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

Diplomarbeit Nr. 2616

BPEL with Explicit Data Flow: Model, Editor, and Partitioning Tool

Javier Vázquez Fernandez

Course of Study:	Computer Science
Examiner:	Prof. Dr. F. Leymann
Supervisor:	M. Eng. R. Khalaf, Dipl.-Inf. O. Kopp
Commenced:	2007-01-24
Completed:	2007-07-23
CR-Classification:	C.2.4, D.2.2, D.2.3, D.2.7, H.4.1

"As you set out for Ithaca hope your road is a long one,

full of adventure, full of discovery...

...Keep Ithaca always in your mind.

Arriving there is what you're destined for."

Ithaca, Konstantino Kavafis

Abstract

The objective of this thesis is composed of two parts. The first one consists of defining a concept of how data connectors can be implemented in BPEL (called BPEL-D) as a solution to split a BPEL process into partitions being executed by different participants.

The second part consists of creating a graphical tool that allows one to create a BPEL-D process, assign different activities to different participants, and create an XML representation of the partition, the resulting WSDL and BPEL files based on the given algorithm in [KL06] and [Pal07], and an XML representation of a wiring file. The tool itself is based on the Eclipse BPEL Designer.

Stuttgart, July 2007 Javier Vazquez

Acknowledgments

To my family. They are always an example for me to follow, of hard working, sacrifice and integrity. Without them this would have not been possible.

To my friends in Spain. Even being hundreds of kilometres far away, their support has helped me during the difficult moments.

To my "Stuttgart Erasmus" friends. For convert this place in a new home for me and share it with me during this time.

To Prof. Dr. F. Leymann. For offer me the opportunity of developing this work in the Institut für Architektur von Anwendungssystemen.

To Dipl.-Inf. O. Kopp and M.Eng. R. Khalaf. For his work, help, and supervision during these months. One part of this work belongs to them.

TABLE OF CONTENTS

Part 1: BPEL-D Syntax and Semantics design 11

1. Introduction and Motivation. 13
1.1. Motivation 13
1.2. Introduction 15

2. Background. 17
2.1. Related Works 17
2.2. Business Process Execution Language (BPEL). 18
2.2.1. Versions. 18
2.2.2. Architectural concepts. 18
2.2.3. Related technologies. 20
2.2.3.1. XML 20
2.2.3.2. Xpath. 20
2.2.3.3. Web Services. 20
2.3. WSFL 21
2.4. Production Workflow Metamodel. 23
2.4.1. Notion of metamodel 23
2.4.2. Data store managing 23
2.4.3 Data Flow managing 24
2.4.4. Conditions and restrictions 26

3. Modeling explicit Datalinks into BPEL. 29
3.1. Input and output containers 30
3.1.1 Integration of containers into the process 31
3.1.2. Containers internal structure 32
3.1.2.1 Semantics 32
3.1.2.2 Syntax possible options and argumentation 33
3.1.2.3 Empirical arguments 34
3.1.3. Containers integration in the BPEL-D process 36
3.1.4. Containers definitive syntax 37
3.2. Data links 39
3.2.1. Datalinks syntax definition. 39
3.2.2. Datalinks syntax definition 40
3.2.3. Maps syntax definition 41
3.2.4. Datalinks integration in the BPEL-D process 42
3.2.5. Datalink final syntax 44
3.3. BPEL Activities 46
3.3.1. Simple activities 46

3.3.1.1 The receive activity 46
3.3.1.2. The invoke activity. 49
3.3.1.3. The reply activity. 50
3.3.1.4. The assign activity. 51
3.3.1.5. The throw activity. 54
3.3.1.6. Rest of simple activities 55
3.3.2. Structured activities. 56
3.3.2.1. Some common questions. 56
3.3.2.2. The switch activity 61
3.3.2.3. The pick activity. 62
3.3.2.4. The while activity. 65
3.3.2.5. The sequence activity 67
3.3.2.6. The flow activity. 69
3.3.2.7. The scope activity. 70
3.4. Summary 74

Part 2: Implementation of Eclipse BPEL-D Designer 75

4. Implementation Background. 77
4.1. Eclipse Development Platform. 77
4.2. Eclipse Modelling Framework (EMF). 78
4.2.1. Introduction 78
4.2.2. Architecture 79
4.3. Graphical Editing Framework (GEF). 80
4.4. The Eclipse BPEL Project. 80

5. Eclipse Designer Overview. 83
5.1. High level vision. 83
5.2. Plug-in org.eclipse.bpel.model 84
5.2.1 Parsing and writing BPEL files 88
5.3. Plug-in org.eclipse.bpel.ui 90
5.4. System classes interaction to create VPC elements 96
5.5. System classes interaction to create a flow 99

6. BPEL Designer Implementation. 103
6.1. Introduction 103
6.2. Adapting the EMF model to BPEL-D. 104
6.3. Providing the UI with facilities to manage BPEL-D 107
6.3.1. Modifications in the model package. 107
6.3.2. Modifications in the ui package. 108
6.3.3. Creating a Container from the GUI 113
6.3.4. Creating a DataLink from the GUI 116
6.4. Browsing variables usagefrom the model. 118
6.5. Splitting algorithm 119
6.5.1 Overview 119
6.5.2 Design and implementation 120
6.5.3. Splitting the process from the GUI. 122

7. Conclusions and future works. 125
Appendix A: BPEL-D Samples 127

Part 1

BPEL-D Syntax and Semantics Design

CHAPTER 1

INTRODUCTION AND MOTIVATION

First, this chapter presents an introduction about the structure of this paper, explaining the chapters it is composed of, and what is done in each of them. After that, it introduces the issues which motive this research, and explains its targets.

1.1. Motivation

The business world becomes each day more competitive, demanding from companies a more efficient way of production. Business Process Reengineering (BPR) and Continuous Process Improvement (CPI) are two important practices to achieve this goal.

These practices consist in detecting potential deviations from the efficient company business course, with the intention of avoiding them or deriving the appropriate measures for improving their performance. In some of these cases, the company finds that the best option is to delegate problematic tasks to a third-party. This option is commonly called “Outsourcing”.

Sometimes, delegating a fragment of the process will be enough for solve the problem. Normally, this cut fragment must interact with the rest of the parts of the main process, keeping the relations and dependencies it had when it was part of the main process.

Then, the process will be partitioned in different activities sets. The responsibility of executing each one of them will correspond to different partners. Once the responsibilities have been chosen, the splitting algorithm described in [KL06] is used to fragment the main process into a different process model for each partner. At the same time, we must be able of bring the control and data dependencies existing in the original

process, to the new ones. This will occur by using messages between the processes. We can see this decomposition in Figure 1.

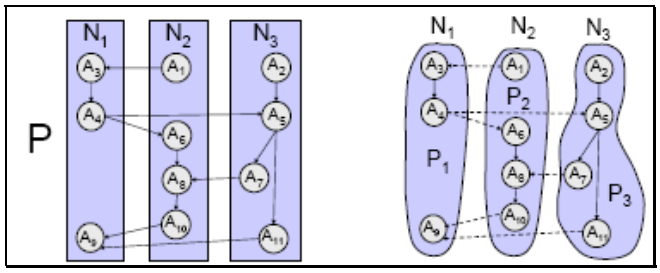

Figure 1: Decomposition and reconnection of business process [KL06]

BPEL 1.1 works with scoped variables, as the classic programming paradigm. It means that data dependencies are implicit in the code. To fragment the process and translate these dependencies into message flows between the fragments we want to specify these dependencies in an explicit way.

The target of the research in this thesis is to design a mechanism to make the process data flow explicit to thus be able to disconnect the process into partitions. It will simplify the reassignment of activities to the different entities.

The approach followed to achieve these targets is presented in Figure 2. The "Transform" block takes as input the process model, in BPEL-D which is the modified version of BPEL defined in this thesis as well as the WSDL files. The output will be one BPEL process for each participant as well as the new WSDL files necessary to describe the communications between fragments.

Figure 2: Overview of the approach [KL06]

This BPEL-D process consists of a subset of BPEL with data connectors in the place of the classic programming style with scoped variables. The subset of BPEL used to define the main process is described in [KL06] to which scopes and loops have been added in [Pal07].

Our intention in this thesis is to extend the BPEL-D syntax design to be able of embracing the whole classic BPEL 1.1 syntax. For that we add to the initial subset, the rest of BPEL activities: throw, compensate, switch, pick, sequence, etc.

The fact of having a wide view over the whole BPEL set let us to find common patterns between the activities. Then we can define a more coherent model, than if we do it just with the first set. It will also introduce some needed restrictions.

In relation with the implementation of the BPEL-D Designer is important to mention the fact that the original Eclipse BPEL Designer is built following BPEL 2.0 specification. In a different way BPEL-D is designed on top of BPEL 1.1, since the splitting algorithm introduced in [KL06] is designed on top of it. The treatment in relation how this fact affects to the research is explained in Chapter 6.

1.2. Introduction

The thesis is organized in two different parts: the first part of it presents the process of creating a new syntax for a variant of BPEL with data connectors, introduced in [KL06], which we call along this paper as BPEL-D for easy reference. The second part of the thesis presents the implementation of the Eclipse BPEL-D designer tool.

The first chapter provides the motivation and an overview about the definition of BPEL-D and the implementation of its editor, as well as a brief introduction to the process developed along the thesis.

The second chapter provides the background information to the basics as a brief description to BPEL 1.1 [BPEL11], WSFL [WSFL] and Workflow Theory (see [LR00]), which are essential to understand the rest of the thesis, as well as a brief introduction to related works.

The Workflow model explained in [LR00] will provides as with a theoretical reference to follow in the task of achieving a more solid syntax for BPEL-D. This metamodel will contribute to our research with some important concepts and constructs which we will use in the BPEL-D syntax design explained in Chapter 3. WSFL [WSFL] will be also an interesting reference for the design of the syntax.

The third chapter covers a substantial section of the first part, focusing on the design of the data links syntax. First, it explains how to design the datalinks and the containers to store information that is flowed using data links. After that, the paper describes the BPEL activities explaining the semantics with the new elements.

The fourth chapter introduces a little overview about the basics of the IBM Eclipse Platform [Eclipse], as well as over other essential tools necessaries for our development as the Eclipse Modelling Framework [EMF] and the Graphical Editing Framework [GEF]. An introduction over the Eclipse Plug-in development is included too in this part of the paper, focusing in the Eclipse BPEL Project [EclipseBPEL].

The fifth chapter goes deep in the implementation of the Eclipse BPEL Designer, whose understanding is essential to the developing the BPEL-D designer. It starts with a high level view of the plug-in architecture to, afterwards focus on the most important modules for our research and in the normal behaviour of the system. Taking into account the fact of that does not exist any documentation about this Designer, this chapter will had an essential importance in the research.

The sixth chapter explains the necessary model extensions to support BPEL-D. It includes the introduction and creation of the new model objects as well as the exclusion of the BPEL variables user usage, and which modules and in which way are affected for this issues. The final part goes through the splitting algorithm introduced in [KL06] and explains its implementation.

The seventh exposes the conclusions achieved during the research and the future works to be done continuing with this topic.

CHAPTER 2

BACKGROUND

This chapter introduces to the essential basics to understand the thesis contents. It starts with an introduction to works related with our research. After, it introduces BPEL 1.1, base language version of BPEL-D. Next presents WSFL which we take as reference for the design of BPEL-D syntax. The last section contains an introduction to Production Workflow Theory following [LR00], whose metamodel is used as guide in our research.

2.1. Related Works

In relation with transferring data along an environment, we find two different approaches in workflow universe: blackboard and explicit data flow (for more see [ACK03] and [MMP06]).

BPEL 1.1, like several other workflow languages (see, e.g. [HPOpen], [BPML] and [SWSL]), follows the blackboard approach. The blackboard is composed of a set of variables where the different operations take and gather data. In this aspect, each structured element or scope has its own blackboard in a similar way as every program execution has its own values for the variables of the program.

On the other hand, explicit data flow specifies the data flow between activities using data connectors. This approach is followed by some workflow language such as BioOpera [BOpera], MQSeries Workflow [MQSeries] or WSFL [WSFL]. This way the process designer specifies how an activity takes its input information directly from the output of another one that previously executed. This approach is simpler for people who think visually in terms of data flow and not programmatically in terms of shared variables. Additionally, it makes the workflow engine more flexible but at the same

time more complex for advanced data sharing scenarios. This is the approach to follow for designing BPEL-D.

In relation to interacting processes, we also find two different approaches: the use of conversation languages to model message exchanges [WSCDL], and breaking up the process into different fragments to be run by different partners as in [MWW+98] and [CD00].

The design of BPEL-D is based in the second one, enabling its use as input to a splitting algorithm for fragmenting the main process between the participants. For this issue we break up the BPEL process in several fragments by a process designer following the approach in [KL06].

2.2. Business Process Execution Language (BPEL).

BPEL is an XML-based language for standardizing business processes in a distributed or grid computing environment that enables separate businesses to interconnect their applications and share data.

Designed as a combination of IBM's Web Services Flow Language [WSFL] and Microsoft's XLANG spec [XLANG], BPEL is platform-independent and allows enterprises to keep internal business protocols separate from cross-enterprise protocols. Then, internal processes can be changed without affecting the exchange of data between different sections of one enterprise, or even from enterprise to enterprise.

The BPEL usage to describe business process can be understood in two different ways. On one hand it can be used as the format to exchange processes between partners without exposing the process's internal behaviour. On the other hand, it can be used to detail the internal behaviour of one of them. For more see [WCF+05].

2.2.1. Versions.

The last release BPEL 2.0 was published on April of 2007 and it is becoming the version used nowadays by the computing industry. BPEL-D is designed on top of the previous version, BPEL 1.1 which is the version on which the splitting algorithm has been created [KL06]. BPEL 2.0 becomes more relevant in the implementation of the editor, which is described in Chapter 6.

2.2.2. Architectural concepts.

As has been said, BPEL process describes interaction between parts. These parts are called partners who participate in the interaction are based on three elements:

- ***Partner Link Type:*** defines a generic link for a certain category of web services, as a set of operations and roles that offer or use them.
- ***Partner Link:*** defines the web service that will be actually invoked.
- ***Partner:*** groups all the links that relate to the same entity.

Instances of these typed connectors provide either one or both of a role that the process implements and one that it expects from a partner. The roles refer to defined WSDL portTypes.

Data is written to and read from scoped variables. The type of these variables may be a WSDL message, a XML Schema simple type or a XML Schema element. Variables associated with message types can be specified as input or output variables for some of the simple activities as invoke, receive and reply. Figure 3 illustrates the explained structure.

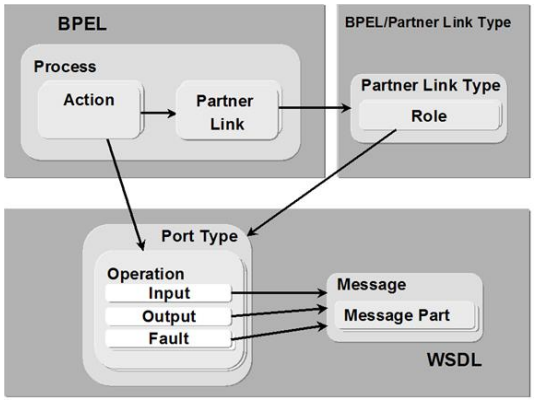

Figure 3: BPEL Process communications

A correlation mechanism is used to route the messages to correct instances of a running process. A correlation set refers to a set of properties; each property is aliased to fields in one or more WSDL messages. Incoming messages are checked for the set, which is matched against existing values mapped to running process instances. If a match is not found, the process definition is checked for the ability to create one, based on the message and the 'receive' activities in the process with the "createInstance" attribute set to "yes".

A BPEL process is defined by using a set of activities which model the different tasks of standard business process behaviour. BPEL describes two different kinds of activities: simple and structured. Simple activities can be combined inside structured activities that impose control dependencies on them to achieve higher-level business logic. For more clarity we introduce a brief explanation of the different activities in their respective sections in Chapter 3 before bring them into BPEL-D.

The execution context for every activity is provided by scopes, with the process considered as one top level scope. Each scope provides handlers (fault, compensation and event), variables and correlation sets in a local way. For more about BPEL see [SWSBPEL] and [BPELGuide].

2.2.3. Related technologies.

BPEL4WS [BPEL11], or simply BPEL for short, is an extensible workflow XML based language that aggregates services by choreographing service interactions.

As has been said, BPEL4WS is layered on top of several XML specifications: WSDL 1.1 [WSDL], XML Schema 1.0 [XMLSchema], and XPath 1.0. [XPath]. WSDL messages and XML Schema type definitions provide the data model used by BPEL4WS processes. XPath provides support for data manipulation.

2.2.3.1. XML

XML (Extensible Mark-up Language, see [XML]) is a general-purpose mark-up language. Its primary purpose is to facilitate the sharing of data across different information systems, particularly via the Internet

It is a simplified subset of the Standard Generalized Mark-up Language (SGML), and is designed to be relatively human-legible. By adding semantic constraints, application languages can be implemented in XML. Moreover, XML is sometimes used as the specification language for such application languages.

XML is recommended by the World Wide Web Consortium. It is a fee-free open standard. The W3C recommendation specifies both the lexical grammar, and the requirements for parsing.

2.2.3.2. XPath.

XPath (XML Path Language, see [XPath]) is an expression language for addressing portions of an XML document, or for computing values (strings, numbers, or boolean values) based on the content of an XML document.

The XPath language is based on a tree representation of the XML document, and provides the ability to navigate around the tree, selecting nodes by a variety of criteria.

2.2.3.3. Web Services.

The W3C [W3C] defines a Web service as a software system designed to support interoperable Machine to Machine interaction over a network. Web services are frequently just Web APIs that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services.

2.3. WSFL

The Web Services Flow Language [WSFL] is an XML-based language for the description of Web Services compositions as part of a business process definition. It was designed by IBM [IBM] to be part of the Web Service technology framework and relies and complements existing specifications like SOAP [SOAP] and WSDL [WSDL]. WSFL considers two types of Web Services compositions:

- The first type specifies an executable business process known as a flow Model.
- The second type specifies a business collaboration known as a Global Model.

As in BPEL, the unit of work in WSFL is the activity. These activities are represented as linked nodes of a graph. The activities are connected by two different kinds of links: control links and data links. In BPEL 1.1 we have these explicit control links which let the designer describe the work flow along the process. But as has been said before, in BPEL the data links are implicit.

A WSFL Data Link specifies that a source activity passes data to the flow engine as part of the process instance context, which in turn has to pass (some of) this data to the target activity of the Data Link. Following this syntax, data must always flows along Control Links. In other words, Data Flow must always follow Control Flow.

It means that to specify a Data Link between two activities, we have to be able to reach the target activity of the link from the source one. This way we ensure that we avoid a couple of error-prone situations in an easy way. On the other hand, it does not mean that data must be always passed to the immediate successor activity. We can pass trough many different activities along the path until reaching the source activity.

As explained in section 2.4, this condition is commonly used in workflow systems. Following the WSFL structure will easily allow BPEL-D to comply as well.

To simplify data flow, WSFL supplies a mechanism called a map-specification. A map construct (see Listing 1) specifies which part of the source activity's output message will be transferred into which field of the target activity's input message. It means how the information between the different activities must be exchanged.

```
<dataLink name="connection"
          source="receive1"
          target="invoke1"
     <map sourceMessage="creditInformationMessage" targetMessage="request"
          sourcePart="creditInformationMessage" targetPart="request"/>
</dataLink>
```

Listing 1: WSDL Datalink example

Listing 1 shows an example of a WSFL Data link. The source and target attributes specify the two activities and their data exchange relation. Listings 2 and 3 show the definition of these two constructs.

```
<element name="map">
   <complexType>
      <attribute name="sourceMessage" type="NCName"/>
      <attribute name="targetMessage" type="NCName"/>
      <attribute name="sourcePart" type="NCName"/>
      <attribute name="targetPart" type="NCName"/>
      <attribute name="sourceField" type="NCName"/>
      <attribute name="targetField" type="NCName"/>
      <attribute name="converter" type="string"/>
   </complexType>
</element>
```

Listing 2: WSDL Map syntax [WSFL]

```
<complexType name="dataLinkType">
   <complexContent>
      <extension base="linkType">
         <sequence>
            <element ref="map" minOccurs="0"/>
         </sequence>
      </extension>
   </complexContent>
</complexType>
```

Listing 3: WSDL DataLinkType syntax [WSFL]

These constructs are a good way to describe activity communications. For this reason we take both constructs as model to design the BPEL-D new elements. But we still need a way to store the information. The Workflow metamodel of [LR00] will help us with this issue.

2.4. Production Workflow Systems.

This section introduces the Workflow metamodel taken from [LR00]. As has been said, we will take it as a model to provide BPEL-D with a theoretic reference to follow providing it with syntax and semantic elements.

2.4.1. Notion of metamodel

The metamodel description given in [LR00] is the following "constructs and associated language used to formulate a data model as well as the precise prescription of the properties and behaviour of instances of an associated data model". It means, a precise way of to explain how the behaviour of our system must be.

The metamodel provides us with this prescription in two different parts. On one hand, it provides the syntax: the necessary constructs and the language which we will need to describe a process. On the other hand, it provides the semantics: the behaviour and the properties of the elements constructed with the syntax. Both are based on graph theory.

2.4.2. Data store management

The set of data required for the normal execution of a process includes the input information to the different activities contained in it as well as the data associated with the transition conditions which rule the control flow. To store this information activities are provided with a construct called a *container*.

A container is a collection of data elements used as input and output data for the different activities of the process. Attending to the formal definition introduced in [LR00]:

The map i assigns to each activity, process model and condition its input container

$$i : H \cup C \rightarrow \wp(V)$$

$$\forall \, \mathrm{X} \in H \cup C : i(\mathrm{X}) \subseteq \mathrm{V} \text{ with } \mathrm{card}\, i(\mathrm{X}) < \infty$$

The map o assigns to each activity and process model its output container:

$$o : H \rightarrow \wp(V)$$

$$\forall X \in H \cup C : i(X) \subseteq V \text{ with card } o(X) < \infty$$

The collection is unordered. This description assigns one input container and one output container to each activity. A container is structured following a tuple structure style. Figure 4 shows and activity with these containers

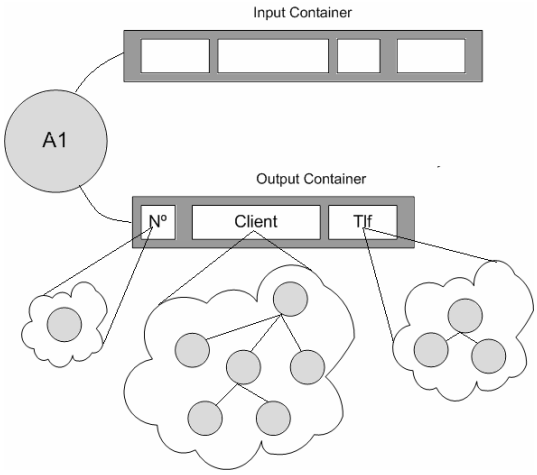

Figure 4: Input and output containers

We use the Workflows Metamodel containers as a starting point to define the ones to be used in the BPEL-D syntax, associating them to BPEL-D activities in a similar way.

As has been said the metamodel associates information with control links by storing it into containers referred to in transition conditions. Transition conditions are business rules or predicates which affects the work-flow transition between two consecutive activities. The information necessary for deciding about these conditions have to be taken from somewhere. It means that transition conditions also have input containers. Output ones are not necessary because the output will be always a Boolean value.

2.4.3 Data flow management

In reference to the process flow, the metamodel provide us with two different kinds of connectors: Control Connectors and Data Connectors. As we said before, in BPEL control links are explicit, and follow quite precisely the approximation shown in this model.

If in the previous section, we explained that the problem of data storing is solved in the metamodel, the next step in relation with the data question is how to transfer data

through the process flow. The way to model this question is solved in two different parts: data dependencies and data mapping.

The first one refers to describing data dependencies between activities. This is which activities or predicates are related by transfer of data.

In graph theory, the dependencies are described by the use of edges. Edges can be understood as the graphic description of a data connector. We will draw an edge between two activities when the first one expects data from the other, it means when both activities are data dependent. This edge is called *data connector*. Figure 5 illustrates the data dependent activities of a process connected by data links.

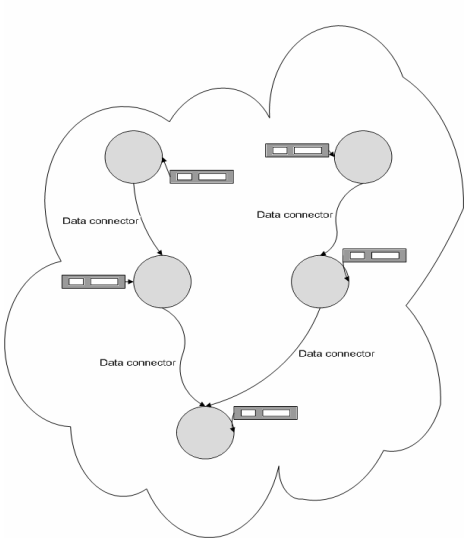

Figure 5: Datalinks in a process

The second one refers to data mapping. It means how data is exchanged, or more concretely, which parts of one input container are expected in which part of one output container.

In relation with that, the metamodel specifies the concrete relation between input and output container data simply specifying the par of elements in relation (d1, d2), in which, the first element will be the part of source activity output container that contains the data to be transferred, and the second one will be the part of target activity's input container that needs this information.

The set of all pairs between two related activities is denoted by Δ (A, B), and we will refer to it as *data connector map*. Each pair of activities that are data dependent will have their own data connector map. Figure 6 illustrates the data mapping between containers.

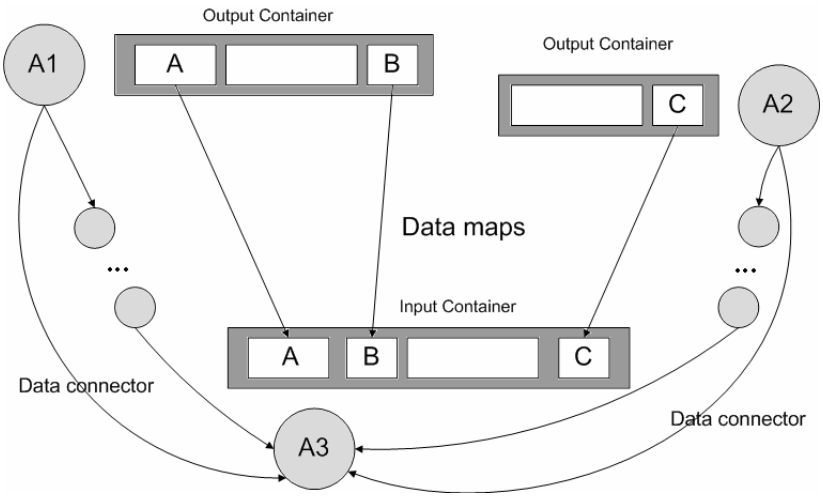

Figure 6: Containers with data maps

After that we have the three constructs that we will need for the definition of the BPEL-D syntax; *containers*, *data connectors and maps*.

2.4.4. Conditions and restrictions

In relation with process data flow there are some restrictions that we must apply when we are using data connectors.

One activity can expect data just from those activities that must run before it; that means that an activity can only produce data for others in the cases that it runs before them. The practical implication of that is we demand the existence of a control path between activities to allow data connectors between them. An activity cannot expect data from another one that runs in parallel.

If we come back to the section 2.3, we find that this issue turned up before when we introduced WSFL as an example to take into account. In this situation we realize that Data Flow always must follow Control Flow, even if it does not to do it directly. This implicates some restrictions in BPEL-D that are explained through Chapter 3.

Following the mathematical model we can define a *Data Connector Map* as follows:

Let A ϵ N be an activity, and let B ϵ N U C, be an activity or a predicate. The map

$$\Delta : N \times (N \cup C) \rightarrow \bigtimes_{A \in N, B \in N \cup C} \wp(o(A) \times i(B))$$

Satisfying the following three conditions:

1. $\Delta(A_1, A_2) \in \wp(o(A_1) \text{ x } i(A_2))$,
2. $\Delta(A_1, A_2) \neq \phi \Rightarrow A_2$ is reachable from A_1
3. $\forall A_2 \in N : (x, z), (y, z) \in \bigcup_{A_1 \in N} \Delta(A_1, A_2) \Rightarrow x = y$,

An element (v_1,v_2) is called data map. The set of all data connectors E is defined as

$$E := \{(A, B, \Delta(A, B)) \in N \text{ x } N \text{ x } \wp(V \text{ x } V) \mid \Delta(A, B) \neq \phi \}$$

The first condition states that a data connector can only be specified between the input container of a source activity and the output container of a target activity.

The second condition states that a data connector is allowed between two activities only in the case of both activities are connected for the control flows; in other words when we can achieve the target activity form the source one following the workflow.

The third condition states that two data maps cannot have the same element as targets. However, even in [LR00], this restriction is relaxed in practice. In business processes with parallelism, (and other capabilities like switching, loops and so on) paths will fork and join, making it pragmatic to relax this condition and allow different maps to target the same element in an input container.

Apart from that, the metamodel works with simple activities. Therefore, we must relax some of these conditions because it is not prepared to work with structured activities in BPEL.

We will try to follow these restrictions explained for the mathematical model as closely as we can to achieve a solid syntax. In the case of finding an important reason for breaking or relaxing them we will still try to respect them in the closest way.

Chapter 3

Modeling explicit Datalinks into BPEL

This is the core chapter of the first part of the thesis. Taking as background the issues explained before, specially referring to the metamodel explained in Chapter 2, we start to design the syntax to bring a data connector solution to BPEL. This section starts with the container design. After that, data connectors are explained. Finally it goes on to the different BPEL 1.1 activities to discuss the data connectors' semantics in relation with them.

Some previous questions

Namespace and naming conventions.

BPEL-D will be defined in a new namespace, which is "urn:IAAS:BPEL:BPEL-D/2007/07/25". It will include all the elements of BPEL 1.1. Therefore, the namespace of the activities change to it. If a listing references to BPEL, the original namespace of BPEL is used as default namespace.

In the listing references to BPEL-D (figure title beginning for "BPEL-D)", the BPEL-D namespace is used as default namespace. If a caption starts with "BPEL" the source is the BPEL specification.

Correlation Sets

Correlation Sets are not affected for the use of datalinks because they are context data used by the BPEL engine and not application data belonging directly to the process: An activity does not read or write to a correlation set, but the BPEL engine. Therefore, correlation sets are not more deeply studied in this research.

One can image correlation sets as data stored in a global store at the engine. The same applies for endpoint references stored in partner links. For this reason BPEL-D does not modify their previous syntax or semantics.

3.1. Designing BPEL-D Containers.

The execution of a business process often takes a long time, so we talk about long-running processes. During this time, errors can occur and affect the correct execution of our processes. For this reason, it is necessary to store persistently some information, the "business context", to be able to resume execution in the case of problems and system errors. In the case of BPEL processes, data stored in variables can be part of the business context. A variable (see Listing 4) may contain, among other things, the data exchanged between partners as instances of WSDL messages (see [BPEL11]).

```
<variables>
  <variable name="ncname" messageType="qname"?
              type="qname"? element="qname"?/>+
</variables>
```

Listing 4: BPEL Variable definition

The first question that we must confront in the process of building BPEL-D syntax is the way of storing the information. As has been said the target of our research is changing the data transfer process using variables, for a new one in which information will be transferred along the process using data links. In the standard BPEL syntax, data is stored in variables; now we must find a new construct in order to work with data connectors.

Following the metamodel introduced in [LR00], we will bring the idea of containers for how to store information in BPEL-D. As has been explained in the Chapter 2, the metamodel solution organizes the information to be used for an activity in collections called *Containers*. It defines them as sets of data elements without order. Following that the information will be organized based on its use; it means that each activity will be associated with a pair of containers, an input container and an output container.

The aim of the input container is get the information that the activity needs to do its work. The aim of the output container is offer the information processed by the activity to the rest of process activities.

As has been said the metamodel proposes providing activities and transition conditions both with containers. This approach is not usable as the input to the process splitting algorithm at hand [KL06]. For this reason we decide not imitate this structure in BPEL-

D. Transition conditions will read information from the containers of their source activity (see Figure 7). Only activities will have containers.

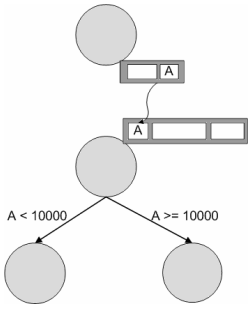

Figure 7: Transition conditions

3.1.1 Integration of containers into the process

Attending to a BPEL process structure where the syntax is XML and therefore must be organized as a tree, it is necessary to provide a section to declare the containers elements. We find two options for where to place this section:

1. Use a process containers declaration section.

2. Each activity has each own containers declaration section.

The first option stays closer to BPEL 1.1 which uses a variables section (see Listing 5) in the process definition, to describe the data variables used by the process (global variables), or in the case of scoped variables, a variables section into the adequate scope. The reason is that all activities in the process or scope must be able to access the information stored in the variables. It means that BPEL variables belong to a whole process or a whole scope, and for this reason they must be defined in a shared place.

```
<process name="loanApprovalProcess">
        ...
        <variables>
              <variable messageType="lns:creditInformationMessage" name="request"/>
              <variable messageType="lns:riskAssessmentMessage" name="risk"/>
              <variable messageType="lns:approvalMessage" name="approval"/>
              <variable messageType="lns:errorMessage" name="error"/>
       </variables>
       ....
```

Listing 5: BPEL Variables declaration section

But this option does not fit adequately the situation of BPEL-D, for the following reason: In BPEL-D, information passes through the process using data connectors; it means that at each moment, data belongs to a concrete activity and not to the global process as in BPEL. Therefore, it does not make sense to place this information in the process scope.

The second option is placing the information in direct relation with the activity which is using it. This option fits with the BPEL-D situation whose semantics determines that the place to store information (that is, to place the *Containers*) must belong to a concrete activity.

Based on the above, the conclusion is to provide each activity with a container declaration section; it means to follow the second option.

3.1.2. Containers internal structure definition.

3.1.2.1 Semantics

The next question is how to manage the information into these containers. To solve this problem, it is necessary to analyze the possible contents of a container. These contents will be the same as those stored in BPEL 1.1 variables: an XML Schema Element, an XML Schema Simple Type or a WSDL Message.

As has been said, the metamodel calls each member of the process data as *data element*; our objective will be storing each one of these data elements in a part of the container. We call each of these parts as *Department*. This way each container will be composed of a number of departments in which is placed the information which in BPEL used to be stored in variables.

Once we have these two constructs, the next questions are about their semantics. For example; how can we determine the size of a container? We will know the number of inputs to the container previously or will it extend dynamically? How does one reference an element in the container? What will the content of each department be? And so on.

The situation is quite close to that of variables: we know “a priori” the information to be stored in each container because we know the whole description of the process. For this reason we are able determine the structure of the container previously.

But the main question is how to map the information that we receive from the partners, and which we transfer through the business process, into the containers.

As we can see in Listing 6, a WSDL message use to be composed of different parts (in this example: client, productName and amount). In BPEL 1.1 the options were map the whole message into a variable, or map each part of it into a different variable with the intention of working just with simple types or at least with more granular information.

```
<message name="orderInformationMessage">
    <part name="client" type="xsd:string"/>
    <part name="productName" type="xsd:string"/>
    <part name="amount" type="xsd:int"/>
</message>
```

Listing 6: BPEL Message

3.1.2.2 Syntax possible options and argumentation

If we go again to the [LR00] metamodel, we will see that it works independently with simple and structured types, so it does not provide a solution for our problem. Then, taking into account the situation of BPEL, we must choose between the following options for BPEL-D:

1. Map each part of the message to a different department of the container.
2. Map the whole message to a department.
3. Let the programmer choose one or the other for each message.

The advantages of the first option are clearer code which will be easier to understand, an easier way to manage the information when we must fragment it into different parts to transfer it along the process or when we have to modify it by *assign* activities. Listing 7 shows how this solution could be.

```
<outputContainer>
    <department name="client" sourcePart="client" />
    <department name="productName" sourcePart="productName" />
    <department name="amount" sourcePart="amount" />
</outputContainer>
```

Listing 7: BPEL-D Container definition (1)

The main advantage of the second option is that it needs much less code lines and it make programming easier. Another aspect to consider is that normally, we will transfer the complete block of information; that is, the structured element that we received as a message. So this normally seems to be a better option than to separate it in parts. Listing 8 shows the same example choosing this option.

```
<outputContainer>
    <department name="request"/>
</outputContainer>
```

Listing 8: BPEL-D Container definition (2)

The third option is less involved. We prefer find key reasons to choose one of the other two options and clearly fix how the information exchanged in the process must be described. Therefore, we drop the third option of letting the programmer choose.

Finding not enough arguments in the theory to take a final decision between the first two, the best way to choose between both options is to test them in a real environment.

3.1.2.3 Empirical arguments

To test this situation in practice, we choose two classical BPEL business process examples: Loan Approval, and Market Place [BPELSamples] (supposing composed messages). It will provide us empirical information, as important as our reasoning process, and will show aspects that we can not see without this kind of test.

Along the thesis, some snippets of Loan Approval are been shown to illustrate the decisions taken about the BPEL-D syntax. We test them too in Market Place and some more but is not practical show all them in each situation. Appendix shows both examples translated to BPEL-D.

```
...
<dataLink sourceActivity="receive1"
          targetActivity="invoke1">
    <map sourceDepartment="client"
         targetDepartment="client" />
    <map sourceDepartment="productName"
         targetDepartment="productName" />
    <map sourceDepartment="amount"
         targetDepartment="amount" />
</dataLink>
...

<receive name="receive1" createInstance="yes"
         operation="request">
    <outputContainer>
        <department name="client"/>
        <department name="productName"/>
        <department name="amount"/>
    </outputContainer>
    ...

</receive>

<invoke name="invoke1" operation="check">
    <inputContainer>
        <department name="client"/>
        <department name="productName"/>
        <department name="amount"/>
    </inputContainer>
    <outputContainer>
        <department name="accept"/>
        <department name="coment"/>
    </outputContainer>
    ...
</invoke>
...
```

```
...
<dataLink sourceActivity="receive1"
          targetActivity="invoke1">
    <map sourceDepartment="request"
         targetDepartment="request" />
</dataLink>
...

<receive name="receive1" createInstance="yes"
         operation="request">
    <outputContainer>
        <department name="request"
    </outputContainer>
    ...

</receive>

<invoke name="invoke1" operation="check">
    <inputContainer>
        <department name="request"
    </inputContainer>
    <outputContainer>
        <department name="risk"
    </outputContainer>
    ...
</invoke>
...
```

Listing 9: BPEL container structure samples (left) mapping separately (right) mapping directly

```
...
<message name="request">
        <part name="client" type="xsd:string"/>
        <part name="productName" type="xsd:string"/>
        <part name="amount" type="xsd:int"/>
</message>
<message name="risk">
        <part name="accept" type="xsd:string"/>
        <part name="coment" type="xsd:string"/>
</message>
...
```

Listing 10: Messages sample

After trying both options in both examples (see Listing 9, supposing the messages shown in Listing 10) we find the following conclusions:

- As we said, normally, we must transfer the whole content of the message. Using the first approach implies writing, for each part of the message, one department and one map in the opportune datalink. However, when bringing this conclusion to extreme situations, that will result in many more code lines, and much more work for the programmer

- Therefore, the code using the first option is not as clear as we had initially thought. In fact it seems to be clearer when we use the second approach.

- Even when we divide the message in different parts, we will not always obtain simple types data. A recursive decomposition seems not be a useful option, for that we have to discard one of the reasons for choosing the first approach.

Options	**Map separately**	**Map directly**
Pros	A bit clearer with composed messages. Makes easier splitting information.	Good situation to apply it is more common. Less code to write. Global process easy to follow.
Cons	Not achieving simple data. Message information use to be used together.	

Table 1: Mapping options

Taking into account these conclusions, the more appropriate option is the second one, it means that the message will be mapped directly to the container departments, and

transferred this way along the process flow, and one would use an assign activity to discompose it like in BPEL.

3.1.3. Containers integration with activities.

We started the information storing design, taking the Workflow metamodel as point of divergence. For this reason the idea of associating to each activity one input and one output container was to be the right way.

But the translation of this idea to BPEL-D syntax can be improved. After deciding that each container will have just one department, we find that each section will have just an element declared and this is not efficient. Another possibility is combining both sections in one. Then we must decide between the following options:

1. Maintain the initial syntax having a different declaration section for input and output containers, as shown in Listing 11.

2. Combine both sections in just one specifying the type of each one (see Listing 12).

```
<invoke operation="check" partnerLink="assessor"">
    <inputContainer>
        <department name="request"/>
    </inputContainer>
    <outputContainer>
        <department name="risk"/>
    </outputContainer>
</invoke>
```

Listing 11: BPEL-D Invoke with containers

```
<invoke operation="check" partnerLink="assessor"">
    <container>
        <department name="request" type="input"/>
        <department name="risk" type="output"/>
    </container>
</invoke>
```

Listing 12: BPEL-D Invoke with containers (2)

The first option implies writing 2 more code lines for each activity, and is a little less clear than the second one. If we bring this situation to a process with one hundred activities it means much more work for the programmer and less clarity understanding the process understanding.

The second option makes the code much easier to write for the programmer, and at the same time makes understanding the process clearer and easier. And it is still close to the metamodel. These advantages are easier to see in activities that have both input and

output data; in other cases the benefit is not as obvious. The Listing 13 illustrates both situations.

```
...
<invoke operation="check">
    <inputContainer>
        <department name="request"/>
    </inputContainer>
    <outputContainer>
        <department name="risk"/>
    </outputContainer>
</invoke>

<assign>
    <inputContainer>
        <department name="decision"/>
    </inputContainer>
    <outputContainer>
        <department name="approval"/>
    </outputContainer>
    <copy>
        <from expression="decision" />
        <to department="approval"/>
    </copy>
</assign>

<invoke operation="approve"">
    <inputContainer>
        <department name="request"/>
    </inputContainer>
    <outputContainer>
        <department name="approval"/>
    </outputContainer>
</invoke>
...
```

```
...
<invoke operation="check">
    <container>
        <department name="request" type="input"/>
        <department name="risk" type="output"/>
    </container>
</invoke>

<assign>
    <container>
        <department name="decision" type="input"/>
        <department name="approval" type="output"/>
    </container>
    <copy>
        <from expression="decision" />
        <to department="approval"/>
    </copy>
</assign>

<invoke operation="approve"">
    <container>
        <department name="request" type="input"/>
        <department name="approval" type="output"/>
    </container>
</invoke>
...
```

Listing 13: Different container designs (left) following first option (right) following second option

Taking this into account we decide to follow the second one, it means that each activity will have associated a single space to store information, a *Container*.

This way each container will have *Departments* for input and output data. For this reason we add a new attribute to the department construct to specify when data is input or output. We can considerate this information as redundant, but this way it can be understood easier in a quick glance. We call this attribute *type*, and it could take as value *input* or *output*, depending on the kind of the information.

Along our research process we will find that this definition is not complete adequate for our targets. For this reason types *toPartner*, *fromPartner* and *faultData* will be added to this set. These changes will be explained into the section 3.3.1.1 during the explanation of BPEL-D simple activities.

3.1.4 Containers definitive syntax

After all the changes and decisions taken through the design process, we are close to obtaining the final syntax for our two first constructs. But looking again to BPEL-D syntax, we find that the names, *departments* and *container,* are not the most adequate option.

The reason is the following: in a BPEL process, the section to declare a variable is called variables section, the section to declare a correlation set is called correlationSets. The same happens with fault handlers, compensation handlers, etc. Following that, why call the section that declares departments a container?

For this reason, and following the BPEL syntax style, we propose two more appropriate names, renaming department to *container*, and the container declaration section as *containers*.

This is the last aspect to finalize the with containers syntax. Listing 14 illustrates the *container* syntax definition:

```
<element name="container">
    <complexType>
        <attribute name="name" type="NCName" use="required" />
        <attribute name="type" type="input|output|toPartner|fromPartner|faultData" use="required"/>
    </complexType>
</element>
```

Listing 14: BPEL-D Container definition (3)

During the implementation process, we will need to add some attributes to this definition to imitate the variables behaviour. It is explained in the section 6.2.

And the *containers* declaration section is illustrated in Listing 15:

```
<complexType name="containers">
    <complexContent>
        <sequence>
            <element ref="containers" minOcurrs="1" maxOcurrs="unbounded" />
        </sequence>
    </complexContent>
</complexType>
```

Listing 15: BPEL-D Containers definition

3.2. Data links

In the previous section we talked about the way of storing the information in BPEL-D: the containers. Now is time to solve the main part of the research; how to pass this information through a BPEL-D process. As we know, in BPEL 1.1 information pass through the different activities in an implicit way, by sharing globally visible data variables. Now our intention is make these relations explicit.

Firstly, several options were considered to decide how to confront the problem of sending data between two activities that will be put into different process fragments. These options, explained in [KL06] were the following:

- Shared database, with engine level data replication to synchronize and retrieve data on-demand.
- Data sent with the next control link:
- Data links and broken using a data service at each partner.
- Data links, broken using WS message exchanges.

Data links is the chosen option. This option is the easiest to split, has been commonly used in workflow languages and can be derived by analyzing data process dependencies.

The next sections explain the syntax and semantics in relation with them.

3.2.1. Datalinks semantics

Coming back to WSFL 1.0 [WSFL], one of the precursors of BPEL, we find that data links idea is not a new concept. WSFL did not use variables; it shared the information directly by using data links between activities. That means that it shows the information exchanged in an explicit way, exactly what we want for BPEL-D. For this reason we will observe WSFL syntax as an approximation to define our own data links.

A data link must specify how a source activity passes data to the flow engine to the target activity. Two activities are data dependent when the first one expects information from the second. This dependency will be expressed by a new BPEL-D construct we will call *data link*. The data link definition expresses that it must define the data exchange process along the business process.

The first step in the process of defining this construct is to answer the next question: what must it connect: activities or containers? The following arguments justify each one of these options:

- Connect containers: we have to express how information passes from a part of an input container to a part of an output container. Both containers must be connected.

- Connect activities: we have to express how one target activity expects data from a source activity. Both activities must be connected.

To answer this question we take a look in the [LR00] metamodel to find some issues regarding this. Recall from Chapter 2 that the metamodel encompasses the definition of the process data along a process in two parts:

- First part, *data dependencies*: how to express which activities or predicates expect information from other activities.

- Second part, *data mapping*: how data elements of an input container are composed from data elements of the output containers of these other activities.

Taking that into account the answer is that we will need not one but two different constructs to explain the information interchange in a correct way. We find these two concepts also in the metamodel definition:

The metamodel uses the term of *data connector* to express a data dependency between two activities. To define which data elements of one input container expects data from which data elements of which output container, it uses another structure called *map*.

With this information we find the two constructs that we where looking for. They are *data links* and *maps*. We use data links as solution to the first question: we express the data dependency between two activities using a data link between them. On the other hand, we use maps to define which part of the source input container will be mapped into which part of the target output container. This way when the input container is materialized at runtime, the correct element in it will receive a copy of the actual instance of the correct element of the output container.

3.2.2. Datalinks syntax definition

Having the concepts, we now define their syntax. The FDL (based on the metamodel) and WSFL data links provide two examples that we can take as starting point in our design.

The FDL syntax defines data connectors by the clause DATA, expressing source and target activities preceded by the keywords FROM and TO (see Listing 16). Data maps are associated with the data connector and identified by the keyword MAP. The source

data part is followed by the keyword TO, which will be followed by the target part. We can see it clearer in the next picture. For more information see [LR00].

```
DATA FROM 'receive1'
     TO 'invoke1'
     MAP 'receive1.request' TO 'invoke1.request'
```

Listing 16: FDL Map

Next, we look at the WSFL syntax [WSFL] and find a very close concept; a <dataLink>. This construct uses two attributes, source and target, to specify the linked activities. The information exchange process is described using a <map> element which normally appears nested in a datalink. Each map will include attributes to explicitly state which part of the input message originates the information, and which part of the output message will receive it. We can see it in Listing 17:

```
<dataLink name="connection"
          source="receive1"
          target="invoke1"
     <map sourceMessage="creditInformationMessage" targetMessage="request"
          sourcePart="creditInformationMessage" targetPart="request"/>
</dataLink>
```

Listing 17: WSFL Datalink

This is an interesting starting point. The most important subject of a *data link* is relating the activities. For this reason the option to use the names of both activities to define the relation seems to be a good idea. For this reason the first two attributes in the new structure will be *source activity* and *target activity* (this issue is detailed discussed in section 3.2.4). The idea of having a name for the *data link* seems not to be very important but it is interesting for the implementation; for this reason and we also import this attribute for the data link.

3.2.3. Maps syntax definition

The next step is defining the information exchange between the containers, or in other words, defining the *map* structure. The first option follows WSFL and FSL, even if here the information is not taken from an activity member or in an interchanged message. In this case the information will pass from containers to containers, but the structure to exchange it is always the same: one source element, and one target element.

A different possibility consists in imitating the syntax of the copy structure from the <assign> activities which is closer to the FDL example. That means each map will be composed from two different constructs a <from> and a <to>. Each one of them will refer to a concrete container, a part or a query into it.

Then, the two possible options to define the map syntax are:

1. Create the construct following the structure used in WSFL (Listing 17), as shown in Listing 18.

2. Imitate the <assign> copy structure as illustrated by Listing 19.

```
<dataLink sourceActivity="receive1" targetActivity="invoke1">
        <map sourceContainer="request" targetContainer="request" />
</dataLink>
```

Listing 18: BPEL-D Datalink (1)

```
<dataLink sourceActivity="receive1" targetActivity="invoke1">
        <map>
            <from sourceContainer="request"/>
            <to targetContainer="request" />
        </map>
</dataLink>
```

Listing 19: BPEL-D Datalink (2)

We do not find important reasons for choosing between these two options. The question seems to be just a question of style, because it does not represent an important change in our model. In our situation, we decide to take the first option which looks clear enough and save code lines, (and it has been already used in WSFL as solution to describe explicit data flow). During the implementation process we will realize that this option is more efficient.

Taking into account the structure of the elements that we can store in the containers, it is necessary to add some attributes to specify more concretely the information transfer process. Taking as example the from-to construct of the *assign* and the WSFL map definition, we describe three attributes for doing it. These three attributes will be, for both source and target elements are: xContainer, xPart and xQuery where 'x' is replaced by either 'source' or 'target'.

This is an efficient solution which let us have implicit assignments into the datalinks.

3.2.4. Datalinks integration in the BPEL-D process

Once the elements themselves are defined we find some little difficulties to be solved. The beginning of the datalink syntax definition process locates us with the same problems we had when where working with containers. The first question is: where must the data links be defined? The options are similar too:

1. Include them into the activities, like the control flows.

2. Use a section to declare them as the same way as variables, etc.

About *containers*, we decide to include them into activities because each one was really related with one activity. In this case we find a different situation; A *data link* does not belong to an activity (at least it would belong to the activities which share it). We can say that it belongs to the whole process description like BPEL variables.

For these reasons we decide declare them in a new section called <datalinks>, it means to follow the first option. Listing 20 illustrates the result.

```
<dataLinks>
    <dataLink sourceActivity="receive1" targetActivity="invoke1">
        <map sourceContainer="request" targetContainer="request" />
    </dataLink>

    <dataLink sourceActivity="receive1" targetActivity="invoke2">
        <map sourceContainer="request" targetContainer="request" />
    </dataLink>

    <dataLink sourceActivity="assign1" targetActivity="reply1">
        <map sourceContainer="approval" targetContainer="approval" />
    </dataLink>
    ...
</dataLinks>
```

Listing 20: BPEL-D Datalinks section

The activities name question

Another problem that we find here, in our model is the following: we are going to relate activities using their names. The problem is that BPEL 1.1 do not force the programmer to name activities, it means the name attribute is optional. Even worse, even if an activity is named, the name does not have to be unique so more than one activity can have the same name. So, what can we do when an activity is not named? We find possible the next options:

1. Express the dependencies creating links between activities in a similar way as BPEL control links.

2. Include the data dependencies into an existing control links.

3. Force the programmer to uniquely name the activities.

The first option will have a very detrimental effect on the code, because it means writing three more code lines for each data link. If we think in a normal BPEL process with for example, eight activities and ten data links, the result will be thirty more code lines, and more complex code which will be more difficult to understand.

The second option is not a good possibility either, because it will mean changing existing BPEL constructs: control links. That is not very coherent. Apart from that, data links can relate activities which are not related for control links. For this reason we must discard this option.

The third one seems to be the simpler one, because normally the programmer use to name activities, and in the case that he does not, the fact of forcing him to do it does not represent a big change in the code.

For these reasons, our decision is solve this problem forcing the programmer to name the activities; it means to follow the third option.

3.2.5. Datalink final syntax

The next Listings illustrate the syntax of the four new BPEL-D elements described in this section. Listings 21 and 22 show the syntax of the BPEL-D map and maps section declaration.

```
<element name="map">
    <complexType>
        <attribute name="sourceContainer" type="NCName" />
        <attribute name="sourcePart" type="NCName" />
        <attribute name="sourceQuery" type="string" />
        <attribute name="targetContainer" type="NCName" />
        <attribute name="targetPart" type="NCName" />
        <attribute name="sourceQuery" type="string" />
    </complexType>
</element>
```

Listing 21: BPEL-D Map syntax

```
<complexType name="maps">
    <complexContent>
        <sequence>
            <element name="map" type="map"  minOcurrs="1" maxOcurrs="unbounded" />
        </sequence>
    </complexContent>
</complexType>
```

Listing 22: BPEL-D Maps syntax

Listings 23 and 24 show the syntax of the BPEL-D datalink and datalinks section declaration.

```
<complexType name="dataLink">
    <complexContent>
        <attribute name="sourceActivity" type="NCName" />
        <attribute name="targetActivity" type="NCName" />
        <sequence>
            <element name="maps" type="maps" minOccurs="1"/>
        </sequence>
    </complexContent>
</complexType>
```

Listing 23: BPEL-D Datalink syntax

```
<complexType name="dataLinks">
    <complexContent>
        <extension base=tExtensibleElements">
            <sequence>
                <element name="dataLink" type="dataLink" minOcurrs="1" maxOccurs="unbounded"/>
            </sequence>
        </extension>
    </complexContent>
</complexType>
```

Listing 24: BPEL-D Datalinks syntax

3.3. BPEL Activities

Having described the necessary new constructs it is time to bring them into the normal behaviour of BPEL activities (see [BPEL11]), and define how they must change the way of taking and passing information in the BPEL-D style without variables.

As we mentioned in the first steps of our research, the first intention of this work is to apply the BPEL-D syntax to a limited subset of BPEL (see [KL06] and [Pal07]). We are extending this subset with the intention of create a more complete syntax.

Treating a bigger subset to apply our syntax we obtain a more difficult design process but some important advantages. As has been said, working with this bigger horizon, we will have a wider vision which will let us obtain a more coherent syntax comparing the result of our decisions in more situations.

For this reason we will study the whole set of BPEL activities as well as the constructs they use. Following that we start introducing containers and data links into BPEL simple activities and continue after that with the structured ones.

3.3.1. Simple activities.

This part explains the semantics of containers and datalinks in relation with the BPEL simple activities: <receive>, <invoke>, <reply>, <assign>, <throw>, etc.

3.3.1.1 The receive activity.

"The <receive> activity, allows the business process to do a blocking wait for a matching message to arrive" [BPEL11]. The <receive> defines the point where the process expects to be invoked by a partner. A BPEL process must contain at least one <receive> activity as the first activity of the process having the "createInstance" attribute set to "yes". A received message data can be stored to a variable using this activity.

In another words, the main work of a receive construct will be receive a message from a partner and introduce this information into our business process.

```
<receive partnerLink="ncname" portType="qname" operation="ncname"
         variable="ncname"? createInstance="yes|no"?
         standard-attributes>
  standard-elements

</receive>
```

Listing 25: BPEL Receive activity

If we take a look at the BPEL description of this construct (Listing 25), we can see where the variable takes part. As has been said, it refers to a variable to store directly the content of the incoming message, but the variable is optional (see the "?" symbol near the attribute).

BPEL-D does not use variables. Because of that, the information coming from the partner must be stored in a different way: using *containers*.

As has been said, the information processed by the activity to offer to the rest of process activities is stored in output containers. This is the case of the receive activity, which can take data from the received partner message to pass it into the process. Then *output containers* are the solution for this issue.

When defining *containers*, we mapped directly from the content of a WSDL message into a BPEL-D container. For this reason one *output container* will be enough for this issue. As in the case of using variables this container will be optional too.

As has been introduced in section 3.1 about containers definition, we have another question to solve here. Receive, invoke and reply are the activities used to communicate our process with partners; it means getting and sending information. For this question we must differentiate the two kinds of data involved in the process: data from and to the process itself and data from and to the partners.

The solution we decided on is to create new different types of containers to specify in a better way which kind of information are we managing in the process. We will use these containers into the activities implied in communication with partners. The new types are the following:

- *fromPartner*: a special kind of output container which we will use in receive, pick and invoke activities for differentiating the information coming from a partner (outside) from that which comes from the process.

- *toPartner*: a special kind of input container which we will use in invoke and reply activities when the data is to be send to a partner (outside).

Attending to *receive* it is necessary to provide it with zero or one *fromPartner container* to write the data coming via the WSDL message sent by the partner.

This decision seems to be adequate, but there are still some other questions that must be solved respecting with these issues.

The main theme in this research is the data flow through a business process, but we can not forget the links between activities: control flow. As has been introduced before, the

control links between activities which define the control flow along the process, can need information to decide chose between two branches or simply for continue the flow or not; it is the *transition conditions*.

In BPEL 1.1 this information can be taken from the shared variables of the process, but now we substitute them for containers, so: how can we obtain this information? As has been said, our approach does not let transition conditions having their own containers. They take the necessary information from their source activity containers.

Then, the more logical answer is providing the *receive* construct with zero or more input containers to store information coming from other activities in the business process, to be taken by its *transition conditions*. Figure 8 illustrates the communications of this construct.

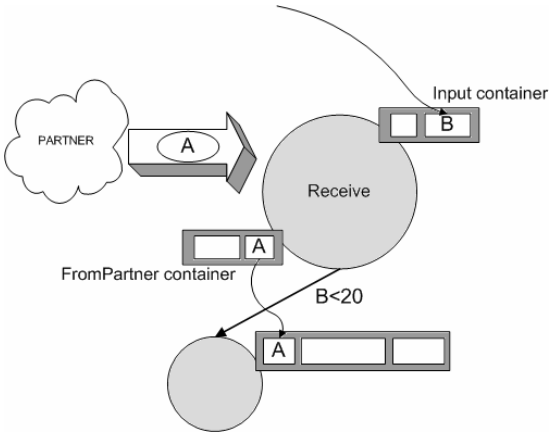

Figure 8: Receive with containers

Another aspect to take in account is that the *receive* activity and the *pick* activity are the only BPEL activities which can create a business process instance. For this reason both activities have the attribute *createInstance*. As has been said, when this attribute has a true value, it means that this activity is the first one in the process and the one responsible for creating it. For this reason, it cannot receive information from a previous one, because simply it does not exist.

For this reason we must differentiate between creating process receives and not creating process receives. In the second case we must provide the activity with zero or more input containers, depending on the information needed by the transitions conditions. But in the first one it is always zero as we have noted.

After the research process, we can define the syntax for the <receive> BPEL-D activities, shown in Listing 26:

```
<receive partnerLink="NCName"
         portType="qname"
         operation="NCName"
         createInstance="yes|no"?
         standard-attributes>
         standard-elements
        <containers>?
            <container name="NCName" type="input"/>*
            <container name="NCName" type="fromPartner"/>?
        </containers>
        <correlations>?
            <correlation set="NCName" initiate="yes|no" />+
        </correlations>
</receive>
```

Listing 26: BPEL-D Receive activity with containers

As we can see in this listing, the changes between the BPEL and BPEL-D syntax are the inclusion of the containers (zero or more input ones, and zero or one fromPartner) and the exclusion of variables.

3.3.1.2. The invoke activity.

"This construct allows the business process to invoke a one-way or request-response operation on a portType offered by a partner" [BPEL11]. The <invoke> activity is maybe the most complex simple BPEL activities. The reason of it is that an invoke operation can work in a synchronous or in an asynchronous manner. Therefore, we must treat the activity differently in one case or in another.

An asynchronous invocation will require just an input variable because in this case we do not expect a response. In the other hand when we use it in a synchronous invocation we can need both variables. In BPEL both variables are optional (see Listing 27).

```
<invoke partnerLink="ncname" portType="qname" operation="ncname"
        inputVariable="ncname"? outputVariable="ncname"?
        standard-attributes>
  standard-elements
  ...
</invoke>
```

Listing 27: BPEL Invoke activity

It is necessary to have this situation to account for translating this activity to BPEL-D syntax. Another aspect to treat is the new containers construct. We must therefore consider the following needs of input and output information in invoke:

- Zero or more *input container*, to store the information that come from the process that we use for the transition conditions.
- Zero or one *toPartner* container, to store information from our business process that gets sent to our partners.

- Zero or one *fromPartner* container, to store the response from the partner for the case of a synchronous invocation.

Listing 28 shows the BPEL-D syntax for the invoke activity:

```
<invoke partnerLink="NCName"
        portType="qname"
        operation="NCName"
        standard-attributes>
        standard-elements
        <containers>?
            <container name="NCName" type="input" />*
            <container name="NCName" type="toPartner"/>?
            <container name="NCName" type="fromPartner"/>?
        </containers>
        <correlations>?
            <correlation set="NCName" initiate="yes|no" pattern="in|out|out-in" />+
        </correlations>
        ...
</invoke>
```

Listing 28: BPEL-D Invoke activity with containers

The <invoke> activity can be associated with a fault or a compensation handler. These constructs have their own containers as will be explained later in the scope section (3.3.2.7) and thus do not affect the containers in the invoke activity.

3.3.1.3. The reply activity.

The <reply> activity is used to send a response to a previously accepted though a receive activity. In a different way from invoke, the reply is used only for synchronous interactions.

```
<reply partnerLink="ncname" portType="qname" operation="ncname"
       variable="ncname"? faultName="qname"?
       standard-attributes>
  standard-elements
  <correlations>?
     <correlation set="ncname" initiate="yes|no"?>+
  </correlations>
</reply>
```

Listing 29: BPEL Reply activity

As we can see in Listing 29, reply has an optional variable containing the data to send to the partner. This variable must be substituted by BPEL-D data containers.

Firstly we solve the problems of normal data flow. <reply> only sends information to partners and never receives it. For this reason we provide it with zero or one *toPartner* container containing the information to be sent in reply.

The situation for transition conditions data is exactly the same as for "receive" and "invoke". For this reason our solution will be exactly the same: provide it with zero or more input containers for this target.

```
<reply partnerLink="NCName"
       portType="qname"
       operation="NCName"
       faultName="QName"?
       standard-attributes>
       standard-elements
       <containers message="NCName">?
           <container name="NCName" type="input"/>*
           <container name="NCName" type="toPartner"/>?
       </containers>
       <correlations>?
           <correlation set="NCName" initiate="yes|no"/>+
       </correlations>
</reply>
```

Listing 30: BPEL-D Reply activity with containers

In Listing 30 we can see the final result of the BPEL-D reply syntax.

3.3.1.4. The assign activity.

"The assign activity can be used to copy data from one variable to another, as well as to construct and insert new data using expressions" [BPEL11]. These expressions can operate with different subjects to obtain new values for a variable. These subjects can be message selections, properties and literal constants.

```
<assign standard-attributes>
  standard-elements
  <copy>+
    from-spec
    to-spec
  </copy>
</assign>
```

Listing 31: BPEL Assign activity

The structure of this activity is a bit more complicate than the others (see Listing 31). It copies a value from the source "from" to the destination "to". There must be type-compatibility between both constructs. The element of the "from" can be one of the following:

- A complex variable, which contains a complete message
- A message part contained into a variable
- An XPath expression; a constant, a query acting over a message or an XPath function.

```
<complexType name="tFrom">
    <complexContent>
        <extension base="bpws:tExtensibleElements">
            <attribute name="variable" type="NCName"/>
            <attribute name="part" type="NCName"/>
            <attribute name="query" type="string"/>
            ...
        </extension>
    </complexContent>
</complexType>
```

Listing 32: BPEL From definition

As we can see in Listing 32, from construct can take information from a BPEL variable. The special structure of this construct leads to more aspects that must be studied than in the cases treated before. The first decision point is: what is the best place for place the data containers?

The assign activity can be composed from various "copy" elements, for this reason we have two different options:

- Following the syntax of the previous activities, the containers will belong to the assign; it means to the general activity.
- Each copy element will have its own, input and output, containers closer to the BPEL 1.1 assign definition.

The second option does not seem to be the appropriate solution. For example, if we think of an assign activity with five copy elements, taking the second option will mean writing approximately twenty code lines just for the containers (about ten more for the data links).

The result will be more work for the programmer and a less clear code. At the same time, it is not efficient because much of these copy elements can use the same input container (a reason for not writing it several times).

Apart from that, the first option is more coherent with the syntax of the other simple activities, and if we see an example we will see that it is as close to the BPEL 1.1 assign's syntax as the other one. For all these reasons we decide to take the second option. The result is shown in Listing 33:

```
<assign name="assign">
    <containers>
        <container name="approval" type="output"/>
    </containers>
    <copy>
        <from expression="'yes'" />
        <to container="approval"/>
    </copy>
</assign>
```

Listing 33: BPEL-D Assign activity with containers

Having defined the syntax and identified from where to take the input data and where to put the output data, we study the other BPEL elements implied in the assignment. Recall that BPEL 1.1 assign used variables in the "from-spec" and "to-spec". Therefore, the solution can be simply to change the reference to variables to a reference to containers (see Listing 34).

```
<complexType name="tFrom">
    <complexContent>
        <extension base="bpws:tExtensibleElements">
            <attribute name="container" type="NCName"/>
            <attribute name="part" type="NCName"/>
            <attribute name="query" type="string"/>
            ...
        </extension>
    </complexContent>
</complexType>
```

Listing 34: BPEL-D From definition

The next aspect of the "assign" construct to be adapted to the syntax is the theme of the possible contents of the "from-spec" construct. These can be variables, part of variables, constants or XPath functions applied on variables. The changes that we must apply in this case are the following:

- Complete message stored in a variable: complete message stored in a container.
- Part of a message stored in a variable: part of a message stored in a container.
- Constant: does not need changes.

The last question is the case of applying an XPath function on a variable. We redefine BPEL's XPath functions to adapt them to BPEL-D syntax as follows:

- GetVariableData(), for GetContainerData().

- GetVariableProperty(), for GetContainerProperty().
- GetLinkStatus(), continue with the same syntax.

The final decision in relation with this activity is the cardinality of its input and output containers. As has been said, assign activity can include an indeterminate number of copy elements but they cab even read from the same container. For this reason we provide them with zero or more input and output containers as we provided the other activities.

Apart from that, we must include the classic input containers used for getting information for the transition conditions (this does not modify the described cardinality). We are now ready to define the BPEL-D syntax of the activity with data links (see Listing 35).

```
<assign standard-attributes>
        standard-elements
        <containers>
            <container name="ncname" type="input" />*
            <container name="ncname" type="output" />*
        </containers>
        <copy>+
            from-spec
            to-spec
        </copy>
</assign>
```

Listing 35: BPEL-D Assign activity with containers

3.3.1.5. The throw activity.

"The throw activity can be used when a business process needs to signal an internal fault explicitly" [BPEL11]. Every fault is required to have a globally unique QName. The throw activity is required to provide such a name for the fault and can optionally provide a variable containing data that provides further information about the fault.

```
<throw faultName="qname" faultVariable="ncname"? standard-attributes>
  standard-elements
</throw>
```

Listing 36: BPEL Throw activity

The activity definition (see Listing 36) shows that the use of variables in this construct takes place in the faultVariable. This variable must be substituted for a container. But taking a look along the containers type definition, we find that none fits exactly with the nature of this case, because the input information comes not from a partner or is used for the transition conditions.

The solution is to define a new type of container which will be used. We call it *faultData* container. This will be a kind of hybrid container, because the data will go here for the fault and there will be an implicit data link to the fault handler's input container from this (really it is an input container but the data will be sent at runtime to the right fault handler). Following that, this container is provided with zero or one *faultData* container.

Listing 37 shows the BPEL-D throw construct new syntax:

```
<throw faultName="qname"
       standard-attributes>
       standard-elements
       <containers>
            <container name="ncname" type="faultData"/>
       </containers>
</throw>
```

Listing 37: BPEL-D Throw activity with containers

3.3.1.6. Rest of simple activities.

As we said in the beginning of our research, our intention is to cover the whole BPEL syntax. That means studying all the activities described for the language.

In the different sections of section 3.3.1 we have treated "receive", "invoke", "reply", "assign" and "throw". There are two more simple activities: "terminate" and "empty". They are not covered extensively here because they do not access BPEL variables. In spite of that they can have transition conditions following them. For this reason they will need to be provided with zero or more input containers.

Wait activity situation is a bit different. In BPEL, this activity can read from variables to fix the waiting time. Taking into account this as well as the transition conditions information, wait must be provided with zero or more input containers too.

3.3.2. Structured activities.

Having introduced the syntax and semantics of BPEL simple activities, we now move on to BPEL's structured activities that enable one to build a high-level logic business process.

"Structured activities prescribe the order in which a collection of activities take place. They describe how a business process is created by composing the basic activities" [BPEL11].

Bringing the behaviour of these activities to BPEL-D will represent a complex task. Again, the BPEL activities we are going to use will be the whole set of BPEL structured activities: "sequence", "switch", "while", "pick", "flow" and "scope".

The fact that each one of these activities has a specific behaviour means finding special situations in each case, treated in the specific activity's subsection. First, we address some questions in relation with the whole set. Then, we move on to each individual activity.

3.3.2.1. Some common questions.

Taking into account the nature of the structured activities, we find that there are some common points between most of them that we find opportune treat together. Maybe not all them are applicable to the whole set; in this case we will explain these specific questions in the respective sections along section 3.3.2.

The logical first question is: how can the structured activities manage the information? This question is in reality composed of three questions:

- How can structured activities get information from the process?
- How can they transmit it inside; to their nested activities?
- How can they pass it outside, to the business process again?

This question is directly related with the semantics of the input and output containers (the other three types does not participate in most of these activities). When we treated the simple activities we found quite a simple behaviour in relation with this, because it was similar to the usage of variables. Here these questions will be more complex.

Input information

In simple activities all the incoming information from the process arrives to the *input* containers of the activity (or in same concrete cases to the *fromPartner* one). In a similar way all the outgoing information goes passes through the output containers (or

through the *toPartner* one). Because of that the translation to BPEL-D syntax had not a high complexity. This translation is not as easy with complex activities.

The first aspect to treat will be how to get the information from the business process. The usage of an input container seems to be mandatory, at least for continuing the design process with coherence. But the very nature of the structured activities provokes some unknowns about the nature of the input container.

The question we are referring is: must all the information pass into the activity through the input container, o can some part of that can go directly to the activity/activities inside? This question comes from the issue that in structured activities we can find two types of incoming information:

- On one hand, the information which the main structured activity needs; it means the information to select a branch in a "switch", the loop condition information in a "while", or information used for transition conditions, as some examples (see maps (a) in Figure 9).
- On the other hand, the information which the activities inside needs. For example: the data to be send for a "reply", to be modified for a "assign" or even the first type of information in the case of a nested structured activities (see maps (b) in Figure 9).

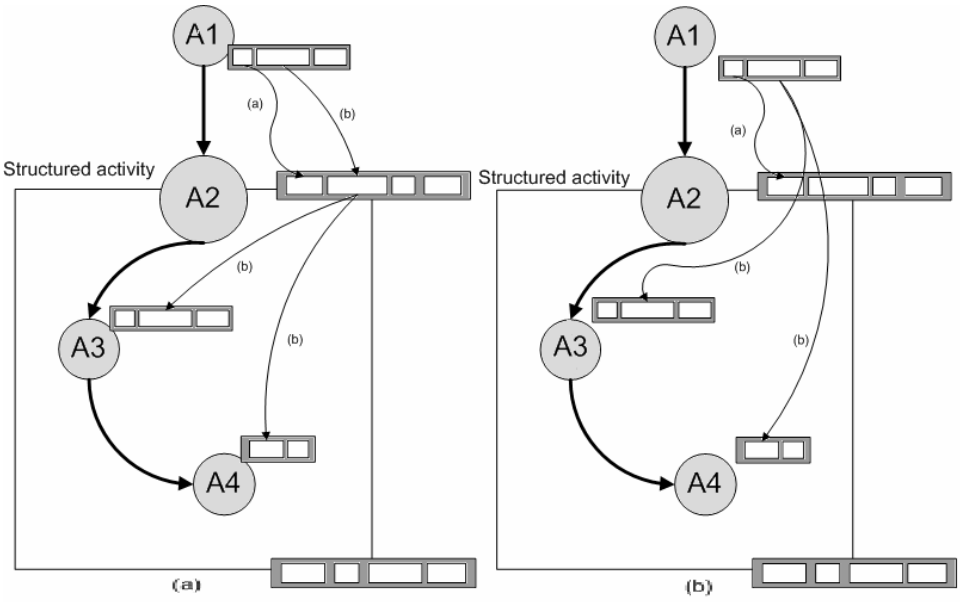

Figure 9: Structured activities datalinks options (a) using input containers for every structured activity, (b) not using input containers for each structured activity

In summary, must the input container be a proxy for all the information into the structured activities? We find three possible solutions to answer this question:

1. Both types of information must pass through the input container.

2. Just the first type of information will be from the input container. The second type can pass directly to the target activities.

3. Allow both solutions and let the programmer choose between them.

To solve the problem we search into the Workflow Production Metamodel explained in [LR00]. In this case it does not provide an answer as clear as it did in the previous situations. The problem is that the metamodel does not consider the possibility of structured activities. The only approximation that we can find here is the case of the global process input and output containers, which is not a very good example for our needs.

In this case the metamodel propose using both, input and output containers, for each business process following the same rules as for simple activities. The main reason of that is to understand a business process as an independent piece of code which can be reusable as a unit in other more complex business processes. Reusing code is not a very common situation in this environment. Therefore, we will need arguments more relevant to our environment to aid us in making a decision.

No one of the options contradicts the metamodel or is clear supported for it. For this reason we will need to research other aspects to find more important reasons to decide.

One of the arguments which we are taking into account along our research is the code clarity in relation to make reading it easier, and also the more comfortable writing code to make easier the labour of the programmer. Attending of that we must prove how our options relate to this theme.

There is a situation in which the use of input containers to pass the information into the structured activities is especially inefficient. This situation is illustrated in Figure 10, when we have a process with a great number of structured activities nested, in which the data is necessary for a deep activity. Case (a) illustrates how it will be using input containers to pass the data into a structured activity. Case (b) illustrates passing the data directly to the nested activity.

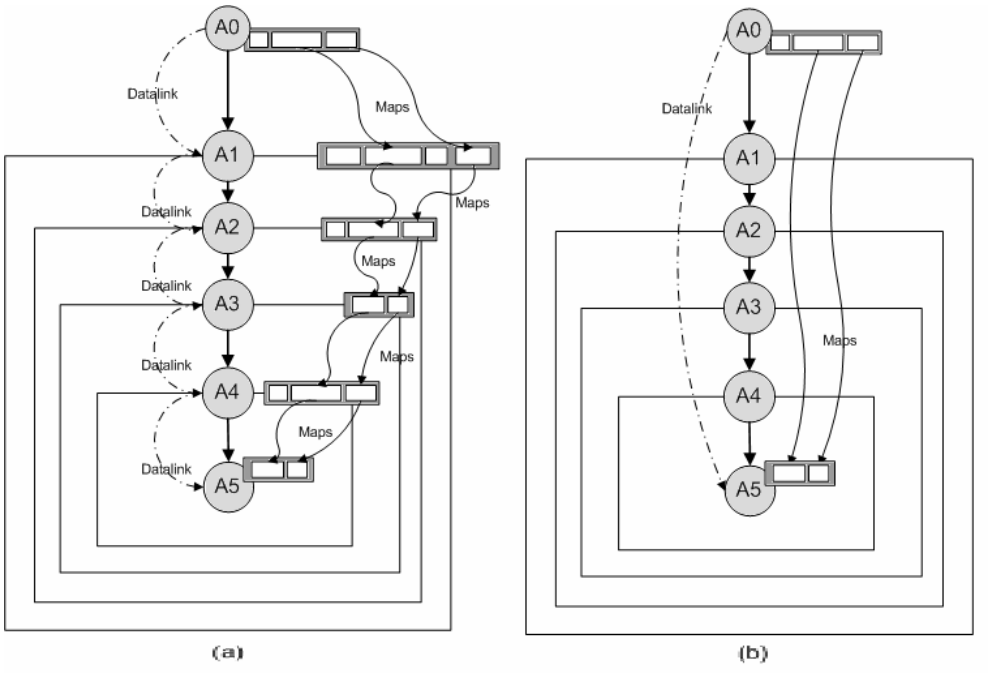

Figure 10: Structured activities with datalinks (a) using input containers for every structured activity, (b) not using input containers for each structured activity

Calling *n* the number of nested activities to go along, and *m* the number of maps, we will have *n-1* datalinks more, *(n-1)*m* maps more and *(n-1)* containers more. If we bring this situation to code writing (4 code lines to declare a datalink, 1 for a map, and 3 for a container) it means writing approximately 8*(n-1) new lines. It the previous example it will mean 40 more code lines; more work and a more difficult to understand process.

Apart from that, it can make the business process comprehension maybe a bit more complex to follow. It means that having a quite long process, it will find from where comes the information closer as in the other option (in which for example can succeed that we must go to the first activity to find where comes the information needed in the last one).

In the other hand this is not a very common situation in the environment of BPEL. In normal cases we do not need pass the information along a great number of structured activities.

The table below summarizes the advantages and handicaps for both solutions:

Options	**Pass through containers**	**Pass directly to target activity**
Pros	Can make easier understanding the information passed through structured.	Much less code to write. Application situation is more common.
Cons	8*(n-1) code lines more to write.	

Table 2: Structured activities input containers options

Therefore, the best option seems to be letting the programmer choose between both possibilities depending on the situation.

Output information

The next question is the opposite of the previous one but directly related; can the information go out from the structured activity directly, or must they have an output container acting as proxy?

This is a difficult question to answer, taking into account the different nature of the activities that we are treating. For this reason we will try to achieve a good approximation which will be discussed particularly in the parts of the paper dedicated particularly to each activity.

Most of structured activities can contain just one activity inside them. “Switch” and “pick” can enclose more than one, but they choose between different branches; then the real performance is as just having one activity. When we treat with “Sequence” (section 3.3.2.5) we will find some important different aspects which will require a different treatment. “Flow” encloses more than one activity that can run in parallel but sequentially too; for this reason it will need a special explanation too (section 3.3.2.6).

As in the case of incoming information, we find the same three different options to solve the problem. As has been said, they were: force the programmer to use the output containers, do not let him to use it, or allow him to choose.

In general, we are not in the same exact situation as in the previous section, when we talked about incoming information. There we result that some part of this information was used for the structured activity itself. This is not the case with outgoing information (except some concrete cases discussed afterwards like the “while”).

The arguments to take a decision are similar too, to those found in the previous section. If we try to find arguments into the metamodel, we find the same situation; it does not provide us with more reasons than code reusability and that the data links must follow control flow. The reason is again that the metamodel is not prepared for structured activities.

The other argument that we followed with input information was create a syntax which makes code easy to write and to read. In relation with that, results are quite similar too: forcing the information to pass through the output container (Figure 11 (a)) means writing a substantial quantity of data links, maps and containers which are really just redundant information. The option shown in Figure 11 (b) is clearer.

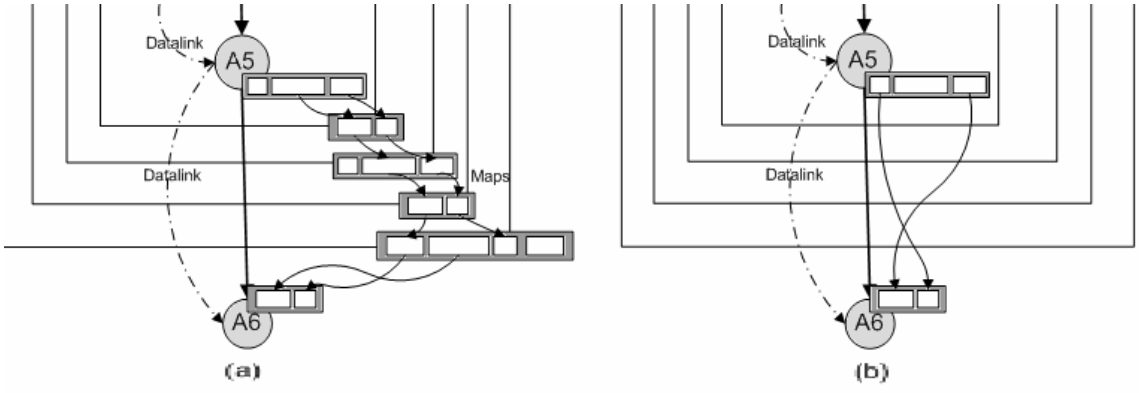

Figure 11: Structured activities with datalinks (a) using output containers for every structured activity, (b) not using output containers for each structured activity.

Maybe it makes the code easier to understand, but at the same time, it can imply read the same information in one line and in the next and so on. That is not clearer code.

Then we have almost the exact situation as we had in the case before and the best option seems to let the programmer choose between both options following his own opinion. Some explanations are introduced in the opportune sections when the treated activity needs it.

Managing the information into the structured activity

The last question is how to distribute the information into a structured activity. But if we think in a more abstract way, we can find that this question is already answered because it is a mix between the other two questions.

The explanation of that is that transmitting the information into a structured activity is just a recursive way of passing data in and out of nested activities, whether they are simple or structured.

Then the solution will be the same too, and supported by the same arguments. We let the programmer choose the solution more adequate in each situation.

3.3.2.2. The switch activity.

This activity let us model the situations when we must choose between different actions to take in our business process. The construct provide a list of conditional branches defined as <case> elements. The first of them whose condition holds true will be the branch performed for the switch.

Having different <case> branches can suggest the idea of provide each of them with its own container, but it is not the right option. We can consider the fact of choosing between branches in a similar way as the transition conditions, with the difference that we will just evaluate them until finding a true condition.

If we take a look into some BPEL process, we will find that it is common for all the conditions of a "switch" activity take information from the same variable; it means just one variable must be evaluated to decide the branch to perform. For this reason this information must be factored into the "switch" input container.

Respect of the information used and produced in a concrete branch of the process, we maintain the common decisions of letting the programmer choose between using redundant input and output containers in the "switch" and passing it directly to the activities which are going to use it.

As a result, we provide the switch activity with zero or more *input* containers and zero or more *output* containers. In Listing 38 we can see the syntax resulting of our decisions.

```
<switch standard-attributes>
        standard-elements
        <containers>?
            <container name="NCName" type="input" />*
            <container name="NCName" type="output" />*
        </containers>
        <case condition="bool-expr">+ activity</case>
        <otherwise>? activity</otherwise>
</switch>
```

Listing 38: BPEL-D Switch activity with containers

3.3.2.3. The pick activity.

"The pick activity awaits the occurrence of one of a set of events and then performs the activity associated with the event that occurred" [BPEL11]. The <pick> activity is composed for a set of branches according with the structure event/activity. The occurrence of the event associated to one of them will determine the execution of the opportune branch.

The semantics of each <onMessage> element is quite similar to the semantic of the <receive> activity. It means that each branch regards the possibility of storing information incoming from the partner message. Again, we substitute BPEL's variables for containers.

```
<pick createInstance="yes|no"? standard-attributes>
  standard-elements
  <onMessage partnerLink="ncname" portType="qname"
             operation="ncname" variable="ncname"?>+
    <correlations>?
       <correlation set="ncname" initiate="yes|no"?>+
    </correlations>
    activity
  </onMessage>
  <onAlarm (for="duration-expr" | until="deadline-expr")>*
    activity
  </onAlarm>
</pick>
```

Listing 39: BPEL Pick activity

We can think of this activity like a special type of "switch" activity, and for this reason we could define its syntax in the same way: providing it with an input container to storing the incoming information from the partner.

But if we take a closer look into the syntax, we see that each branch has its own variable declared. It is reasonable because each message can provide different information than the others. This fact disapproves the first option. The more logic solution is to follow the BPEL syntax and provide each <onMessage> branch with its own input container.

The next example illustrates this situation: if we think in a "pick" activity with the following three branches:

- 1º expects message from partner "buyer" providing data called "lineItem".
- 2º expects message from partner "buyer" providing data called "compDetail".
- 3º expects message from partner "seller" providing data called "revocation".

In this situation if we provide the activity with just one container to store the incoming message, then the business meaning in a container's name is lost and we would need to find a random name to identify it. It means that provide each <onMessage> with the opportune containers is more appropriate.

After discussing these issues the containers provided to each <onMessage> element will be the following:

- One or zero *fromPartner* containers to store the information from the partner.
- Zero or more *input* letting the programmer pass the information used for the transition conditions through them or directly.

- Zero or more *output* containers letting the programmer choose between passing it through a <onMessage> input container or passing it directly to the enclosed activity.

Figure 12 illustrates the communications of pick constructs.

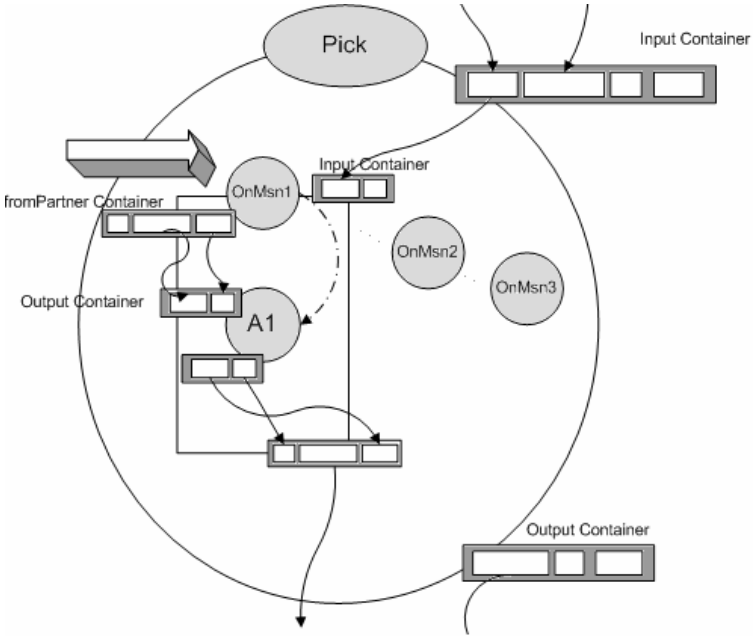

Figure 12: BPEL-D Pick with datalinks

In relation with the <pick>, the situation is the same as has been introduced in the assign and in the common issues section (3.3.2.1). Then we provide it with zero or more input and output containers for the same reasons as before. Listing 40 shows the BPEL-D syntax of this activity including cardinality, situation and type of containers for both elements: <onMessage> and <pick>.

```
<pick createInstance="yes|no"
      standard-attributes>
      standard-elements
      <containers>
            <container name="NCName" type="input" />*
            <container name="NCName" type="output" />*
       </containers>
      <onMessage partnerLink="NCName"
                 portType="qname"
                 operation="NCName">
                 standard-elements
                 <containers>?
                    <container name="NCName" type="fromPartner" />?
                    <container name="NCName" type="output" />*
                 </containers>
     </onMessage>
     ...
</pick>
```

Listing 40: BPEL-D Pick activity with containers

3.3.2.4. The while activity.

"The while activity supports repeated performance of a specified iterative activity" [BPEL11]. The iterative activity is performed until the given Boolean "while" condition no longer holds true.

The while activity is the solution to bring the classic loops of programming into BPEL syntax. If we look to the syntax (see Listing 41), it seems to be a very simple activity, but "while" is one of the most interesting cases of study in this research.

```
<while condition="bool-expr" standard-attributes>
   standard-elements
   activity
</while>
```

Listing 41: BPEL While activity

The main question is about the Boolean condition. Looking in some BPEL process examples we note that this condition consists usually in an expression about one variable in a similar way as the branches of the "switch" activity.

But the iterative nature makes the situation a bit more complicated. In BPEL, the variable used to evaluate the condition could be rewritten during the execution of the activities enclosed in the "while". This situation can affect also the incoming information used for the enclosed activities.

If we substitute the variable used for the activity for an input container to store this information we are solving the first interaction. But, what happens if this information is modified by the enclosed activities? Can a container be rewritten?

In this situation we must discuss about two possibilities:

- Rewrite the input container.
- Use an output container to store the output information, and redirect it to the activity input.

The first option is a bit strange. During our research, and according with the Workflow Metamodel we are following, we talked about input and output containers, but never a hybrid between both. Using this option means that once that the "while" activity finishes its work; the information would pass to the process from a hybrid input/output container.

If we remember the definition and the conditions of the Workflow Metamodel about data connectors, we find that data connectors must always be between input and output

containers but not between both input containers, or even from it to itself. The second condition rules this issue:

$$\Delta(A_1, A_2) \in \wp(o'(A_1) \times i'(A_2))$$

Apart from that, this option is not so coherent with the rest of our BPEL-D syntax, because as we said we never had hybrid containers and no one activity writes and reads from the same container. At the same time this solution is less intuitive, and hard to understand.

In front of these reasons, the only advantage of the first option is that it lets writing a little less code, but that is not enough confronting it with the arguments against it: metamodel conditions, BPEL-D coherence, intuitiveness, etc.

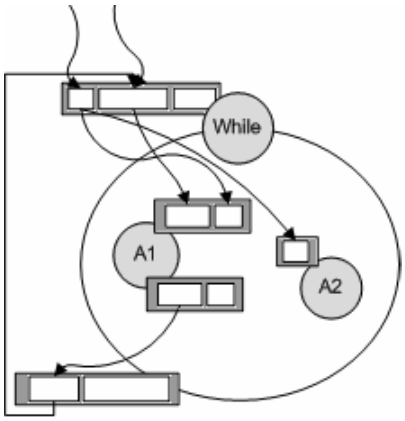

Figure 13: BPEL-D While with containers

For this reason we decide using one container in the output of the activity as we can see in Figure 13. The data links to its input container may come from other activities as well as from its own output container. It means we must add this iteration datalink to the set of normal datalinks used by the while.

The semantics related are that the iteration datalink will write in the while input container after the loop has completed the first iteration. Otherwise, the other data links will be responsible of that. Normally the condition of the loop will take the data from the input container. For more information about this issue see [KL07] and [Pal07].

As in the rest of structured activities we must include the containers for the transition conditions information, and the "proxy" containers to introduce the information into the while body.

But this activity has still another aspect to take in account. We find a condition in the BPEL specification that we must respect in this case: "a link MUST NOT cross the boundary of a while activity, a serializable scope, an event handler or a compensation

handler" [BPEL11]. For this reason, in this case we MUST FORCE the programmer to pass the information through the output container of the while activity.

The question here is that we must not transfer information until the while iteration is complete, because of problems of coherence, synchronization, determinism, etc. This question could affect too to the sequence activity. It is explained in the section 3.3.2.5.

The containers are still optional because it is possible that the while condition is not contained in the data, as well as not be necessary to pass information out of the while. For this reason it is provided with zero or more *input* and *output* containers. Listing 42 shows the new syntax for the "while" activity.

```
<while condition="bool-expr"
       standard-attributes>
       standard-elements
       <containers>
            <container name="NCName" type="input" />*
            <container name="NCName" type="output" />*
       </containers>
       activity
</while>
```

Listing 42: BPEL-D While activity with containers

3.3.2.5. The sequence activity.

"A sequence activity contains one or more activities that are performed sequentially, in the order in which they are listed within the <sequence> element, that is, in lexical order. The sequence activity completes when the final activity in the sequence has completed" [BPEL11].

```
<sequence standard-attributes>
  standard-elements
  activity+
</sequence>
```

Listing 43: BPEL Sequence activity

This last sentence of the activity description has a special importance for our research. We said in the section 3.3.2.1, dedicated to common questions, that we let the programmer pass the information out of a structured activity through an output container or just directly. We talk about two important exceptions to this rule, one is the while activity already explained. The other one is the sequence activity.

The main question here is: if we let data links cross the boundaries of the sequence, what happens in the moment in which the activity fails? In classic BPEL the fault handler would run and correct the changes affecting to this activity, but in BPEL-D it

could not achieve the external activities which have received information from the sequence and which would work with no correct information.

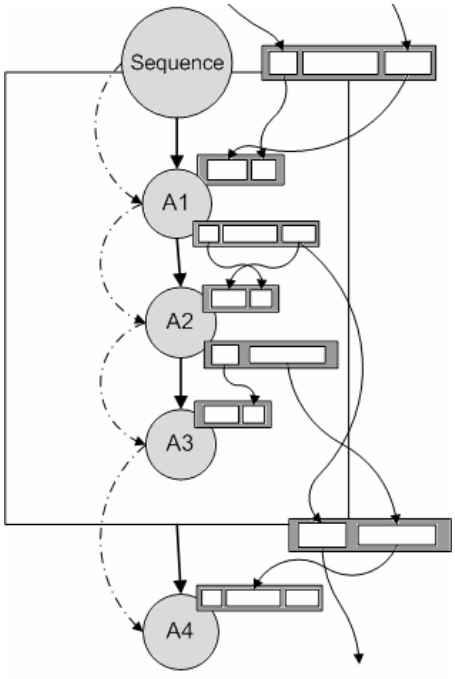

Figure 14: BPEL-D Sequence with datalinks

This situation could provoke problems about coherence with the information that the different activities manage along the business process, or in another words a wrong execution of it. Apart from that, other consequences of this situation are that it would changes the determinist condition of the process, problems of dead lock between activities, serialization, etc.

Following the graph theory of Workflow Metamodel, we know that an output container is materialized in the moment in which the activity completes its work. Adding this to the fact of the sequence activity completes when the final activity enclosed finish, the best option to solve the problem is force the programmer to pass out the information through the output container of the structured activity (see Figure 14).

But there are some situations in which following this option can be problematic. If we bring this situation to the limit, it means if we imagine a "sequence" activity with one hundred enclosed activities, and a very long processing time (for example one year). If we must wait until all the activities are complete to communicate the information, the rest of the activities which are waiting for it could not work until it finish.

We explained the most dangerous situations that we can meet following both options, but going inside BPEL real processes we find that these are not common situations. For this reason we can relax our opinion and maintain our first decision of let the programmer the responsibility of chooses between both possibilities.

After all, in normal cases and for achieving a good programming style, clearer and more solid, we suggest the programmer take more in account the first option and use the output container to pass the information out of the sequence activity.

As a conclusion and in coherence with our research we will provide this activity with zero or more input containers and zero or more output containers. The sequence activity BPEL-D syntax is shown in Listing 44.

```
<sequence standard-attributes>
          standard-elements
          <containers>
            <container name="NCName" type="input" />*
            <container name="NCName" type="output" />*
          </containers>
          activity+
</sequence>
```

Listing 44: BPEL-D Sequence activity with containers

3.3.2.6. The flow activity.

"The flow construct provides concurrency and synchronization" [BPEL11]. This activity completes after that all the enclosed activities have finished. The "flow" provides a construct to create synchronization dependencies: the construct called "link" (see Listing 45).

```
<flow standard-attributes>
  standard-elements
  <links>?
    <link name="ncname">+
  </links>
  activity+
</flow>
```

Listing 45: BPEL Flow activity

As the "sequence" activity, this construct lets us organize a subset of activities but this time they can be run in parallel. These common points appoint to a similar treat as with "sequence", but the fact of concurrency need to clear some questions about that.

If we search about concurrency and parallelism in the metamodel explained in [LR00] and introduced in the section 2.4, we find the following: "An activity A can only produce data that is expected as input by another activity B if A runs before B… In particular, an activity cannot expect input from an activity that can run in parallel".

The reason for it is that again the main properties of the metamodel data connectors. This time is the second condition: the target of a datalink must be reachable from the source of the datalink, or formally.

Let A ϵ N be an activity, and let B ϵ N U C be an activity or predicate:

$$\Delta(A_1, A_2) \neq \phi \Rightarrow A_2 \text{ is reachable from A1.}$$

But it has no more consequence than that we must take care about the data links that we define between the activities into the structured "flow". It means that we can write data links between the activities of a flow if they have a path between both activities.

Apart from that we are in a similar situation like in the case of sequence. It means that we will have similar problems of coherence, process integrity, synchronization, etc. in the case of passing data directly to outside activities when the structured "flow" fails. But we have too the problem of the very long waiting for part of the activities which need information from a very big "flow".

For this reason we make the same recommendations of the previous case and allow the programmer to of choose which solution is more appropriate for his case, but suggesting the usage of output containers. As we can see in Listing 46 the cardinality of containers will be exactly the same as in the previous cases.

```
<flow standard-attributes>
      standard-elements
      <links>
        <link name="NCName"/>+
      </links>
       <containers>
            <container name="NCName" type="input" />*
            <container name="NCName" type="output" />*
      </containers>
      activity+
</flow>
```

Listing 46: BPEL-D Flow activity with containers

3.3.2.7. The scope activity.

"The scope activity provides a behaviour context for a primary activity, which normally will be a structured activity with many nested activities" [BPEL11]. The context provided by the scope can include fault, event and compensation handlers, data variables and correlation sets.

This fact will make this construct quite complex and for this reason this section will be too one of the most complex and interesting. The fact of substituting the use of variables for input and output containers will not require more complexity than in the previous activities (same cardinality zero or more of both types). Following them the scope syntax is shown in Listing 47.

```
<scope variableAccessSerializable="yes|no"
   standard-attributes>
   standard-elements
   <containers>
        <container name="NCName" type="input" />*
        <container name="NCName" type="output" />*
   </containers>
   ...
   activity
</scope>
```

Listing 47: BPEL-D Scope with containers

But the main questions here are the fault handling, and the use of compensation handlers. We will treat both questions separately.

Fault handling

The fault handling in BPEL is treated as "reverse work" which undoes the partial and unsuccessful work of a scope in which a fault has occurred.

The construct that let us correct the failed situations is the "catch". Each of these activities is prepared to detect a specific kind of fault. The role of the variables here comes from the need of passing information associated with the fault to the activities inside the "catch".

In Listing 48 we can see that each <catch> element has an optional attribute called "faultVariable". That is what we must substitute for the use of data links and containers.

```
<faultHandlers>?
  <catch faultName="qname"? faultVariable="ncname"?>*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
</faultHandlers>
```

Listing 48: BPEL Fault Handler

Then the first step is providing it with an optional container in the place of the main variable. The input data could pass to it from the activity that faulted. Once the activity faults, the data with the fault is copied into the container of the fault handler, by the runtime.

The special nature of the faults makes it impossible to write specific data links as we are doing in BPEL-D. In this case the incoming data link to the container is implicit because it can not be determined until the moment in that the fault is thrown at runtime.

Taking into account that the way of receiving information is not the same as we find before we must use the same type of container created for faulting situations: *faultData* container.

In this case we cannot pass information to the activities enclosed in the structured construct because of the special nature we talked about. In the "catch" no data links are allowed when their sources are outside the fault handler and their targets inside it. Then, and as a difference with the rest of BPEL-D structured activities, we will need one container as much for this element. Figure 15 illustrates an example of the behaviour of this construct.

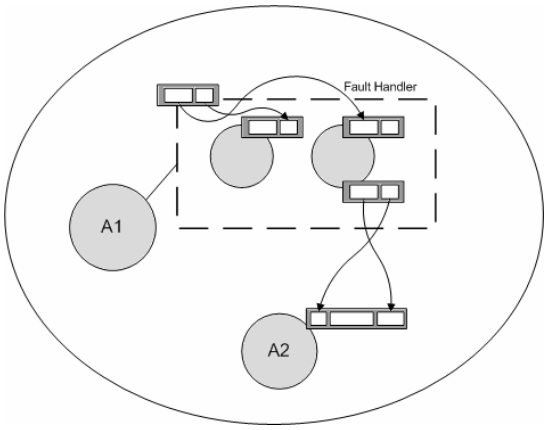

Figure 15: BPEL-D Fault Handler representation

On the other hand, data links are allowed to leave a fault handler by having their sources at an activity in the fault handler but their target outside. Here we are in same situation as in the "pick" or the "switch", and we have the same answer too. The use of output containers for this activity is just a question of style and we let using them or not to the programmer decision.

After clarify this question, we can see in Listing 49 the syntax of the fault handlers declaration section into the scope.

```
<faultHandlers>?
     <catch faultName="qname"?>*
         <containers>
             <container name="NCName" type="faultData"/>?
             <container name="NCName" type="output"/>*
         </containers>
         activity
     </catch>
     <catchAll>?
         activity
     </catchAll>
</faultHandlers>
...
```

Listing 49: BPEL-D Fault Handler with containers

Compensation Handlers

As we said in the introduction, BPEL processes can spend a long time computing. In this time some part of the process can fail or be cancelled after many ACID transactions have been committed during its progress. This partial work done must be undone as best as possible and for this reason some kind of mechanism is necessary to restore the situation when the process or one of the nested scopes fails. This mechanism is the "compensation handlers".

"A compensation handler in BPEL is simply a wrapper for a compensation activity. It will need to receive in many scenarios data about the current state of the world and return data regarding the results of the compensation" [BPEL11].

```
<compensationHandler>?
  activity
</compensationHandler>
```

Listing 50: BPEL Compensation Handler

As we can see from the definition of this construct, a "compensation handler" will need to receive some information incoming from the process. In BPEL 1.1 a compensation handler takes this information directly from variables which would have in each moment the necessary information.

Now that we cannot use variables, we will take this information from the activities which produce it. For this reason we must provide the compensation handler with an indeterminate number of input containers (zero or more) according with the enclosed activities needs. Figure 16 illustrates the semantics of compensation handlers data links.

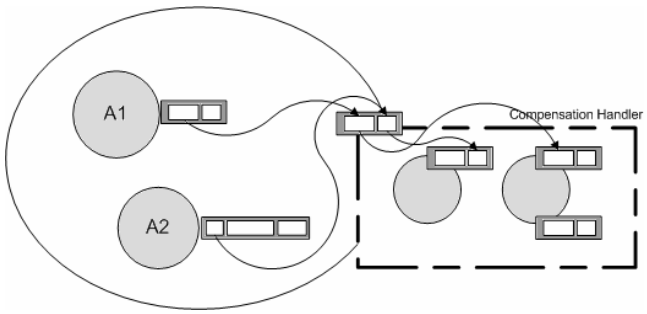

Figure 16: BPEL-D Compensation Handler representation

On the other hand, attending to outgoing information, a compensation handler in BPEL 1.1 is not allowed to write data which the rest of the process can see. For this reason no data links can have their source into the compensation handler and their target outside it because. It means that in this case we will not need output containers for this activity

(see [KL07] for more information in agreement with these conclusions). The final syntax is shown in Listing 51.

```
...
<compensationHandler>?
     <containers>
          <container name="NCName" type="input"/>*
     </containers>
     activity
</compensationHandler>
...
```

Listing 51: BPEL-D Compensation Handler with containers

3.4. Summary

For the BPEL-D syntax activities, we have followed WSFL as a language reference and the workflow theory of [LR00] as a theoretical model. The result of this process includes the creation of three new constructs: (1) containers, element responsible for storing information, (2) datalinks, responsible for connecting data dependent activities and (3) maps, responsible for describing the container mapping between activities. Depending on its nature, the container type can be: *input*, *output*, *toPartner*, *fromPartner* and *faultData*.

Depending on its semantics, each BPEL-D activity (and some constructs too) is provided with a determined number of containers of the different types, substituting the use of variables.

Part 2

Implementation of Eclipse BPEL-D Designer

CHAPTER 4

IMPLEMENTATION BACKGROUND

This section introduces a brief description about the implementation environment and the essential basic for its understanding. First an introduction the Eclipse Development Platform is given. Afterwards it introduces a little explanation about the Eclipse Modelling Framework and the Graphical Editing Framework tools. To conclude this chapter some issues about the Eclipse BPEL Project are explained.

4.1. Eclipse Development Platform.

Eclipse is an Open Source software framework based in Java. The main role of this framework is to provide tool creators with mechanisms and rules that lead to seamlessly-integrated tools. To expose this mechanism, it offers the use of API interfaces, classes and methods.

The basic requirements which Eclipse offers are introduced in [Eclipse]:

- Support the construction of a variety of tools for application development.
- Support an unrestricted set of tool providers.
- Support tools to manipulate arbitrary content types.
- Facilitate integration of tools with different content types and tool providers.
- Support both GUI and non-GUI-based application development environments.
- Run on a wide range of operating systems, including Windows and Linux.
- Capitalize on the popularity of the Java programming language for writing tools.

Eclipse has no specific knowledge about a concrete domain. This does not limit it to the development of Java language applications as it offers the possibility of plug-in use with support for other programming languages as C++. The Eclipse SDK includes the Eclipse Platform, a Java Development Tool (JDT) and the Plug-in Development Environment (PDE)

The basis for Eclipse is the Rich Client Platform (RCP). The major components which constitute the Eclipse RCP are shown in the next picture. All of these components are necessary for the GUI version of the Eclipse Platform.

The main point of this platform is the possibility of extending its capabilities by installing plug-ins written for the Eclipse software framework. A plug-in is the smallest unit of function that can be developed and delivered separately. Usually a small tool is written as a single plug-in, whereas a complex tool has its functional split across several plug-ins.

When the Platform is launched, the user is presented with an integrated development environment (IDE) composed of the set of available plug-ins. Except for a small kernel known as the Platform Runtime all of the Eclipse Platform's functionality is located in plug-ins. Some of these plug-ins conform to the Eclipse Modelling Framework and the Graphical Editing Framework.

4.2. Eclipse Modelling Framework (EMF).

4.2.1. Introduction

Originally based in MOF (Meta Object Facility), Eclipse Modelling Framework (EMF): “is a modelling framework and code generation facility for building tools and other applications based on a structured data model” [EMF]. The Java Framework provides a code generation facility in order to keep the focus on the model itself and not on its implementation details. The key concepts underlying the framework are: meta-data, code generation, and default serialization.

The main advantage of this Framework is the low cost entry of this model. When we talk about modelling, we think about things like Class Diagrams, Collaboration Diagrams, State Diagrams and so on, combined to complete the model specification. EMF requires just a small subset of the things you can model in UML, specifically simple definitions of the classes and their attributes and relations. Therefore the graphical modeling capabilities of an EMF based tool are limited to this subset.

Once you specify an EMF model, the EMF generator is used to generate the corresponding set of Java implementation classes. After it is possible to edit these

classes to add methods and instance variables, having still the possibility of regenerate them from the EMF model and giving the option of preserve the user additions during the regeneration.

EMF uses XMI (XML Metadata Interchange [XMI]) as its canonical form of a model definition. For this reason it can offer several ways of getting your model into that form:

- Create the XMI document directly, using an XML or text editor
- Export the XMI document from a modeling tool such as Rational Rose
- Annotate Java interfaces with model properties
- Use XML Schema to describe the form of a serialization of the model

In addition EMF provides several other benefits like model change notification, persistence support including default XMI and schema-based XML serialization, a framework for model validation, and a very efficient reflective API for manipulating EMF objects generically. For more see [EMF] and [RedBook].

4.2.2. Architecture

EMF is based of the following three parts:

EMF Ecore: the core includes a meta model for describing models which provides basic generation and runtime support to create Java implementation classes for a model. It provides Java interfaces and implementation classes for all the classes in the model, a factory and package (meta data) implementation class, and even adapter implementation classes (called ItemProviders) that adapt the model classes for editing and display.

EMF.Edit: includes generic reusable classes for building editors for EMF models, easing the task of adding user interface to the model. For that it provides:

- Content and label provider classes, property source support, and other convenience classes that allow EMF models to be displayed using standard desktop (JFace) viewers and property sheets.
- A command framework, including a set of generic command implementation classes for building editors that support fully automatic undo and redo.

EMF.Codegen: the EMF code generation facility is capable of generating everything needed to build a complete editor for an EMF model. It includes a GUI from which generation options can be specified, and generators can be invoked. The generation facility leverages the JDT (Java Development Tooling) component of Eclipse.

4.3. Graphical Editing Framework (GEF).

Graphical Editing Framework (GEF) is a framework that was developed for the Eclipse platform. In spite of having a very steep learning curve it offers great benefits to for GUI development.

GEF provides the groundwork to build a greater part of a software application, including: activity diagrams, GUI builders, class diagram editors, state machines, etc.

This framework includes the following components:

- The org.eclipse.draw2d plug-in provides a layout and rendering toolkit for displaying graphics. It is used for the View components.
- Requests/Commands to be used to edit the model.
- Palette of Tools that are offered to the user.

GEF employs an MVC (model-view-controller see [GHJ+05]) architecture which enables simple changes to be applied to the model from the view. In addition to MVC, the most recurring design patterns under GEF are:

- Factory: Creating models from palette, editparts and Figures
- Observer: Typically a controller (read editpart) listening on Model and View.
- Chain of Responsibility: To decide which EditPolicy should handle a Request.
- State pattern: A typical example of which is opening editors for an input model.

For more information about this issue see [Redbook] and [GHJ+05].

4.4. The Eclipse BPEL Project.

The Eclipse BPEL Designer Editor was created as an open source project under the Eclipse Technology Project [EclipseBPEL] in May of 2005, covered mainly by IBM and Oracle Corporation. It was around that time that BPEL became a key component of the SOA stack, finding a great acceptance into the community of users.

"The goal of the BPEL Project was to add comprehensive support to Eclipse for the definition, authoring, editing, deploying, testing and debugging of BPEL processes [EclipseBPEL]". This project will provide broad support for integrating BPEL4WS design time experience with the Eclipse platform.

The key pieces of functionality to be provided were:

- ***Designer:*** GEF-based editor to provide a graphical means to design BPEL processes.
- ***Model:*** an EMF model that represents the WS-BPEL 2.0 specification.
- ***Validation:*** validator which operates with the model and produces errors and warnings based on the specification.
- ***Runtime Framework:*** extensible framework which to allow deployments and execution of BPEL processes from the tools into a BPEL engine.
- ***Debug:*** A framework which will allow the user to step through the execution of a process, including support for breakpoints.

One of the main points of the implementation was to achieve being extensible to third-party vendors in a number of ways, to support new activity types, property pages for extensibility of existing constructs, an extensible palette, and product-specific branding capabilities. For this reason the idea was to use the standard Eclipse mechanism of extension points to allow 3rd party integration of various such elements

The main components of the Eclipse BPEL project are the visual and source design of the flow and interpretation of the XML schema model used as definition for BPEL variables and messages. The project focuses on the design time aspect of BPEL which includes construction, definition, inspection, and documentation. The runtime aspect of BPEL is typically vendor specific and includes such parts as validation, compilation, deployment, debugging, versioning, and migration.

The Project Principles introduced in [EclipseBPEL] were:

- Leverage the Eclipse Ecosystem; to extend the Eclipse user experience as well as leverage other Eclipse project functionality where applicable.
- Vendor Neutrality; to build the tool that so that it is not biased toward any particular BPEL runtime.
- Extensibility; to provide a set of extension points to allow for the binding the runtime time environment and to allow extensibility of component palettes.
- Incremental Development; an agile development process will be employed to manage and respond to changing requirements.

Chapter 5

Eclipse Designer Overview

In this chapter, the thesis goes deep into the Eclipse BPEL Designer, focusing into the architecture, the most important modules and the essential parts to understanding the functionality of the system described using UML (see [BRJ05] and [UML]). This task is essential to find the points to treat in the construction of the BPEL-D Designer.

As has been said, the Designer is not documented. This fact complicates considerably the task of implementing the BPEL-D Designer. All the documentation presented through this chapter is result of our research. It can be taken as staring point for a complete documentation that facilitates the work of future designers.

Apart from that, this chapter does not aim to present complete system documentation, but to focus in the parts in direct relation with our research. That is, to describe the aspects affected for the management of the containers and datalinks of the BPEL-D syntax.

In relation with the UML diagrams used to describe the system, we adopt the following convention for achieving more clarity and an adequate extension. In Chapter 5 diagram's, most of classes methods are not shown because they follow the same schema of that ones shown in Chapter 6. In Chapter 6 only related methods are shown in diagrams. If methods are duplicated in classes we show the schema only once in a class.

5.1. High level vision.

Based in EMF and GEF (see [Redbook]), the implementation of the system follows an architecture based on the patterns MVC and Layers (see [GHJ+05]).

The BPEL Designer implementation is composed of five different subsystems, each one implements a different functionality element. Next, it introduces a brief description concerning about them:

- ***org.eclipse.bpel.common.model*:** general extension model framework for xmi-serialized extension models which live in a separate file (in our case, our ***.bpelex*** file) from the main file yet which is semantically linked to the model elements.

- ***org.eclipse.bpel.common.ui*:** provides the Graphical User Interface elements to be extended into org.eclipse.bpel.ui.

- ***org.eclipse.bpel.model*:** implements the EMF model core of the system, including the bpel.ecore and bpel.genmodel files. Includes the BPEL specification elements modelization as well as proxies and factories.

- ***org.eclipse.bpel.ui*:** supports the main Graphical User Interface elements of the BPEL Designer. It is based in EMF and GEF.

- ***org.eclipse.bpel.runtimes*:** extensible framework which to support deployment and execution of BPEL processes from the tools into a BPEL engine.

We focus on the most important systems to study relating to our research: ***org.eclipse.bpel.model*** and ***org.eclipse.bpel.ui.*** For this reason the next sections are focused in the description of both subsystems.

5.2. Plug-in org.eclipse.bpel.model

As we have already explained, this plug-in supports the EMF model that represents the WS-BPEL 2.0 specification [BPEL20]. As an EMF model, the system definition is described in the bpel.ecore responsible for the code generation. Figure 17 shows the package general diagram referring to this plug-in.

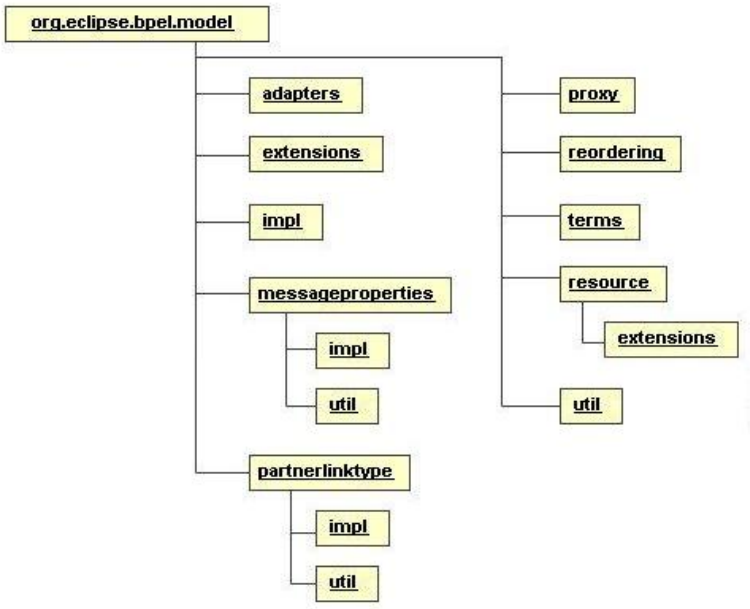

Figure 17: org.eclipse.bpel.model UML Package Diagram

The EPackage *model* contained in the ***bpel.ecore*** file generates three Java packages: these packages are ***org.eclipse.bpel.model***, ***org.eclipse.bpel.model.impl*** and ***org.eclipse.bpel.model.util***.

Each EClass in the EPackage generates two elements, an interface in the base package, and a Java class that implements it in the implementation package. In the case that the EClass inherits from another EClass, the generated interface and implementation extend the respective interface and implementation classes of their super types.

The Package ***org.eclipse.bpel.model*** provides the model classes' interfaces as well as an object factory and package access class. All the model classes inherit from the Interface *ExtensibleElement* which assumes the role of root element. These classes include *Process*, *Variables, PartnerLink*, *Link*, *Copy*, etc. At the same time each BPEL activity is represented by an interface which inherits from the interface *Activity*. The UML diagram Figure 18 illustrates this package.

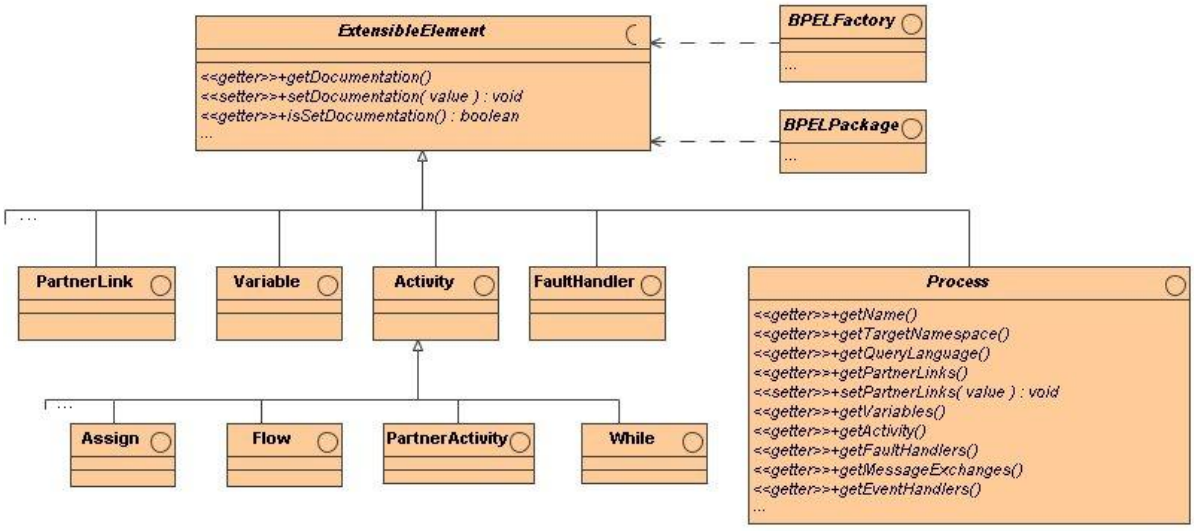

Figure 18: Model objects UML Classes diagram

This package contains many more classes inheriting from *ExtensibleElement* and from *Activity*. It shows just a reduced number because the target is showing the package organization and showing all of them would just add clutter. For the same reason not all class methods or attributes are shown either. We will do the same with the rest of the UML diagrams.

Back to the package, the interface *BPELFactory* provides the creation methods for each non-abstract class of the model, while the *BPELPackage* contains methods to access to the meta objects represented in the model. All these classes are directly generated by the ***bpel.ecore*** file introduced before.

As has been said all the interfaces contained in this package have their implementation classes into the package ***org.eclipse.bpel.model.impl***. These implementations contain the attributes and the getter and setter methods. In the case of non-volatile features a field to cache the value is also generated by the EMF. Figure 19 shows a summary of it.

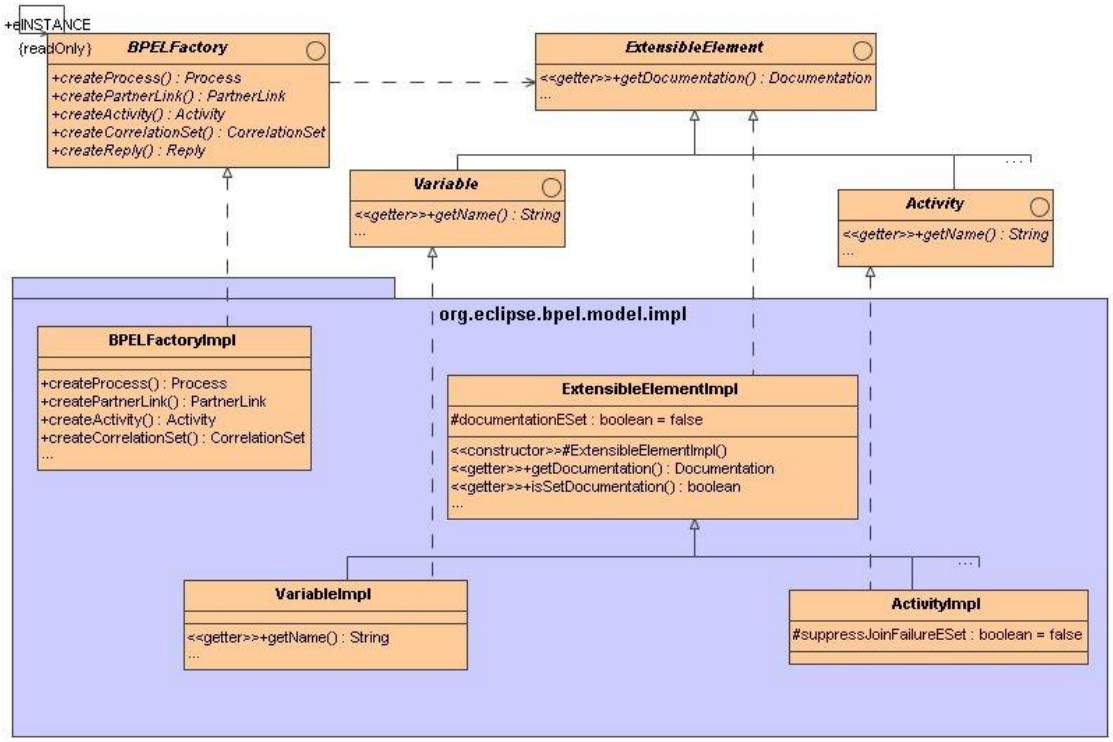

Figure 19: Model objects UML Classes Diagram (2)

The package ***org.eclipse.bpel.model.util***, use to be optional, and its presence will depend on the code generation properties. In our situation, there are some quite important elements which require to be mentioned.

- *BPELAdapterFactory*: support the creation of adapters for the model objects.
- *BPELUtils*: contains helping methods to manage the model objects an offer them as a help to superior level classes.
- *BPELConstants*: contains the declaration of constants used by the model objects of the plug-in.

If we come back to the main package, we can find two more ****.ecore*** files apart from the bpel.ecore which brings the BPEL specification into an oriented object model. These two files are ***messageproperties.ecore*** and ***partnerlinktype.ecore***. Each one contains an EPackage messageproperties and partnerlinktype respectively.

Both of them provide an object structure for the WSDL Properties and PartnerLinkTypes in a similar way as the ***bpel.ecore*** does with the model classes. The respective ****.genmodel*** files are directly linked with the bpel.genmodel. In a similar way as happens with the EPackage model, three Java packages will be generated by each one of the EPackages. The model generation will produce the following packages:

- ***org.eclipse.bpel.model.messageproperties, *.impl, *.util.***
- ***org.eclipse.bpel.model.partnerlinktype, *.impl, *util.***

One of the most interesting packages of this plug-in and one of the most important is the ***org.eclipse.bpel.model.resource***. The important classes that stand out in relation with our research are *BPELReader* and *BPELWriter*.

BPELReader (see Figure 20) carries out the role of parsing the ****.bpel*** file. This class is the one responsible for translating each xml element of the file into a Java object of the model. For this task, this class implements a different method to translate each BPEL element into an EMF model object. These methods will follow the form ***xmlToxxx***, ***xxx*** being the name of the BPEL element (for example xmlToInvoke or xmlToVariable). When one of these methods recognizes a BPEL element it uses the *BPELFactoryImpl* to create the new model object and a DOMParser to parse the files.

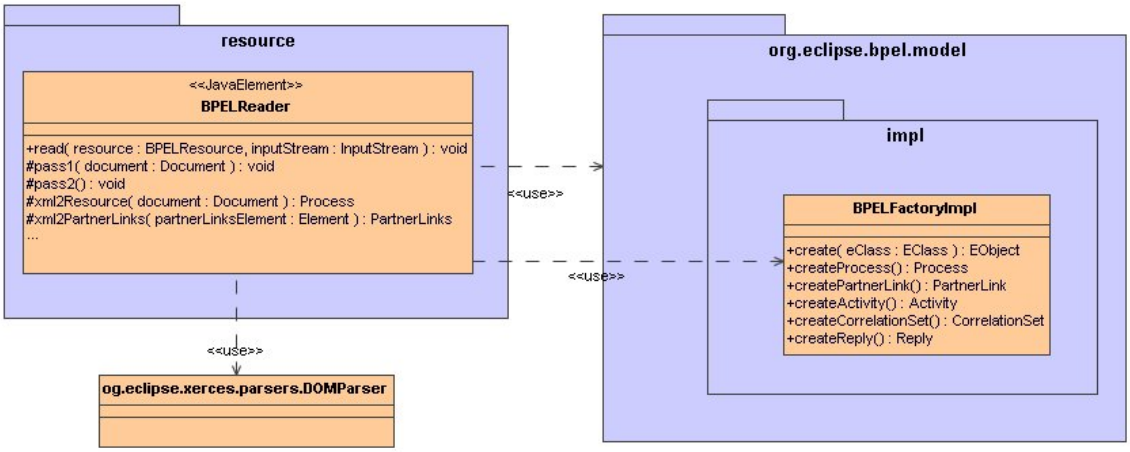

Figure 20: BPELReader UML Classes Diagram

In the opposite direction *BPELWriter* is the one responsible for serializing the BPEL model into the correspondent ****.bpel*** file. Both processes are explained in detail in section 5.2.1.

The rest of classes of the plug-in are not so much important in our research. For this reason we just pretend to present a brief introduction of the functionality of the correspondent packages into the subsystem:

- ***org.eclipse.bpel.model.adapters:*** contains an Adapter hierarchy which will be extended in the user interface plug-in.
- ***org.eclipse.bpel.model.extensions****:* provides serialization and des serialization of extension elements into a DOM.
- ***org.eclipse.bpel.model.reordering*** and ****.extensions****:* supports the management of model extensibility elements lists.
- ***org.eclipse.bpel.model.terms:*** contains the class *BPELTerms*, which is the main plug-in class to be used in the desktop.

5.2.1 Parsing and writing BPEL files

The processing of a BPEL process starts with parsing the ****.bpel*** file during the deployment process. The *BPELReader* recognize the XML elements found in the file and creates the corresponding definition objects. That happens with every activity or element, and for each of those such a definition object will exist. As we explained, the *BPELReader* will use for this task the creation methods provided by the *BPELFactoryImpl*, which will return the required object in each case. Figure 21 illustrates this task.

Since BPEL syntax is XML, this class uses a *DOMParser* object to create the definition objects. DOM, as is generally known, implements a tree-based representation of an XML document. Then each instance will contain a reference to its contained elements in a recursive way until the complete set of the represented objects have been produced.

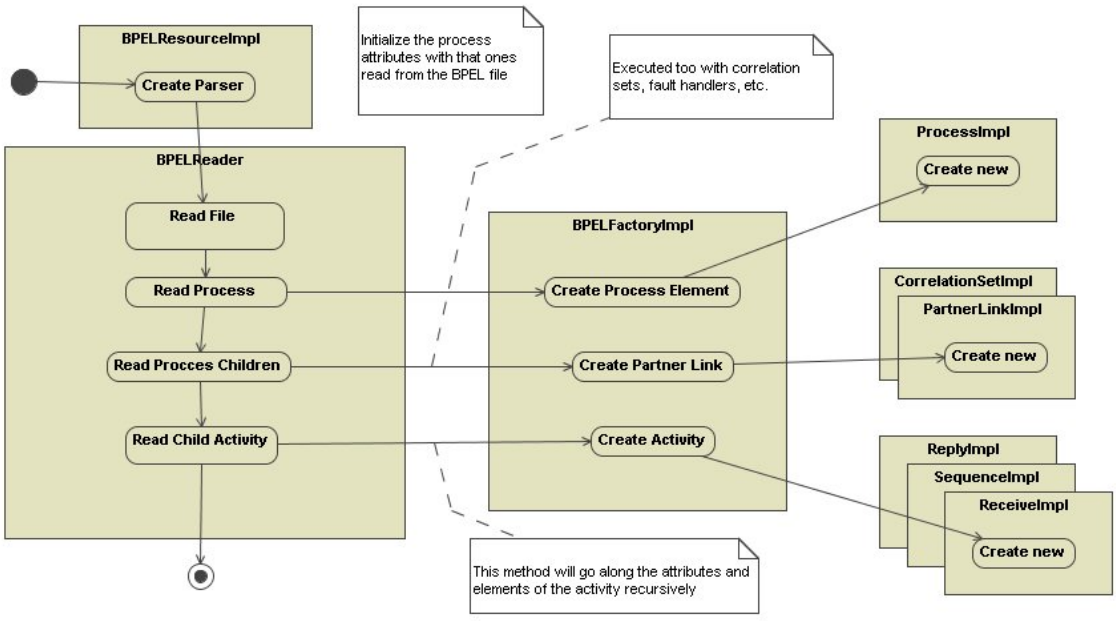

Figure 21: Parsing process UML Activity Diagram

After this process, each XML element will have its own implementation object. For activities, an implementation objects contains and handles instance-specific activity information like the state of an activity, a reference to the definition object of an activity, a reference to the process implementation object, fault information etc. From an implementation point of view it is a hierarchy of classes.

In the moment that we want to write down the object model into a BPEL file, the opposite process takes place. In this case, the *BPELWriter* class (see Figure 22) is responsible for this task.

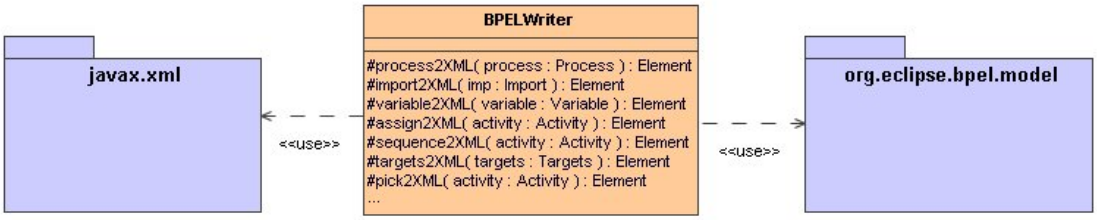

Figure 22: BPELWriter UML Classes Diagram

As has been said the whole process is referenced as a containing object tree, whose root element is the process element. The writing task is the following. The parser will read the process child elements and translate them into the BPEL file as XML elements (see Figure 23). For each process child the parser will repeat the task in a recursive way until it has covered all the model elements contained in the process.

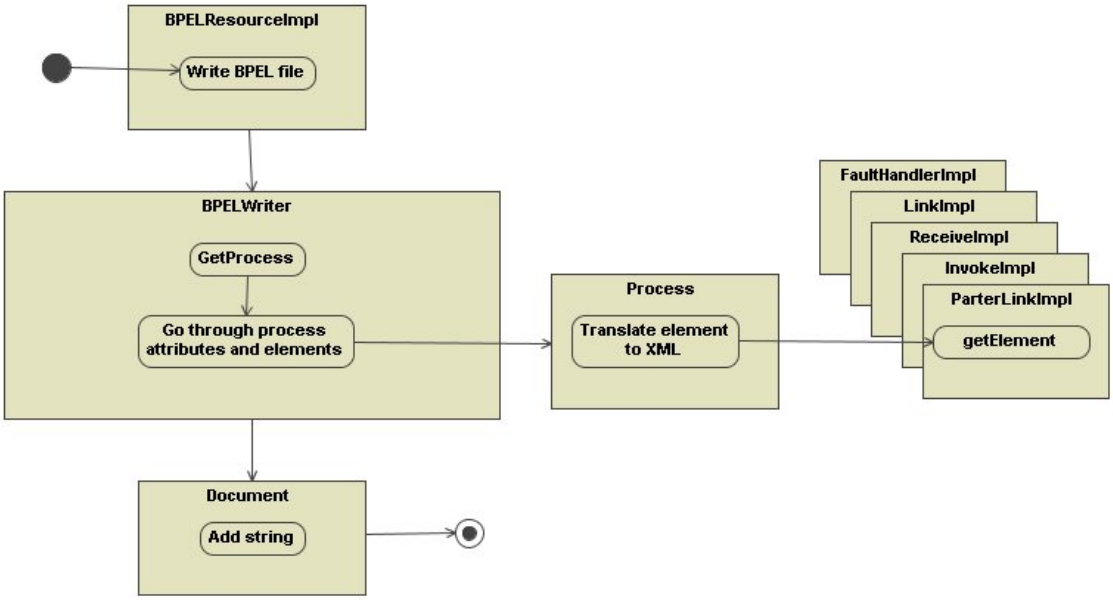

Figure 23: Writing process UML Activity Diagram

5.3. Plug-in org.eclipse.bpel.ui

As has been said, this plug-in is based in MEF and GEF which allows one to easily develop graphical representations for a concrete model. It supports the main part of the Controller and View layers. Figure 24 shows the plug-in package diagram.

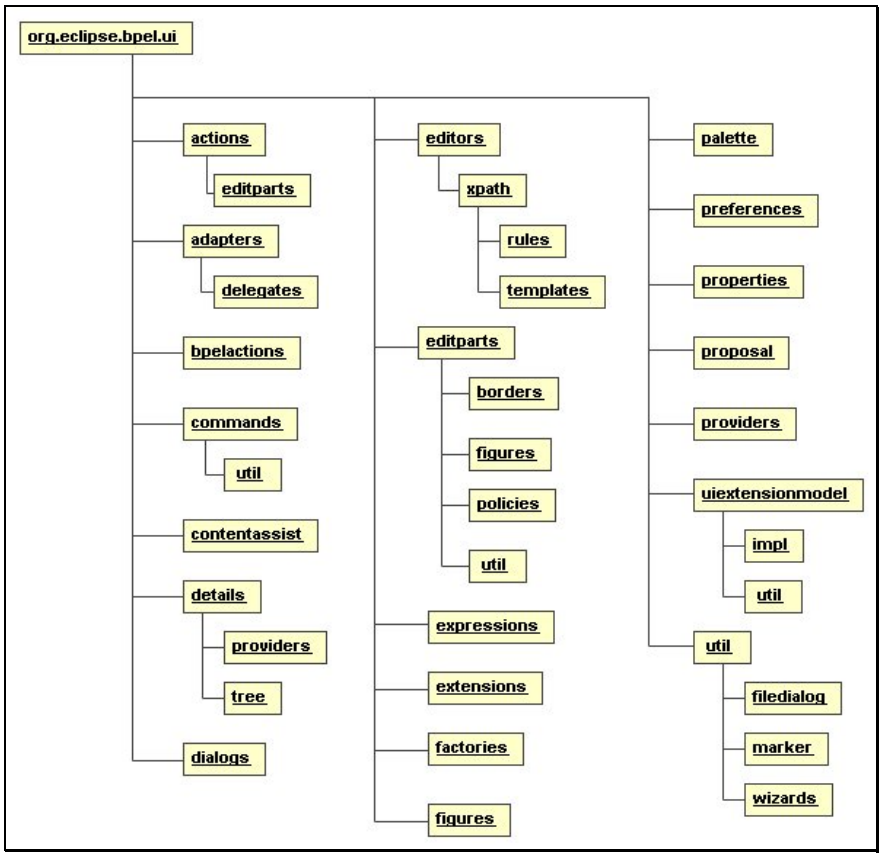

Figure 24: org.eclipse.bpel.ui UML Package Diagram

Next, we introduce a brief description of the packages that are most important in relation to the development of the BPEL-D Designer:

- ***org.eclipse.bpel.ui*:** contains the basic classes of the GUI, of which the most important are:
    - *BPELEditor:* The main class of the plug-in (see Figure 25). It is the one responsible for the Editor creation, including creating the options palette, the general, tray and outline views, calling the parser, instantiating and managing the actions, managing the changes in the file, etc.
    - *ModelListenerAdapter:* observer responsible for notifying the changes between view and model. For the special nature of flow links, they have their own observer the *LinkNotificationAdapter*.

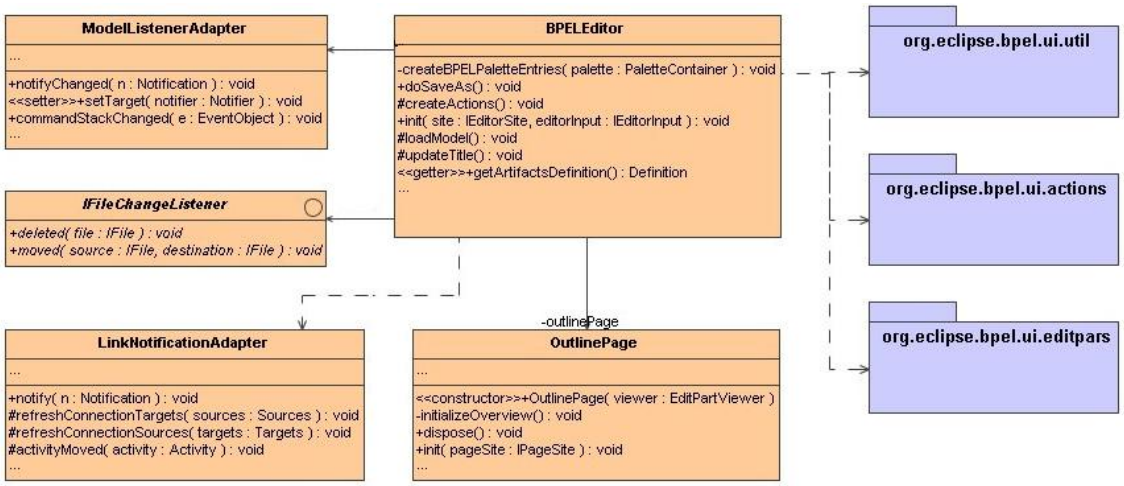

Figure 25: BPELEditor UML Classes Diagram

- *BPELResourceChangeListener:* listens to changes to the BPEL file and reacts accordingly.
- *BPELUIPlugin*: manages the plug-in system integration issues.
- *IBPELUIConstants and Messages:* factors out the UI constants referring to icons and images and works as a proxy to the language support provided by *messages.properties*.
- *messages.properties:* factors out the labels used in the GUI as easier way to manage them and supporting language internationalization.
- *ProcessContextMenuProvider:* provides the context menu of the GUI that manages the actions invocations.

- ***org.eclipse.bpel.ui.actions*** and ****.editpart***: provides two kinds of action classes: one to support the classic software options (base package) such as copy, paste, print, rename, etc., The other to support the model objects creation as well as management of VPC[1] objects relation with the process activities (editpart package). These actions are instantiated from the Dialog classes and from other actions. Figure 26 illustrates this hierarchy.

[1] VPC: abbreviation of Variables, PartnerLinks and CorrelationSets for simply.

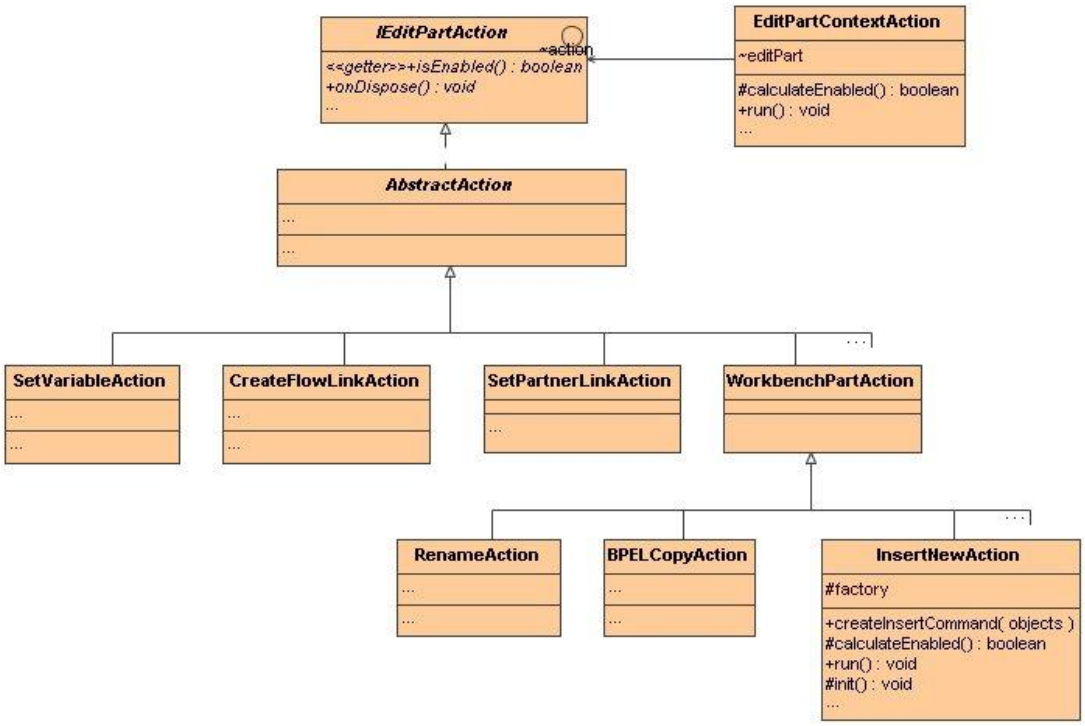

Figure 26: Actions hierarchy UML Classes Diagram

- ***org.eclipse.bpel.ui.adapters*** and ****.delegates***: contain the hierarchy which provides adapters for all the model objects including VPC objects and activities. Their function is to adapt each model object with its correspondent graphical representation or editpart. Figure 27 illustrates the adapter hierarchy.

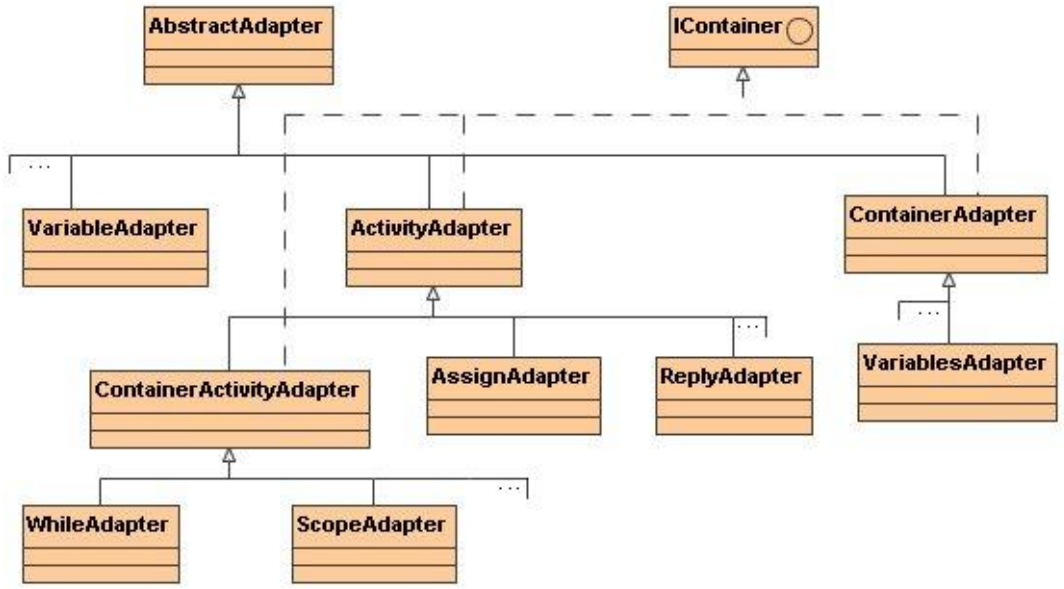

Figure 27: Adapters hierarchy UML Classes Diagram

- ***org.eclipse.bpel.ui.commands*** and ****.util***: implement the pattern command (for more see [GHJ+05]), and provides the classes that actually act with the model to create and manage the VPC objects (creation, elimination, setting, etc.). This includes, providing execution limitations, undo and redo, combining and chaining actions, etc. The "doExecute()" method is responsible of test the possibility of let the command execution or not. Figure 28 illustrates the commands.

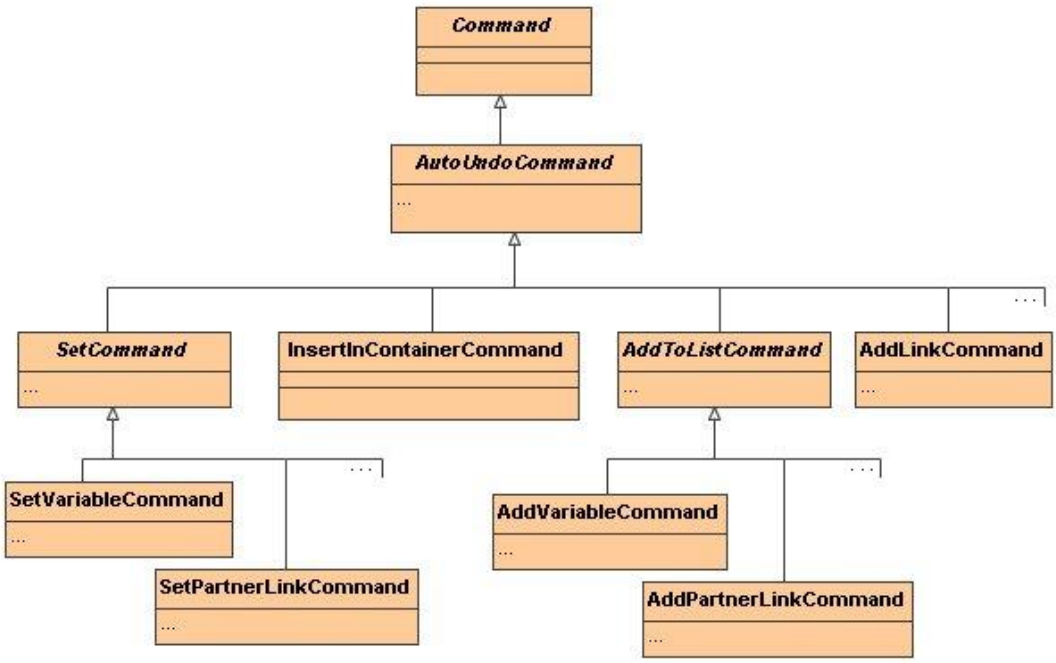

Figure 28: Commands hierarchy UML Classes Diagram

- ***org.eclipse.bpel.ui.detail.providers***: contains content providers for the EMF objects to read them from the Workspace, and list filters.

- ***org.eclipse.bpel.ui.detail.tree***: supports the tree structure to organize the model objects as an image of the XML file tree.

- ***org.eclipse.bpel.ui.dialogs***: provides the GUI dialog interface to manage the VPC objects; supporting creation, relating to activities, deleting, etc. They are the ones responsible for calling the actions. Figure 29 illustrates these dialogs.

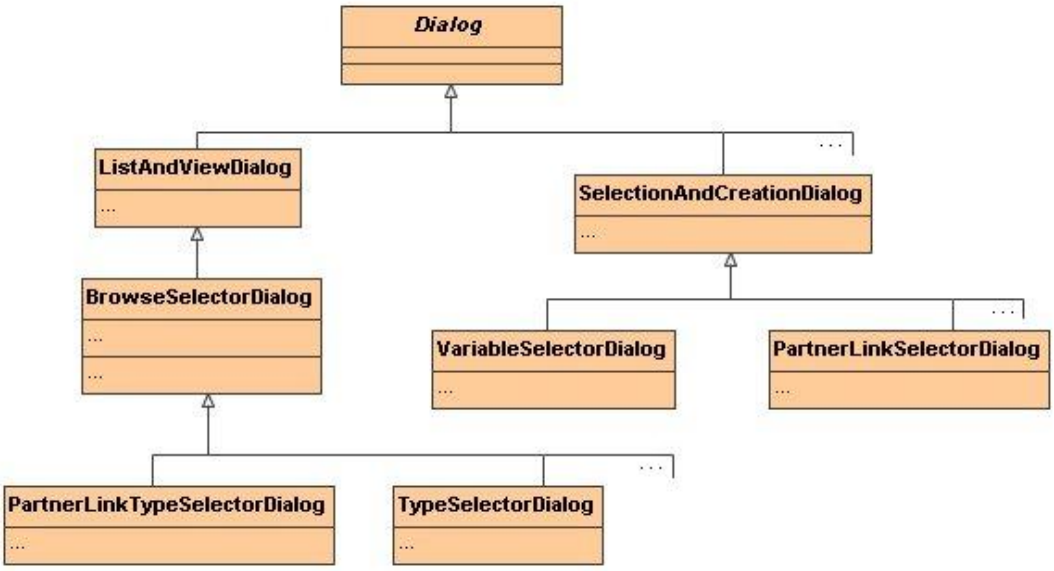

Figure 29: Dialogs hierarchy UML Classes Diagram

- ***org.eclipse.bpel.ui.editparts***: contains the controllers which map the model elements to visual Figures. Figure 30 illustrates the VPC object and activities editparts.

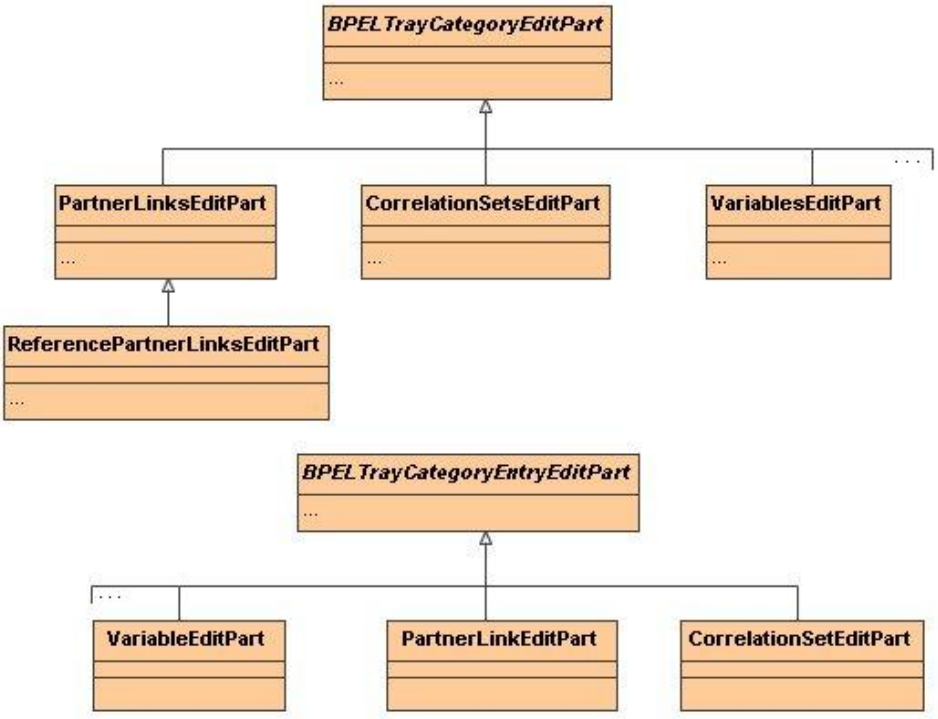

Figure 30: Editparts hierarchy UML Classes Diagram

- ***org.eclipse.bpel.ui.editparts.borders, *.Figures, *.policies*** and ****.util***: contain features provided by Draw2d to draw the model editparts. Special importance is given here to parts related with the Flow Links drawing.

- ***org.eclipse.bpel.ui.factories***: provide the necessary factory to construct the graphical representation of the model objects and associating the correspondent labels, icons, descriptors and more UI information. Figure 31 illustrates the provided factories.

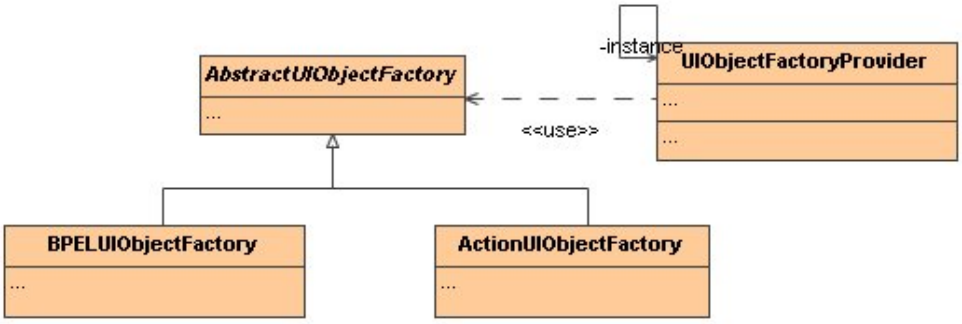

Figure 31: Factories UML Classes Diagram

- ***org.eclipse.bpel.ui.properties***: contains GUI elements which support managing and editing the BPEL model objects

- ***org.eclipse.bpel.ui.proposal.providers***: contains a hierarchy of content providers to provide content to the model objects.

- ***org.eclipse.bpel.ui.util***: provides a wide set of util classes to help the rest of the classes in different issues. It contains different classes such adapters, zoom tools, buffer helper and different utility classes. Some of them require a more concrete explanation:

    - *ModelHelper:* this class provides a common interface to access (i.e. setXX/getXX/isXXAffected) to certain properties which exist across several model object types.

    - *BPELUtil*: is the class in which to place the static helper methods to be used for the BPEL editor.

    - *BPELReader*: Reads a BPEL file and makes it compatible with the BPEL tooling.

5.4. System classes interaction to create VPC elements.

The use of the BPEL Designer can be summarized in creating BPEL specification elements and organizing them into the XML tree.

The BPEL-D Designer we will need the capability to create, manage and delete Containers, Datalinks and Maps. Taking into account that variables, partnerlinks and correlationSets are organized into the process tree in their respective child sections in a similar way, it is logical that the implementation of all them will be quite similar too.

How the Designer manages these elements can be a good example to design how it must manage the new elements to introduce. For this reason it is necessary to understand how the classes explained through sections 5.2 and 5.3 interact. It will be the example to follow for designing the new structures to manage the containers, datalinks and maps. In this case we will follow the process or creating a new variable and linking it to a BPEL activity.

This process is explained with detail along the next lines.

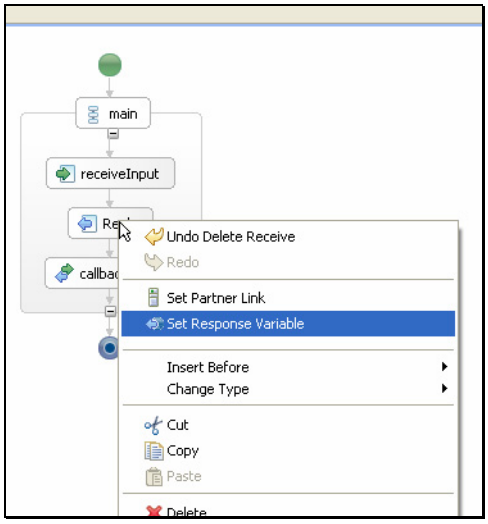

Figure 32: Variable creation from the GUI (1)

1. This process begins when the user clicks right over and activity and the 'Set ResponseVariable' option from the contextual menu (see Figure 32). In this case the execution begins by calling the *ActivityAdapter* which is extended in this case by the *ReplyAdapter* (see next page Figure 34, message (1)). It instantiates the correspondent controller action *SetVariableAction* (message (2)).

2. The action class is responsible for managing the whole variable creation process. At first it must instantiate the correspondent dialog class, *VariableSelectorDialog* (message 3). This class creates a GUI Dialog which shows the user the existing variables.

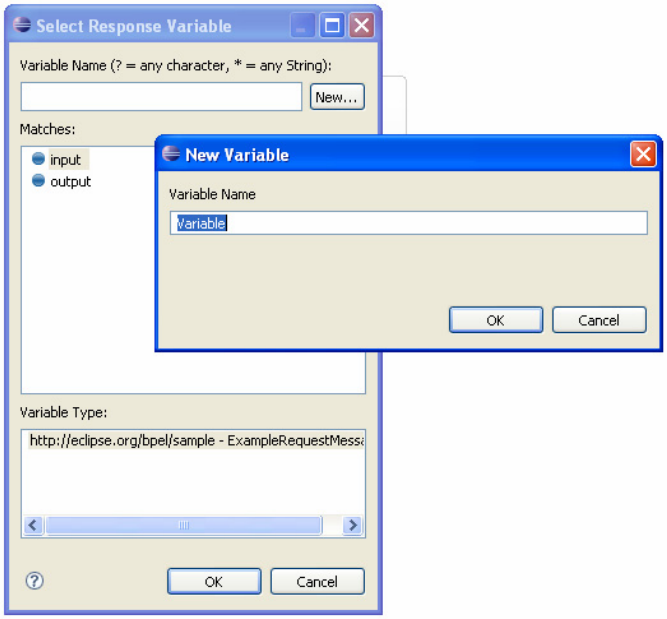

Figure 33: Variable creation from the GUI (2)

3. In the case of selecting one of the existing variables the control turns back to the action class which executes the command *SetVariableCommand*, whose execution links the variable to the activity.

4. In the case of creating a new variable (see Figure 33) the dialog class uses the helper methods provided by *ModelHelper* (who uses the *BPELFactory*) to create the variable representation model object. The factory invokes the constructor of the *VariableImpl* (message (6)).

5. After creating the model object, the dialog prepares a *CommpoundCommand* (message (7)) which is composed of three different commands, *InsertInContainerCmmd*, *SetNameCmmd* (8) and *SetVariableTypeCmmd* (9).

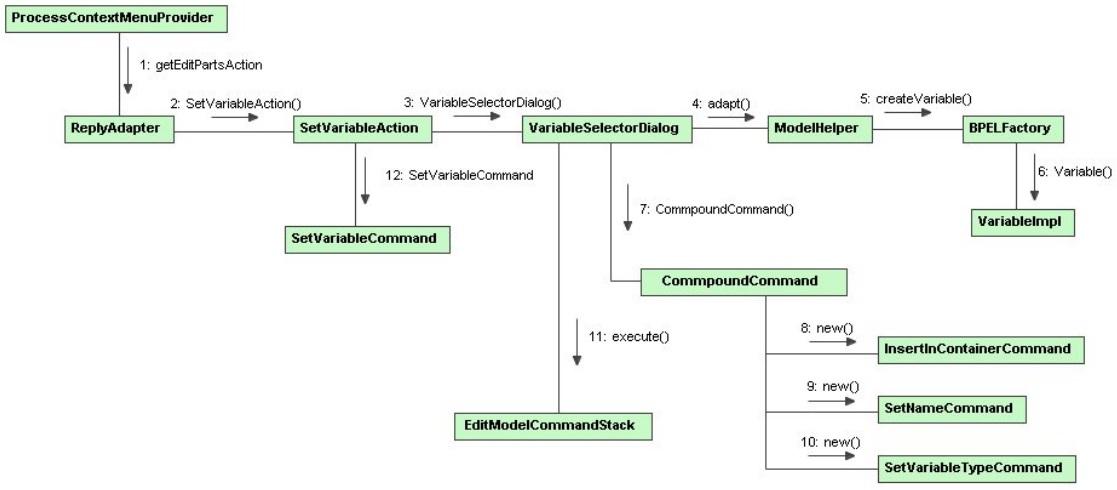

Figure 34: Variable creation UML Communication Diagram (1)

6. *EditModelCommandStack* is responsible for the command execution (Figure 35, message (2). This class will test the different commands of the *CommpoundCommand* by going through the list which contains them. If all the commands can be executed it will ask them to execute; if some cannot, an exception is thrown.

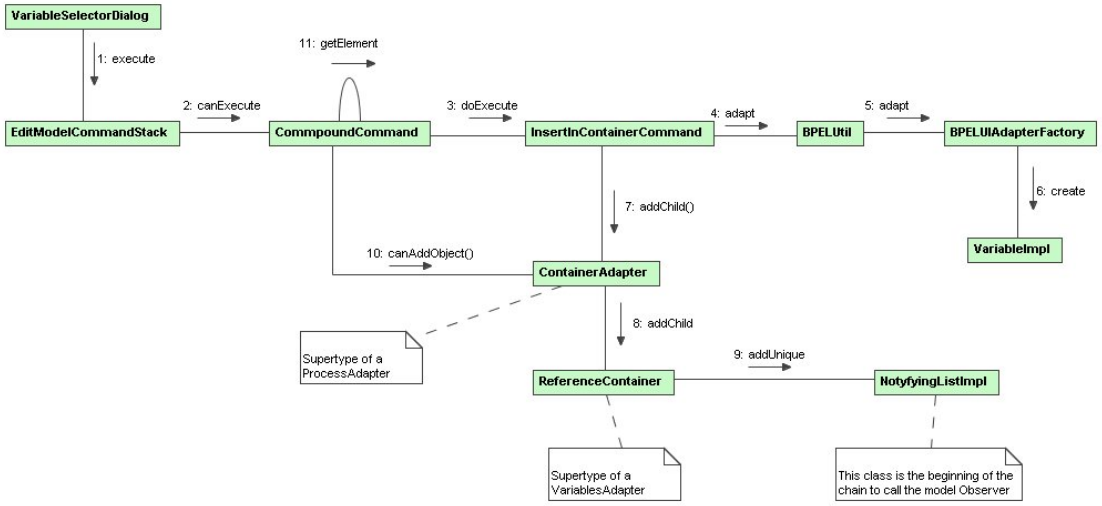

Figure 35: Variable creation UML Communication Diagram (2)

7. *InsertInContainerCommand* is the most important of them and responsible for taking the created object parent element (recall the tree structure) and relating it with the recent created child. It uses *BPELUtil* to provide a graphical representation or editpart when it needs it. For this issue *BPELUtil* will create both, the editpart and the model object.

8. The creation of the graphical representation of the model object is unleashed by the *ReferenceContainer* who uses *NotyfingListImpl* (9). This class uses some classes external to our system to call *ModelListenerAdapter* who will use the helping methods offered by the *ModelHelper* to create the correspondent editpart (by using the opportune factory) and an adapter for it. This process is illustrated in Figure 36.

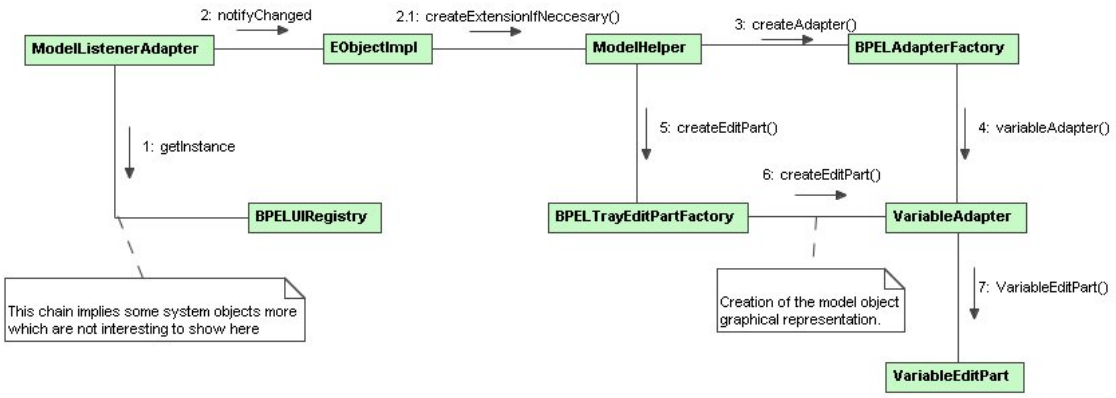

Figure 36: Variable creation UML Communication Diagram (3)

9. After this the control comes back to the action class which executes the *SetVariableCommand* who link the variable to the activity. This is the last issue of the variable creation.

The creation of a partnerlink or a correlation set follows a very similar set of steps.

5.5. System classes interaction to create a flow link.

The creation of a flow link process in the BPEL Designer is very interesting for our research as a possible example to follow in the designing of the datalink creation. For this reason this section presents and explains the classes' interaction in building the link. This process is a little different from the variable creation because of the graphical representation of it.

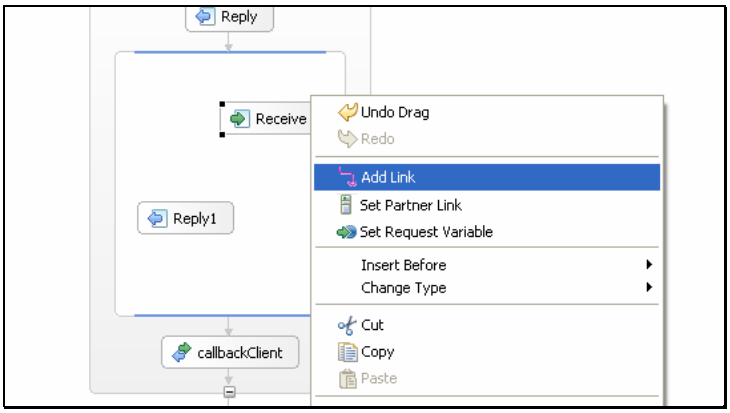

Figure 37: Link creation from the GUI (1)

As in the case explained previously the process begins with the invocation of the opportune action. This action is invoked from the contextual menu as Figure 37.

1. When the user clicks in the 'Add Link' button, the action class *CreateFlowLinkAction* is instantiated (see Figure 37).

2. This class creates a *BPELConnectionCreationTool*, which will be the responsible for managing the process of selecting the target activity. For this issue *CreateConnectionRequest* will help it, refreshing the editpart pointed by the mouse until finding the target activity. This process is illustrated in Figure 38.

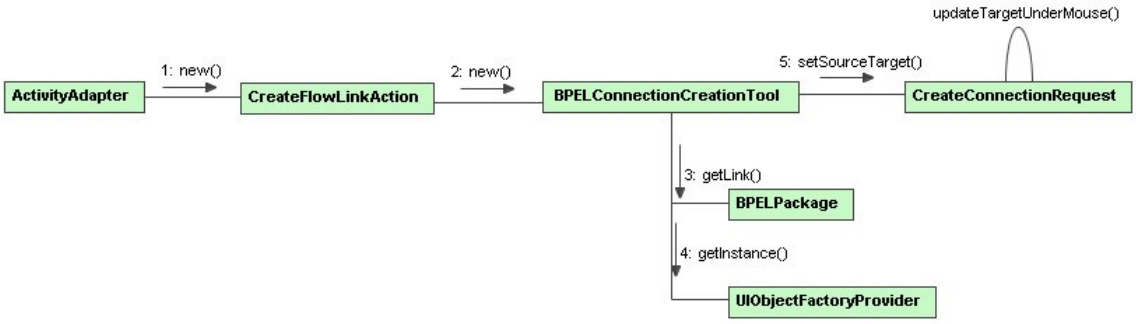

Figure 38: Flow link creation UML Communication Diagram (1)

3. At the same time the creation of the link model representation is provided by *GraphicalNodeEditPolicy* that creates the model object representation of the BPEL flow link (Figure 39, message (2)) and creates the command responsible of the link creation: *AddLinkCommand*.

4. This command is put into the top of the command stack. When the user clicks in the target activity the execution of the stack is fired (message (4)).

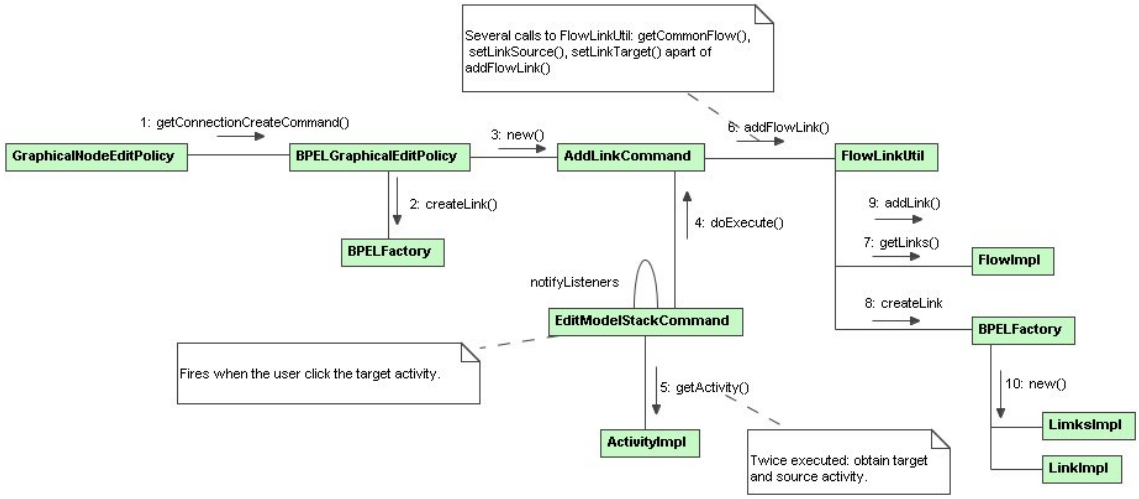

Figure 39: Flow link creation UML Communication Diagram (2)

5. *EditModelStackCommand* is responsible for this execution. Similar to the creation of variables, it tests if the command *AddLinkCommand* can be executed and if so invokes its *doExecute()* method. The command uses a helper class for this issue: the previously explained *FlowUtilLink*.

6. *FlowUtilLink* provides methods to get the parent flow, add the target and source activity to the targets and sources link sets, etc. The process is the following: first it gets the parent flow links set; after that creates the new link model object and finally adds it to the links set (Figure 39, messages (7), (8) and (10)).

7. In a parallel way *EditModelStackCommand* notifies the listeners of the link creation process. The most important of them is *ModelListenerAdapter* as in the variables creation. This class will be the responsible for creating the graphical representation of the link. The interaction here is the same as the variables case, using *BPELAdapterFactory* and *BPELTrayEditPartFactory* to create the link editpart and the correspondent adapter. The picture 40 illustrates this process,

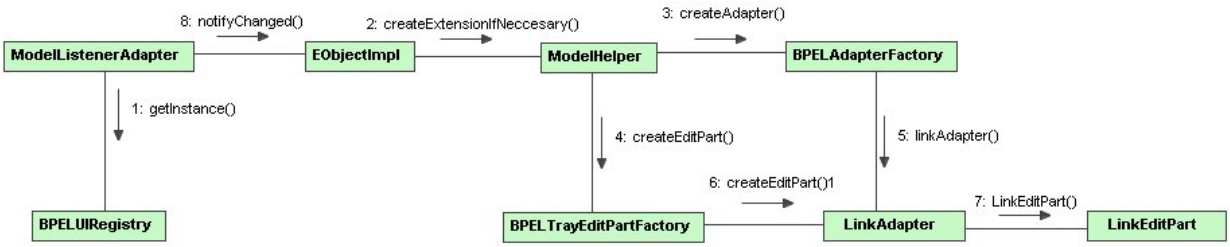

Figure 40: Flow link creation UML Communication Diagram (3)

Figure 41 illustrates the result of the flow link created between these two activities.

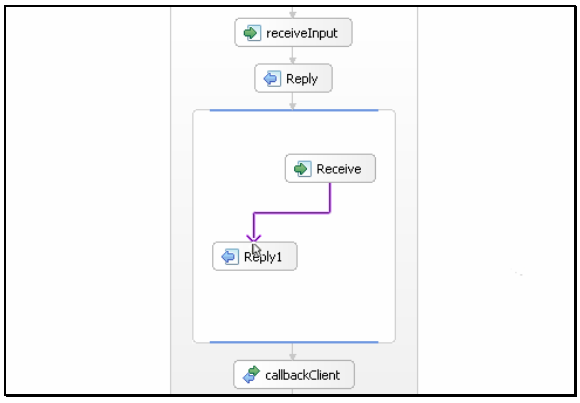

Figure 41: Link creation from the GUI (2)

In the next chapter we will see that the interaction between the classes used for data link creation has some common parts with the flow link one. For this reason part of the classes implied in this process will be used or modified to achieve data link creation.

Chapter 6

BPEL Designer implementation

This chapter explains the design and the implementation of the BPEL-D designer. This tool it is based in the Eclipse BPEL Designer, whose behaviour has been analysed in the Chapter 5 to find that parts that will require being modified to support the BPEL-D syntax (i.e.: using data connectors). It begins with the modifications to be done in the model package, to continue with that ones to be done in the GUI. Finally the splitting algorithm is explained.

6.1. Introduction.

As has been said BPEL-D is designed on top of BEPL 1.1, since the splitting algorithm introduced in [KL06] is designed on top of it. But the Eclipse BPEL Designer implementation is based in BPEL 2.0. Taking into account that the BPEL 1.1 structures studied during the syntax design are also in BPEL 2.0 the main question is to decide what to do with new structures in 2.0 that were not in 1.1. The option followed is restrict the use of the new activities into the BPEL-D Designer.

As we explained before, the objective of this thesis is to define a how data connectors can be implemented, and to create a graphical tool that allows one to create and manage a BPEL-D process. As a requirement, the tool should also be able to take the process information and create an XML representation of the partition, the resulting WSDL and BPEL files based on the given algorithm, and an XML representation of a wiring file.

The development process will require the following steps:

- Adapt the EMF model to BPEL-D syntax.
- Provide the User Interface with the facilities to manage BPEL-D.

- Introduce the splitting algorithm into the Designer.

Another aspect to be treated is to restrict the usage of variables in the Designer. They are not used in the design of a BPEL-D process, but they will be necessary, when splitting the process, the datalinks are converted into variables again.

The extension of the BPEL Designer implementation makes difficult to reflect in this paper all the modifications realized. For this reason it is focused in that ones which are really important to understand the BPEL-D Designer behaviour.

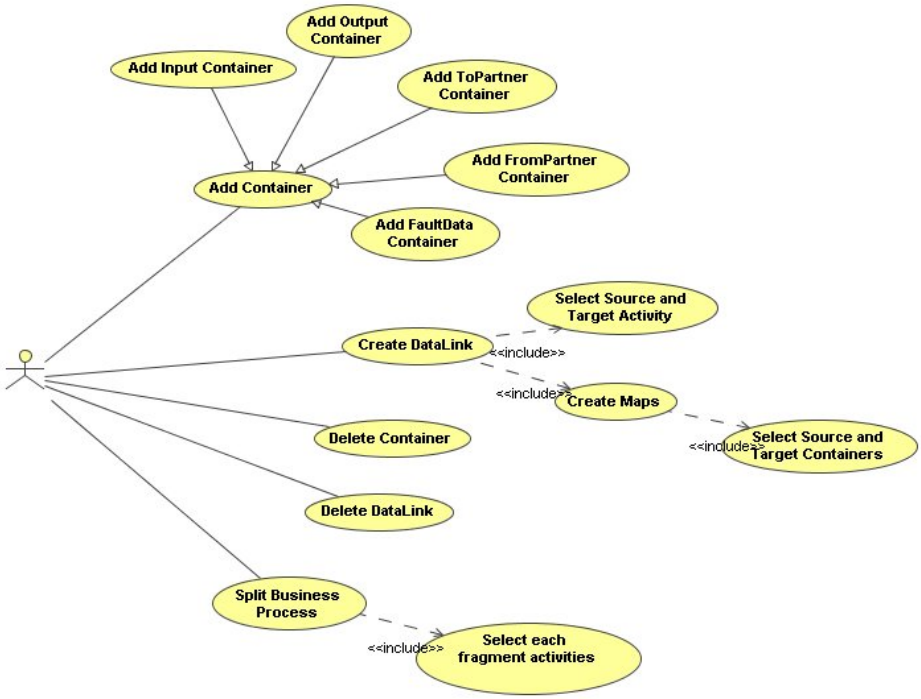

Figure 42: BPEL-D Designer new cases UML Use Cases Diagram

Figure 42 illustrates the user cases to be added to the BPEL Designer to achieve the functionality required for working with datalinks and splitting processes.

6.2. Adapting the EMF model to BPEL-D.

Following the explanation in Chapter 5, the model that represents the BPEL specification is supported by EMF. Then the first task in the BPEL-D Designer implementation is to modify the ***bpel.ecore*** file to fit the new syntax. The steps to make in this process are the following:

1. Introduce in the model the new objects required by the syntax.

2. Adapt the existing elements to interact with the new ones.

In the BPEL-D syntax analysis and design process documented in Chapter 3 of this thesis, we found three new elements to be introduced in the model. These elements were containers, datalinks and maps. Following the tree structure of a BPEL file, three more elements are necessary to support the new requirements: the sections to declare these elements.

To modify the model we use the Sample Ecore Model Editor, even to edit the XMI directly or use the Omondo Eclipse UML plug-in. The modification of the model EPackage means the creation of six new EClasses to represent the concepts of Container, Containers, DataLink, DataLinks, Map and Maps. Apart from that an EEnum element is necessary to represent the different types of container that we described in the design process. All the classes will be implemented as extensions of the root element of the model: ExtensibleElement.

The Container EClass implementation will a bit different with respect to the design of the syntax definition. The reason of that is to fit it with the system implementation, and then we will imitate the BPEL variable structure. The fact of an existing type attribute will require renaming the Type attribute as Direction. Then the Container EClass will be composed by the following attributes:

- *Name*: EAtributte of type EString which refers the container name.
- *Direction*: EReference of type Direction which refers the container direction.
- *MessageType, XSDElement, Type* and *From*: equal to variable ones.

The DataLink EClass implementation is consistent with the syntax declaration. It will be composed by the following attributes:

- *Name*: EAtributte of type EString which refers the container name.
- *SourceActivity* and *TargetActivity*: EReferences of type Activity to the target and source activities of the datalink.
- *Maps*: EReference of type Maps which refers to the Map elements section.

The Map EClass implementation is consistent with the syntax declaration too. Its attributes will be:

- *SourceContainer* and *TargetContainer*: EReferences of type Container to the target and source containers of the map.
- *SourcePart* and *TargetPart*: Optional EReferences of type Part to the target and source container parts of the map.

- *SourceQuery* and *TargetQuery*: Optional EReferences of type Query to the target and source containers queries of the map.

The section declaration elements Containers, DataLinks and Maps, all of them follow the same structure. The only attribute will be an EReference to its child element list called children. The type element of each list will be respectively Container, DataLink and Map.

The EEnum Direction is composed by five Numeral elements. As explained in the Chapter 5, these elements are: *input*, *output*, *toPartner*, *fromPartner* and *fauldData*.

The next step, once the new model objects are defined, is to describe the relation of the new objects with the rest of the BPEL model's representations. The Maps EClass, as well as the Direction EEnum have relations just with our BPEL-D new EClasses; they do not interact directly with the rest of the model.

On the other hand, the Containers EClass will interact with most of the EClasses that represent BPEL activities. As described in the syntax definition, each activity is provided with a containers declaration section. For this reason, each Activity EClass will contain an EReference to a Containers EClass.

The Eclipse BPEL Designer works with BPEL 2.0 activities, some of which did not exist in BPEL 1.1 specification, so we prefer to add this ERefence in each EClass individually instead of doing it in the super class Activity. The activities that have to add the Containers EReference are: *PartnerActivity* (which represents *Invoke*, *Receive* and *Reply*), *Throw*, *Assign*, *Sequence*, *While*, *Flow* and *Scope*. Some constructs are affected too: *Catch* and *OnMessage*.

The *DataLinks* EClass just has to be referenced in the Process EClass.

Once the EMF Model is ready (taking into account that the BPEL variables browsing is postponed), we can regenerate the object model from the ***bpel.genmodel*** file. As we explained before, for each of our new EClass two new Java elements will be generated: an interface in the base package ***org.eclipse.bpel.model***, and a Java Class that implements it in the implementation package ***org.eclipse.bpel.model.impl***. For the EEnum Direction an implementation extending AbstractEnumerator will be generated in the base package too. Figure 43 illustrates then NMO[2] relations.

[2] NMO: New Model Objects for simplification

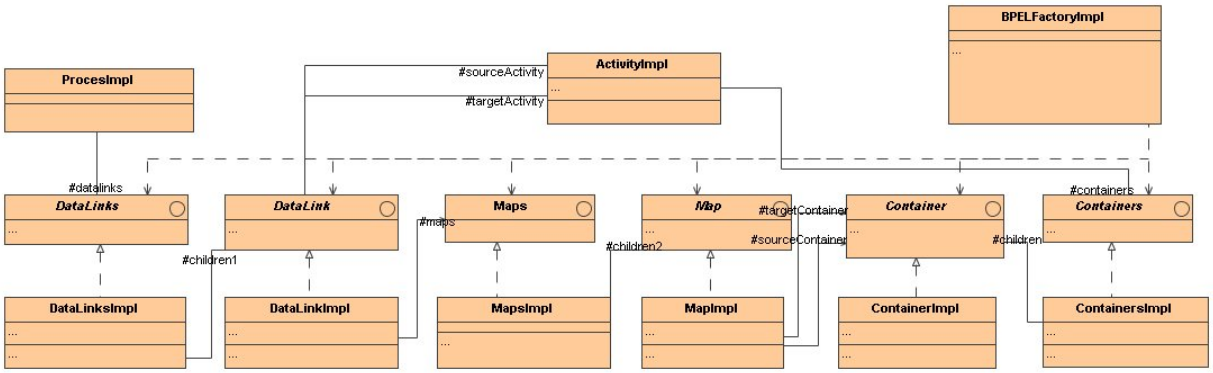

Figure 43: New Model Objects UML Classes Diagram

Apart from that, *BPELFactory*, *BPELPackage* and their implementation classes, as well as *BPELSwitch* will be directly modified by the EMF model to give support to the creation and management of the new specification model classes.

6.3. Providing UI with facilities to manage BPEL-D.

This section explains the BPEL Designer structure modifications done to create a tool which lets the user create and manage data connectors in a BPEL process. After studying the plug-in architecture, we decided to take the existing structure for manage the BPEL Variables, PartnerLinks, CorrelationSets and Flow Links as a model to the managing Containers, DataLinks and Maps.

In the beginning we introduce which modules must be modified and the new classes to be implemented. We start with the modifications referring to the package ***org.eclipse.bpel.model***, and continue after with ***org.eclipse.bpel.ui***.

6.3.1. Modifications in the model package.

As explained in Chapter 5. The BPEL-D elements representations are generated directly from the ***bpel.genmodel***. Apart from that, we will need to create and modify some classes. Attending to the package structure order, the changes required and the new classes added are the following:

- ***org.eclipse.bpel.model.proxys***:
    - *ContainerProxy*, *DataLinkProxy* and *MapProxy*: support the design pattern proxy (see [GHJ+05]) to the model's new specification objects. They extend the implementation classes (see Figure 44).

Figure 44: Model elements hierarchy UML Classes Diagram

- ***org.eclipse.bpel.model.resource***:

  - *ContainerResolver*, *BPELContainerResolver*, *DataLinkResolver* and *BPELDataLinkResolver*: interface and implementation of content providers which provide an extension mechanism for the resolution of containers and datalinks with a name and content. They are used by the *BPELReader* (see Figure 45).

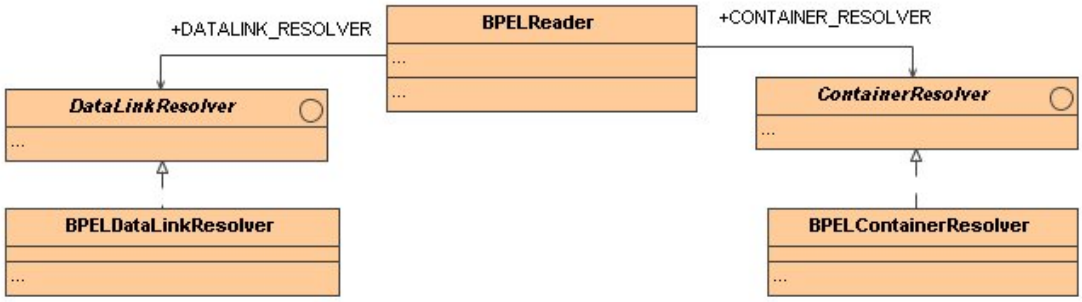

Figure 45: Contain solvers UML Classes Diagram

  - *BPELReader*: we must provide it with new methods to translate the new syntax elements into objects, and modify existing methods which read the process and activities so that they look into them for containers and datalinks elements.

  - *BPELWriter*: the same modifications of *BPELReader* but in the opposite direction.

- ***org.eclipse.bpel.model.util*:**

  - *BPELUtils*: we must add some utility methods to help the rest of the model objects to manage with the new specification objects.

6.3.2. Modifications in the ui package.

Attending to the package structure order, the changes are the following (The NOM acronym means New Model Objects for simplification):

- ***org.eclipse.bpel.ui***:
    - *DataLinkNotificationAdapter*: A new adapter which gets notified of all model changes and refreshes the connections of the appropriate edit parts when there are model changes involving data links.
    - *BPELEditor*: add creation support to the new actions about the new model objects creation into the editing context.
    - *ProcessContextMenuAdapter*: add access reference in the process context menu to the new actions previously described.
    - *Messages* and *IBPELUIConstants*: add the references to the new interface elements messages and icons.
- ***org.eclipse.bpel.ui.actions.editpart***:
    - *CreateDataLinkAction*, *SetContainerAction*, *SetMapAction*: new actions to implement the creation and setting of the new model objects. Figure 46 illustrates them.

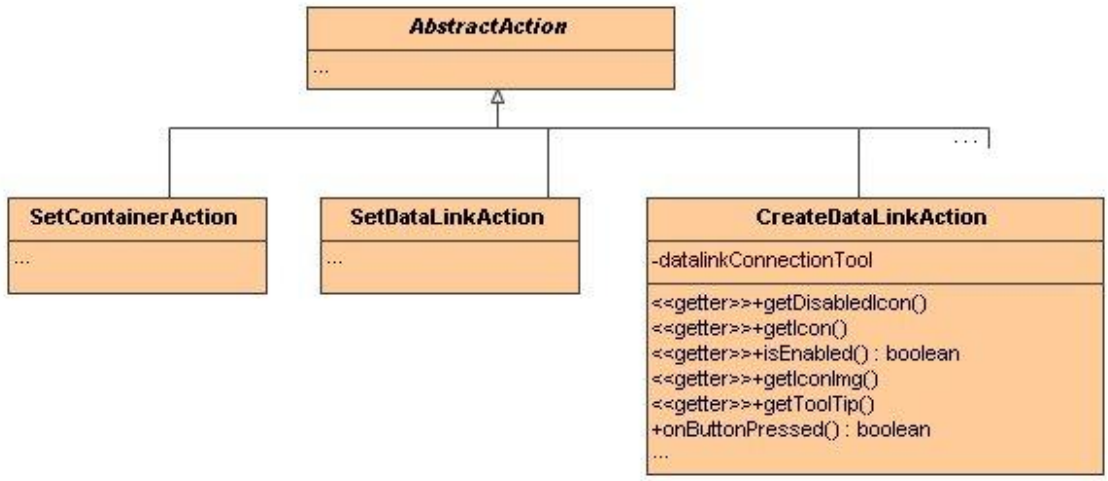

Figure 46: New editparts UML Classes Diagram

- ***org.eclipse.bpel.ui.adapters***:
    - *Container/sAdapter*, *DataLink/sAdapter* and *Map/sAdapter*: adapters for the new model objects and their respective sections to adapt their graphical representation to their modelling representation. Figure 47 illustrates these new adapters.

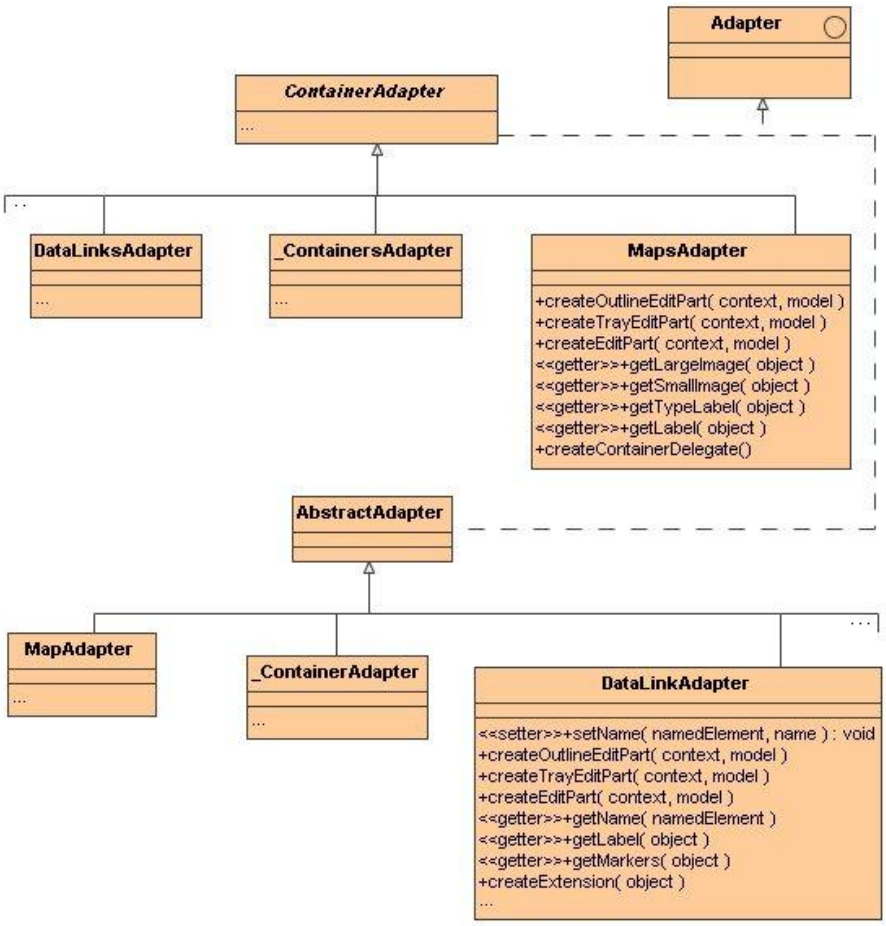

Figure 47: New adapters UML Classes Diagram

- *BPELUIAdapterFactory*: add the adapters' creation methods for the NMO.

- *AssignAdapter*, *FlowAdapter*, *OnMessageAdapter*, etc.: all activity adapters must be modified to add the menu context option of adding an appropriate container (input, output or whatever). Apart from that we need to change their extension super class from *ActivityAdapter* to *ContainerActivityAdapter*.

- ***org.eclipse.bpel.ui.commands***:

  - *SetContainer*, AddDataLink*, DeleteDataLink** and, *SetMapCommand*: new commands which actually manage the creation, setting and deleting of NMO into the model representation. Figure 48 illustrates the new commands.

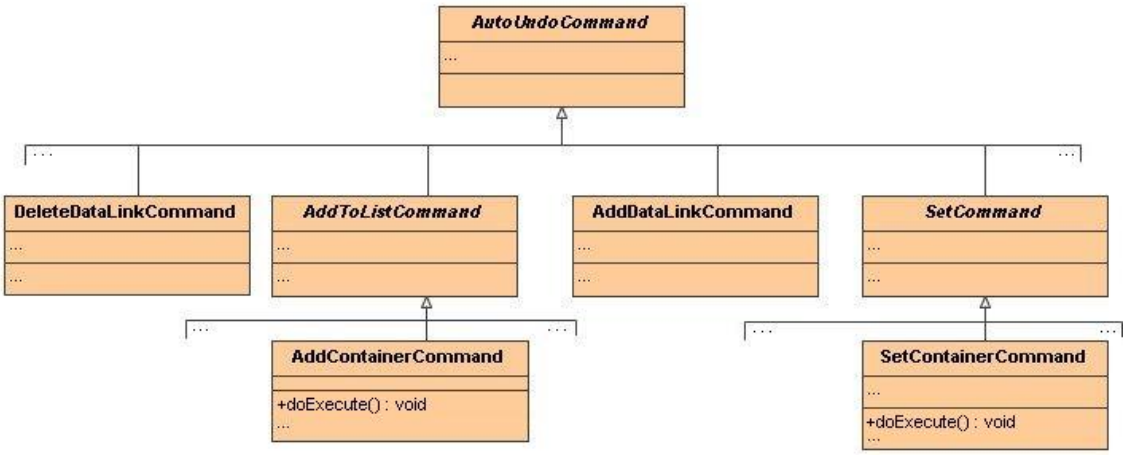

Figure 48: New commands UML Classes Diagram

- ***org.eclipse.bpel.ui.details.providers****:*
  - *ContainerContentProvider*, *DataLinkContentProvider* and *MapContentProvider*: provide the containers, datalinks and maps of visible containment in a given context.
- ***org.eclipse.bpel.ui.details.tree****:*
  - *BPELContainerTreeNode*: represent a Container model object in a similar way as a Variable.
- ***org.eclipse.bpel.ui.dialogs****:*
  - *ContainerSelectorDialog* and *MapSelectorDialog*: support GUI dialog objects to let the user specify the creation of the NMO (See Figure 49).

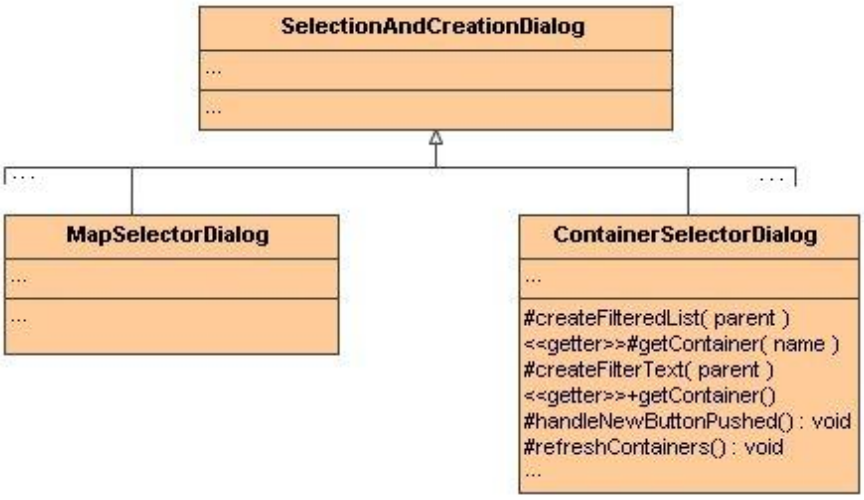

Figure 49: New dialogs UML Classes Diagram

- ***org.eclipse.bpel.ui.editparts***:

    - *Container/sEditPart*, *DataLink/sEditPart* and *Map/sEditPart*: new controllers to specify how the NMO are mapped to visual Figures. Figure 50 shows the new editparts classes.

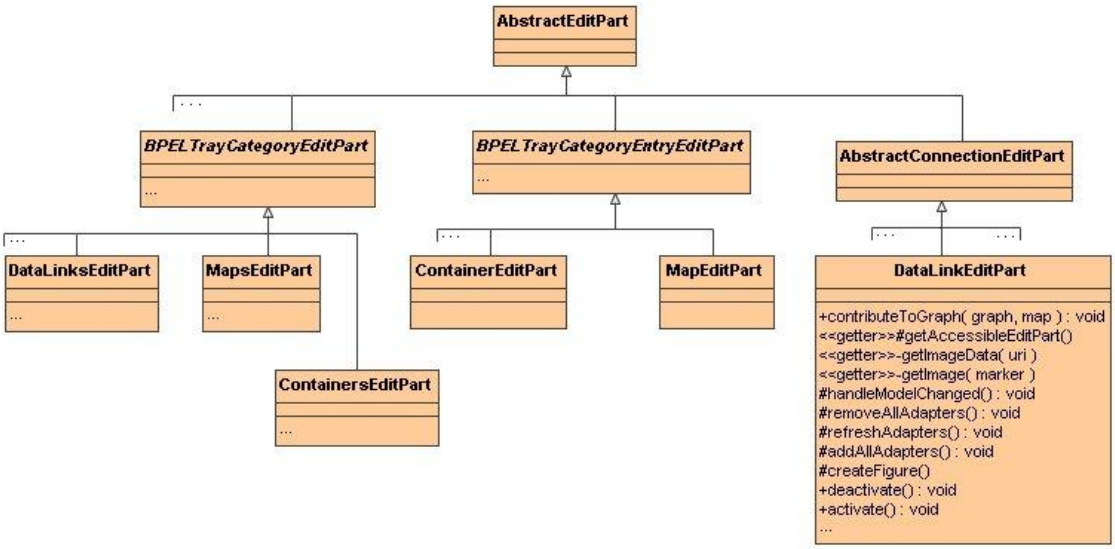

Figure 50: New editparts UML Classes Diagram (2)

    - *ReplyEditPart* and *SequenceEditPart*: modify the "not having child" activities graphical representations to the fact of now all them can have container child elements.

    - *ProcessOutlineEditPart* and *ProcessTryEditPart*: add support to manage the NMO elements.

- ***org.eclipse.bpel.ui.editpars.policy***:

    - *BPELGraphicalEditPolicy*: modify the graphical link management to support also the datalink graphical management.

    - *DataLinkConectionEditPolicy*: new policy to handles deleting datalink requests.

- ***org.eclipse.bpel.ui.factories***:

    - *BPELUIObjectFactory*: add methods to create the NMO objects as well as modify the creation and initialization of some constructs affected by them.

- ***org.eclipse.bpel.ui.util:***
    - *ProcessDataLinkUtil*: provides some helper method to deal with datalinks into the process
    - *BPELReader*: add support to initialize the activities' child container lists during the parsing process.
    - *BPELUtil and ModelHelper*: add some utility methods to help the rest of the model objects manage with the new specification objects.

Figure 51 illustrates the relations between the new classes created for managing the containers. The classes implied on datalinks and maps management follow a quite similar schema.

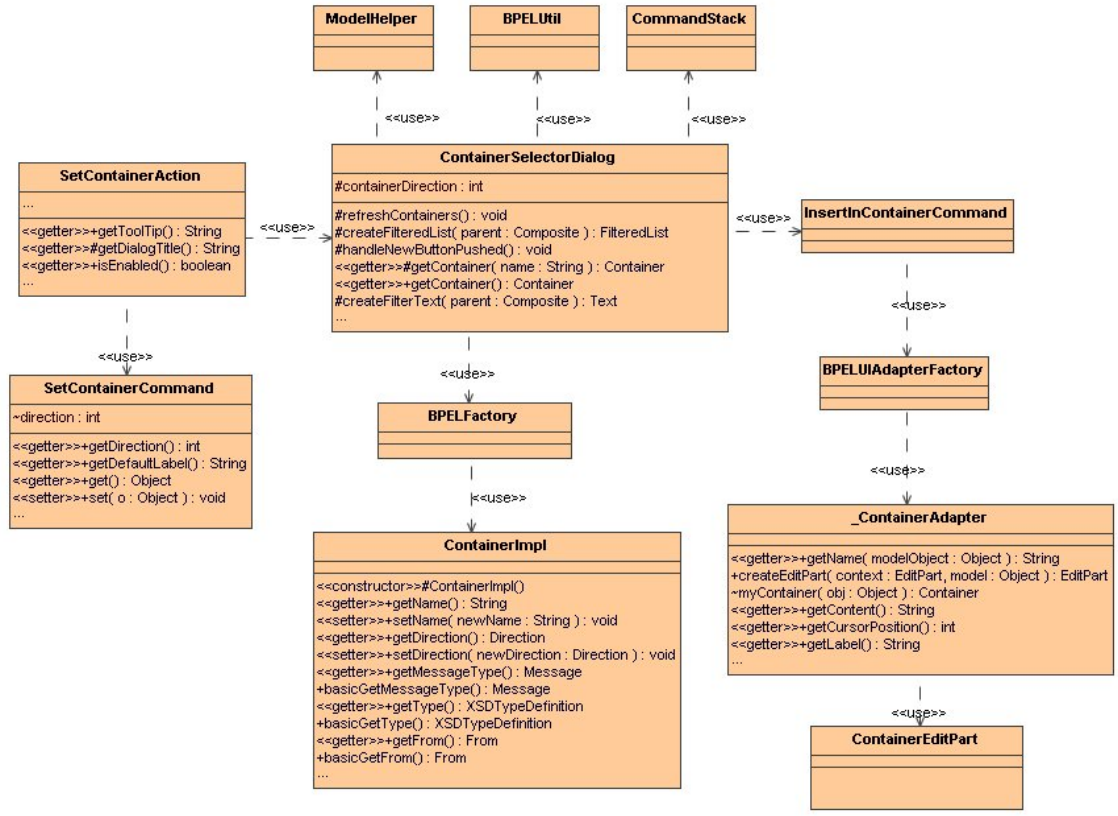

Figure 51: Container creation UML Classes Diagram

6.3.3. Creating a Container from the GUI

As we explained, there are two ways of instantiating containers in the BPEL Designer; the first one was during the parsing process, the second one is choosing the option from the context menu of an activity in the User Interface. The Figure 52 introduces some communication diagrams to illustrate this process. A concrete explanation follows the picture.

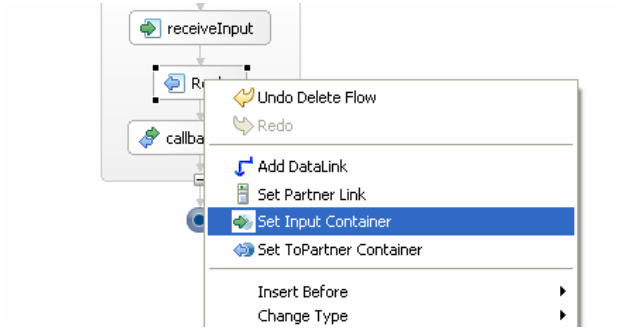

Figure 52: Container creation from the GUI (1)

1. First, the user selects to add a new container from the context menu of one BPEL Activity. The *ActivityAdapter* instantiate the correspondent action class *SetContainerAction* and pass the control to it (Figure 53 message (2)).

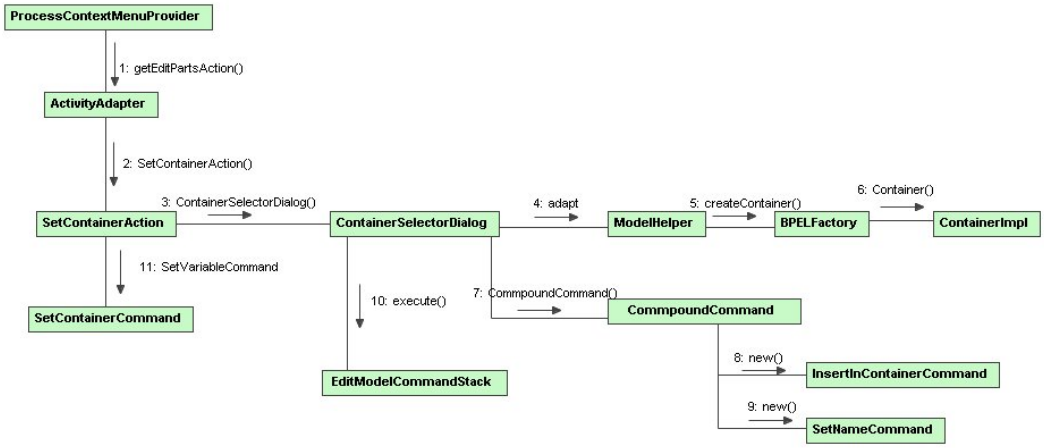

Figure 53: Container creation UML Communication Diagram (1)

2. *SetContainerAction* instantiates *ContainerSelectorDialog* (message (3)), which will be responsible for the major process part. This class displays the GUI dialog which lets the user create a new container (see Figure 54).

3. After the user chooses a name for the container and clicks ok, *ContainerSelectorDialog*, helped by methods from the *ModelHelper* and *BPELUtil*, creates an instance of Container by using the *BPELFactory* ((4), (5), and (6).

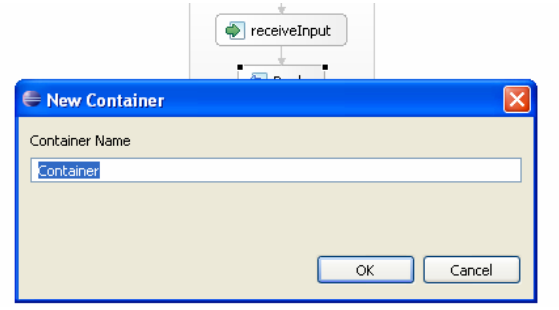

Figure 54: Container creation from the GUI (2)

4. After the *ContainerSelectorDialog* instantiates a *CompoundCommand* composed by *InsertInContainerCommand* and *SetNameCommand* (messages (8) and (9). The first one has a real importance for us.

5. As in the variables situation, the *CompoundCommand* will be responsible for testing if the commands compounded in it can be executed. Because of that it goes through its commands list calling the method "canExecute()" on each one of them (Figure 55, message (4). If one returns false, an exception is thrown.

6. When the command execution is fired *InsertInContainerCommand* is responsible for adding the new container to its place (through *ContainerAdapter*). For this issue it must get the containing activity and its containers element (Figure 55). After this it notifies the listeners to create the graphical representation for it.

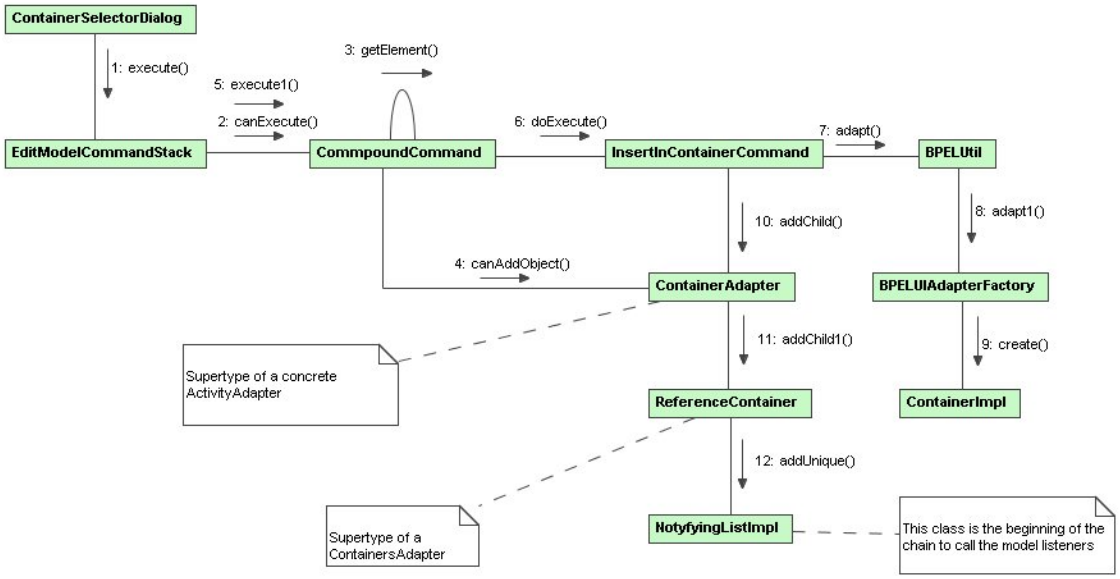

Figure 55: Container creation UML Diagram Communication (2)

7. The *ModelListenerAdapter* is responsible for creating the container editpart and the adapter for it in a similar way as the previously explained ones. For this issue it uses *BPELAdapterFactory* and *BPELTrayEditPartFactory* (Figure 56).

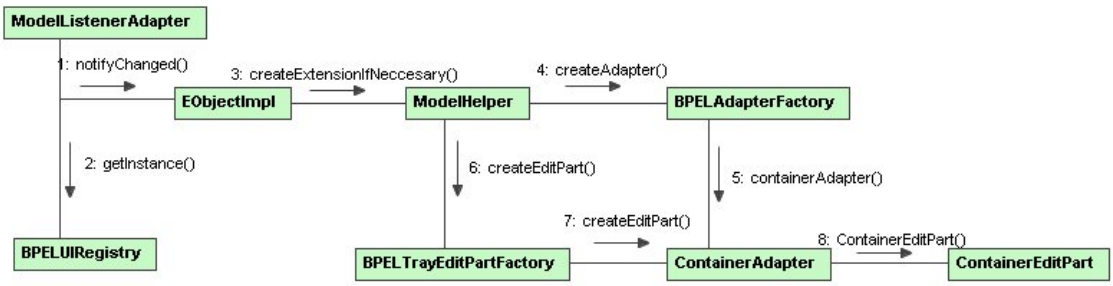

Figure 56: Container creation UML Diagram Communication (3)

8. After that the control comes back to the Dialog who takes the *CommandStack* which will be the responsible of trying to execute the existing commands.

6.3.4. Creating a DataLink from the GUI.

The creation of a datalink is a bit more complicated process. The reason is that a datalink creation represents a graphical representation quite more complex than that one of the container. In spite of that, the processing is quite similar to the flow link creation process. For this reason the common aspects are not in detail described; for more see the link creation process description and its UML communication diagrams in section 5.5. Figure 59 illustrates the classes' interaction.

The datalink creation begins with the user clicking in an activity and choosing the option 'Add Data Link' in the contextual menu (Figure 57). In this moment the pointer becomes in an edge and the opportune command is created. The user must achieve another activity with containers with this edge and click again. These steps correspond to the 4 first steps in link creation (see Section 5.5) being *CreateDataLinkAction* the action class and *CreateDataLinkCommand* the command class.

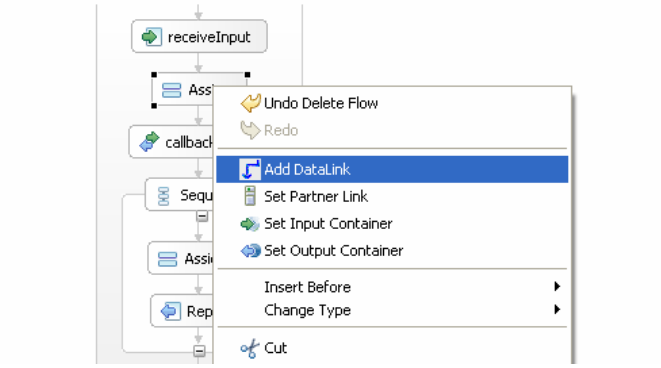

Figure 57: Datalink creation from the GUI (1)

In this moment the Dialog pop-up showing the different containers which can be related (see Figure 58) from the source to the target activity. This process is ruled by *MapSelectorDialog* (see Figure 59) which show the user the containers he can relate in the map.

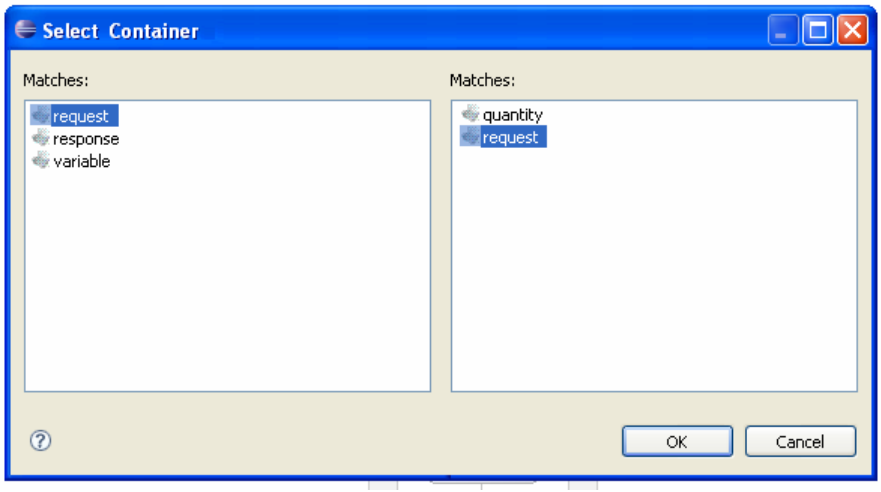

Figure 58: Datalink creation from the GUI (2)

The containers shown will depend of the source and target activity nature. In the case of relating two single activities, the user can choose between output, the fromPartner or the faultData (in case of being a throw) container from the source activity. Also from the input container if source activity is a structured one. In the case of targetActivity the user can choose between input containers, the toPartner or the fauldData container.

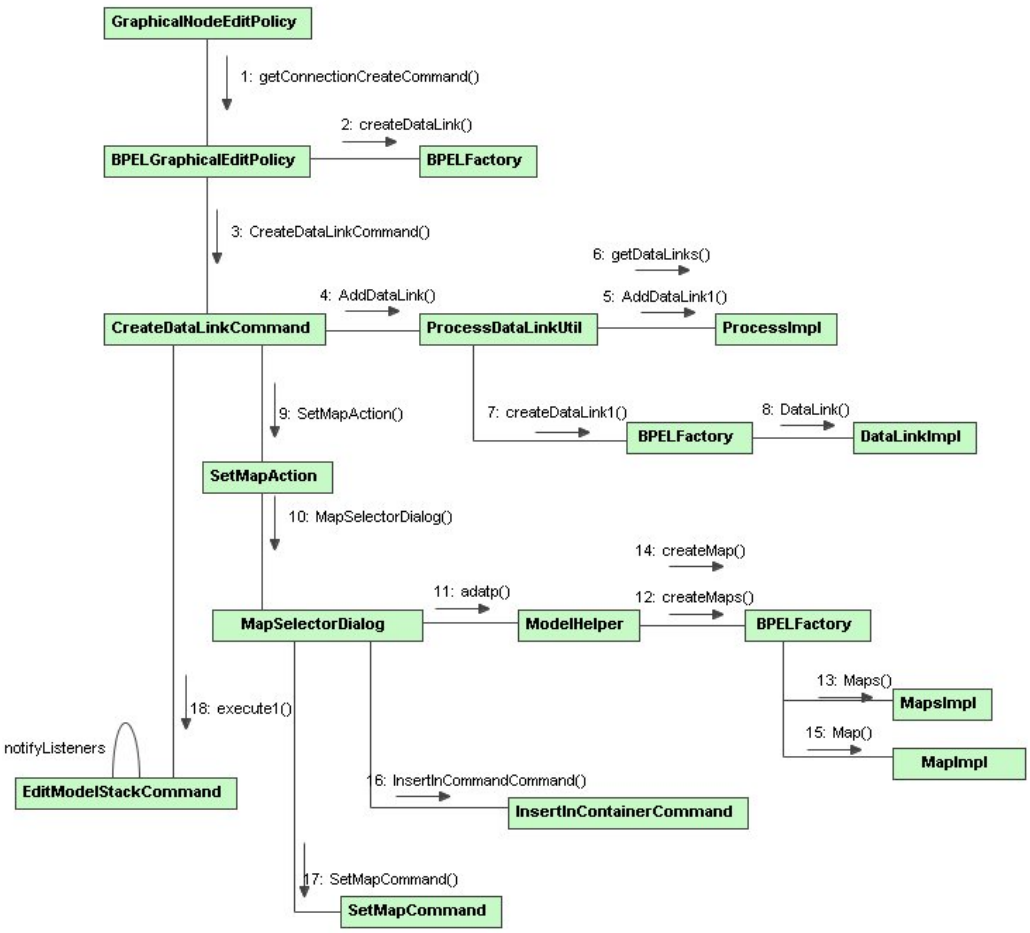

Figure 59: Datalink creation UML Communication Diagram

The dialog class uses the *ModelHelper* to create the objects and sets the commands *InsertInContainerCommand* and *SetMapCommand* in the *CommandStack*. Once the user decides which containers to relate and click ok, the map creation is fired by the *EditModelStackCommand* (see section 5.5, steps 5 and 6).

As in the case of links and VPC objects, *ModelListenerAdapter* is responsible for creating the graphical representation of the datalink (see section 5.5 Figure 40) and the map which corresponds with step 7 of link creation being *DataLinkEditPart* and *MapEditPart* the necessary editparts. Then the datalink between the activities is shown in the GUI (see Figure 60).

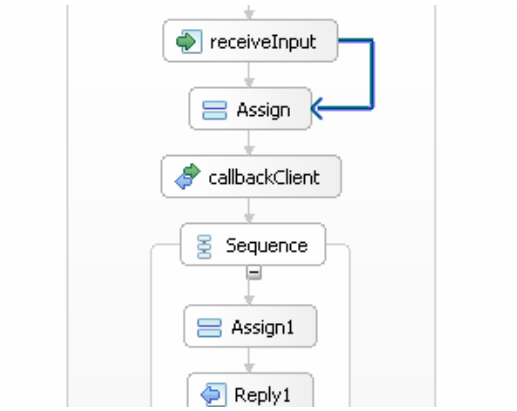

Figure 60: Datalink creation from the GUI (3)

6.4. Browsing variables usage from the model.

Once the BPEL-D Designer is ready to work with data connectors, the next step is browse the BPEL variables usage from the model. This way the information exchange along the process will be realized just by the datalinks. This task has not as much interest as the rest of the research, for this reason this section introduces just a brief explanation to the process.

The references to the variables in the EMF model must persist because they will be necessary to build the BPEL fragments from the BPEL-D process.

We need to remove all that classes involved in the management of the variables into the process or offer to the user this management on the GUI, and modify the reference to them in that ones which realize another tasks. Just the splitting algorithm and the *BPELWriter* will need to work with them. We find not much more interesting explain this process in much more detail.

Completed this process the model works just with data connectors. Then is time to introduce the splitting algorithm as last step in our implementation process.

6.5. Splitting algorithm.

6.5.1 Overview

This section introduces a brief description of the mechanics of the partitioning algorithm. This algorithm is detailed descript in [KL06] (more issues are introduced in [KL07] and [Pal07]). As explained there, the aim is to "cut out" parts of a process model to delegate their performance to third-parties, or in another words Business Process Outsourcing. To achieve this target a way of fragmenting the process is needed. The algorithm creates a separate processes model for each different partner, translating to them the flow and data control dependencies between activities from the original process

The decomposition is created by defining a partition of the set *A* of all activities in the process. Every participant, *n,* belonging to the set of participants, *N,* consists of a name, *s*, and a set of one or more activities, *M,* such that $N = \{n\} = \{(s, M) \mid M \subseteq A\}$ Using $\pi_i(f)$ to represent the i-th projection map, the restrictions on *N* are the following:

$$\forall n_i \in N, \pi_2(n_i) \neq \phi$$
$$\forall n_i, n_j \in N, i \neq j \longrightarrow n_i \text{ I } n_j = \phi$$
$$\text{Y}_{n \in N} \pi_2(n) = A$$

In other words, a participant must have at least one activity, no two participants share an activity or a name, and every activity of the process is assigned to a participant.

New portTypes and partnerLinkTypes must be created for each participant to reflect the communications between the fragments. Each fragment is converted in a BPEL process. Some elements as process partnerlinks, correlations set must be added to each fragment. BPEL-D datalinks that stay within each fragment are replaced in variables by using an 'assign' linked between the data source activity and the data target activity. Invoke and receive activities will be added to communicate the fragments.

To avoid problems related with DPE, the algorithm must ensure that a datalink does not write data for the target activity unless its source activity has complete. To solve this problem it adds a new activity between the receive and the target activity (see [KL06] for more). This is shown in Figure 61, being the 'assign' the added activity.

This assures skipping the new activity if the source one fails. The join condition of the target activity is ignores the status of the new link from the added activity, because whether or not the target activity will run is not a function of its datalinks.

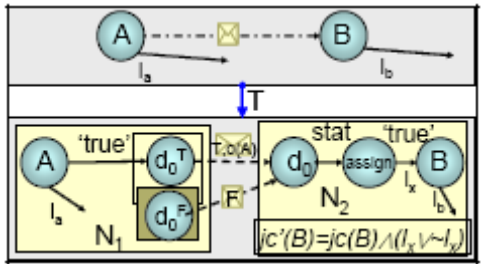

Figure 61: Splitting up a data link across local processes [KL06]

Depending of the source activity completion, data and 'true' or 'null' and 'false' will be send by the fragment. Fault handling is used to propagate the activity completion. Additional data is sent in the positive case, an activity buffers the 'receive' from the original target, and the join condition of the target is modified. The join condition of the target is amended as for 'B' above.

Finally the fragments need to be wired together because they can not relate others partnerLinks by alone. For this reason is necessary a minimal wiring: this will be a set of pair-wise connectors in which one part must be a BPEL Process For this issue its designed a deployment descriptor element. This element is descript in [KL07].

6.5.2 Design and implementation.

To convert the explained solution in a concrete algorithm which allows us to fragment business process we will need to realize the next modifications in the system.

The first step to support the splitting algorithm is modifying the model which represents the BPEL specification elements. It means modify again the ***bpel.ecore*** file.

The new EClasses Participants and Participant must be added. Participants follow the same structure as the classes Datalinks, Containers or Maps having just one EReference: children, its child element list.

EClass Participant is composed by the following attributes:

- *Name*: EAtributte of type EString which refers to the participant name.
- *Activities*: EReference of type List which refers to the list of activities related to the participant.

An EReference to Participants is added to the EClass Process. Also the following attributes must be added to support the semantics defined in [Pal07]. This modifications are added to BPEL-D syntax:

To EClass Process:

- *Belongs-to*: EAtribute of type EString which specifies the name of the overall process.

To EClass Scope:

- *Fragmented:* EAtribute of type EBoolean which denotes whether the scope is a fragment or not. Default value “no”.

To EClass While:

- *Fragmented:* EAtribute of type EBoolean which denotes whether the loop is a fragment or not. Default value “no”.
- *Is-responsible:* EAtribute of type EBoolean which specifies whether this fragment of the loop is responsible for the loop condition. Default value “no”.

To support the graphical representation of the participants next classes are designed: *ParticipantEditPart* and *ParticipantsEditParts*, as well as their respective adapters *ParticipantAdapter* and *ParticipantsAdapter*. This classes follows the same schema as the NMO do (see Section 6.3.1).

The next step is to provide the GUI with the necessary facilities no manage this functionality. This task can be divided in the following subtask:

1. Define the participants to execute the different fragments.
2. Assign a participant to each activity.
3. Split the process.

According with the plug-in structure, the next elements have been designed and implemented to achieve this task.

CreateParticipantAction, *SetParticipantAction* and *SplitProcessAction*: action classes responsible of implementing the creation of new participants relate them with the activities and split the process respectively. All they inherit from *AbstractAction*.

CreateParticipantDialog, *ParticipantSelectorDialog* and *SplitProcessDialog*: GUI dialogs that let the user, respectively, to create a new participant and relate an activity with a participant. Inherit the two first from *SelectionAndCreationDialog* and the last one from *AbstractDialog*.

AddParticipantCommand, *SetParticipantComand* and *SplitProcessCommand*: new commands which actually manage the creation of a new participant, fix the relation of it

with an activity, and split the process respectively. Inherit from *AddToListCommad*, *SetCommand* and *AutoUndoCommand* respectively.

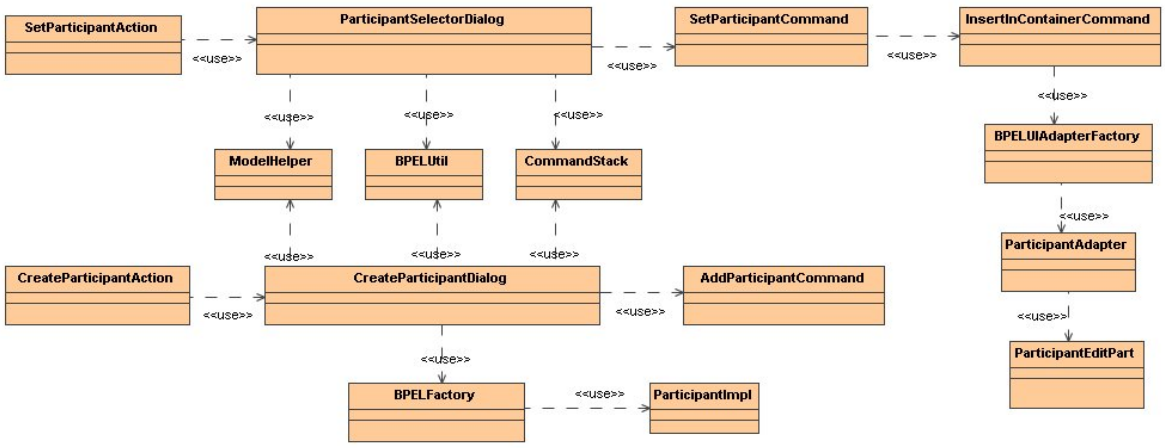

Figure 62: Participants UML Classes Diagram

Figures 62 and 63 illustrate the relation between the recently created classes.

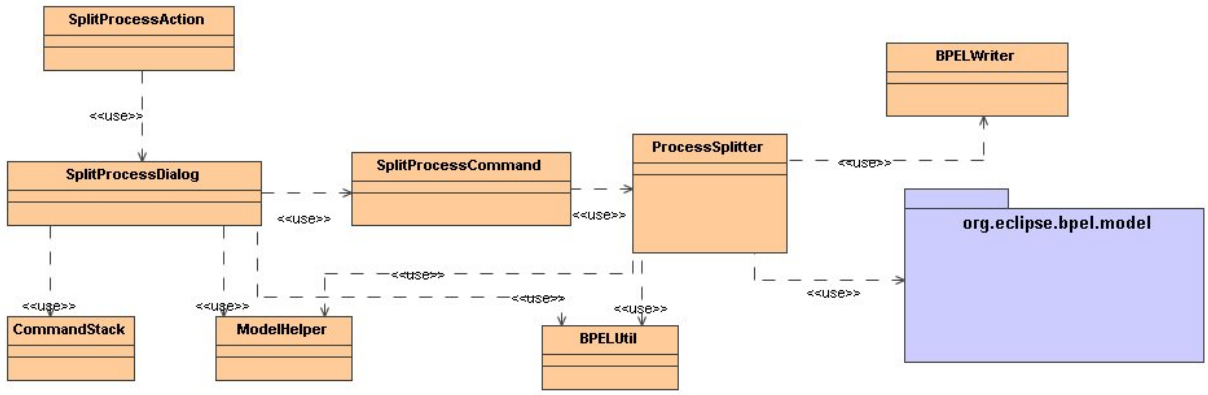

Figure 63: Splitting process UML Classes Diagram

Apart of that the next classes have been modified to relate the activities with the participants: *Messages*, *IBPELUIConstants*, *ActivityAdapter* and *ActivityEditPart*.

The most important class of this process is *ProcessSplitter* which is responsible of taking the existing model representation, create the different fragments based in it and write them into different ****.bpel*** files.

6.5.3. Splitting the process from the GUI.

The first step to split the process is the user to define the participants between which the process must be defined. The creation one participant follows the same schema as the creation of VPC elements or containers (see sections 5.4 and 6.3.3). For this task *CreateParticipantAction*, *CreateParticipantAction*, *CreateParticipantDialog*,

AddParticipantCommand, *ParticipantEditPart* and *ParticipantAdapter* substitute the classes used in Figures 34, 35 and 36 (section 5.4).

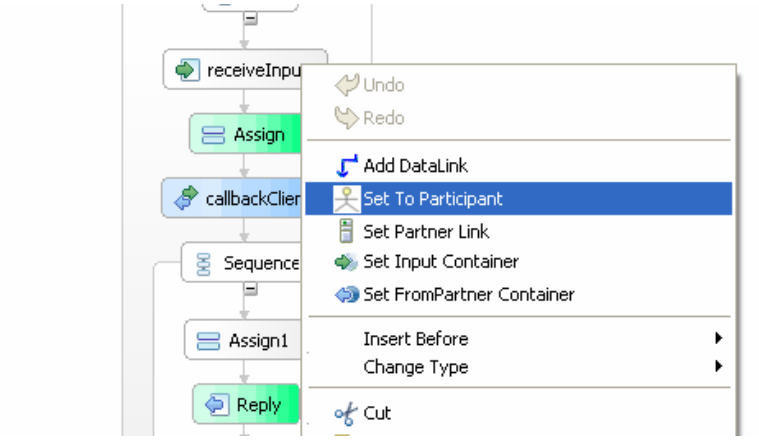

Figure 64: Splitting the process from the GUI (1)

Once the participants are defined, the user must relate each activity with one of them. For this issue he must select the option 'Set To Participant' in the contextual menu related to the opportune activity. The GUI expresses belonging to a participant changing the background of the activity to a different colour (See Figure 64). These colours are predefined in class *IBPELUIConstants*.

When all the activities have been assigned to a participant, user must click in the splitting button (See Figure 65). This button fires *SplitProcessAction* that, through the correspondent classes *SplitProcessDialog* (in this case, this is not a GUI dialog, just follow the habitual chain of responsibility) and *SplitProcessCommand,* instantiate *ProcessSpliter* (See Figure 63).

Figure 65: Splitting the process from the GUI (2)

As has been said this class is responsible of the whole splitting. *ProcessSpliter* implements the algorithm introduced in [KL07] and detailed in [Pal07]. Figure 66 illustrates a high view of this class's work.

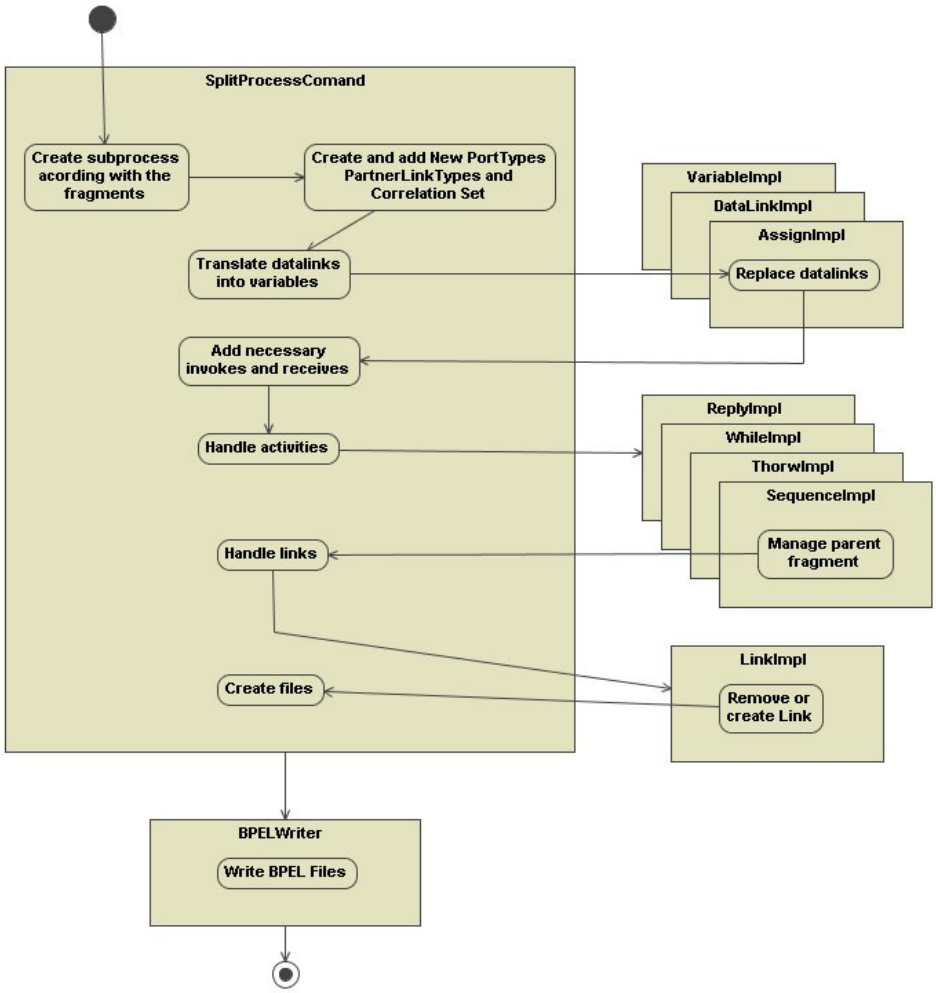

Figure 66: Splitting algorithm UML Activity Diagram

Chapter 7

Conclusions and future works

The aim of the thesis is to design the syntax of the variant of BPEL with explicit data flow defined in [KL06] an implement a Designer Tool for it basing in Eclipse BPEL Designer. First we design the necessary extensions defined to specify the data flow between the activities of the process illustrating the syntax and semantics definition process. After that we analyze how the Eclipse Designer is designed to after make the necessary modifications to allow working with BPEL-D files instead of BPEL 2.0.

The result is a tool which allows the designer build a BPEL process with explicit data flow and split it in several really BPEL fragments that will run locally in different partners, using Web Services to propagate data and control between them.

As has been said the approach defined in [KL06] takes BPEL 1.1 as background to define an explicit data flow variant. Nowadays, industry uses BPEL 2.0 has standard. For this reason future work will be adapting the syntax defined through this thesis to tackle the BPEL 2.0 activities not supported by the approach. Other aspects to be treated for the enrichment of BPEL-D are: supporting BPEL 2.0 Compensation handlers which can read and write data on the process, or let arbitrary activities write into fault handlers (only the throw can do it in our approach).

Another interesting point to be treated in later work is the possibility of designing an algorithm to convert directly from a BPEL 1.1 file to a BPEL-D one, saving the designer to translate by himself this task which in case of long described process it can achieve a big complexity.

In relation with Eclipse BPEL-D Editor, there are many aspects that can be treated in the future; End-to-end integration, to letting the user the whole BPEL-D – BPEL

processing: design the process, assigns participants to activities, chooses a coordinator, assign engines to the participants, split the processes and deploy them on the assigned engines.

APPENDIX A

BPEL-D SAMPLES

Loan approval sample in BPEL-D syntax

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

<process name="loanApprovalProcess" suppressJoinFailure="yes"
        targetNamespace="urn:active-endpoints.samples.bpel1_1.2005 10.loan_approval"
        xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
        xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
        xmlns:lns="urn:active-endpoints.resources.wsdl.2005-10.loan_approval"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:bpeld="urn:IAAS:BPEL:BPEL-D/2007/07/25">

<partnerLinks>
        <partnerLink myRole="loanService" name="customer"
                partnerLinkType="lns:loanPartnerLinkType" />
        <partnerLink name="approver"
                partnerLinkType="lns:loanApprovalLinkType" partnerRole="approver" />
        <partnerLink name="assessor"
                partnerLinkType="lns:riskAssessmentLinkType" partnerRole="assessor" />
</partnerLinks>

<faultHandlers>
        <catch name="catch1" faultName="lns:loanProcessFault">
                <containers>
                        <container name="error" type="faultData"/>
                </containers>
                <reply name="reply2" faultName="lns:unableToHandleRequest"
                        operation="request" partnerLink="customer"
                        portType="lns:loanServicePT">
                        <containers>
                                <container name="error" type="input"/>
                        </containers>
                </reply>
        </catch>
</faultHandlers>

<dataLinks>
```

```
        <dataLink sourceActivity="receive1" targetActivity="invoke1">
                <map sourceContainer="request" targetContainer="request" />
        </dataLink>

        <dataLink sourceActivity="receive1" targetActivity="invoke2">
                <map sourceContainer="request" targetContainer="request" />
        </dataLink>

        <dataLink sourceActivity="assign1" targetActivity="reply1">
                <map sourceContainer="approval" targetContainer="approval" />
        </dataLink>

        <dataLink sourceActivity="invoke2" targetActivity="reply1">
                <map sourceContainer="approval" targetContainer="approval" />
        </dataLink>

        <dataLink sourceActivity="invoke2" targetActivity="reply1">
                <map sourceContainer="approval" targetContainer="approval" />
        </dataLink>

        <dataLink sourceActivity="catch1" targetActivity="reply2">
                <map sourceContainer="error" targetContainer="error" />
        </dataLink>
</dataLinks>

<flow>
        <links>
                <link name="receive-to-assess" />
                <link name="receive-to-approval" />
                <link name="assess-to-setMessage" />
                <link name="assess-to-approval" />
                <link name="setMessage-to-reply" />
                <link name="approval-to-reply" />
        </links>

        <receive name="receive1" createInstance="yes" operation="request"
                partnerLink="customer" portType="lns:loanServicePT">
                <containers>
                        <container name="request" type="fromPartner"/>
                </containers>
                <source linkName="receive-to-assess"
                        transitionCondition="getContainerData('request','amount') &lt;
                        10000" />
                <source linkName="receive-to-approval"
                        transitionCondition="getContainerData('request','amount') &gt;=
                        10000" />
                <source linkName="request-to-check"/>
        </receive>

        <invoke name="invoke1" operation="check" partnerLink="assessor"
                portType="lns:riskAssessmentPT">
```

```
            <containers>
                <container name="request" type="toPartner"/>
                <container name="risk" type="fromPartner" />
            </containers>
            <target linkName="receive-to-assess" />
            <source linkName="assess-to-setMessage"
                transitionCondition="getContainerData('risk','level')='low'" />
            <source linkName="assess-to-approval"
                transitionCondition="getContainerData('risk','level')!='low'" />
            </invoke>

        <assign name="assign">
            <containers>
                <container name="approval" type="output"/>
            </containers>
            <target linkName="assess-to-setMessage" />
            <source linkName="setMessage-to-reply" />
            <copy>
                <from expression="'yes'" />
                <to container="approval"/>
            </copy>
        </assign>

        <invoke name="invoke2" operation="approve" partnerLink="approver"
            portType="lns:loanApprovalPT">
            <containers>
                <container name="request" type="toPartner"/>
                <container name="approval" type="fromPartner" />
            </containers>
            <target linkName="receive-to-approval" />
            <target linkName="assess-to-approval" />
            <source linkName="approval-to-reply" />
        </invoke>

        <reply name="reply" operation="request" partnerLink="customer"
            portType="lns:loanServicePT">
            <containers>
                <container name="approval" type="toPartner" />
            </containers>
            <target linkName="setMessage-to-reply" />
            <target linkName="approval-to-reply" />
        </reply>
</flow>
</process>
```

Market place sample in BPEL-D syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="marketplace"
        targetNamespace="urn:samples:marketplaceService"
        xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
        xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
        xmlns:tns="urn:samples:marketplaceService"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<partnerLinks>
        <partnerLink myRole="sales" name="seller" partnerLinkType="tns:salesplnk" />
        <partnerLink myRole="buying" name="buyer" partnerLinkType="tns:buyingplnk" />
</partnerLinks>
<correlationSets>
        <correlationSet name="negotiationIdentifier" properties="tns:negotiatedItem" />
</correlationSets>

<datalinks>
        <dataLink sourceActivity="receive1" targetActivity="MarketPlaceSwitch">
                <map sourceContainer="sellerInfo" targetContainer="sellerInfo" />
        </dataLink>
        <dataLink sourceActivity="receive2" targetActivity="MarketPlaceSwitch">
                <map sourceContainer="buyerInfo" targetContainer="buyerInfo" />
        </dataLink>
        <dataLink sourceActivity="SucessAssign" targetActivity="SellerReply">
                <map sourceContainer="negotiationOutcome"
                     targetContainer="negotiationOutcome" />
        </dataLink>
        <dataLink sourceActivity="SucessAssign" targetActivity="BuyerReply">
                <map sourceContainer="negotiationOutcome"
                     targetContainer="negotiationOutcome" />
        </dataLink>
        <dataLink sourceActivity="FailedAssign" targetActivity="SellerReply">
                <map sourceContainer="negotiationOutcome"
                     targetContainer="negotiationOutcome" />
        </dataLink>
        <dataLink sourceActivity="FailedAssign" targetActivity="BuyerReply">
                <map sourceContainer="negotiationOutcome"
                     targetDepartment="negotiationOutcome" />
        </dataLink>
</datalinks>

<sequence name="MarketplaceSequence">
<flow name="MarketplaceFlow">
        <receive name="receive1" createInstance="yes" name="SellerReceive"
                  operation="submit" partnerLink="seller" portType="tns:sellerPT">
                <containers>
                        <container name="sellerInfo" type="fromPartner"/>
                </containers>
```

```
        </receive>
        <receive name="receive2" createInstance="yes" name="BuyerReceive"
                 operation="submit" partnerLink="buyer" portType="tns:buyerPT">
                 <containers>
                       <container name="buyerInfo" type="fromPartner"/>
                 </containers>
        </receive>
</flow>
<switch name="MarketplaceSwitch">
        <containers>
                <container name="sellerInfo" type="input"/>
                <container name="buyerInfo" type="input"/>
        </containers>

        <case condition="getContainerData('sellerInfo', 'askingPrice')
                &lt;= getContainerData('buyerInfo', 'offer')">
                <assign name="SuccessAssign">
                        <containers>
                                <container name="negotiationOutcome" type="output"/>
                        </containers>
                        <copy>
                                <from expression="'Deal Successful'" />
                                <to container="negotiationOutcome" part="outcome"/>
                        </copy>
                </assign>
                </case>
        <otherwise>
                <assign name="FailedAssign">
                        <containers>
                                <container name="negotiationOutcome" type="output"/>
                        </containers>
                        <copy>
                                <from expression="'Deal Failed'" />
                                <to container="negotiationOutcome" part="outcome"/>
                        </copy>
                </assign>
        </otherwise>
</switch>
<reply name="SellerReply" operation="submit" partnerLink="seller"
        portType="tns:sellerPT">
        <containers>
                <container name="negotiationOutcome" type="toPartner"/>
        </containers>
</reply>
<reply name="BuyerReply" operation="submit" partnerLink="buyer" portType="tns:buyerPT">
        <containers>
                <container name="negotiationOutcome" type="toPartner"/>
        </containers>
</reply>
</sequence>
</process>
```

BIBLIOGRAPHY

[ACK03] G. Alonso, F. Casati, H. Kuno. Web Services: Concepts, Architectures and Applications. Springer, 2003.

[BOpera] W. Bausch, C. Pautasso, and G. Alonso. Programming for dependability in a service-based gridIn proceedings of the International Symposium on Cluster Computing and the Gid, Tokyo, Japan, May 2003.

[BPML] Business Process Modeling Language Version 1.0, Specification, 2002 http://www.ebpml.org/bpml_1_0_june_02.htm

[BPEL11] Business Process Execution Language for Web Services Version 1.1 Specification, 2003 http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf

[BPEL20] Web Services Business Process Execution Language Version 2.0 Specification, 2007 http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[BPELGuide] Business Process Execution Language (BPEL) Resource Guide http://bpelsource.com/

[BPELSamples] Active BPEL Open Source Project. BPEL Samples http://www.activebpel.org/samples/samples-3/BPEL_Samples/doc/index.html

[BRJ05] G. Booch, J. Rumbaugh, and I. Jacobson Unified Modeling Language User Guide. Addison-Wesley, 2005.

[CD00] F. Casati, A. Discenza. Supporting Workflow Cooperation Within and Across Organizations, SAC 2000, Como, Italy, March 2000.

[Eclipse] The Eclipse Platform, http://ww.eclipse.org/

[EclipseBPEL] The Eclipse BPEL Project, http://www.eclipse.org/bpel/

[EMF] Eclipse Modelling Framework, http://www.eclipse.org/emf/

[GEF] Graphical Editing Framework, http://www.eclipse.org/gef/

[GHJ+05] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Addison-Wesley, 2005.

[IBM] IBM Software Group. http://www.ibm.com/us/

[HPOpen] Hewlett-Packard Company. HP Open View. http://www.openview.hp.com.

[KL06] R. Khalaf, F. Leymann. Role-based Decomposition of Business Processes using BPEL, 2006. ICWS 2006 International Conference of Web Services – Application Services and Industry Track. IEEE, June 2006.

[KL07] R. Khalaf, F. Leymann. Note on Syntactic Details of Split BPEL-D Business Processes (Technical Report 2007/02, Institute of Architecture of Application Systems, University of Stuttgart)

[LR00] F. Leymann, D. Roller. Production Workflow Concepts and Techniques. Prentice Hall, 2000.

[MMP06] A. Marconi, M. Pistore, and P. Traverso. Implicit vs. Explicit Data-Flow Requirements in Web Service Composition Goals. Proc. of Fourth International Conference on Service Oriented Computing (ICSOC06), Chicago, USA, December 2006.

[MQSeries] IBM Corporation. MQ Series Workflow for Business Integration, http://www.ibm.com

[MWW+98] P. Muth, D. Wodkt, J. Wiessenfels, D.A Kotz, G. Weikum, From Centralized Workflow Specification to Distributed Workflow Execution, Journal of Intelligent Information Systems, 1998.

[Pal07] M. Paluszek. Coordinating Distributed Loops and Fault Handling, Transactional Scopes using WS- Coordination protocols layered on WS-BPEL services, (Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Diplomarbeit Nr. 2586, 2007)

[RedBook] Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework. IBM Redbooks, 2004. http://www.redbooks.ibm.com/redbooks.nsf/redbooks/

[SOAP] World Wide Web Consortium. Simple Object Access Protocol Version 1.2, 2007. http://www.w3.org/TR/soap/

[SWSBPEL] Search Web Services. BPEL Learning Guide. http://searchwebservices.techtarget.com/general/0,295582,sid26_gci1172072,00.html

[SWSL] Semantic Web Services Language. World Wide Web Consortium http://www.w3.org/Submission/SWSF-SWSL/

[UML] Unified Modeling Language Resource Page. http://www.uml.org/

[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, 2005.

[WSCDL] N. Kavantzas, D. Burdett, G. Ritzinger, Y. Lafon. Web Services Choreography Description Language Version 1.0. W3C Candidate Recommendation, 2005. http://www.w3.org/TR/ws-cdl-10.

[WSDL] World Wide Web Consortium. Web Service Description Language (WSDL) Version 1.1. http://www.w3.org/TR/wsdl/

[WSFL] Web Services Flow Language 1.1 IBM Software Group http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[XLANG] Microsoft Corp, 2001. http://xml.coverpages.org/xlang.html

[XMI] XML Metadata Interchange. OMG Group.
http://www.omg.org/technology/documents/formal/xmi.htm

[XML] World Wide Web Consortium. Extensible Markup Specification (XML) 1.0, 2006. http://www.w3.org/TR/REC-xml/

[XMLSchema] World Wide Web Consortium. XML Schema
http://www.w3.org/TR/xmlschema-0/

[XPath] World Wide Web Consortium. XML Path Language (XPath) Version 1.0, 1999. http://www.w3.org/TR/xpath

[W3C] World Wide Web Consortium. http://www.w3.org/

All references visited on the 20th July, 2007

Table of Figures

Figure 1: Decomposition and reconnection of business process [KL06] 14
Figure 2: Overview of the approach [KL06] 14
Figure 3: BPEL Process communications 19
Figure 4: Input and output containers 24
Figure 5: Datalinks in a process 25
Figure 6: Containers with data maps 26
Figure 7: Transition conditions 31
Figure 8: Receive with containers 48
Figure 9: Structured activities datalinks options (a) using input containers for every structured activity, (b) not using input containers for each structured activity 57
Figure 10: Structured activities with datalinks (a) using input containers for every structured activity, (b) not using input containers for each structured activity 59
Figure 11: Structured activities with datalinks (a) using output containers for every structured activity, (b) not using output containers for each structured activity. 61
Figure 12: BPEL-D Pick with datalinks 64
Figure 13: BPEL-D While with containers 66
Figure 14: BPEL-D Sequence with datalinks 68
Figure 15: BPEL-D Fault Handler representation 72
Figure 16: BPEL-D Compensation Handler representation 73
Figure 17: org.eclipse.bpel.model UML Package Diagram 84
Figure 18: Model objects UML Classes diagram 85
Figure 19: Model objects UML Classes Diagram (2) 86
Figure 20: BPELReader UML Classes Diagram 87
Figure 21: Parsing process UML Activity Diagram 88
Figure 22: BPELWriter UML Classes Diagram 89
Figure 23: Writing process UML Activity Diagram 89
Figure 24: org.eclipse.bpel.ui UML Package Diagram 90
Figure 25: BPELEditor UML Classes Diagram 91
Figure 26: Actions hierarchy UML Classes Diagram 92
Figure 27: Adapters hierarchy UML Classes Diagram 92
Figure 28: Commands hierarchy UML Classes Diagram 93
Figure 29: Dialogs hierarchy UML Classes Diagram 94
Figure 30: Editparts hierarchy UML Classes Diagram 94
Figure 31: Factories UML Classes Diagram 95
Figure 32: Variable creation from the GUI (1) 96
Figure 33: Variable creation from the GUI (2) 97
Figure 34: Variable creation UML Communication Diagram (1) 97
Figure 35: Variable creation UML Communication Diagram (2) 98
Figure 36: Variable creation UML Communication Diagram (3) 99
Figure 37: Link creation from the GUI (1) 99
Figure 38: Flow link creation UML Communication Diagram (1) 100
Figure 39: Flow link creation UML Communication Diagram (2) 100
Figure 40: Flow link creation UML Communication Diagram (3) 101
Figure 41: Link creation from the GUI (2) 101
Figure 42: BPEL-D Designer new cases UML Use Cases Diagram 104
Figure 43: New Model Objects UML Classes Diagram 107

Figure 44: Model elements hierarchy UML Classes Diagram 108
Figure 45: Contain solvers UML Classes Diagram 108
Figure 46: New editparts UML Classes Diagram 109
Figure 47: New adapters UML Classes Diagram 110
Figure 48: New commands UML Classes Diagram 111
Figure 49: New dialogs UML Classes Diagram 111
Figure 50: New editparts UML Classes Diagram (2) 112
Figure 51: Container creation UML Classes Diagram 113
Figure 52: Container creation from the GUI (1) 114
Figure 53: Container creation UML Communication Diagram (1) 114
Figure 54: Container creation from the GUI (2) 114
Figure 55: Container creation UML Diagram Communication (2) 115
Figure 56: Container creation UML Diagram Communication (3) 116
Figure 57: Datalink creation from the GUI (1) 116
Figure 58: Datalink creation from the GUI (2) 117
Figure 59: Datalink creation UML Communication Diagram 117
Figure 60: Datalink creation from the GUI (3) 118
Figure 61: Splitting up a data link across local processes [KL06] 120
Figure 62: Participants UML Classes Diagram 122
Figure 63: Splitting process UML Classes Diagram 122
Figure 64: Splitting the process from the GUI (1) 123
Figure 65: Splitting the process from the GUI (2) 123
Figure 66: Splitting algorithm UML Activity Diagram 124

Table of Listings

Listing 1: WSDL Datalink example 21
Listing 2: WSDL Map syntax [WSFL] 22
Listing 3: WSDL DataLinkType syntax [WSFL] 22
Listing 4: BPEL Variable definition 30
Listing 5: BPEL Variables declaration section 31
Listing 6: BPEL Message 33
Listing 7: BPEL-D Container definition (1) 33
Listing 8: BPEL-D Container definition (2) 33
Listing 9: BPEL container structure samples (left) mapping separately (right) mapping directly 34
Listing 10: Messages sample 35
Listing 11: BPEL-D Invoke with containers 36
Listing 12: BPEL-D Invoke with containers (2) 36
Listing 13: Different container designs (left) following first option (right) following second option 37
Listing 14: BPEL-D Container definition (3) 38
Listing 15: BPEL-D Containers definition 38
Listing 16: FDL Map 41
Listing 17: WSFL Datalink 41

Listing 18: BPEL-D Datalink (1) 42
Listing 19: BPEL-D Datalink (2) 42
Listing 20: BPEL-D Datalinks section 43
Listing 21: BPEL-D Map syntax 44
Listing 22: BPEL-D Maps syntax 44
Listing 23: BPEL-D Datalink syntax 45
Listing 24: BPEL-D Datalinks syntax 45
Listing 25: BPEL Receive activity 46
Listing 26: BPEL-D Receive activity with containers 49
Listing 27: BPEL Invoke activity 49
Listing 28: BPEL-D Invoke activity with containers 50
Listing 29: BPEL Reply activity 50
Listing 30: BPEL-D Reply activity with containers 51
Listing 31: BPEL Assign activity 51
Listing 32: BPEL From definition 52
Listing 33: BPEL-D Assign activity with containers 53
Listing 34: BPEL-D From definition 53
Listing 35: BPEL-D Assign activity with containers 54
Listing 36: BPEL Throw activity 54
Listing 37: BPEL-D Throw activity with containers 55
Listing 38: BPEL-D Switch activity with containers 62
Listing 39: BPEL Pick activity 63
Listing 40: BPEL-D Pick activity with containers 64
Listing 41: BPEL While activity 65
Listing 42: BPEL-D While activity with containers 67
Listing 43: BPEL Sequence activity 67
Listing 44: BPEL-D Sequence activity with containers 69
Listing 45: BPEL Flow activity 69
Listing 46: BPEL-D Flow activity with containers 70
Listing 47: BPEL-D Scope with containers 71
Listing 48: BPEL Fault Handler 71
Listing 49: BPEL-D Fault Handler with containers 72
Listing 50: BPEL Compensation Handler 73
Listing 51: BPEL-D Compensation Handler with containers 74

Table of Tables

Table 1: Mapping options 35
Table 2: Structured activities input containers options 60

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.

(Javier Vazquez)