControlLinkMessage">
2      <wsdl:part name="status" type="xsd:boolean"/>
3      <wsdl:part name="correlation" type="xsd:string"/>
4   </wsdl:message>
```

Listing 4.1: Control Link Message Specification

enables the CREATE-SENDING-BLOCK($proc,\ dfn,\ l$) at line 11 and the CREATE-RECEIVING-BLOCK($proc_t,\ dfn_t,\ l$) at line 12. The former procedure will be introduced in section 4.2.3–Creating Sending Block and the latter in section 4.2.4–Creating Receiving Block.

In the *If*-branch at line 13, the expression $proc \neq proc_t$ evaluates to `false`, which means that the link l does not cross processes. The COMBINE-NONSPLIT-LINK($proc,\ l$) at line 14 will combine the link l with its other part in the process $proc$.

The procedure SPLIT-CONTROL-LINK terminates after each fragment process has been processed. It iterates in the following sequence: (1) process (2) activity (3) source (4) link. Each iteration consumes one outgoing link of an activity and the corresponding incoming link that associated to another activity. As a result, when the procedure is ended, all the outgoing links and their associated incoming links are processed.

4.2.2 Constructing Prerequisite

The procedure CONSTRUCT-PREREQUISITE($dfn,\ dfn_t$) in Algorithm 4.1 at line 10 creates the prerequisite artifacts for the *sending block* and *receiving block*. The dfn is WSDL definition of the fragment process that contains the source of the link l, and the dfn_t is WSDL definition of the fragment process that contains the target of the link l.

Note that the construction of prerequisites for *sending block* and *receiving block* agrees with the Table 3.1–WSDL Artifacts Re-usability and Multiplicity.

The major steps of the procedure are as follows:

1. Create Control Link Message
2. Create PortType and Operation
3. Create PartnerLinkType and Role

We now examine these steps in a more concrete manner.

1. Create Control Link Message

   As shown in the Listing 4.1, the control link message contains two parts: one part is named “status” of type `boolean`, the other part is named “correlation” of type `string`. The part “status” (`true` or `false`) represents the transition condition of the split control link,

and the part "correlation" is previously set in main process and should be copied from main process for message routing in the scenarios of parallel process instances.

Only one control link message is necessary for the inter-communication between two fragment processes. The message is to be generated in the WSDL definition of the process to which one sends the message. Consider that before a control link l_2 crossing process boundaries, there was already another control link l_1, which also crosses the same fragment processes and was handled previously. In that scenario, the reusable control link message should have been created in the WSDL definition, which is exactly the same place we want to create the prerequisite message definition for splitting the control link l_2. So at the beginning it should be tested whether the message existed in the WSDL definition dfn_t. Should it not exist, then a new such control link message is generated, otherwise, skip.

2. Create PortType and Operation

   The PortType between two processes is reusable, thus, the existence of the PortType for control flow from invoking process $proc$ to invoked process $proc_t$ is tested, before a new one gets created in the WSDL definition of the invoked process, in our case $defn_t$. The test condition for existence in this thesis is based on some naming conventions that all the PortTypes agree with.

   In contrast to PortType, a new operation must be created with a unique name for each control link, and be placed in the PortType that the operation is associated to.

   As a result, the PortType is referred by attribute `portType` in the `<invoke>` activities and the `<receive>` activities that are responsible for exchanging message from the invoking process $proc$ to the invoked process $proc_t$. And the attribute `operation` in `<invoke>` and `<receive>` activities varies each time.

3. Create PartnerLinkType and Role

   PartnerLinkType and Role are also reusable, analog to the PortType. One PartnerLinkType with a Role is created to describe the Partner-Relationship between the invoking process $proc$ and the invoked process $proc_t$. The existence checking happens to the creation of PartnerLinkType and Role too.

4.2.3 Creating Sending Block

The procedure CREATE-SENDING-BLOCK($proc,\ dfn,\ l$) creates the constructs for sending message, which contains the status of the control link l, in the process $proc$.

Before the details of the procedure are explained, we go in the issue of *variable initialization* mentioned at the beginning of the ascendant section 4.2. Variables in the process $proc$ are of message type as defined in section 3.1. The attribute `variable` in out-bound activity of

```
<variable name="NCName" messageType="ControlLinkMessage">+
   from-spec?
</variable>
```

Listing 4.2: Syntax for Declaring Variable referred in Sending Block

```
(1)<from variable="BPELVariableName" part="NCName"?>
     <query queryLanguage="anyURI"?>?
        queryContent
     </query>
  </from>
(2)<from partnerLink="NCName" endpointReference="myRole|partnerRole" />
(3)<from variable="BPELVariableName" property="QName" />
(4)<from expressionLanguage="anyURI"?>expression</from>
(5)<from><literal>literal value</literal></from>
(6)<from/>
```

Listing 4.3: From-Spec Variants in BPEL [OAS07]

the sending block will refer to a pre-defined message of type *ControlLinkMessage* as shown in Listing 4.1. Implicitly, the out-bound activity e.g. `<invoke>` in the *sending block* must be equipped with a variable of type *ControlLinkMessage*, whose 'status' part is set to `true` or `false`, before it sends the message.

Variable Initialization

There are three possible ways to initialize variable in BPEL process:

1. Variable initialization with `<assign>` activity

   The conventional method to initialize a BPEL variable is to use `<assign>` activity. The idea is to create a `<assign>` activity in front of each out-bound activity, and assign the `true` or `false` value to the message referred by the out-bound activity. Regarding the concept introduced (cf. Figure 4.2), we create two `<assign>` activities. One is for assigning `true` to the variable referred by the out-bound activity, and the other for assigning `false`.

   It is obvious that this method is straightforward and simple. The disadvantage is there will be two `<assign>` more for each link. The generated artifact will look quite complex.

2. Variable initialization within the variable declaration

   An alternative to initialize BPEL variable is to use a *from-spec* inside the variable declaration. It is stated in the WS-BPEL 2.0 Specification that "a variable can optionally be initialized by using an in-line from-spec" [OAS07, Section 8.1, Page 48].

We now do some analysis about the BPEL variable declaration and the *from-spec*, then see how it can be used in our scenario.

The BPEL variables that are used in the sending block are of message type "*Control-LinkMessage*" (cf. Listing 4.1). The BPEL compliant syntax for declaration is illustrated in Listing 4.2. The *from-spec*, if given, works like a virtual-assign, which copies the value specified by the *from-spec* to the variable.

In the BPEL specification there are 6 variants of *from-spec* shown in the Listing 4.3. Since the variable we used is of message type and the endpoint assignment is also not relevant (cf. BPEL subset definition in page 20), the <from>-variants that can copy value to the sending block variable narrow down to variant (1) and (5).

If variant (1) is used to assign our variable, it means that we need another variable that can be set in the attribute of the <from>. It must be of exact the same message type so that it can be copied to the sending block variable directly. Because in this so called virtual assign, the target is the whole sending block variable, which makes it not possible to do the copy operations from different parts of the other variables to the corresponding parts of the sending block variable. Since we do not have such a variable in the process previously, now just variant (5) may work.

Variant (5) requires that one creates the whole XML structure as the desired message with the information we want for the sending block and set it to the sub-element <literal> in the <from>. In our scenario it is not trivial.

3. Variable initialization within the out-bound activity <invoke>

   Another alternative to initialize BPEL variable is to use the <toParts> inside the <invoke>. <toParts> is used to have data from WS-BPEL variables directly copied into an anonymous WSDL message used by the <invoke>. The <toParts> contains one or multiple <toPart>, each of the <toPart> matches to one part of the WSDL message i.e. "*ControlLinkMessage*". It is like an anonymous message of type "*ControlLinkMessage*". The desired value for each part of this message is given explicitly in the <toParts>, and instead of being labeled with a name, it is directly used by the <invoke> activity as inputVariable. Note that if the <toParts> presents in the <invoke>, the attribute inputVariable must *not* exist. The co-existence of both the <toParts> and the attribute inputVariable in <invoke> is forbidden [OAS07, Section 10.3.1, Page 88].

The target work-flow engine[1] in this thesis does not support the advanced feature with virtual assign via in-line *from-spec*, nor the <toParts> in <invoke>. Therefore, the second and third choices are excluded. The first one is the only candidate, therefore it is chosen to initialize the variables. After the variable initialization emerges into the concept introduced in Figure 4.2, a representation of that concept with variable initialization presents in Figure 4.3.

[1] http://ode.apache.org/ Apache ODE

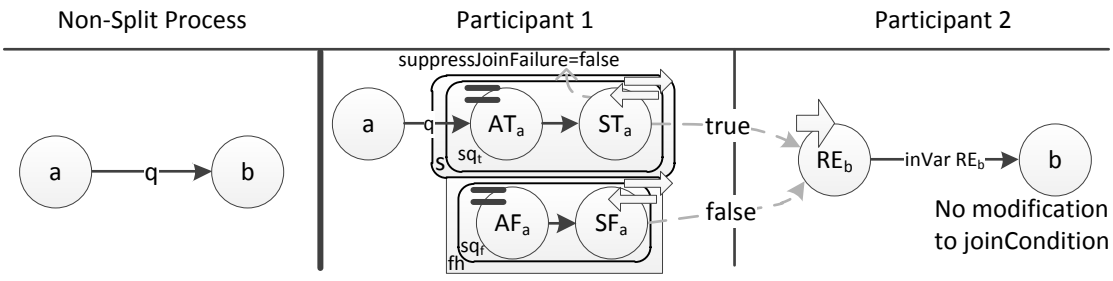

Figure 4.3: Splitting Control Link Concept with Variable Initialization

Figure 4.3 shows the renewed concept for fragmenting control link. Comparing with the concept in Figure 4.2, the difference is that an <assign> activity (AT_a and AF_a) is created before each of the two <invoke> activities in the *sending block*. Since the <scope> 's' can not accommodate multiple activities, a <sequence> activity 'sq_t' is introduced into the <scope> '*s*'. Similarly, the <sequence> activity 'sq_f' is created in the <faultHandler> '*fh*'.

Procedure Details

The procedure CREATE-SENDING-BLOCK($proc$, dfn, l) at line 11 in Algorithm 4.1 creates the *sending block* shown in the renewed concept (cf. Participant 1 in Figure 4.3) in process $proc$. The dfn is WSDL definition of $proc$ and the l is the link which crosses processes. The procedure contains the following main steps:

1. Create constructs to send message containing `true`
2. Create constructs to send message containing `false`
3. Add all constructs into process $proc$
4. Combine the source activity of the link l (cross processes) to sending block
5. Collecting Information for message link in BPEL4Chor artifact <topology> and <grounding>

Now we examine the steps in details:

1. Create constructs to send message containing `true`

   As shown in Participant 1 in the Figure 4.3, a <invoke> activity (ST_a) , whose attribute *suppressJoinFailure* is set to `false`, is created. A `variable` of message type *ControlLinkMessage* is created and set into the attribute `inputVariable` of ST_a. A <assign> activity (AT_a)

is created to initialize the `inputVariable` of ST_a, such that the variable's `status` part evaluates to `true`.

A `<sequence>` activity (sq_t) is created to contain the AT_a and ST_a, with the execute sequence that sq_t comes before ST_a. The sq_t then is set as the child of a new `<scope>` (s).

2. Create constructs to send message containing `false`

   A `<invoke>` activity (SF_a) is created. Its attribute `inputVariable` is set by a new created `variable` of message type *ControlLinkMessage*. Then a `<assign>` activity (AF_a) is created. It sets the `status` part of the `inputVariable` in the SF_a as `false`.

   A new `<sequence>` activity (sq_f) is created. The AF_a and SF_a are added into sq_f with the execute sequence that AF_a is before SF_a. After that, a `<faultHandler>` (fh) is created. Then sq_f is set as child of the fh.

   Add the fh into the scope s.

3. Add all constructs into process $proc$

   After all the constructs are created add the `<scope>` s into top level `<flow>` activity in the process $proc$.

4. Combine the source activity of the link l (cross processes) to sending block

   We set the target activity of the link l as the `<sequence>` activity sq_t in the `<scope>` activity s.

5. Collecting Information for message link in BPEL4Chor artifact `<topology>` and `<grounding>`

   For each split link, there is a new `<messageLink>` created in both the `<topology>` and `<grounding>`. Note that only a part of the attributes in each of the new message links are known in this procedure. The attributes such as '`name`' and '`sender`' are related to the sending block, therefore are available. The rest of the information for the message links will be filled in the procedure CREATE-RECEIVING-BLOCK.

4.2.4 Creating Receiving Block

The procedure CREATE-RECEIVING-BLOCK($proc_t$, dfn_t, l) at line 12 in Algorithm 4.1 creates the *receiving block* shown in the renewed concept (cf. Participant 2 in Figure 4.3) in process $proc_t$. The dfn_t is WSDL definition of the process $proc_t$ and the l is the link from the process $proc$ where the sending block is created. Recall that in $proc_t$ exists one link l_t that has the same name as l. The procedure has the following steps:

1. Create <receive> activity and add into process $proc_t$

   create the <receive> activity (RE_b) and its inputVariable ($inVar$). The $inVar$ is of message type *ControlLinkMessage*. After the creation, add it into the top level <flow> activity.

2. Combine the <receive> activity to the target of the link l (cross processes)

   We lookup the corresponding link l_t in the process $proc_t$, then put the source activity of l_t as the receive activity RE_b. The attribute transitionCondition in the link l_t must be set as the status part of the $inVar$ from RE_b, i.e. the transition condition of l_t contains an expression that accesses the part of status in *"ControlLinkMessage"*.

3. Add information to the message link (BPEL4Chor artifact) regarding the link l

   We continue to update the message links created in the CREATE-SENDING-BLOCK. The attributes e.g. 'receiver' and the 'portType' that relate to the receiving block can be set in this procedure.

4.3 Summary

In this chapter, an algorithm (Algorithm 4.1) applying the concept for splitting control link in [Kha08, Section 5.8] is introduced. In the algorithm, if the original control link crosses partition boundaries, it is split by creating sending block in it source process, creating receiving block in its target process, and exchanging message between the sending- to receiving block. If the original link does not cross partition boundaries, then the source and target of the link are reconnected. Additionally, the issue of variable initialization is discussed and addressed. Some information for BPEL4Chor message links is also collected inside the algorithm.

5 Data Dependency Fragmentation

As depicted in Figure 5.1, data dependency fragmentation (step 6) will be executed after the control link fragmentation.

Unlike the BPEL-D method, the data dependency fragmentation in this thesis splits the data dependency in an implicit way. We analyze the data-flow of the main process (cf. Section 5.1) first. Then against a (data) variable and its reader (activity), the analysis result is able to tell us which writers the reader is dependent on. Based on that knowledge we create the constructs (local resolver and receiving flow, cf. Section 5.4) in the fragment process that contains the writers of the data and in the fragment process in which the reader of the data presents. The local resolver is responsible for summarizing the data from the various writers and sending it to the receiving flow per message (in the case of the writers and the reader being in different fragment processes), while the receiving flow is responsible for collecting the data sent by the local resolver, assembling that data in its previous order, and eventually rerouting the data to the reader. The message, which is exchanged between the two sides, contains not only the data but also the information of whether the writers succeed (`true` or `false`). A Writer Dependency Graph (WDG) is created in order to re-generate the control dependency of the writers in the

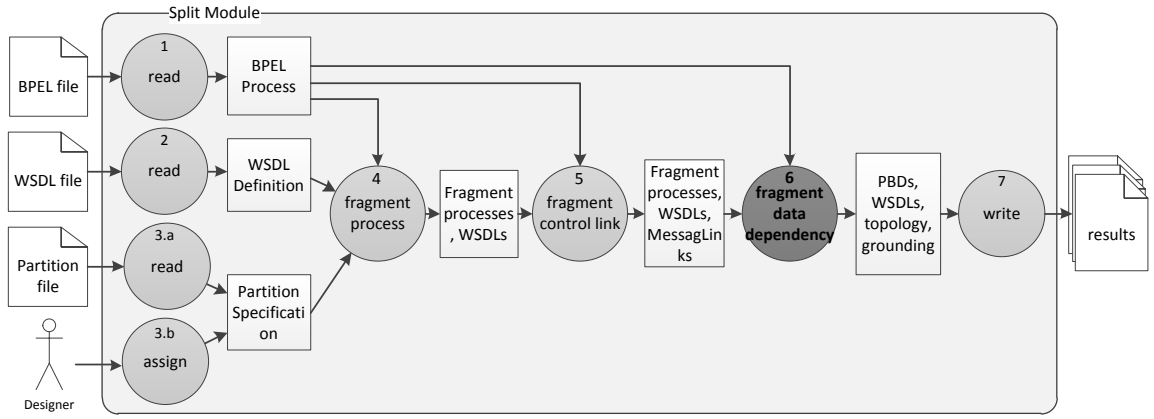

Figure 5.1: The Overview of the Split Module with Fragment Data Dependency (Step 6) emphasized

Activity	Variable	Query	$\text{Writes}_\circ$			
			Possible Writers ($\text{poss}_\circ$)	**Disabled Writers ($\text{dis}_\circ$)**	**Invalid Writers ($\text{inv}_\circ$)**	**May Be Dead ($\text{mbd}_\circ$)**
a	x	.m	$\{w_1, w_2\}$	$\emptyset$	$\emptyset$	false
a	x	.n	$\{w_3, w_4\}$	$\emptyset$	$\emptyset$	false
a	x	.k	$\{w_3, w_4\}$	$\emptyset$	$\emptyset$	false
a	x	ϵ	$\{w_5\}$	$\emptyset$	$\emptyset$	false

Table 5.1: Sample Data-Flow Analysis Results

receiving flow and therefore realize the '*last writer wins*' policy. Then a Partitioned Writer Dependency Graph (PWDG) based on the WDG is created to reduce the quantity of those exchanged messages. The WDG and PWDG are explained in Section 5.2 and 5.3.

5.1 Data-Flow Analysis of BPEL Process

One input of the algorithm for fragmenting data dependency in this thesis is the data-flow analysis of the main BPEL process. It serves the purpose of determining the data dependencies between activities. Such a concept for data-flow analysis has been presented by Kopp et al. [KKL08b, KKL07]. It has been extended by Breier [Bre08] and implemented by Gao [Gao10].

Assume that a variable *x*, which contains two parts, i.e. 'm' and 'n', is read by activity *a* in a process. An example of the write state of variable *x* when activity *a* is reached is presented in Table 5.1.

In Table 5.1, the query is stated as e.g. '.m' which means the part 'm' in variable *x* is read. The column of "Possible Writers" in that table provides the set of writers, which possibly write to the variable element (interpreted by concatenating the variable and the query) as the activity *a* is reached.

The white circle ($\circ$) in the subscript position indicates the write state of the variable *x* before the activity *a* is executed. The '$\text{poss}_\circ(a, x.\text{m})$' is the function that returns the possible writers of 'x.m' before the execution of the activity *a*. It is the only function we are interested in due to the data dependencies. As shown in the table, '$\text{write}_\circ$' is a tuple set, each of the tuples contains a set of possible writers ($\text{poss}_\circ$), a set of disabled writers ($\text{dis}_\circ$), a set of invalid writers ($\text{inv}_\circ$), and the boolean value for 'May Be Dead ($\text{mbd}_\circ$)'. Regarding the possible writers in Table 5.1, we also use the function '$\text{poss}_\circ$' as in [KKL08b], which results in follows:

1. $\text{poss}_\circ(a, x.\text{m}) = \pi_1(\text{writes}_\circ(a, x.\text{m})) = \{w_1, w_2\}$ for row 1
2. $\text{poss}_\circ(a, x.\text{n}) = \pi_1(\text{writes}_\circ(a, x.\text{n})) = \{w_3, w_4\}$ for row 2

3. $\text{poss}_\circ(a, x.\text{k}) = \pi_1(\text{writes}_\circ(a, x.\text{k})) = \{w_3, w_4\}$ for row 3
4. $\text{poss}_\circ(a, x) = \pi_1(\text{writes}_\circ(a, x)) = \{w_5\}$ for row 4, in this case the symbol ϵ represents query to the whole variable, therefore the $\text{poss}_\circ$ contains the writers that write to the whole variable *x*, furthermore the writers that only write to part of the variable *x* do not belong to that writer set.

Recall that $\pi_i(f)$ is used to project the *i*-th component of the tuple *f*. The first component of the $\text{write}_\circ$ tuple is $\text{poss}_\circ$, therefore, $\pi_1(\text{writes}_\circ(a, x.\text{m}))$ is equal to $\text{poss}_\circ(a, x.\text{m})$. The symbol ϵ is used as an empty string and indicates the query on the whole variable, thus, when we concatenate the variable *x* to ϵ, the outcome is still *x*.

To encode the data dependencies determined by the data-flow analysis, we use the function $Q_s(a, x)$ as in [KKL08a] to describe a set of tuples. Each of those tuples consists of a query set and a writer set, and both of the sets are based on a variable *x* that is read by activity *a*. Given a tuple in $Q_s(a, x)$, the writers in the writer set write to variable *x* with the queries in the associated query set. Let a tuple be $(\{.n, .k\}, \{w_3, w_4\}) \in Q_s(a, x)$, that means writer w_3 and w_4 specifically write to the part '$.n$' and '$.k$' of the variable *x* i.e. the writer set $\{w_3, w_4\}$ writes to variable *x* with query set $\{.n, .k\}$.

Let q_s be a tuple in the $Q_s(a, x)$ for variable *x* read by activity *a*. Let (qs, ws) be a tuple in the $Q_s(a, x)$ with the qs as a query set and ws as a writer set. qs contains the queries $\{q_1, q_2, \dots\}$ on variable *x* and ws contains the writers $\{w_1, w_2, \dots\}$ that write to variable *x* using that queries in qs. The function $Q_s(a, x)$ is defined as follows:

$$Q_s(a, x) := \{\ q_s \mid q_s = (qs, ws)\ \wedge\ qs = \{q_1, q_2, \dots\}\ \wedge\ ws = \{w_1, w_2, \dots\}\} \tag{5.1}$$

$Q_s(a, x)$ (equation (5.1)) is designed for the algorithms used in this thesis and therefore has the properties as follows:

1. $\forall q_s \in Q_s(a, x) :\ \pi_1(q_s) = qs = \{q_1, q_2, \dots\} \neq \emptyset$ and $\pi_2(q_s) = ws = \{w_1, w_2, \dots\} \neq \emptyset$.
2. $\forall q_{s_i}, q_{s_j} \in Q_s(a, x),\ i, j \in \mathbb{N} :\ q_{s_i} \neq q_{s_j}\ \Rightarrow\ \pi_1(q_{s_i}) \neq \pi_1(q_{s_j})\ \wedge\ \pi_2(q_{s_i}) \neq \pi_2(q_{s_j})$, i.e. each of the tuples in $Q_s(a, x)$ has a different query set and a different writer set. If either the query sets or the writer sets were equal, the tuples would have been merged. Assume that $(\{.m\}, \{w_3, w_4\})$ and $(\{.k\}, \{w_3, w_4\})$ are two tuples in $Q_s(a, x)$, in this case, they must be merged into one tuple by merging the query sets, that is $(\{.m, .k\}, \{w_3, w_4\})$.

Now we investigate the generation of $Q_s(a, x)$ from the data-flow analysis of the main process. For an instance, based on the analysis results on the variable *x* and activity *a* in Table 5.1, we can create the $Q_s(a, x)$ as follows:

$$Q_s(a, x) = \{\underbrace{(\{.m\}, \{w_1, w_2\})}_{q_{s_1}},\ \underbrace{(\{.n, .k\}, \{w_3, w_4\})}_{q_{s_2}},\ \underbrace{(\{\epsilon\}, \{w_5\})}_{q_{s_3}}\} \tag{5.2}$$

Note that in this example the q_{s_2} is merged from the row 2 and 3 in the Table 5.1, due to the same writer set. In other words, the results of the data-flow analysis must be parsed properly into $Q_s(a, x)$, so that they can meet the expectation from the algorithms in this thesis.

An observation over the Table 5.1 implies that the results of the data-flow analysis for variable *x* read by activity *a* are tuples, each of which is a pair of an individual query and a writer set. The difference from the tuple $Q_s(a, x)$ is that the first part of the tuple is a query instead of a query set in $Q_s(a, x)$. Based on that we merge the tuples from that results whose writer set is the same by forming a query set out of the single queries. At the end, we get the $Q_s(a, x)$. The calculation for $Q_s(a, x)$ based on Table 5.1 is given as follows:

$$\left\{ \begin{array}{l} (.m, \{w_1, w_2\}), \\ (.n, \{w_3, w_4\}), \\ (.k, \{w_3, w_4\}), \\ (\epsilon, \{w_5\}) \end{array} \right\} \Rightarrow Q_s(a, x) = \left\{ \begin{array}{l} (\{.m\}, \{w_1, w_2\}), \\ (\{.n, .k\}, \{w_3, w_4\}), \\ (\{\epsilon\}, \{w_5\}) \end{array} \right\}$$

Function PARSE-RESULT(res, *a*, *x*) is created to parse the data-flow analysis against the variable *x* and the activity *a* that reads it, then the function returns the $Q_s(a, x)$. The input 'res' of the function is the result of data-flow analysis on a BPEL process.

The main steps of the procedure are as follows:

1. Retrieve the tuples for variable *x* and activity *a*
2. Create a single query to writer set map
3. Merge single queries that have equal writer set
4. Return $Q_s(a, x)$

The steps are detailed as follows:

1. Retrieve the tuples for variable *x* and activity *a*

   From the result *res*, we get all the queries, that are pointed to variable *x* and read by activity *a*, and their corresponding writers set, e.g. for query '.m' in Table 5.1 get the poss$_\circ$(*a*, '*x*.m') = $\{w_1, w_2\}$.

2. Create a single query to writer set map

   Based on the tuples from last steps, we create a map *m* with the single query as key and its associated writers set as value, e.g. *m*(.m) = $\{w_1, w_2\}$.

3. Merge single queries that have equal writer set

   Create a new tuple set for $Q_s(a, x)$. Check every two tuples of *m* whether their writers set are equal, if yes then merge them as a new tuple for $Q_s(a, x)$, e.g. the *m*(.n)=$\{w_3, w_4\}$,

and *m*(.k)=$\{w_3, w_4\}$, so they get merged as q_s=($\{.n, .k\}$, $\{w_3, w_4\}$) and the q_s is inserted into $Q_s(a, x)$. The rest of the tuples in the map *m* make it self a tuple for $Q_s(a, x)$.

4. Return $Q_s(a, x)$

   The created $Q_s(a, x)$ is returned.

5.2 Writer Dependency Graph (WDG)

Consider a data-flow analysis of the main process against variable *x* for activity *a* has been done. The outcome is $Q_s(a, x)$. By projecting the second part of the tuples $q_s \in Q_s(a, x)$ and by adding them up, we get all the writers that activity *a* is dependent on. In order to describe the control dependencies between the writers, a Writer Dependency Graph (WDG), in which the nodes represent the writer activities and the edges between them represent the control dependencies, is needed.

First of all, we introduce a function $A_d(a, x)$ that returns all the writers that activity *a* depends on due to reading variable *x*. The function $A_d(a, x)$ is defined as the union of the writer sets that get projected from every tuple q_s in $Q_s(a, x)$ [KKL08a]:

$$A_d(a, x) := \bigcup_{q_s \in Q_s(a,x)} \pi_2(q_s) \tag{5.3}$$

We compute the $A_d(a, x)$ with the example equation (5.2) in last section, the result is as follows:

$$\begin{aligned} A_d(a, x) &= \pi_2(q_{s_1}) \cup \pi_2(q_{s_2}) \cup \pi_3(q_{s_3}) \\ &= \{w_1, w_2\} \cup \{w_3, w_4\} \cup \{w_5\} \\ &= \{w_1, w_2, w_3, w_4, w_5\} \end{aligned}$$

5.2.1 Definition of WDG

Let V_d be the set of nodes, and E_d be the set of edges. Let the tuple (v_s, v_t) denote an edge in a WDG, v_s be source of the edge, and v_t be target of the edge. The Writer Dependency Graph (WDG) for the activity *a* and the variable *x* is a graph with the nodes V_d and the edges E_d, and is formally defined as follows [KKL08a]:

$$WDG_{a,x} := (V_d, E_d) \tag{5.4}$$

where $V_d := A_d(a, x)$ and $E_d \subset \{V_d \times V_d\}$. The nodes of the WDG are BPEL activities ($A_d(a, x)$) and each of the edges consists of a source node (v_s) and a target node (v_t).

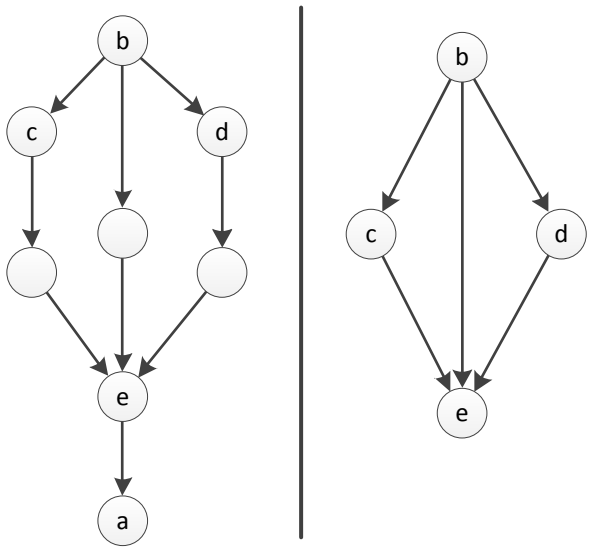

Figure 5.2: Construction of WDG, left: $A_d(a, x) = \{b, c, d, e\}$, right: $\text{WDG}_{a,x}$

Since the BPEL process graph is acyclic [OAS07], a WDG is a directed acyclic graph, therefore has the constraints as follows:

1. The edges are directed.
2. The source and the target of the edge are two different individual writers of *x* in the BPEL process, i.e. $\forall(v_s, v_t) \in E_d : v_s \neq v_t$.
3. The source activity (v_s) and the target activity (v_t) in an edge are not necessarily in the main process directly connected. As long as there is a path in the BPEL process from v_s to v_t and between them there are no other activities that are WDG nodes too. In other words, let $p = v_s, u_1, u_2, \ldots, v_t$ be one path in the main process from v_s to v_t, then $(v_s, v_t) \in E_d \Leftrightarrow u_i \notin V_d$.
4. The graph is directed and acyclic, i.e. there is no such path in the graph in which the start node and the end node is the same one.
5. WDG is independent of the partition specification.

An illustration for generation of the edges is given in Figure 5.2. The process is given on the left side, and the activities, on which the activity *a* depends due to reading variable *x*, are defined with $A_d(a, x) = \{b, c, d, e\}$, which wrote to the variable *x* before activity *a*. Additionally, the blank nodes denote the intermediate activities that do not write to variable *x*. On the right side is the WDG derived from left side. In that WDG one can find that the edge (c, e) and (d, e)

are created because there is a path between them in the process from c (and d) to e, and no activity that belongs to other WDG nodes is between them. The WDG for Figure 5.2 can be defined as follows:

$$WDG_{a,x} = (\underbrace{\{b, c, d, e\}}_{V_d}, \underbrace{\{(b, c), (b, d), (c, e), (d, e)\}}_{E_d})$$

5.2.2 Construction of WDG

Algorithm 5.1 Construct WDG

```
1:  procedure CREATE-WDG(A_d(a, x))
2:      V_d ← A_d(a, x)
3:      E_d ← ∅
4:      for all v_1, v_2 ∈ V_d ∧ v_1 ≠ v_2 do
5:          if |Paths(v_1, v_2)| > 0 ∧ ∃p ∈ Paths(v_1, v_2) : p ∩ V_d = {v_1, v_2} then
6:              E_d ← E_d ∪ {(v_1, v_2)}
7:          end if
8:      end for
9:      return (V_d, E_d)
10: end procedure
```

The Algorithm 5.1 presents the approach to construct a Writer Dependency Graph and requires the $A_d(a, x)$ and the underlying original process. The original process is implicitly used in this algorithm, because it is already given in the input of the splitting procedure (cf. Step 6, Figure 5.1), then we assume that it is also accessible from here. Given the activities $A_d(a, x)$ (line 1) as input parameter, in order to construct the WDG we take all the activities in $A_d(a, x)$ as nodes (line 2). Then we create an empty edge set E_d (line 3) to hold the edges that represent the control dependencies between the writers. To fill the edge set E_d, we test all pairs of the (v_s, v_t) in the V_d whether they form an edge, if they do, create a new edge with them in the edge set (line 4-8). We now examine in particular the condition of being an edge. The function *paths*(v_s, v_t) [Kha08, Section 5.10.4] that finds all ordered paths from v_s to v_t, is used to test whether there is such an ordered path, in which there is no activity that belongs to the other WDG nodes i.e. $V_d \setminus \{v_s, v_t\}$ from v_s to v_t in the process graph. If the expression in line 5 evaluates to `true`, then there is an edge combining v_s and v_t. Consider an example path p from v_s to v_t is $v_s, u_1, u_2, \ldots, v_t$ which make the expression in line 5 evaluate to `true`, that means $p \cap V_d = \{v_s, v_t\}$ and furthermore $u_i \notin V_d$.

The path searching in the underlying BPEL activities presents a challenge in this algorithm. The issue whether there is an edge between v_1 and v_2 can be resolved by (1) searching 'all paths between two nodes' in the process graph, and then (2) testing whether any of the intermediate nodes in a path is another WDG node. All this graph calculation must be run while respecting

the BPEL process graph mechanism. In the BPEL world restricted by the previously defined subset, to navigate from a node *b* to its only successor *c* means that (i) one gets the *b*'s 'source', then (ii) gets the outgoing link from this 'source', next, (iii) gets the 'target' of the link, finally, (iv) gets the 'activity' *c* that is associated to the 'target'. Note that when the node '*b*' has multiple successors then the iteration must go in each of the successors.

After the factor in BPEL navigation is considered, the searching paths in the process graph relates to *Breadth-First Search*. In this case, we refer the interested readers to [CSRL01, Chapter 22].

Recall that the purpose of creating WDG is to re-create the control dependencies of the writers of the variable *x* from which activity *a* reads when it is reached. The writers normally spread across different participants. If these writers and the reader are not in the same participant, the constructs for collecting written results will be created in the process where the writers locate, and the information including the written data and whether the writers succeeded will be sent from that constructs to the reader's process, where the information will be assembled together before it is passed on to the reader. All these are enabled by messaging. To avoid too many messages, some of the results from the writers in the same participant can be merged into one message. The Section 5.4 will handle the creating constructs and sending messages. The next section addresses how to form the writer sets whose information can be merged together.

5.3 Partitioned Writer Dependency Graph (PWDG)

5.3.1 Definition of PWDG

The *partitioned writer dependency graph* (PWDG) characterizes the control dependencies among the subsets of writers, which are in a WDG and write to the variable *x* before the reader activity *a* is reached, based on a partition specification provided [KKL08a].

Consider that we have got the Writer Dependency Graph ($WDG_{a,x}$) against the activity *a* and the variable *x*, and the Partition Specification P. Recall that

$$WDG_{a,x} = (V_d, E_d)$$

is as in equation (5.4). The partition specification P is given as in equation (3.1):

$$P = \{p_1, p_2, \dots, p_k, \dots\} = \{\underbrace{(s_1, M_1)}_{p_1}, \underbrace{(s_2, M_2)}_{p_2}, \dots, \underbrace{(s_k, M_k)}_{p_k}, \dots\}$$

where for $\forall k \in \mathbb{N}$, p_k is a participant, and s_k is the name of participant p_k i.e. $s_k{=}\pi_1(p_k)$, and M_k is the set of activities that are assigned in the participant p_k i.e. $M_k{=}\pi_2(p_k)$.

Khalaf [KKL08a] provided an informal description of the PWDG, which is formalized in the following. Assume that the $WDG_{a,x}$ and the partition P are given, let $M_{k,l}$ be a '*region*' (or

subset) of the WDG nodes in the participant p_k, then the PWDG against the activity *a*, the variable *x*, and the partition specification P is formally defined as follows:

$$PWDG_{a,x,P} := (V_P, E_P) \tag{5.5}$$

with the node set $V_P := \{n_i \mid n_i = (s_k, M_{k,l})\}$ where $\bigcup_{l\in\mathbb{N}} M_{k,l} = V_d \cap M_k$ and $n_i = (s_k, M_{k,l})$ implies $p_k = (s_k, M_k) \in P$, and the edge set $E_P \subset \{V_P \times V_P\}$.

Note that the difference between the M_k and the $M_{k,l}$ is subtle. The former is the set of activities, that are assigned in the participant p_k, and the latter is a '*region*' or a subgraph of the $WDG_{a,x}$, which also belongs to participant p_k. From the graph aspect, the subgraph $M_{k,l}$ is a part of the intersection of the V_d of $WDG_{a,x}$ and the M_k in participant p_k, i.e. $M_{k,l} \subseteq V_d \cap M_k$. The subgraph $M_{k,l}$ and the participant name s_k together form a PWDG node n_i=$(s_k, M_{k,l})$ in the participant p_k, which also implies that a participant holds one or multiple PWDG nodes. If all the subgraphs that belong to the same participant unite, they become the intersection of the V_d and the M_k, i.e. $\bigcup_{l\in\mathbb{N}} M_{k,l} = V_d \cap M_k$.

Each edge in PWDG represents a control dependency between the source- and target PWDG node. The necessary condition to create one edge between two nodes is that there is at least one link whose source and target is in either of the nodes. For instance, let n_1=(x, {B}) and n_2=(z, {D}) be two PWDG nodes, and there is a link l(B, D, true). Consequently, there is a PWDG edge (n_1, n_2).

Until now we have no means to describe the group of WDG nodes that present in a particular participant. Let P be a partition specification, $p \in P$ with the name s and the set of activities M that belong in p, and $WDG_{a,x} = (V_d, E_d)$ the Writer Dependency Graph against variable *x* for activity *a*. We introduce a function $A(a, x, p)$ to represent the group of WDG nodes that are in the participant p. It is defined as the intersection of the node set of the WDG and the activities in the participant:

$$\begin{aligned} A_d(a, x, p) :=& A_d(a, x) \cap \pi_2(p) \\ =& V_d \cap \pi_2(p) \end{aligned} \tag{5.6}$$

where $p = \{s, M\} \in P = \{(s_i, M_i) \mid i \in \mathbb{N}\}$ and $\pi_2(p) = M$.

Figure 5.3 derives from Figure 5.2. We put the partition $P3$ additionally on the process at the left side, so one can see that in one participant there may simultaneously be WDG nodes and non-WDG nodes. Nonetheless, our focus is the WDG, so, we use $A_d(a, x, p)$ to describe the group of WDG nodes that are in a certain participant p. As in Figure 5.3, we get $A_d(a, x, p_1) = \{b, c\}$, $A_d(a, x, p_2) = \{d, e\}$, and $A_d(a, x, p_3) = \emptyset$. Note that in participant p_3 there is no WDG node i.e. no activity in p_3 writes to variable x, consequently, $A_d(a, x, p_3)$ is empty set.

In this section, the characteristics of the PWDG, particularly about the PWDG nodes and edges, are described. Nonetheless, how we create a PWDG node is not mentioned, and nor which

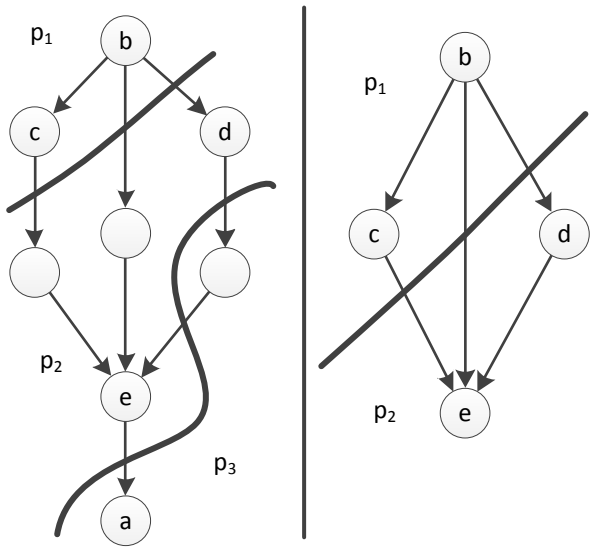

Figure 5.3: WDG with Partition P3=$\{p_1, p_2, p_3\}$, left: $A_d(a, x) = \{b, c, d, e\}$, right: $\text{WDG}_{a,x}$ presents in two participants

criterion we choose a WDG node into a PWDG node with. The following section will cover this topic and introduce the algorithm to create a PWDG base on a given WDG.

5.3.2 Construction of PWDG

The construction of a PWDG based on a given WDG can be divided into two major parts: (1) creating PWDG nodes (2) creating PWDG edges .

For creating PWDG nodes, the main steps are as follows [Kha08, Section 5.10.5]:

1. Insert one temporary root node for each participant where there are WDG nodes and connect it to the nodes which present in that participant and have no incoming edge from that same participant.
2. Create the PWDG nodes by forming the largest weakly connected components in the WDG subgraph cut out via the participant, which do not violate the path constraint.
   a) Recall that a weakly connected component is maximal subgraph of a directed graph, where for any two of the nodes there is a path between them regardless of the direction.

b) Path constraint means that between any two nodes of a (WDG) weakly connected component in one participant, there is no such path where there are (WDG) nodes from other participants, i.e. no path may cross participants.

c) The purpose of the path constraint is to avoid the cyclic dependencies. For instance, let WDG nodes *u* and *v* be in one participant p_1 and there is a path *p*=$u, k_1, k_2, \ldots, v$, assuming that WDG node $k_i \in \{k_1, k_2, \ldots\}$ presents in participant p_2. It means if we put u and v in one WDG component in p_1, it would lead to a dependency from this WDG component (containing u and v) in p_1 to that WDG component (containing k_i) in p_2, due to the path $u, k_1, k_2, \ldots, k_i$, and a dependency from that component in p_2 to this component p_1 due to the path $k_i, k_{i+1}, k_{i+2}, \ldots, v$.

3. Remove the temporary roots and the associated edges from the PWDG

For creating edges, it has been mentioned that a PWDG edge is created when at least one link between the WDG nodes in the source and target PWDG node exists.

Procedure CREATE-PWDG

Algorithm 5.2 Construction of PWDG

```
1: procedure CREATE-PWDG(WDG_{a,x}, P)                    // P: partition specification
2:     R ← INSERT-TEMP-ROOT-NODE(WDG_{a,x}, P)            // R: array of temp root nodes
3:     V_P ← FORM-PWDG-NODES(WDG_{a,x}, P, R)
4:     REMOVE-TEMP-ROOT-NODE(V_P, R)
5:     E_P ← CREATE-PWDG-EDGE(WDG_{a,x}, V_P)
6:     PWDG_{a,x,P} ← (V_P, E_P)
7:     return PWDG_{a,x,P}
8: end procedure
```

The procedure CREATE-PWDG in Algorithm 5.2 creates a PWDG using a given $\text{WDG}_{a,x}$ and partition specification P. The steps of the algorithm are identical to what we discussed for creating PWDG nodes and edges. They are detailed as follows:

1. The function INSERT-TEMP-ROOT-NODE in line 2 takes a $\text{WDG}_{a,x}$ and partition P as input, then creates a temporary node as root for each participant where WDG nodes present and connect it to the WDG nodes that have no incoming edges from the very participant by creating new edges between them. As output the function returns a data structure R, which is an array of temporary root nodes and the array is indexed using participant. It works like a key-value map, i.e. $R[p_1] = r_1$ means that for participant p_1 a temporary root node r_1 is created. In case of the participant p_1 containing no WDG node, there is no temporary root node inserted in this participant, then $R[p_1]$=NIL, where the value of NIL is null.

A (temp) root node is inserted in every participant, because at the beginning of forming largest weakly connected components we need a root to start with, and not every participant have such one. It will be removed again after all is done.

2. In line 3, the PWDG nodes are created by function FORM-PWDG-NODES as shown in Algorithm 5.3, which takes the $\text{WDG}_{a,x}$, partition specification P, and the R as input, and returns the set of newly created PWDG nodes i.e. V_P.
3. After the PWDG nodes are formed, the temporary roots are to be removed. It is done by function REMOVE-TEMP-ROOT-NODE in line 4, which retrieves the root nodes from R and delete them from V_P.
4. Creating the edges happens in function CREATE-PWDG-EDGE (line 5). The function's input includes the Writer Dependency Graph for activity a and variable x ($\text{WDG}_{a,x}$) and the nodes in $PWDG_{a,x,P}$ (V_P). What it does is to search in the PWDG nodes V_P, once any link exists between the WDG nodes that present in two PWDG nodes is found, a PWDG edge is generated between that two PWDG nodes.
5. Finally, the PWDG is instantiated in line 6 and returned in line 7.

In line 2 presents the array of (temp) root nodes, which is likely to raise the question why we need to explicitly store the information about which root node has been created in which participant. This array is needed for two reasons: (i) the partition specification P does not know about these root nodes, since they are inserted in WDG afterward, the (temp) root nodes will be excluded in $A_d(a, x, p_i)$ due to the intersection $A_d(a, x, p_i) = A_d(a, x) \cap \pi_2(p_i)$, where $A_d(a, x)$ is the set of all writers in $\text{WDG}_{a,x}$, therefore, we can not get the root node associated to participant p_i using $A_d(a, x, p_i)$ (cf. line 8 and 9 in Algorithm 5.3). Furthermore, (ii) both the WDG and PWDG are defined as graph, and not designed to facilitate the *find-root* operation easily in a 'subgraph' of them as in any other 'tree' data structure. As a trade-off, we create a array data structure (R) to save the information about which root node is in which participant as the temporary root nodes are being created.

Strategy of Forming PWDG Nodes

The Algorithm 5.3 and 5.4 originate in KOPP's [Kop11a]. They are combined together to form the PWDG nodes. In these two algorithms the PWDG nodes are formed by applying the *Greedy Strategy* [CSRL01, Chapter 16]. They work top-down choosing a WDG node to put into the weakly connected component (PWDG node) as long as no path constraint is violated.

The pattern is that we navigate from the (temp) root node to nodes that reach the border of the partition, and we always make a choice first, whether the path between the parent and the current child violates constraint, before we put the current child node in the component (PWDG node). During the navigation, we choose the WDG nodes from those children that are compatible with those already chosen i.e. they are connected and there is no path constraint

violation, into the weakly connected component. If the path constraint is broken, and the WDG node that is currently reached still presents in the same partition, then we take that WDG node as start point and begin to form a new weakly connected component (PWDG node) again; otherwise it terminates.

Function FORM-PWDG-NODES

Algorithm 5.3 Creation of PWDG Nodes

```
1:  function FORM-PWDG-NODES(WDG_{a,x}, P, R)        // P: partition, R: temp root array
2:      toCheck ← ∅
3:      q ← [ ]
4:      n ← NIL
5:      V_P ← ∅
6:      V_d ← π_1(WDG_{a,x})
7:      for all p_i ∈ P do
8:          toCheck ← A_d(a, x, p_i)
9:          q ← q ∪ {R[p_i]}                          // Begin with (temp) root
10:         while |q| > 0 do
11:             top ← dequeue(q)
12:             n ← (ϵ, ∅)                            // new PWDG node
13:             π_1(n) ← π_1(p_i)
14:             π_2(n) ← π_2(n) ∪ {top}
15:             V_P ← V_P ∪ {n}
16:             for all c ∈ {children(top)} do
17:                 ADD-TO-PWDG-NODE(top, c)   // Recursively, respecting Path Constraint
18:             end for
19:         end while
20:     end for
21:     return V_P
22: end function
```

Function FORM-PWDG-NODES in Algorithm 5.3 presents the main function to form the PWDG nodes. Its input consists of the $WDG_{a,x}$, the partition specification P, and the array of temporary root nodes R.

Line 2 to 6 define the prerequisite variables. A set $toCheck$ (line 2) is defined to hold the WDG nodes that present in a participant. A queue q (line 3) is used to save the start point (line 11) of each PWDG node. The variable n (line 4) is for new PWDG node, while the V_P is the set of PWDG nodes and the V_d set of WDG nodes.

For each participant p_i, PWDG nodes are created, filled, and stored into the set V_P (line 8-18). As preparation for forming PWDG nodes, the WDG nodes inside the current participant p_i (cf. equation (5.6)) are assigned to $toCheck$ (line 8), and with the temporary root node of those WDG nodes, the queue q is initialized (line 9). As long as the queue q is not empty (line 10), we iterate once for forming each new PWDG node. Using the head element of queue q as first WDG node top (line 11) that is filled in the new PWDG node n (line 12-15), we navigate recursively down in the children of the WDG node top (line 16), and add recursively the children satisfying the path constraint into the PWDG node n (line 17). As shown in Algorithm 5.4, if there is any path from the parent WDG node to the current one that crosses other participants, then the current WDG node will be appended to the queue q (line 9 in Algorithm 5.4), and latter be handled as start point for the next new PWDG node (line 11). As result, the V_P is returned in line 21.

A special case of the input parameter R is that in certain participant e.g. p_i there is no presence of any WDG node, then $R[p_i]$ evaluates to NIL, therefore the queue q in line 9 is empty, it leads to $|q| > 0$ in line 10 evaluating to *false*, consequently, the iteration for the participant p_i is simply skipped.

One conjoint observation of Algorithm 5.3 and 5.4 is that the WDG nodes in $toCheck$ are removed one by one and added into corresponding PWDG node. The element of queue q represents the start point of one PWDG node, each of the elements in q means one iteration of the loop, where a PWDG node is created and filled.

Additionally, the function $children(c)$ (line 16) find the successors of *c* in the WDG given (not in BPEL process). The function $dequeue(q)$ (line 11) removes and returns the head element of the queue q.

Function ADD-TO-PWDG-NODE

Function ADD-TO-PWDG-NODE in Algorithm 5.4 recursively handles the children of current WDG node, which is the function's second parameter. And the first parameter of the function is the parent of the current node.

Since function ADD-TO-PWDG-NODE recurses in the scope of the Algorithm 5.3, we assume that the global variables in that algorithm are visible in this function, so that we do not need to pass them into the function ADD-TO-PWDG-NODE using too many parameters. The global variables in the scope of Algorithm 5.3 include n, $toCheck$, and q.

For termination of the function ADD-TO-PWDG-NODE we test whether the node is already placed in another PWDG node or whether the participant that holds it is different from the current one. In either case, the function is to be ended (line 2-7).

If the node c has made it to the line 8, it proves that it does still belong to the nodes that present in the current participant. We test whether there is such a path from the parent p to c

Algorithm 5.4 Add to PWDG Node with Satisfaction of Path Constraint

```
1:  function ADD-TO-PWDG-NODE(p, c)          // p : Parent WDG node, c : child of p
2:      if c ∉ toCheck then                  // Node already in another PWDG node?
3:          return
4:      end if
5:      if participant(c) ≠ p_i then         // Same Participant?
6:          return
7:      end if
8:      if PATH-VIA-OTHER-PARTICIPANT(p, c) then
9:          append(c, q)
10:         return
11:     end if
12:     π2(n) ← π2(n) ∪ {c}                  // Add c to PWDG node n
13:     toCheck ← toCheck \ {c}              // Remove c from toCheck
14:     for all s ∈ children(c) do
15:         ADD-TO-PWDG-NODE(c, s)
16:     end for
17: end function
```

that crosses over other participants (line 8). If so, i.e. PATH-VIA-OTHER-PARTICIPANT returns `true`, the node is appended in the queue q, and the function is ended returning to the superior stack (line 9-10). When no path constraint is violated, i.e. PATH-VIA-OTHER-PARTICIPANT returns `false`, then the node c is added in to the PWDG node n that is being formed (line 12). Subsequently, it is removed from the set $toCheck$ (13).

After the current node has been handled, we process its children recursively by calling the function itself again (line 14-16).

Note that the function $participant(c)$ (line 5) returns the participant of the node given. And the function $append(c,\ q)$ inserts the node c in the tail of the queue q. The function PATH-VIA-OTHER-PARTICIPANT($p,\ c$) is resolved in two steps: (i) find all paths between the parent node p and child node c in the underlying WDG, (ii) test whether the path crosses participants. If any such path is found, the return value is `true`. In the end, if no such path is found, then the returned value is `false`. The former point is similar to the "path searching" problem (cf. page 53) in WDG construction, which can normally be resolved by a *Breadth-Firth Search* with some tweaks. The difference is that the "path searching" in WDG is based on the BPEL process, and the "path searching" in PWDG is based on WDG, i.e. its graph calculation relies on WDG.

A PWDG is dependent on a WDG, therefore its construction relies on the graph calculation of the given WDG. As a matter of fact, if we regard the WDG as a abstract layer on top of the BPEL process, which consists of the writers as nodes and their control dependencies as edges, then a PWDG upon this WDG can be regarded as one abstract layer higher, where a PWDG node is

an aggregation of one or multiple WDG nodes and an PWDG edge represents the connections between the WDG nodes in the source- and target PWDG node. When we project a PWDG node to the underlying main BPEL process, we get the writers in the BPEL process, whose writing results are what we want to merge, in order to reduce the messages number.

Demonstration of PWDG Construction

Figure 5.4 demonstrates the steps of forming the largest weakly connected components in the WDG subgraph, which agree to the path constraint i.e. no path between any pair of the parent-child nodes in the weakly connected component crosses over participants.

To illustrate the construction of PWDG, we examine the Figure 5.4 with the steps in the algorithms.

In box 1, The $WDG_{a,x}$ and partition $P = \{p_1, p_2, p_3\}$ are given, with the properties as follows:

$$\begin{aligned} V_d = \pi_1(WDG_{a,x}) &= \{R, S, T, U, V, W, X, Y\} \\ A_d(a, x, p_1) &= (R, S, X, Y) \\ A_d(a, x, p_2) &= (T, W) \\ A_d(a, x, p_3) &= (U, V) \end{aligned}$$

After the procedure CREATE-PWDG in Algorithm 5.2 begins, it inserts a temporary root node in each participant (line 2), as result, we see three nodes more in box 2 i.e. r_1, r_2, and r_3. Additionally, we get the R with $R[p_1] = r_1$, $R[p_2] = r_2$, and $R[p_3] = r_3$.

Then, it enters the function FORM-PWDG-NODES in Algorithm 5.3. From box 3 to 7, the iteration upon the participant p_1 between line 7 to 20 in function FORM-PWDG-NODES is illustrated. In the iteration, the set '$toCheck$', as well as the queue 'q', is maintained. The $toCheck$ is assigned with WDG nodes in the participant p_1 as initial value, i.e. $toCheck = \{R, S, X, Y\}$, and the q with r_1, i.e. $q = [r_1]$.

We examine the box 3 to 7 as follows:

Box 3 The root node r_1 is dequeued from q and inserted into the newly created PWDG node n_1. Next, for each child of r_1, we recursively choose the child to add into the PWDG node n_1 if no path constraint is violated and the one still belongs in participant p_1 (line 16 to 18 in function FORM-PWDG-NODES).

Box 4 R is removed from $toCheck$ and added into n_1 (line 12 in function ADD-TO-PWDG-NODE), since there is no such path from r_1 to R, that crosses participants. The recursion goes further in R's children, but stops at T while it is already outside of the participant p_1.

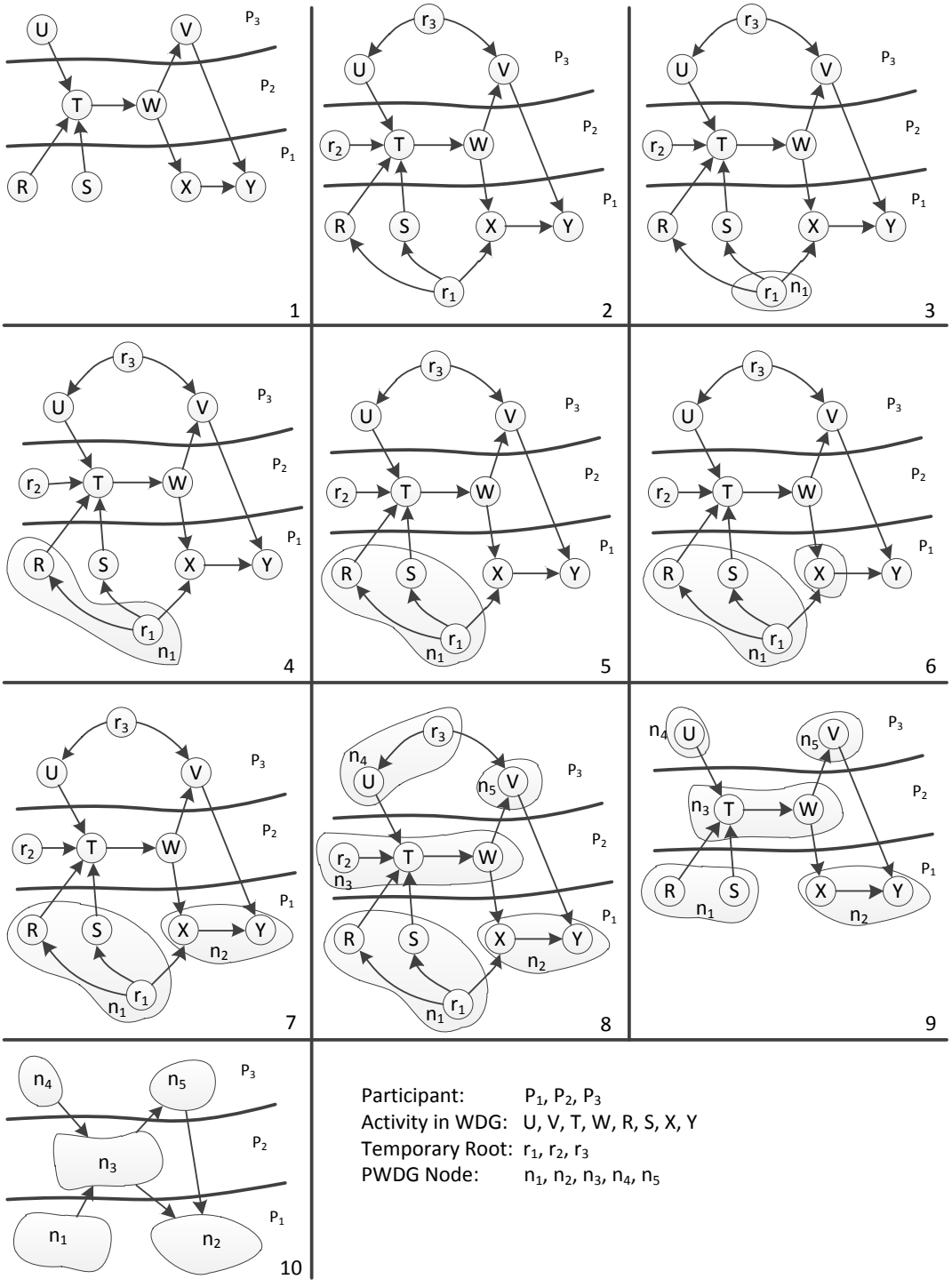

Figure 5.4: Construction of a PWDG (10) from a WDG (1) derives from [Kha08, Fig:41]

Box 5 The same routine iterates in the direction of S. As shown in box 5, S is also removed from $toCheck$ and added into n_1, after that the recursion breaks up at T. At the time, $toCheck$ still contains X and Y.

Box 6 As the procedure goes on, the X is removed from $toCheck$. However, unlike R and S, X is not added into n_1 due to the path r_1, S, T, W, X, where T and W present in other participant p_2. Instead, X is added into q (now $q = [X]$) and subsequently the function ADD-TO-PWDG-NODE executes the line 10 and returns to FORM-PWDG-NODES at line 16. Nevertheless, at this moment, all of the children of r_1 have been investigated, then the currently running function jumps back to line 10 in function FORM-PWDG-NODES, where the expression $|q| > 0$ is evaluated. As of this moment $|q| > 0$ evaluates to `true` because $q = [X]$, then it runs further. The head element of q is dequeued, in this case the X, and is added into a new PWDG node n_2, whose name is the same as p_1's.

Box 7 The recursion goes out of X to Y, and leads to the Y being removed from $toCheck$ and added into PWDG node n_2. Then recursion stops, since there is no children any more, i.e. the recursion for forming the PWDG node n_2 is finished. Finally, it returns to the line 10 in function FORM-PWDG-NODES again, this time $|q| > 0$ evaluates to `false` (now, $toCheck = \emptyset$ and $q = [\,]$), then the iteration for the whole participant p_1 is finished.

In box 8, the PWDG nodes n_3, n_4, and n_5 are also created in other participants using the same principle. Then the function FORM-PWDG-NODES ends up with returning the V_P back to the line 3 in procedure CREATE-PWDG.

After line 4 of procedure CREATE-PWDG is ended, the (temp) root nodes are removed from the PWDG nodes, as in box 9.

Finally, the edges will be created in line 5 of procedure CREATE-PWDG, as in box 10.

5.4 Local Resolver and Receiving Flow

In the construction of local resolver and receiving flow, an expression is needed to describe which query set is written by which writer set inside a PWDG node. Recall that $Q_s(a, x)$ in equation (5.1) denotes the tuples, each of which contains the query set and the writer set inside of the main process. Now, let n be a PWDG node in a PWDG, and let $Q_{sp}(n, a, x)$ denote the tuples out of $Q_s(a, x)$ whose writer sets contain only such writers that appear in the PWDG node n. Similarly, each tuple in $Q_s(n, a, x)$ contains a query set qs and a writer set ws_n, whose subscript n denotes a PWDG node. It is formally defined as follows [KKL08a]:

$$Q_{sp}(n, a, x) = \{(qs, ws_n)\} \tag{5.7}$$

where $qs = \{q_1, q_2, \ldots\}$ and $ws_n = \{w | w \in \pi_2(q_s) \cap \pi_2(n) \ \wedge \ \forall q_s \in Q_s(a, x)\}$.

```
1 <!-- Sample definition for message with single status part-->
2 <message name="Y2ZMessage">
3   <part name="status" type="boolean"/>
4   <part name="data" type="any"/>
5 </message>
```

Listing 5.1: Sample Message Snippet for Sending Variable with Single Query-Set

```
1 <!-- Sample definition for message with multiple status parts -->
2 <message name="Y2ZMessage">
3   <part name="statusa" type="boolean"/>
4   <part name="statusb" type="boolean"/>
5   <part name="data" type="any"/>
6 </message>
```

Listing 5.2: Sample Message Snippet for Sending Variable with Multiple Query-Sets

After we have the PWDG graph, the writer conflicts can be resolved locally in each PWDG node, and constructs are created to collect the writers' data and send them to the reader's process. On the other hand, a `<flow>` activity is created in the reader's process to receive the writer's data and re-create the writer dependencies. The former is named Local Resolver (LR) and the latter Receiving Flow (RF) [KKL08a].

Sending values via local resolver in PWDG node and receiving the values in the reader's process with `<flow>` activity are well illustrated in [KKL08a] and [Kha08]. Therefore, the details about the algorithms i.e. procedure CREATE-LOCAL-RESOLVER-MULTIPLE-WRITERS for local resolver and procedure CREATE-RECEIVING-FLOW are not handled in this section. Instead, the definition of the messages that are sent between writers' processes and reader's process is addressed. Then the preparation of the required artifacts that facilitate the inter-communication is also explained.

5.4.1 Message Specification

The first concern while realizing the splitting data dependency using local resolver and receiving flow is to setup the proper message to encode the status and data of the variable that is written when the reader of that variable is reached. The message depends not only on the data of the variable but also on the status of the write state (`true` or `false`). Additionally, if there are multiple query sets that are required by multiple writers in the PWDG node, then all write states of these query sets should be sent together to the reader. Consider a PWDG$_{a,x}$: in the PWDG there is a PWDG node $n = (s_1, \{B, C\})$ in which the B writes to *x*.a and the C writes to *x*.b. In this scenario, both the write status of the query set $\{.a\}$ and $\{.b\}$ should be sent. That way the write conflicts are also encoded, because the write state of each of the query sets

```
1 <!-- equivalent construct for b.toPart=("data", x) -->
2 <assign>
3 <copy>
4   <from variable="x" />
5   <to variable="varY2Z" part="data" />
6 </copy>
7 </assign>
```

Listing 5.3: Construct Snippet Assigning Value of Variable into Message Part

will be sent and the result will be assembled at the reader side while respecting the control dependencies. As such, the message that is used to convey status and data contains one or multiple status parts. The count of the status part is dependent of the query sets. The Listing 5.1 and 5.2 present the message setup for a single query set and double query sets of a variable.

The pseudo code `b.toPart=("data", x)` shown in line 21 of Algorithm 1 in [KKL08a] implies the concept of copying the data that is written by writers into the 'data' part of message that is sent between sending block and receiving flow. Listing 5.3, which copies the variable *x* to the data part of the variable "varY2Z", is equivalent to this pseudo code. Note that the variable "varY2Z" can be of message type as in Listing 5.1 or 5.2 depending on how many query sets it has.

A challenge is that the type of the data part in a message must be explicitly defined as one copies data of a variable into the data part of the message being sent. According to the WSDL Specification [W3C01] the type attribute of a message part can be either simple data type such as 'boolean' and 'string' or complex type that is defined in in-line schema or external schema. For simplification of the data part assignment, we set the type attribute of the data part in the message that we send to 'any' type as in Listing 5.1 and 5.2. With the 'any' type we take advantage of the XML parsing mechanism in BPEL and can simply set the variable that is written to the data part of the message without pointing out which type this part is of, and latter the right message type for the data part of the incoming message will be set at the receiving flow so that it can be correctly parsed.

5.4.2 Creating Prerequisites

To setup the local resolver and receiving flow, the necessary artifact is not only the message which is referred in the inputVariable of the `<invoke>` activity but also the underlying portType, partnerLinkType, and partnerLink. An argument is that these artifacts are already created as the control links are being split, wherever there are control links that cross process boundaries. So one does not need to create them. It is true, when the PWDG nodes are all in the participants that are directly inter-connected with the participant in which the reader activity presents. If

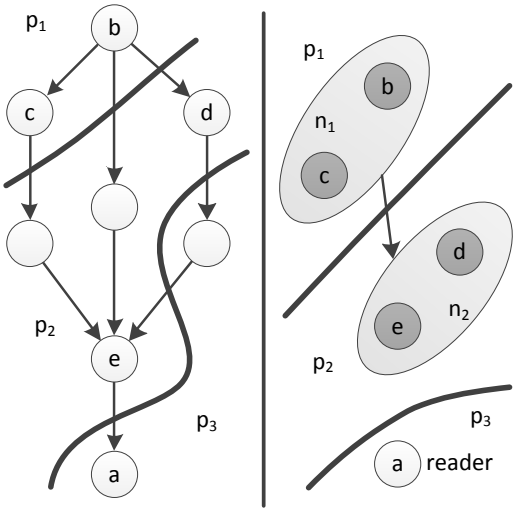

Figure 5.5: Example PWDG Illustrating Scenario for Creating Prerequisites, continued with Figure 5.3. Left: main process with partition $P3 = \{p_1, p_2, p_3\}$ and the dependent writers $A_d(a, x) = \{b, c, d, e\}$, Right: $PWDG_{a,x,P3} = (\{n_1, n_2\}, \{(n_1, n_2)\})$

the reader is in the participant that is not directly inter-connected with some other participants, the prerequisite artifacts for the local resolver and receiving flow may not have been created.

Figure 5.5 shows a scenario where new portType, partnerLinkType, and partnerLink (cf. Table 3.1) should be created. On the left is the main process with the given dependent writers $A_d(a, x) = \{b, c, d, e\}$ against variable *x* for reader *a*. On the right is the PWDG$_{a,x} = (V_P, E_P)$ with $V_P = \{n_1 = (s_1, \{b, c\}), n_2 = (s_2, \{d, e\})\}$ and $E_P = \{(n_1, n_2)\}$. The reader *a* is in participant p_3, which is not directly inter-connected with participant p_1. We examine the creation of the artifacts for the inter-communication in the stage control link fragmentation. We see that in the main process there are control links crossing between p_1 and p_2, as well as between p_2 and p_3, therefore artifacts should have been created in the control link fragmentation and these artifacts can be reused. However, there is no control link across p_1 and p_3 hence no artifacts have been created for the inter-communication between p_1 and p_3, when local resolver and receiving flow are being created due to the PWDG node n_1 and reader *a*. In the case of p_1 and p_3 not being directly inter-connected, new artifacts should be created to support the inter-communication of local resolver and receiving flow.

The function CREATE-PREREQUISITES in Algorithm 5.5 generates the necessary artifacts that are needed while the local resolver and receiving flow are being created. Before the explanation of the algorithm, we introduce the utility functions that are used. We use *participant(a/n)* as an generic function that can accept both activity and PWDG node to get the participant that is

Algorithm 5.5 Creation of Prerequisites for Local Resolver and Receiving Flow

```
1: function CREATE-PREREQUISITES(PWDG, Q_s(a, x))
2:     p_r ← participant(a)
3:     defn_r ← definition(p_r)
4:     for all n ∈ PWDG do
5:         p_n ← participant(n)
6:         Q ← Q_sp(n, a, x)
7:         if p_n ≠ p_r ∧ |Q| > 0 then
8:             CREATE-PREREQUISITE-MESSAGE()
9:             CREATE-PREREQUISITE-PARTNERLINKTYPE()
10:            CREATE-PREREQUISITE-PORTTYPE-OPERATION()
11:            CREATE-PREREQUISITE-PARTNERLINK()
12:        end if
13:    end for
14: end function
```

associated to the input. And consider the *definition(p_r)* as the function that can look up and return the WSDL definition of the input participant p_r. The $Q_{sp}(n, a, x)$ is the tuple set as in equation (5.7) for writers that present in the given PWDG node.

First, the participant p_r which contains the reader *a* and the p_r's WSDL definition, is detected (line 2 and 3). The function iterates through all the PWDG nodes (line 4). In each iteration, the participant p_n that contains the current PWDG node n is detected (line 5) as the tuple set Q is computed using $Q_{sp}(n, a, x)$ (line 6). If the p_n is not the same as p_r i.e. writers and reader are not in the same participant and the tuple set Q is not empty, the necessary prerequisites are created (line 8 to 11).

Note that the functions in line 8 to 11 use the p_n, p_r, $defn_r$, n, and Q as global variables, since the functions are under the scope of function CREATE-PREREQUISITES. Function CREATE-PREREQUISITE-MESSAGE (line 8) creates the messages for local resolver in p_n and receiving flow in p_r. Based on the number of the tuples in the Q ($|Q| == 1$ or $|Q| > 1$), different message is created. If Q contains only single tuple, a message with single status part (cf. Listing 5.1) is created, otherwise a message with multiple status part (cf. Listing 5.2). Should a message with multiple status parts be needed between p_n and p_r, it must be newly created. On the other hand, the message with single status part is reusable between p_n and p_r, since the data part (in that message) that is of type "any" can represent all variable and the status part (in that message) that is of type "boolean" can also be used for all single query set.

Function CREATE-PREREQUISITE-PARTNERLINKTYPE (line 9) creates the necessary partnerLinkType in the WSDL definition $defn_r$ if in the definition there is no partnerLinkType for both of the conversational partners (p_n and p_r). In function CREATE-PREREQUISITE-PORTTYPE-OPERATION (line 10), a new portType in WSDL definition of p_r is to be created and assigned

to one role inside of the partnerLinkType that is created in function CREATE-PREREQUISITE-PARTNERLINKTYPE if there is no portType in WSDL definition of p_r for the conversation between p_n and p_r. After that, a new operation for the link between local resolver and receiving flow is to be created, and this operation should be added into the portType that has been previously created. The function CREATE-PREREQUISITE-PARTNERLINK creates the partnerLink for conversation between p_n and p_r if there is no partnerLink that describes that conversation.

There is an alternative to doing this preparation of the artifacts that enables the conversation between local resolver and receiving flow. One may do the preparation while creating local resolver and receiving flow. The prerequisite artifacts such as a message referred by inputVariable of the <invoke> or the <receive> activity will be created right before the activities are created and need to be equipped with the message. A major disadvantage of this approach is that the abstraction of the logic in the algorithms for creating local resolver and receiving flow will be compromised, and the implementation of the algorithms might get drastically complex.

5.4.3 Creating Message Links for Participant Topology and Grounding

While fragmenting control link, the message links for participant topology and grounding (BPEL4Chor) are created for each split link that crosses process boundaries. The attributes of each message link are collected when the sending block and receiving block are being created. Similarly, a new message link will be created for each pair of local resolver and receiving flow, since a WSDL message will be sent from local resolver to receiving flow. The collection of the attributes can happen while the local resolver and receiving flow are being created.

Note that the original algorithms for creating local resolver and receiving flow are presented in [KKL08a] and are not concerned about the needs of collecting information of BPEL4Chor artifacts, so the logic for creating message link and collecting attributes of that message link should be injected into the algorithms where the local resolver and receiving flow are instrumented.

An alternative to creating the message links while generating local resolver and receiving flow is creating the message links after all the fragmentation of data dependency. In this way one must have a mechanism to iterate in the fragment processes and identify all pairs of local resolver and receiving flow that are present in different processes.

5.5 Putting All Together

The procedure SPLIT-DATA-DEPENDENCY in Algorithm 5.6 presents the high level logic to incorporate the data-flow analysis of the original process, the creation of WDG and PWDG, the

Algorithm 5.6 Splitting Data Dependency

```
1:  procedure SPLIT-DATA-DEPENDENCY(𝒮, 𝒟, process, P)
2:      res ← analyze(process)
3:      for all a ∈ A = {⋃_{p_i∈P} π2(p_i)} do
4:          for all x ∈ resolveVariable(a) do
5:              Q_s ← PARSE-RESULT(res, a, x)
6:              A_d ← ⋃_{q_s∈Q_s(a,x)} π2(q_s)
7:              WDG ← CREATE-WDG(A_d)
8:              PWDG ← CREATE-PWDG(WDG, P)
9:              CREATE-PREREQUISITES(PWDG, Q_s)
10:             for all n ∈ π1(PWDG) do
11:                 CREATE-LOCAL-RESOLVER-MULTIPLE-WRITERS(n, a, x)
12:             end for
13:             CREATE-RECEIVING-FLOW(PWDG)
14:         end for
15:     end for
16: end procedure
```

creation of the prerequisites, the construction of the local resolver and receiving flow together, i.e. to split the data dependency.

Consider *analyze(process)* as the function that invokes the implementation of data-flow algorithm by [Bre08, Gao10] and does the data-flow analysis on the *process* given and returns the result as shown in Table 5.1. The function *resolveVariable(a)* resolves the variables that are read by the activity *a*.

The input of the procedure SPLIT-DATA-DEPENDENCY includes $\mathcal{S}$, $\mathcal{D}$, $process$, and P. $\mathcal{S}$ is the set of fragment processes, as $\mathcal{D}$ is the set of their WSDL definitions. The $process$ is the original process. And P is the partition specification as in equation (3.1) (cf. page 21).

In line 2 the data-flow analysis is run on the original process and the analysis results are returned to the variable res. For all the basic activities in all the participants and the variables that the activities read, i.e. for each possible pair of activity '*a*' and variable '*x*', the operations between line 5 and 13 are carried out.

First, the query set to writer set data structure $Q_s(a, x)$ (cf. equation (5.1)) is created and returned by procedure PARSE-RESULT (line 5), which parses the analysis results, retrieves the information that is associated to (a, x), and forms the $Q_s(a, x)$. Then the control dependencies of the writers are created by the procedure CREATE-WDG in line 7, whose input i.e. the writers set A_d as in equation (5.3) is prepared in line 6. In order to reduce messages, a $PWDG$ is created against the WDG (line 8). Before the local resolver and receiving flow are created, the prerequisite artifacts for the `<invoke>` and `<receive>` activities, that present in the local resolver

and receiving flow, are prepared by the procedure CREATE-PREREQUISITES in line 9, which is shown in the previous Section 5.4.2.

Line 10 to 12 present the logic to generate local resolver ([KKL08a, Section 5.3]) for multiple writers in each of the PWDG node. The procedure CREATE-LOCAL-RESOLVER-MULTIPLE-WRITERS in line 11 creates the corresponding local resolver against the input, i.e. the PWDG node n, the activity a, and the variable x.

After all local resolvers for all the PWDG nodes are ready and are able to send the necessary values, the receiving flow as in [KKL08a, Section 5.4] is created in the participant of the reader (line 13).

5.6 Summary

In this chapter, the details of how to do the fragmentation of data dependency is introduced. At the beginning we have illustrated parsing results of data-flow analysis of the main process to the proper forms i.e. $Q_s(a, x)$ that can be consumed by the algorithms used in this thesis. After that, the creation of Writer Dependency Graph (WDG) and Partitioned Writer Dependency Graph (PWDG) for reducing messages to be sent are elaborated. Then the preparation of local resolver and receiving flow is explained. Finally, a general algorithm (Algorithm 5.6) is introduced to put all the parts together to do the fragmentation of the data dependency.

6 Output in BPEL4Chor Choreography

After the fragmentation of the data dependency, one has split the main process completely. The task at the end is to transform the executable fragment BPEL processes into Participant Behavior Descriptions (PBDs), then output them together with the participant topology and grounding that have been prepared in the previous steps. This step is emphasized in Figure 6.1.

6.1 Participant Behavior Description (PBD)

In this stage the fragment processes are executable BPEL processes. As mentioned in section 2.1, a PBD is an abstract process profile. Therefore, we need to transform each of the executable fragment processes into an abstract process that meets the constraints as follows:

1. Each communication activity contains a namespace wide unique identifier. The identifier is namely the “wsu:id” attribute that is of type “xsd:id”. The attribute “wsu:id” must present especially in the *onMessage* branches of activity `<pick>`.

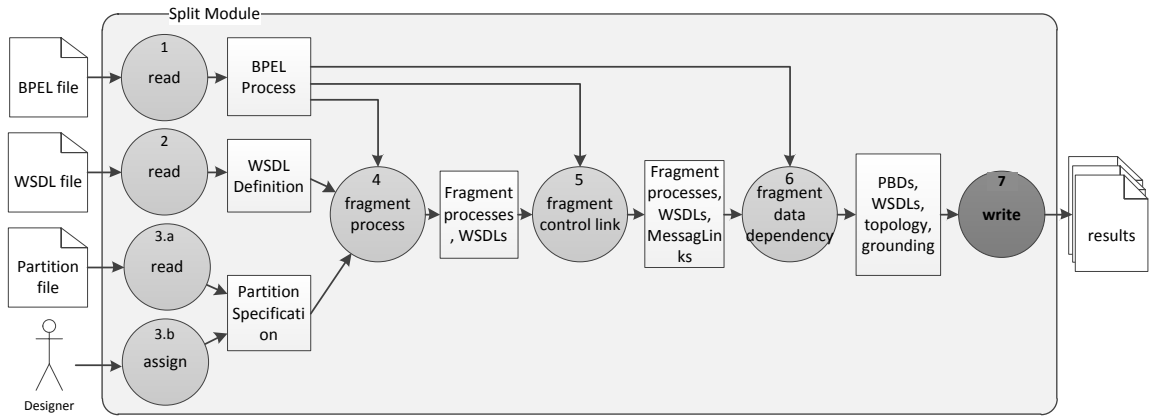

Figure 6.1: The Overview of the Split Module with Output in BPEL4Chor (Step 7) emphasized

```
1  <!-- Syntax definition for participant topology files -->
2  <topology name="NCName" targetNamespace="anyURI">
3    <participantTypes>
4      <participantType name="NCName"
5        (participantBehaviorDescription="QName" processLanguage="anyURI"?)? />+
6    </participantTypes>
7    <participants>
8      <participant name="NCName" type="NCName" selects="NCNames"? scope="QName"? />+
9      <participantSet name="NCName" type="NCName" scope="QName"?
10       forEach="QNames"? >*
11        (<participantSet ... >...</participantSet>
12        |<participant name="NCName" scope="QName"? forEach="QNames"?
13          containment="add-if-not-exists|must-add|required"? />
14        )+
15     </participantSet>
16   </participants>
17   <messageLinks>
18     <messageLink
19       name="NCName"? (default: messageName)
20       (sender="NCName" | senders="NCNames")
21       sendActivity="NCName"?
22       receiver="NCName" receiveActivity="NCName"?
23       bindSenderTo="NCName"?
24       messageName="NCName"
25       (participantRefs="NCNames" copyParticipantRefsTo="NCNames"?)?
26      />*
27   </messageLinks>
28 </topology>
```

Listing 6.1: Participant Topology File Syntax [Kop11c]

2. The partnerLink, portType, and operation attributes in communication activity are excluded.
3. If there is a pair of combined `<receive>` and `<reply>`, an enforced *messageExchange* is created.

6.2 Participant Topology

The participant topology is the structure aspect of the BPEL4Chor choreography. It consists of three main notions: *participantType*, *participant*, and *messageLink*. Listing 6.1 shows the syntax for participant topology that is the output from the splitting procedure. The lines 3 to 6 provide the definition of *participantType*, while the line 7 to 16 is the definition of *participant*. The *messageLink* is defined in the line 17 to 27. Note that there is no web service specific configuration in the topology message link. The *participantTypes* have been prepared in the

```
1  <!-- Syntax definition for participant grounding -->
2  <grounding topology="QName">
3    <messageLinks>
4      <messageLink name="NCName"
5        ((portType="QName" operation="NCName")
6        |(senders="NCNames"? expectedPortType="QName" expectedOperation="NCName"
7          offeredPortType="QName" offeredOperation="NCName"
8          mediator="anyURI"))
9      />*
10   </messageLinks>
11   <participantRefs>
12     <participantRef name="NCName" WSDLproperty="QName" />+
13   </participantRefs>?
14   <properties>
15     <property name="NCName" WSDLproperty="QName" />+
16   </properties>?
17  </grounding>
```

Listing 6.2: Participant Grounding File Syntax [Kop11b]

previous section 3.4 and the *messageLinks* have been prepared in the previous sections: 4.2.3, 4.2.4, as well as 5.4.3.

Note that the syntax in Listing 6.1 uses the informal syntax to describes the XML grammar. The appended character “?” stands for “zero or one”, “+” for “one or more”, “*” for “zero or more”. Elements and attributes separated by “|” and grouped by “(” and “)” are meant to be syntactic alternatives.

6.3 Participant Grounding

The participant grounding provides the web service specific configuration for the choreography. The two main notions are the *messageLink* and *participantRef*. The syntax of the output participant grounding is shown in Listing 6.2. The definition for *messageLink* is in the line 3 to 10. And the *participantRef* is defined in the lines 11 to 17. Note that a port type and operation combination is given in each *messageLink* in participant grounding. Similarly, the information of participant grounding is prepared in the previous sections: 4.2.3, 4.2.4, as well as 5.4.3.

7 Architecture and Implementation

In this chapter, the application implemented in this thesis is introduced. We focus on the architecture, the referred projects, the imported external libraries, and the models that represent XML artifacts or graph.

Note that the implementation is sightly different from the Figure 1.1 and 1.2 in Chapter 1, as a matter of fact the data-flow analysis happens inside of the data dependency fragmentation (cf. Chapter 5).

7.1 Architecture

In this section, the infrastructure of the splitting module is depicted first. Then an overview of the components is presented, as well as their structural behavior and relationship. At last the high level business logic of splitting module is described.

7.1.1 Application Infrastructure

In this thesis, an Eclipse application is created to implement the splitting process approaches, that are presented in [KL06, KKL08a, Kha08]. The main functions of the application are (i) accepting the given BPEL process, WSDL definition, and partition specification as input, then (ii) splitting the process, at the end (iii) outputting the result in the format of BPEL4Chor.

Now we introduce the tools and libraries that are utilized in this thesis. First of all, we take the Eclipse BPEL Model project and its dependent projects out of the Eclipse BPEL Designer for accessing and manipulating the BPEL process. Note that the BPEL standard is based on WSDL 1.1 and this relation also reflects on the runtime dependency of Eclipse BPEL Model. The Eclipse WST WSDL[1] project is the underlying project for reading and writing to the WSDL definition. Besides the projects for BPEL and WSDL, the project implemented by Gao [Gao10] for data-flow analysis is used for analyzing the BPEL process. All of these projects are Eclipse Plug-in application, so we can aggregate them together and build an new Eclipse Plug-in application on top of the Eclipse platform.

[1] http://www.eclipse.org/webtools/wst/main.php Eclipse Web Standard Tools Platform

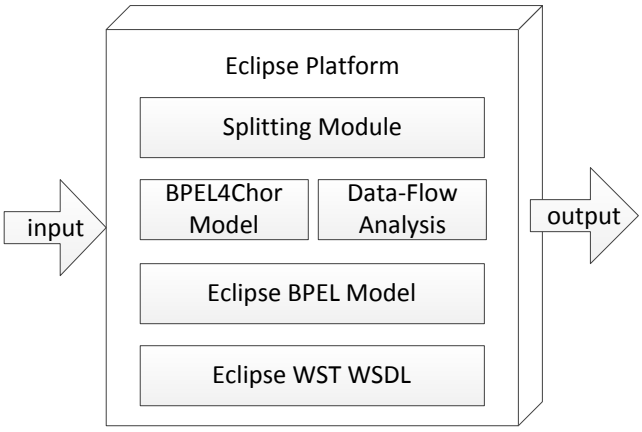

Figure 7.1: Splitting Process Infrastructure

The above mentioned projects can now be regarded as the infrastructure available for the new application. Based on the infrastructure, two new projects are created: the "Splitting Module" and the "BPEL4Chor Model". The former project is mainly responsible for splitting the BPEL process given. The latter project provides a data model of BPEL4Chor and the capability of manipulating the model. The Figure 7.1 illustrates the application infrastructure in this thesis.

As for the external libraries, we utilize StAX[2] and DOM[3] to interact with XML document, Args4j[4] to accept input from command line interface, and JGraphT[5] to provide graph representation for WDG and PWDG.

7.1.2 Component and Data Flow Overview

In this section we observe the design of the application in the aspect of component structure and the aspect of data flow.

Figure 7.2 provides an overview of the major components in the application. The component *org.bpel4chor.splitting* is the new project for splitting process, and the component *org.bpel4chor.model* is the representation of the BPEL4Chor artifacts and provides capability

[2]Streaming API for XML (StAX)
[3]Document Object Model (DOM)
[4]http://java.net/projects/args4j/ (Args4j)
[5]http://jgrapht.org/ (JGraphT)

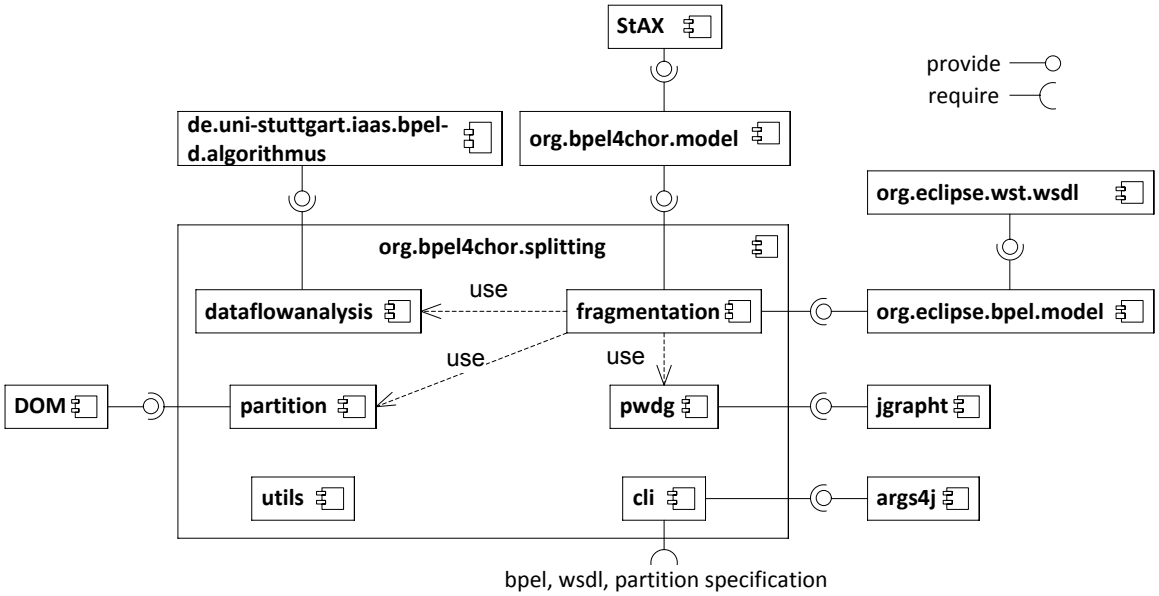

Figure 7.2: Splitting Component Overview

to manipulate the artifacts. Inside the *org.bpel4chor.splitting* are the subcomponents of the project: *cli*, *utils*, *partition*, *dataflowanalysis*, *pwdg*, and *fragmentation*. They are introduced as follows:

1. cli

   The component utilize the external library *args4j* to get the input from the command line and parse the parameters correspondingly.

2. utils

   The component provides the utilities for e.g. resolving variables from an activity.

3. partition

   The component is responsible to create the partition specification instance from either the command line input or the GUI. Note that it uses XML library DOM to parse the partition file from XML format into document object.

4. dataflowanalysis

   The component for data-flow analysis provides the wrapper that calls the analyzer of the process (from the referred project `de.uni-stuttgart.iaas.bpel-d.algorithmus`) and the

parser that converts the analysis result into the format that the algorithms in this thesis can understand.

5. pwdg

   The component is created for construction of WDG and PWDG. It uses the base class `DefaultDirectedAcyclic` and `DifaultEdge` from *JGraphT* to generate the Writer Dependency Graph (WDG) and Partitioned Writer Dependency Graph (PWDG).

6. fragmentation

   The component is created for the process fragmentation, the control link fragmentation, and the data dependency fragmentation. It refers the projects such as `org.eclipse.bpel.model` from Eclipse BPEL Designer for BPEL and WSDL access and manipulation, and also the project `org.bpel4chor.model` for outputting BPEL4Chor artifacts. Furthermore, it utilizes different subcomponents in different steps. In the process fragmentation, the subcomponent `partition` is used for information of the partition specification. During the data dependency fragmentation, the subcomponents `partition`, `dataflowanalysis`, and `pwdg` are used. Note that project `org.eclipse.bpel.model` required by this component requires the further project `org.eclipse.wst.wsdl`.

Figure 7.3 is an overview of the high level business logic. The round circle in the figure means procedure that is executed and the rectangle box in front of and after the round circle means the input and output data. So the application logic can be described in 7 steps as follows:

Step 1 Read BPEL File

Use the `BPELReader` from the project `org.eclipse.bpel.model` to read the BPEL file and generate a BPEL process.

Step 2 Read WSDL File

Use the `WSDLReader` from the project `org.eclipse.wst.wsdl` to read the WSDL file that is associated to the BPEL file and generate the WSDL definition.

Step 3 Read partition specification either from the designer or from a partition file

Either the partition file is given in command line as parameter, the parameter is parsed accordingly and passed to the `PartitionSpecReader` from the component 'partition' in *org.bpel4chor.splitting*, then the `PartitionSpecReader` reads in the file and generates the partition specification, or the designer assigns the activities into partition then partition specification is generated by GUI.

Step 4 Fragment Process

The `ProcessFragmenter` in the *fragmentation* component takes the BPEL process, the WSDL, and the partition specification as input and creates the fragment processes correspondingly. In addition to the fragment processes, the associated WSDL definitions

are also created. Note that the available information for BPEL4Chor artifacts are also collected.

Step 5 Fragment Control Link

The `ControlLinkFragmenter` in the *fragmentation* component takes the output from the process fragmentation and continue to split the control link in the fragment processes. The output of this step is basically the same as the previous step. The differences are that the control links are split and more message links are collected in the BPEL4Chor artifacts.

Step 6 Fragment Data Dependency

The `DataDependencyFragmenter` in the *fragmentation* component takes the output from control link fragmentation and split the data dependencies in the fragment processes. The fragment processes and their WSDL definitions are updated and the information for BPEL4Chor artifacts are also collected. And the fragment processes, WSDL definitions, and the message links for BPEL4Chor are output.

Step 7 Write BPEL4Chor

At the end, the BPEL4Chor artifacts are written into files.

Note that the `ProcessFragmenter`, `ControlLinkFragmenter`, and `DataDependencyFragmenter` are the classes created in the component *fragmentation* for splitting the process.

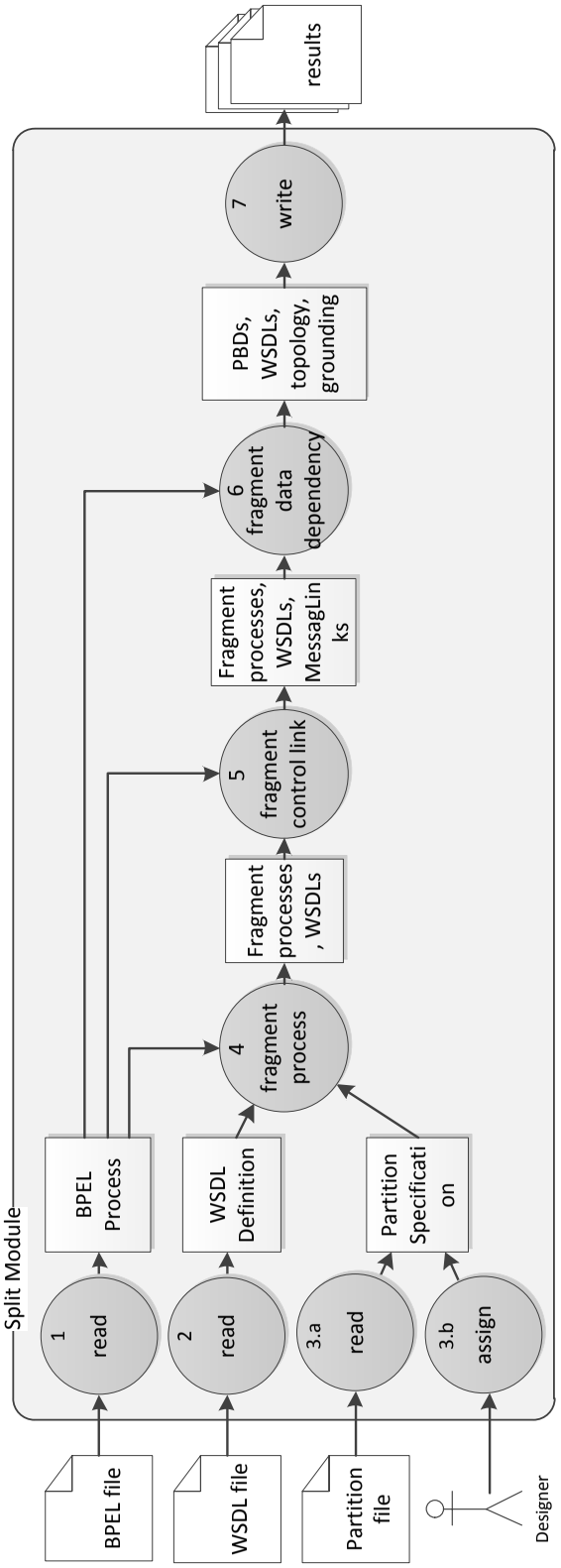

Figure 7.3: The full splitting module

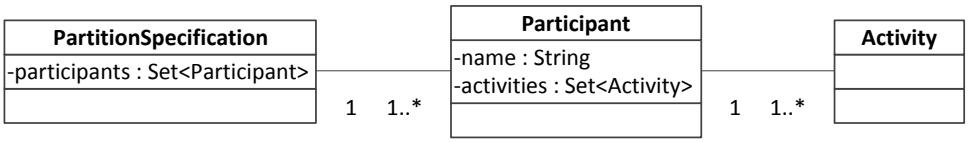

Figure 7.4: Partition Specification Model

7.2 Implementation

In the previous chapters, we have already introduced the algorithms that are combined together to split the process, while the models used in those individual scenarios have not been talked. Then the details of the data models and the graph model is presented in this section.

7.2.1 Partition Specification Model

Recall that one input of the splitting procedure is partition specification (cf. Chapter 3). In the runtime, we need a representation of the partition which tells the splitting algorithm which activity is in which participant, such as the definition (3.1) does to us. As such, we need to design a model for partition specification.

There are two main concerns about the partition specification model. One concern is that it must be reusable, which means it should be resilient to the changes of the underlying BPEL model, as long as the association is not touched. The other concern is that it should be neutral to multiple input variants. The Figure 3.1 has implied that the source of a partition specification can come from a partition file, or it can be provided by designer via some kind of GUI e.g. Eclipse BPEL Designer.

Based on the concerns mentioned above, a model is designed as shown in Figure 7.4. The *Activity* in right side of the model figure presents the BPEL activity. The *Participant* in the middle of the model figure consists of a name and a set of activities. On the left side, is the *PartitionSpecification* that contains a set of participants. In runtime, if one has got an instance of participant, it is simple to get access to the activities that are assigned in the participant.

Since the '*Activity*' is the abstraction of all activity types in the BPEL model, it is very unlikely that it will be changed, unless there is fundamental re-engineering of the BPEL model. Therefore, it fulfills the concern about the re-usability.

On the other hand, the model does not constrain how an instance of `PartitionSpecification` is established. As in Figure 3.1, there are two alternatives to get a `PartitionSpecification`. The

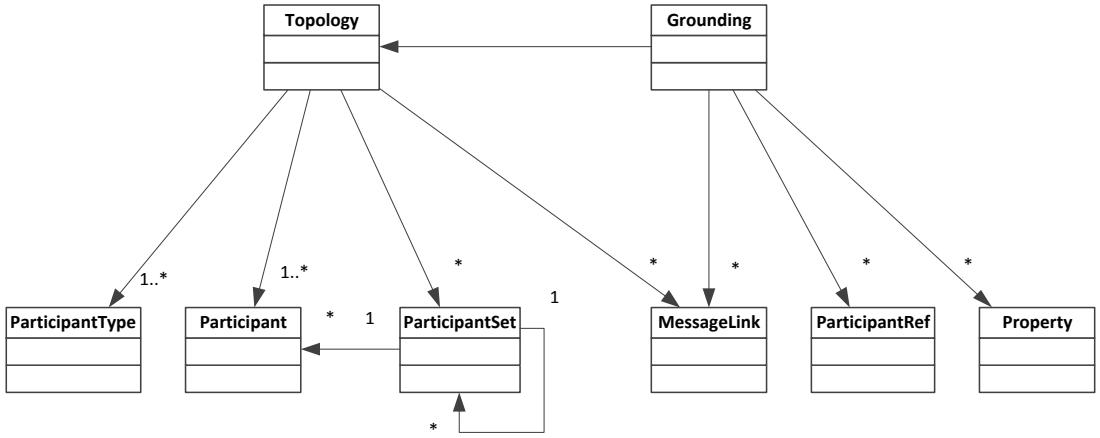

Figure 7.5: Model of Participant Topology and Grounding

first alternative is based on the partition file, and the second one is from the designer, namely based on GUI.

Upon the partition file input, it is assumed that the BPEL process is loaded, but given the circumstance that one does not have direct access to specific BPEL activity in memory. The process is in the memory, but not visible for us. To start the splitting, the split procedure must be told where to find the activity in the BPEL process and which activity is in which participant. A concept to provide the information can be stated as follows: (1) the position of activity is given via XPath stored in the partition file, (2) information about participant is given by grouping the activity XPaths, and (3) the group is labeled with the participant name. This concept is reflected in the Listing 3.1, which is used in this thesis.

The operation pattern upon the partition file input is that the partition file is parsed at the starting time, then the activity XPaths are retrieved, finally the activities in the BPEL process (in memory) are located with help of the XPath, and put into the corresponding participant.

7.2.2 BPEL4Chor Data Model

As introduced in Chapter 6, the participant description behavior (PBD) is an abstract BPEL process, therefore, we use a BPEL process to represent a PBD by explicitly setting the attribute "`abstractProcess`". Additionally, the unique identifier "wsu:id" should be inserted into the communication activities, as well as the onMessage branch in the `<pick>` activity.

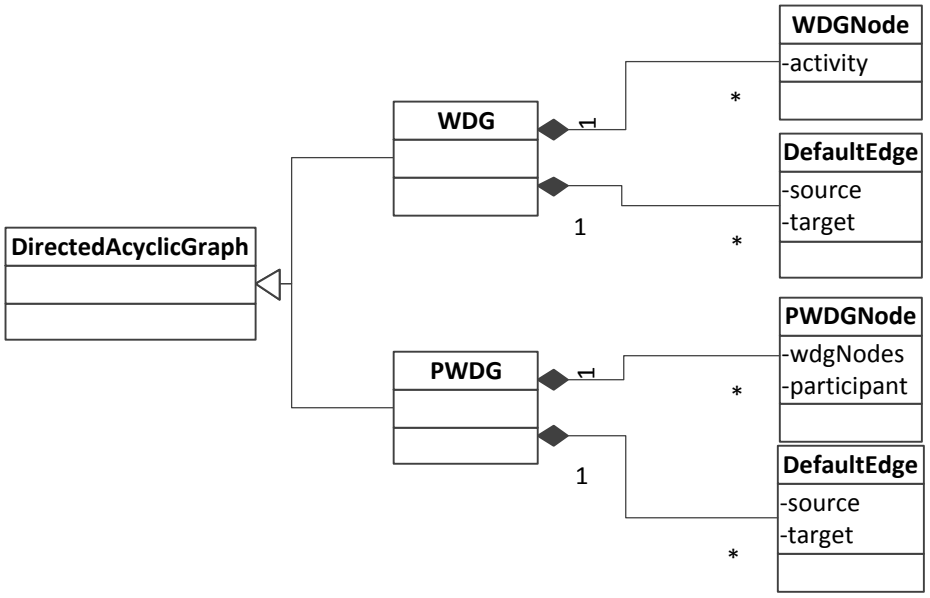

Figure 7.6: Model of WDG and PWDG

Figure 7.5 presents the model for participant topology and grounding. Note that the grounding and the topology refer to the same message link, because the attribute set of the message link is the superset of the attributes that belong to the topology message link and grounding message link. When we write out the message link, which part of the attribute set should be selected can be decided by checking up whether the container is a topology or a grounding.

A challenge in the implementation of the topology and grounding is that one has to manually collect the namespaces in the elements which contain the `QName` attribute, since the XML library StAX that is used to interact with XML document object is not namespace aware.

7.2.3 WDG and PWDG Graph Model

Figure 7.6 shows the graph models for the writer dependency graph (WDG) and the partitioned writer dependency graph (PWDG). We utilize the the `DefaultDirectedGraph` from *JGraph* as base for both graphs, and the `DefaultEdge` as the edge of a graph. A `DefaultEdge` does not constraint the object type of the source and target node, therefore we can use the `DefaultEdge` to represent edges in both graphs. A WDG edge differs from a PWDG edge in the way that it contains two WDG nodes, instead of two PWDG node. Table 7.1 explains the details of the graph models.

Class	Description
WDG	Directed acyclic graph (DAG) which contains a set of nodes and a set of edges
WDGNode	Node of WDG which contains an activity
WDGEdge	DefaultEdge (from JGraphT) which contains a source WDGNode and a target one
PWDG	Directed acyclic graph (DAG) which contains a set of nodes and a set of edges
PWDGNode	Node of PWDG which consists of a participant name and a set of WDG nodes
PWDGEdge	DefaultEdge (from JGraphT) which contains a source PWDGNode and a target one

Table 7.1: Description of the WDG and PWDG Graph Model

The advandage of the both graph models is that they have a clear layering. In other words, from BPEL activity to WDG node, then from WDG node to PWDG node, each layer is just dependent of its underlying layer. If we extend the WDG e.g. to support loop or scope by collapsing it as a WDG node (cf. Chapter 8), the change does not affect PWDG, i.e. the PWDG operates the same way as before.

8 Summary and Future Work

The work in this thesis aims to implement the concepts and algorithms of Khalaf [KL06, KKL08a, Kha08] to generate split BPEL processes out of a plain BPEL process. The concept is extended to output a BPEL4Chor choreography instead of a set of plain BPEL processes.

The contributions of the work start with the implementation of process fragmentation in which the syntax for the partition file is designed, the data model of the partition specification is created, and the main process is split using the partition specification given (cf. Chapter 3).

Based on the fragment processes that are created in the process fragmentation, the control links that are across process boundaries are split using the concept from [KL06]. The concept for splitting control link is implemented with some adjustments due to the variable initialization issue (cf. Chapter 4).

After the control links have been split, we implement the concept from [KKL08a] to split the data dependencies in an implicit manner, i.e., we operate directly on the BPEL processes instead of using explicit data-links in which case one needs to extra transform the BPEL process to an intermediate form (BPEL-D, [KL06]). Implementation for splitting the data dependency includes running data-flow analysis of the main process to derive the explicit data links [KKL08b], creating the writer dependency graph (WDG) and the partitioned writer dependency graph PWDG for avoiding too many messages to be sent, creating the local resolvers (LR) for sending information in the possible nodes of a partitioned writer dependency graph, and creating the receiving flow (RF) for collecting information, reproducing the control flow in a `<flow>` activity, and eventually rerouting the assembled information to the reader activity (cf. Chapter 5).

At the end, the BPEL4Chor artifacts that have been prepared in previous steps are output as result (cf. Chapter 6).

Future Work

During the course of this thesis, the support of loops and scopes is left out, due to the limited capacity . The support of loops and scopes affects all of the steps in the splitting procedure. Therefore these places as follows are identified for future work:

1. Extend process fragmentation to support the loops and scopes (cf. Chapter 3).

2. Extend the control link fragmentation to support the loops and scopes.
3. Extend the data dependency fragmentation to support loops and scopes collapsing in the generation of $Q_s(a, x)$ and writer dependency graph (WDG).
4. Provide coordination protocol for Cross Partner Scope (CPS) using the concept in [Bor10].

A Errata of Related Works

This chapter provides a list of errata found by reading the related works.

Supporting business process fragmentation while maintaining operational semantics : a BPEL perspective [Kha08]

- Page 109, the PARENT_LOOP_OR_SCOPE(child, proc) in function PROCESS_CHILD, at line 1, the function's parameter "proc" should be the main process, NOT a fragment process, because the logic of line 2 and 3 is that one firstly finds the parent of the "child" in the main process, then finds the equivalent construct in the process fragment.
- Page 123, two lines under the Figure 40, the universal quantifier $\forall$ in the equation for an edge existing between v_1 and v_2 as follows:

  $$(v_1, v_2) \in E_d :\Leftrightarrow |Paths(v_1, v_2)| > 0 \ \wedge \ \forall p \in Paths(v_1, v_2) : p \cap V_d = \{v_1, v_2\}$$

  should be changed to existential quantifier $\exists$, namely as follows:

  $$(v_1, v_2) \in E_d \ \Leftrightarrow \ |Paths(v_1, v_2)| > 0 \ \wedge \ \exists p \in Paths(v_1, v_2) : p \cap V_d = \{v_1, v_2\}$$

  . The universal quantifier $\forall$ is not correct because "if there is a path in the process between any two activities in V_d that contains no other activity in V_d, then there is an edge in the WDG connecting these two activities" [Kha08, Section 5.10.4, Page 122].

B Definitions and Notions

This chapter gives a short version of the critical definitions and the notions that are used in this thesis, so that readers can review any particular definition easily and quickly.

B.1 Definitions

B.1.1 Public Functions

Before introducing the definitions, we introduce a function π, which is also used in multiple places in this thesis. Let $x = (x_1, x_2)$ be a tuple, then $\pi_1(x) = x_1$ denotes the projection onto the first coordinate of x, and $\pi_2(x) = x_2$ denotes the projection onto the second coordinate of x.

$$\pi : (x_1, x_2, \dots, x_i, \dots) \times i \mapsto x_i$$

B.1.2 Partition Specification

The partition specification P (cf. page 21) is derived from [KL06]. Let $\mathbb{N}$ be the natural numbers i.e. $\{1, 2, 3, \dots\}$. Let A be the set of all basic activities in the process, p_i be one participant, and P be the set of participants. A participant p_i consists of a name s_i, which is in $\{s_1, s_2, \dots\}$, and a set of activities $M_i \subseteq A$, which is in $\{M_1, M_2, \dots\}$ and holds one or more activities, i.e. Furthermore, each basic activity must be assigned to exactly one set M_i, i.e. $\bigcup_{i \in \mathbb{N}} M_i = A$. The formal definition of the set of participants is as follows [KL06]:

$$P := \{p_i \mid \forall i \in \mathbb{N} : p_i = (s_i, M_i)\}$$

There are several general conditions that the partition P must satisfy. The conditions are initially stated in [KL06] and revised here:

1. $\forall p_i \in P$: $|\pi_2(p_i)| \geqslant 1$ where $\pi_2(p_i) = M_i$ as defined above, i.e. a participant p_i must have at least one activity.
2. $\forall p_i, p_j \in P$: $p_i \neq p_j \;\Rightarrow\; \pi_1(p_i) \neq \pi_1(p_j) \;\wedge\; \pi_2(p_i) \cap \pi_2(p_j) = \emptyset$, i.e. two participants do *not* share a same name or an activity.
3. $\bigcup_{p_i \in P} \pi_2(p_i) = \bigcup_{i \in \mathbb{N}} M_i = A$ where A is the set of all basic activities in the process.

B.1.3 Data-Flow Analysis

$Q_s(a,x)$ (cf. page 49) denotes a set of tuples. Each of those tuples consists of a query set and a writer set, e.g. $(\{.n,.k\},\{w_3,w_4\}) \in Q_s(a,x)$. All the tuples are based on the variable *x* and the reader activity *a*. Let q_s be a tuple in the $Q_s(a,x)$ for variable *x* read by activity *a*. Let (qs,ws) be a tuple in the $Q_s(a,x)$ with qs as a query set and ws as a writer set. qs contains the queries $\{q_1,q_2,\dots\}$ on variable *x* and ws contains the writers $\{w_1,w_2,\dots\}$ that write to variable *x* using that queries in qs. The function $Q_s(a,x)$ is defined as follows [KKL08a]:

$$Q_s(a,x) := \{\ q_s \mid q_s = (qs,ws)\ \wedge\ qs = \{q_1,q_2,\dots\}\ \wedge\ ws = \{w_1,w_2,\dots\}\}$$

$Q_s(a,x)$ is used by the algorithms for fragmenting data dependency in this thesis and therefore has the properties as follows:

1. $\forall q_s \in Q_s(a,x):\ \pi_1(q_s) = qs = \{q_1,q_2,\dots\} \neq \emptyset$ and $\pi_2(q_s) = ws = \{w_1,w_2,\dots\} \neq \emptyset$.
2. $\forall q_{s_i}, q_{s_j} \in Q_s(a,x),\ i,j \in \mathbb{N}:\ q_{s_i} \neq q_{s_j}\ \Rightarrow\ \pi_1(q_{s_i}) \neq \pi_1(q_{s_j})\ \wedge\ \pi_2(q_{s_i}) \neq \pi_2(q_{s_j})$, i.e. each one of the tuples in $Q_s(a,x)$ has a different query set and a different writer set. If either the query sets or the writer sets were equal, the tuples would have been merged. Assume that $(\{.m\},\{w_3,w_4\})$ and $(\{.k\},\{w_3,w_4\})$ were two tuples in $Q_s(a,x)$, in this case, they must be merged into one tuple by merging the query sets, that is $(\{.m,.k\},\{w_3,w_4\})$.

B.1.4 Writer Dependency Graph (WDG)

$A_d(a,x)$ (cf. page 51) is the function that presents all the writers that activity *a* depends on due to reading variable *x*. It is defined as the union of the writer sets that gets projected from all of the q_s in $Q_s(a,x)$ [KKL08a]:

$$A_d(a,x) := \bigcup_{q_s \in Q_s(a,x)} \pi_2(q_s)$$

Let V_d be the set of nodes, and E_d be the set of edges. Let the tuple (v_s,v_t) denote an edge in a WDG, v_s be source of the edge, and v_t be target of the edge. The Writer Dependency Graph (WDG, cf. page 51) for the activity *a* and the variable *x* is a graph with the nodes V_d and the edges E_d and is formally defined as follows [KKL08a]:

$$WDG_{a,x} := (V_d, E_d)$$

where $V_d := A_d(a,x)$ and $E_d \subset \{(V_d \times V_d)\}$.

The WDG is directed acyclic Graph, therefore has the constraints as follows:

1. The edges are directed.

2. The source and the target of the edge are two different individual writers in process, i.e. $(v_s, v_t) \in E_d : v_s \neq v_t$.
3. The source activity (v_s) and the target activity (v_t) in an edge of a WDG graph are not necessarily directly connected in the main process. As long as there is a path in the BPEL process from v_s to v_t and between them there are no other activities that are WDG nodes too. In other words, let $p = v_s, u_1, u_2, \ldots, v_t$ be the path in the main process from v_s to v_t, then $(v_s, v_t) \in E_d \Leftrightarrow u_i \notin V_d$
4. The graph is directed acyclic, i.e. there is no such path in the graph in which the start node and the end node is the same one.
5. WDG is independent of the partition specification.

B.1.5 Partitioned Writer Dependency Graph (PWDG)

Assume that the $WDG_{a,x} = (V_d, E_d)$ and the partition P are given, let $M_{k,l}$ be a '*region*' of the WDG nodes, which are mutually in the participant p_k, the PWDG (cf. equation (5.5) in page 55) against the activity *a*, the variable *x*, and the partition specification P is formally defined as follows:

$$PWDG_{a,x,P} := (V_p, E_p)$$

with the node set $V_P := \{n_i \mid n_i = (s_k, M_{k,l})\}$ where $\bigcup_{l \in \mathbb{N}} M_{k,l} = V_d \cap M_k$ and $n_i = (s_k, M_{k,l})$ implies $p_k = (s_k, M_k) \in P$ and the edge set $E_P \subset \{V_P \times V_P\}$

Let $p \in P$ be participant with the name s and the set of activities M that belong to p. We introduce a function $A(a, x, p)$ (cf. equation (5.6), page 55) to represent the group of WDG nodes that are in the participant p. It is defined as follows:

$$\begin{aligned} A_d(a, x, p) :=& A_d(a, x) \cap \pi_2(p) \\ =& V_d \cap \pi_2(p) \end{aligned}$$

where $p = \{s, M\} \ \in \ P = \{(s_i, M_i) \mid i \in \mathbb{N}\}$ and $\pi_2(p) = M$.

B.1.6 Local Resolver and Receiving Follow

Let $Q_s(a, x)$ be as in equation (5.1), and n be a PWDG node in a PWDG. The $Q_{sp}(n, a, x)$ denotes the tuples from $Q_s(a, x)$ whose writer sets contain only such writers that appear in the PWDG node n. It is defined as follows [KKL08a]:

$$Q_{sp}(n, a, x) := \{(qs, ws_n) \mid qs = \{q_1, q_2, \ldots\} \ \wedge \ ws_n = \{w \mid w \in \pi_2(q_s) \cap \pi_2(n) \ \wedge \ \forall q_s \in Q_s(a, x)\}\}$$

B.2 Notion Summary

Table B.1: Notions used in this thesis

Notion	Description	First Occurrence
$\forall i \in \mathbb{N} : \pi_i(x)$	$\pi_i(x)$ The projection on the i-th component of the tuple $x = (x_1, x_2, x_3, \dots)$ [KL06]	Page 21
$A_d(a, x)$	The set of all writers which the reader of variable *x*, i.e. activity *a* depends on [KKL08a]	Page 51
$A_d(a, x, p)$	The set of WDG nodes that present in a certain participant p	Page 55
$\text{dis}_\circ$	The disabled writers before reader activity is executed [KKL08b]	Page 48
E_d	The edge set of the graph $WDG_{a,x} = (V_d, E_d)$	Page 51
E_P	The edge set of the graph $PWDG_{a,x,P} = (V_P, E_P)$	Page 55
$\text{inv}_\circ$	The invalid writers before reader activity is executed [KKL08b]	Page 48
$l(a,\ b,\ q)$	The control link from activity *a* to *b* with the condition *q* [KL06]	Page 36
$\text{mbd}_\circ$	The 'may be dead' (boolean) value before reader activity is executed [KKL08b]	Page 48
$\mathbb{N}$	The natural numbers i.e. $\{1, 2, 3, \dots\}$	Page 21
NIL	The representation of *null* in algorithm pseudo code [CSRL01]	Page 59
P	The Partition Specification [KL06]	Page 21
$PWDG_{a,x,P}$	The Partitioned Writer Dependency Graph [KKL08a], defined in equation (5.5) (cf. page 55)	Page 5
$\text{poss}_\circ$	The possible writers before reader activity is executed [KKL08b]	Page 48
$Q_s(a, x)$	The tuples of query set and writer set against variable *x* for activity *a* [KKL08a]	Page 49
$Q_{sp}(n, a, x)$	The tuples of query set and writer set in a given PWDG node n [KKL08a]	Page 64
V_d	The node set of the graph $WDG_{a,x} = (V_d, E_d)$	Page 51
V_P	The node set of the graph $PWDG_{a,x,P} = (V_P, E_P)$	Page 55
$WDG_{a,x}$	The Writer Dependency Graph [KKL08a], defined in page 51	Page 5
$\text{writes}_\circ$	The current state of the writes to the given variable element *before* the reader activity is executed [KKL08b]	Page 16

Continued on next page

Table B.1 – continued from previous page

Notion	Description	First Occurrence
writes$_\bullet$	analogous to writes$_\circ$ but *after* the reader activity is executed [KKL08b]	Page 16

Bibliography

[Bor10] S. Bors. *A Runtime for BPEL4Chor Cross-Partner-Scopes*. Diploma thesis nr. 2990, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2010. (Cited on pages 13 and 88)

[Bre08] S. Breier. *Extended Data-flow Analysis on BPEL Process*. Diploma thesis nr. 2726, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2008. (Cited on pages 18, 48 and 70)

[CKLW03] F. Curbera, R. Khalaf, F. Leymann, S. Weerawarana. Exception handling in the BPEL4WS language. In *Proceedings of the 2003 international conference on Business process management*, BPM'03, pp. 276–290. Springer-Verlag, Berlin, Heidelberg, 2003. (Cited on pages 15, 18 and 20)

[CSRL01] T. H. Cormen, C. Stein, R. L. Rivest, C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. (Cited on pages 54, 58 and 94)

[DKLW07] G. Decker, O. Kopp, F. Leymann, M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. pp. 296–303, 2007. doi:10.1109/ICWS.2007.59. (Cited on pages 12, 16, 17 and 18)

[DKLW09] G. Decker, O. Kopp, F. Leymann, M. Weske. Interacting services: From specification to execution. volume 68, pp. 946–972. Elsevier Science Publishers, 2009. doi: 10.1016/j.datak.2009.04.003. (Cited on page 18)

[DS90] T. H. Davenport, J. E. Short. The New Industrial Engineering: Information Technology and Business Process Redesign. *Sloan Management Review*, 31(4):11–27, 1990. (Cited on page 11)

[Fer07] J. V. Fernandez. *BPEL with Explicit Data Flow: Model, Editor, and Partitioning Tool*. Diploma thesis nr. 2616, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2007. (Cited on pages 16 and 18)

[Gao10] Y. Gao. Implementierung einer Datenflussanalyse für WS-BPEL 2.0. (2246):54, 2010. (Cited on pages 18, 48, 70 and 77)

[Ima86] M. Imai. *Kaizen: The Key to Japan's Competitive Success*. McGraw-Hill, New York, NY, 1986. (Cited on page 11)

[Kha07] R. Khalaf. Note on Syntactic Details of Split BPEL-D Business Processes. Technical Report Computer Science 2007/02, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, University of Stuttgart, Institute of Architecture of Application Systems, 2007. (Cited on pages 16 and 18)

[Kha08] R. Khalaf. *Supproting business process fragmentation while maintaining operational semantics : a BPEL perspective*. Doctoral thesis, University of Suttgart, Factulty of Computer Science, Electrical Engineering, and Information Technology, Germany, 2008. (Cited on pages 12, 16, 18, 20, 24, 29, 30, 36, 46, 53, 56, 63, 65, 77, 87 and 89)

[KKL07] O. Kopp, R. Khalaf, F. Leymann. Reaching Definitions Analysis Respecting Dead Path Elimination Semantics in BPEL Processes. Technischer Bericht Informatik 2007/04, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, 2007. (Cited on page 48)

[KKL08a] R. Khalaf, O. Kopp, F. Leymann. Maintaining Data Dependencies Across BPEL Process Fragments. *International Journal of Cooperative Information Systems (IJCIS)*, 17(3):259–282, 2008. doi:10.1142/S0218843008001828. (Cited on pages 11, 12, 18, 20, 21, 22, 49, 51, 54, 64, 65, 66, 69, 71, 77, 87, 92, 93 and 94)

[KKL08b] O. Kopp, R. Khalaf, F. Leymann. Deriving Explicit Data Links in WS-BPEL Processes. In *IEEE International Conference on Services Computing*. IEEE, 2008. (Cited on pages 16, 18, 48, 87, 94 and 95)

[KL06] R. Khalaf, F. Leymann. Role-based Decomposition of Business Processes using BPEL. In *International Conference on Web Services (ICWS 2006)*, pp. 770–780. IEEE Computer Society, 2006. doi:10.1109/ICWS.2006.56. (Cited on pages 11, 12, 18, 20, 21, 24, 36, 77, 87, 91 and 94)

[Kop11a] O. Kopp. Algorithm for Weakly Connected Components with Constraint Satisfaction. E-mail, 2011. (Cited on page 58)

[Kop11b] O. Kopp. Grounding Syntax. E-mail, 2011. (Cited on pages 8 and 75)

[Kop11c] O. Kopp. Topology Syntax. E-mail, 2011. (Cited on pages 8 and 74)

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf. (Cited on pages 8, 15, 16, 23, 28, 42, 43 and 52)

[Off97] U. S. G. A. Office. Business Process Reengineering Assessment Guide. 1997. (Cited on page 11)

[OMG11] OMG. Business Process Model and Notation (BPMN) Version 2.0, 2011. (Cited on page 16)

[W3C99] W3C. XML Path Language (XPath), 1999. URL http://www.w3.org/TR/xpath/. (Cited on page 15)

[W3C01] W3C. Web Services Description Language (WSDL), 2001. URL http://www.w3.org/TR/wsdl. (Cited on page 66)

[W3C05] W3C. Web Services Choreography Description Language Version 1.0, 2005. URL http://www.w3.org/TR/ws-cdl-10/. (Cited on page 16)

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. (Cited on page 15)

All links were last followed on May 4, 2012.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Daojun Cui)