Prototype for Executable EAI Patterns

Xin Yuan

Studiengang: Informatik
Prüfer: Prof. Dr. Frank Leymann
Betreuer: Dipl.-Inf. Thorsten Scheibler
Beginn am: 01. August 2007
CR-Nummer: D.2.11, D.3, H.4.1
Abstract

The workflow of enterprises usually involves the integration of multiple applications from various systems. To lower the difficulty of integration, Enterprise Integration Patterns could be helpful. Each pattern poses a specific design problem, discusses the considerations surrounding the problem, and presents an elegant solution that balances the various forces. The BPEL language is used to model the workflows. The objective of the thesis is to find BPEL representation of several EAI patterns, also with the help of Web services. For this purpose, features of the EAI patterns are investigated and the patterns are parameterized.

An editor was already developed to support combining some EAI patterns to a messaging system and generating BPEL out of it. By extending the editor we are able to generate BPEL files for more patterns, which can be deployed and executed on the ActiveBPEL engine.
CONTENTS

INTRODUCTION 7

FUNDAMENTALS 9

2.1 ENTERPRISE APPLICATION INTEGRATION 9
2.2 MESSAGING AS INTEGRATION STYLE 11
  2.2.1 INTRODUCTION OF MESSAGING 11
  2.2.2 WHY USE MESSAGING? 13
2.3 ENTERPRISE INTEGRATION PATTERNS 14
2.4 BPEL 16
  2.4.1 WEB SERVICES 16
  2.4.2 BUSINESS PROCESS EXECUTION LANGUAGE (BPEL) 16

PARAMETERIZATION OF PATTERNS 19

3.1 PREVIOUS WORK 19
3.2 THE NOTATION 20
3.3 COMPARISON OF THE PaF ARCHITECTURE AND WORKFLOW 21
3.4 PATTERNS 23
  3.4.1 SYSTEM MANAGEMENT PATTERNS 23
  3.4.1.1 Control Bus 25
  3.4.1.2 Detour 29
  3.4.1.3 Wire Tap 30
  3.4.1.4 Message History 32
  3.4.1.5 Message Store 33
  3.4.1.6 Smart Proxy 35
  3.4.1.7 Test Message 37
  3.4.1.8 Channel Purger 43
  3.4.2 MESSAGE ROUTING PATTERNS 45
  3.4.2.1 Composed Message Processor 45
  3.4.2.2 Scatter-Gather 47
  3.4.2.3 Routing Slip 48
  3.4.2.4 Process Manager 50
  3.4.2.5 Message Broker 52
  3.4.2.6 Join Router 54
3.5 COMBINATION OF PATTERNS 55
  3.5.1 COMBINATION POSSIBILITIES 56
  3.5.1.1 System Management Patterns 56
  3.5.1.2 Message Routing Patterns 57
  3.5.2 COMBINATION EXAMPLES 58
  3.5.2.1 Channel Purger and Message Store 59
  3.5.2.2 Composed Message Processor and Scatter-Gather 59
Chapter 1

Introduction

Enterprises are typically comprised of many applications in multiple tiers of different operating system platforms. Enterprise Application Integration (EAI) is regarded as a good solution to build an large enterprise application system and to support common business processes and data sharing across applications.

Integrated applications are independent programs that can each run by themselves and are coordinated in a loosely coupled way. The mix of technologies and the distributed nature of EAI solutions make deployment, monitoring, and trouble-shooting of the system very complex. Therefore we use EAI patterns to model the integration. By combining the patterns, a messaging system that resembles the coordination of applications can be established.

Workflows are used to model business processes within enterprises and between enterprises. Business Process Execution Language (BPEL) is an extensible workflow-based language that aggregates services by choreographing service interactions. It is able to control the data flow passing through multiple nodes according to a predefined process model, while each node may be a data processing step or represent the interaction with an external web service. As BPEL describes how applications are integrated, the EAI patterns could be a good reference when designing BPEL processes.

The possibilities to simulate EAI patterns with BPEL and web services have been investigated in several works [9][12]. Patterns of various types have already been parameterized and the BPEL realization of them have been suggested. This thesis concentrates on some patterns related to the message routing and the management of the messaging system. We will discuss the possible BPEL representation of these EAI patterns at a abstract level.

In the previous works a system “EAI to BPEL” was developed, which is in essence an editor used for constructing messaging systems composed of EAI patterns. The editor can map the messaging system to BPEL code, and generate corresponding BPEL file
and WSDL file for the EAI messaging system. Our task also includes extending the editor so that it supports more patterns, and running the generated BPEL.

The theoretical part of our work is the parameterization of EAI patterns and the mapping from the patterns to BPEL. The practical part consists of the BPEL implementation of some newly parameterized EAI patterns and the deployment and execution of BPEL processes.

The rest of the thesis is structured as follows. Chapter 2 introduces the fundamental knowledge about the enterprise application integration, the messaging system, the EAI patterns and BPEL. In Chapter 3 we explain the parameterization of several EAI patterns and the ideas on how to represent them with BPEL and web services. Chapter 4 describes the usage of the editor “EAI to BPEL” and the modification we made to its implementation. It is also mentioned how to deploy and execute the BPEL files generated by the editor on the ActiveBPEL engine [20]. Finally, we summarize the thesis and bring up possible further works in Chapter 5.
Chapter 2

Fundamentals

The following chapter introduces the fundamentals of concepts and technologies which are related to this thesis. The main concerns are the features of Enterprise Integration Patterns and Business Process Execution Language.

2.1 Enterprise Application Integration

The concept enterprise application is used to describe a system with high performance requirement which supports simultaneous access to large amounts of data, provides various end-user service interfaces and enables efficient, reliable, and secure data exchange with other enterprise applications [1].

However, it is very hard to develop and maintain a single, big enterprise application to run the complete business. Secondly, since under different circumstances we hold different criteria for optimal applications, spreading business functions across multiple applications provides the business with more flexibility in meeting the continuously changing requirements. As a consequence, Enterprise Application Integration (EAI) is regarded as a good solution to build an large enterprise application system and to support common business processes and data sharing across applications.

Enterprises are typically comprised of hundreds, if not thousands, of applications that are custom built, acquired from a third party, part of a legacy system, or a combination thereof, operating in multiple tiers of different operating system platforms [2]. These disparate applications can be integrated so that they work together to produce a unified set of functionality.

When building an integrated enterprise application system we must efficiently reuse what already exists besides adding new applications and data. Integrating the
applications and data sources must be accomplished without requiring significant changes to these existing applications and the data. Some of the applications may be run outside of the enterprise by business partners or customers. Other applications might not have been designed with integration in mind and are difficult to change. These issues and other like them make application integration complicated [2].

Unlike creating a single application with a distributed architecture several computers, integrated applications are independent programs that can each run by themselves. The coordination between them are in a loosely coupled way, which will be discussed later in the following sections.

Despite the loose coupling of applications, users such as customers, business partners, and internal users generally do not sense the boundaries between applications in the system when they interact with them. The business functions that are executed by them may cut across several internal sections or applications of the system, which is often the case. But even if the business processes span several enterprise applications, they are still single business transactions from the customer’s view.

The definition of integration is very broad, it means connecting computer systems, companies, or people. The following six types of integration are repeatedly encountered, many integration projects consist of a combination of multiple types [2].

- Information portals
- Data replication
- Shared business functions
- Service-oriented architectures
- Distributed business processes
- Business-to-business integration

**Information Portal**

In many cases the customers have to access more than one system to retrieve certain information or to perform a single business function. Information portals are applications that provide users a single gateway to personalized information so that it is unnecessary to access multiple systems.

**Data Replication**

Many business systems require access to the same data which may be used and stored individually in different systems. When a customer change the information in one system, all these systems need to change their copy of the same data. This can be accomplished by implementing an integration strategy based on data replication.

**Shared business functions**

In the same way that many business applications store redundant data, they may also implement redundant functionality. These functions can be integrated to a shared business function so that it only needs to be implemented once and afterwards the shared function is available as a service to other systems.

**Service-Oriented Architecture**
Shared business functions are often referred to as services. A service is a well-defined function that is universally available and responds to requests from service consumers. First, applications need some form of service directory, a centralized list of all available services. Second, each service needs to describe its interface in such a way that an application can negotiate a communications contract with the service. These two functions, service discovery and negotiation, are the key elements that make up a service-oriented architecture (SOA) [2].

**Distributed Business Process**

A single business transaction is often spread across many different systems. In most cases, all relevant functions of the transaction are supported by applications in those systems. Thus a business process management component can be added to manage the execution of distributed business processes and to coordinate between the relevant applications.

**Business-to-Business Integration**

Sometimes business functions may also be available from outside suppliers, on the other hand, a system can provide application services to its business partners. Therefore, integration are not limited between applications inside a single enterprise, but occurs between business partners as well.

### 2.2 Messaging as Integration Style

There are various application integration styles which have their own advantages and disadvantages judging with several different criteria. Our concern in this section is the adoption of messaging as an integration style.

#### 2.2.1 Introduction of Messaging

Messaging is a technology that enables asynchronous and program-to-program communication with reliable delivery. Typically, one application sends a message to a common message channel, while other applications can read the message from this channel later. The configuration of the channel and the format of the message must be known to all relevant applications.

The asynchronous feature of messaging can be compared with voice mail. With voice mail, when the receiver does not answer, the caller can leave him a message. The message is queued in the mailbox until the receiver listens to it later, which is easier than trying to connect the receiver and the caller at the same time. Messaging works in such an asynchronous way as well.

Programs communicate by sending packets of data called *messages* to each other. *Channels*, also known as queues, are logical pathways that connect the programs and convey messages. A channel behaves like a collection or array of messages, but one that is magically shared across multiple computers and can be used concurrently by
2.2 MESSAGING AS INTEGRATION STYLE

multiple applications. A *sender* or producer is a program that sends a message by writing the message to a channel. A *receiver* or consumer is a program that receives a message by reading (and deleting) it from a channel [2].

The content of messages is usually information stored in some sort of data structure such as string, record or object. A message can contain simply the data to be processed by an application, or the description of a command to be invoked on the receiver, or the description of an event that occurred in the sender. Messages consist of two parts, a header and a body. The header contains information about the message like the identifier of sender, the destination of message and so on, which is generally used by the messaging system and mostly ignored by the applications using the messages. The body contains the data that is usually used by the applications and in most cases ignored by the messaging system.

Message capabilities are typically provided by a separate software system called a messaging system or message-oriented middleware (MOM), which manages messaging the way a database system manages data persistence. Firstly the channels must be configured to define the paths of communication between the applications, then the messaging system can coordinate and manage the sending and receiving of messages. The main task of a messaging system is to move messages from the sender’s computer to the receiver’s computer in a reliable fashion [2].

Actually, the messaging system is by all means necessary to ensure the reliability of message transmission between computers, because several types of transmission failure may occur without the management and coordination. For example, the sending application never knows whether the receiving application is ready for the message or not if they do not have extra communication for that. Moreover, the network may not be working or may fail to transmit the data properly. A messaging system can overcomes these limitations by various means, such as keeping a record of the transmission status or repeatedly trying to send messages once more until it is finally received.

In essence, a message is transmitted in the following five steps [2]:

- *Create* – The sender creates the message and populates it with data.
- *Send* – The sender adds the message to a channel.
- *Deliver* – The messaging system moves the message from the sender’s computer to the receiver’s computer, making it available to the receiver.
- *Receive* – The receiver reads the message from the channel.
- *Process* – The receiver extracts the data from the message.

It is noticeable that the job of message delivery are taken by the messaging system via the channel, whereas the sender and the receiver only have to interact with the channel in order to add or retrieve messages. During the transmission, there are two important messaging concepts:

*Send and forget* – Once the send step is completed, the sender can go on with other work while the messaging system transmits the message in the background. The
sender can be confident that the receiver will eventually receive the message and does not have to wait until that happens.

*Store and forward* – When the sending application sends the message to the channel, the messaging system stores the message on the sender’s computer, either in memory or on disk. In the *deliver* step, the messaging system delivers the message by forwarding it from the sender’s computer to the receiver’s computer, and then stores the message once again on the receiver’s computer. This store-and-forward process may be repeated many times as the message is moved from one computer to another until it reaches the receiver’s computer [2].

### 2.2.2 Why Use Messaging?

An enterprise has multiple applications that are built independently, with different languages, platforms and data formats. In order to share data and processes in a responsive way, all integration solutions have to deal with information transmission under such circumstances. On the other hand, networks could be slow or unreliable because of delays and interruptions. To overcome those challenges, four main integration approaches are usually used:

- **File Transfer** – have each application produce files of shared data for others to consume and consume files that others have produced.

- **Shared Database** – have the applications store the data they wish to share in a common database.

- **Remote Procedure Invocation** – have each application expose some of its procedures so that they can be invoked remotely, and have applications invoke those to initiate behavior and exchange data.

- **Messaging** – have each application connect to a common messaging system, and exchange data and invoke behavior using messages [2].

Each style has its advantages and disadvantages, thus it would be better to choose the best style for a particular integration scenario than using one style every time. Many decision criteria must be considered when determining a suitable integration style. The applications should reduce their dependencies on each other so that the modifications on them do not bring problems to the other applications. During the integration, the changes to the applications and the amount of integration code should also be minimized. In addition, it is often necessary to support sharing of functionality to provide better abstraction between the applications. Since there are a large number of criteria, applications can integrate using multiple styles so that each part of integration suits some special criteria. As a result, many integration approaches are a hybrid of multiple integration styles.

For the integration patterns that we discuss later, we choose *Messaging* as the integration style for the following reasons:
First, in many cases the sharing of functionality must be supported by the integration. However, File Transfer and Shared Database only enable applications to share their data but not their functionality. Remote Procedure Invocation enables applications to share functionality, but it tightly couples them.

Second, File Transfer allows you to keep the applications well decoupled but the collaborative behavior is too slow. Shared Database keeps data together in a responsive way but at the cost of coupling everything to the database.

Third, remote calls are slower than local calls. You also do not want one application’s failure to bring down all of the other applications, and you do not want each application to know the details about other applications. Thus Remote Procedure Invocation still seems an awkward choice [2].

Asynchronous messaging, on the contrary, may solve some problems of application integration. Sending a message does not require both applications to be ready at the same time. Furthermore, messages can be transformed in transit while either the sender or the receiver do not have to know about the transformation, so that the sending and the receiving applications can have different conceptual models. Besides, applications are able to share data rapidly by sending small messages frequently. The decoupling also allows integrators to choose between broadcasting messages to multiple receivers, routing a message to one of many receivers, or other topologies, which helps to separate integration decisions from the development of the applications [2].

Apparently the messaging solution is not free of problems. Asynchronous design generally makes testing and debugging harder, and the independence between the decoupled applications often requires more costs and efforts (i.e. writing a lot of complicated code) to fit everything together.

2.3 Enterprise Integration Patterns

While developing an EAI solution is no easy task, it could be even more challenging to operate and maintain such a solution. The mix of technologies and the distributed nature of EAI solutions make deployment, monitoring, and trouble-shooting of the system very complex. To lower the difficulty of integration, Enterprise Integration Patterns could be helpful.

A pattern is a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces [3].

Each pattern poses a specific design problem, discusses the considerations surrounding the problem, and presents an elegant solution that balances the various forces. In most cases, the solution is not the first approach that comes to mind, but one that has evolved through actual use over time. As a result, each pattern incorporates
the experience base that senior integration developers and architects have gained by repeatedly building solutions and learning from their mistakes. This implies that patterns are not invented, but rather discovered and observed from actual practice in the field [2].

Enterprise integration patterns have been researched by various people and organizations. The most famous books in this field are “Enterprise Integration Patterns” by G.Hohpe [2] and “Patterns of Enterprise Application Architecture” by Martin Fowler [4]. IBM [5] and Microsoft [6] also published their patterns online.

Since enterprise integration spans many problem domains and levels of abstraction. It makes sense to divide enterprise integration patterns into categories that reflect the scope and abstraction of the patterns. The work of this thesis is based on Hohpe’s book [2], which discusses patterns with Messaging as their integration style. Therefore, the architecture of messaging systems plays an important role in the following pattern categorization.

Message Patterns
Messages contain the information to be exchanged between applications. Message Patterns mainly describe message type and structure

Channel Patterns
Channel is the virtual pipe that connects the sender and the receiver and thereby enables the transmission of messages. All channels must be created and configured before a message system starts to function as designed. Channel Patterns describe different types of channels and communication mechanisms the channels can provide.

Transformation Patterns
Since the sending application and the receiving application may not agree on a common format for the same conceptual data, data may have to be added, taken away or existing data may have to be rearranged. This task is usually taken by a Message Translator. Transformation Patterns describe various translators which add, remove or rearrange the content of messages.

Endpoint Patterns
Most applications do not interact directly with other applications. They usually contain a special part called Message Endpoint, which functions as a bridge between the application and the messaging system and enables the application to send and receive messages. Endpoint Patterns describe the behavior of messaging system clients and illustrate different ways in which applications can produce or consume messages [2].

Routing Patterns
In a large enterprise with numerous applications and channels to connect them, a message may have to go through several channels to reach its final destination. The origin sender sends the message to a Message Router, an application component which determines how to navigate the channel topology and directs the message to the receiver or the next router. Messages may be split up, routed separately and re-merged
later, they can also be duplicated and sent to several destinations. Routing Patterns describe how various types of routers perform their task [2].

**System Management Patterns**

A large amount of messages are generated, routed, transformed and consumed in a message-based integration solution. Error conditions, performance bottlenecks and changes in the participating systems may occur time to time and must be handled with management mechanisms. Some statistical information about the system status also need to be collected and recorded. System Management Patterns provide the tools to keep messaging systems running.

### 2.4 BPEL

#### 2.4.1 Web Services

The goal of the Web services is to achieve interoperability between applications by using Web standards. Loosely coupled integration model with Web services technology allows flexible integration of heterogeneous systems in a variety of domains including business-to-consumer, business-to-business and enterprise application integration [8].

Generally, a Web service is a service available at a particular endpoint in the network, it receives and sends messages and exhibits behavior according to its specification. The service has specific functionality and is deployed with appropriate quality of service at the endpoint. The functional aspects of a service are specified using Web Services Description Language (WSDL), and the constraints and conditions that are associated with the use of the service are specified via policies that can be attached to various parts of the WSDL [7].

WSDL is an XML format for describing services as a set of endpoints that operate on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate [7].

#### 2.4.2 Business Process Execution Language (BPEL)

The full potential of Web services as an integration platform will be achieved only when applications and business processes are able to integrate their complex interactions by using a standard process integration model. The interaction model that is directly supported by WSDL is essentially a stateless model of request-response or uncorrelated one-way interactions. However, models for business interactions
typically assume sequences of peer-to-peer message exchanges, both request-response and one-way, within stateful, long-running interactions involving two or more parties. To define such business interactions, a formal description of the message exchange protocols used by business processes in their interactions is needed [8].

Business Process Execution Language (BPEL) was first known as BPEL4WS but soon renamed as WS-BPEL. It is an extensible workflow-based language that aggregates services by choreographing service interactions. The aggregation is recursive, such that the process exposes WSDL interfaces to those that interact with it, and the corresponding services may be used in other choreographies. Designed to work in a highly dynamic environment in which services might change frequently, a BPEL process is intentionally decoupled from particular instances of the services it choreographs [7].

BPEL utilizes several XML specifications: WSDL 1.1, XML Schema 1.0, XPath 1.0 and XSLT 1.0. WSDL messages and XML Schema type definitions provide the data model used by BPEL processes. All external resources and partners are represented as WSDL services [8].

Data in BPEL is written to and read from lexically scoped, typed variables. The values of these variables are either messages exchanged between the process and its partners, or intermediate data that is private to the process. BPEL variables are typed using WSDL message types, XML Schema simple types, or XML schema elements. XPath is the default language for manipulating and querying variables [7].

The major building blocks of BPEL business processes are nested scopes that contain relationships to external partners, declarations for process data, handlers for various purposes and most importantly the activities to be executed [7].

The activities in a BPEL process are either structured or basic. Structured activities contain other activities and define the business logic between them. Basic activities are, for example, the inbound or outbound Web service interactions or the specific activities for data manipulation. Activities that deal with Web services are receive, reply, pick, invoke, and event handlers. These activities allow a process to exchange messages with the services that it composes. The assign activity copies data from one location to another [7].

Structured activities combine multiple activities to provide higher-level business logic. These include sequence, switch, while, and flow activities. The sequence activity is a simple aggregation of activities executed in the order in which they are specified. The switch activity provides a multi-branch decision construct, and the while activity provides a loop construct. Both switch and while have semantics from corresponding programming language constructs, such as Java. Activities can be executed in parallel when nested in a flow activity. Furthermore, conditional control links can be specified to form a partial (acyclic) order on a set of such parallel activities. A transition condition can be associated with each control link and evaluated at completion of the link's source activity [7].
In addition to Web service interactions and nested structured activities, you can use a couple more basic activities to specify different types of behavior. These activities are called *empty*, *wait*, *terminate*, *throw*, and *compensate*. The *empty* activity can be used as a placeholder for other activities that you add later. The *wait* activity provides a means to interrupt the execution for a specified time interval or until a specified time has been reached, and the *terminate* activity stops the execution of the process immediately. Finally, the *throw* and *compensate* activities are used in conjunction with handlers for error detection and recovery. A BPEL process can also contain handlers to recognize unexpected problems and deal with them. These handlers can include compensation operations that reverse effects from other activities that have been completed successfully before an error situation occurred [7].

A BPEL process refers to the parties that it is interacting with as partners. It interacts with each partner along a set of *partnerLinks*. Partner links are instances of typed connectors that specify the *portTypes* that the process offers to and requires from the partner at the other end of the link. A partner link can be thought as a channel along which a peer-to-peer conversation with a partner takes place. When a process executes, each partner link along which a partner is invoked must be bound to a concrete endpoint. Four binding schemes and their combinations are possible: static design time binding, static deployment time binding, dynamic binding using lookups and dynamic binding using passed-in endpoints [7].

Business processes that are defined in BPEL represent stateful Web services, and might have long-running conversations with other Web services. Whenever you start a new BPEL process, a new instance of that process is created, which might communicate with other business partners. After a process was defined and deployed, you can create multiple instances of it. All these instances can run concurrently and completely independently of each other [7].

The creation and destruction of BPEL process instances is by design implicit. An instance can be created if a partner invokes an operation of the process that corresponds to one of these activities. When the last activity of an instance completes, a terminate activity executes, or the instance experiences a fault from which it cannot recover [7].
Chapter 3

Parameterization of Patterns

In the following chapter the parameterization of EAI patterns is introduced. For each pattern involved, we discuss what inputs it gets from other component, what outputs it delivers to other components, which parameters should be defined by users so that the pattern can function correctly, and the possibility to be configured in the runtime with a certain management console.

Firstly the parameterization is researched at the abstract level, that is, the parameters should reflect the generic characteristics of the patterns, and the main focus is to establish pattern models independent of the concrete implementation methods. Secondly, after the abstract parameterization, we also give suggestions on how the patterns could be realized with BPEL and web services.

3.1 Previous Work

This thesis is based on the works of Florian Schebelle [12] and Bettina Druckenmueller [9]. Florian Schebelle studied the parameterization of selected EAI patterns in several categorizations and designed the mapping mechanism from them to BPEL processes, whereas Bettina Druckenmueller provided a good extension to Florian Schebelle’s work, modeling a variety of patterns in a formal way.

Besides, Florian Schebelle developed a JAVA framework which generates BPEL code for EAI patterns. Bettina Druckenmueller improved the framework so that more patterns are supported and the generation of corresponding WSDL files is enabled. The BPEL standard used in Florian Schebelle’s work was BPEL 1.1 (BPEL4WS), which was upgraded to BPEL Standard 2.0 (WSBPEL) by Bettina Druckenmueller. Details about the implementation will be described in the next chapter.

The parameterized patterns from their theses are listed in the following table:
### 3.2 The Notation

In order to clarify the parameterization, we use a graphical representation of the pattern (shown in Figure 3.1) which illustrates its inputs, outputs and the parameters.
3.3 COMPARISON OF THE PAF ARCHITECTURE AND WORKFLOW

Figure 3.1 Graphical representation of a pattern and its parameters

Usually the pattern itself is illustrated by an icon in the gray box. It has inputs on the left and outputs on the right, which are usually the messages passing through the pattern. Above the gray box we list the parameters related to the configuration of the pattern and other management functionalities. The direction of the arrow indicates whether the control message used for management is transmitted to the component, or to the central console, or there are different types of control messages in both directions. Under the gray box are all the other parameters of the pattern.

Sometimes we do not want to abstract a certain complicated pattern by a single icon, since it may contain multiple components, and we have interest on the inner structure of the pattern as well. In such cases a figure showing the connections between those sub-components is presented instead of the gray box.

3.3 Comparison of the PaF Architecture and Workflow

We not only parameterize the EAI patterns at the abstract level, but also discuss possible BPEL representation of them. In order to have a better understanding of the BPEL realization, it would be helpful to take a look at first at the similarity and differences between the PaF architecture used by EAI messaging system and the architecture of workflow processes.

In a messaging system of the Pipes-and-Filters (PaF) architecture (as illustrated in Figure 3.2), messages are forwarded through a series of pipes and filters sequentially,
3.3 COMPARISON OF THE PAF ARCHITECTURE AND WORKFLOW

while a filter usually represents a message processor and a pipe connects filters and supports the transmission of messages from one end to the other.

We recognize that the messaging system of EAI patterns is a system of the PaF architecture. The message processing or routing components can be regarded as the filters and the channels between them have similar functionalities as the pipes.

The similarity between the illustration of PaF architecture and workflow graphs is obvious, since the workflow graph consists of nodes standing for activities which are connected via control links. However, both architectural styles have fundamental differences [13]. The main distinction was found to be in the support of instances. In workflow based systems an incoming request message starts a new instance of a business process. Hence, messages are associated with a particular workflow instance in contrast to PaF where messages traverse the filters in such a way: when a filter receives a message it processes it, forwards it, and processes the next upcoming message immediately [12].

Figure 3.3 shows the relationship of a workflow process and its instances. Multiple running instances of a single workflow process model can work on different activities respectively at the same time. Once an instance is created, it can only behave as previously defined by the process model. As a result, if we modify the process model in order to change the configuration setting of an activity, all the running instances of this process cannot follow this modification automatically.
This characteristic of the workflow system brings difficulty to the BPEL realization of some dynamically changing contents of EAI patterns. A possible solution is to introduce context information into workflows [11], which provides a mechanism for the central console to actively manipulate running instances by sending event notifications (such as a state change) to those instances that are waiting for the event. However it requires the extension of BPEL standard and the support of Nexus Platform [14]. Alternatively we can overcome this problem by another means, which will be explained in the next section in detail.

3.4 Patterns

The main concerns of the thesis are all the system management patterns and the five message routing patterns that were not parameterized in Bettina Druckenmueller’s work [9]. In addition, a practical routing pattern not mentioned by Hohpe [2] is also introduced here, as it is helpful to the creation of a system management scenario. All those parameterized patterns are listed in the Table 3.2.

<table>
<thead>
<tr>
<th>System Management:</th>
<th>Control Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Detour</td>
</tr>
<tr>
<td></td>
<td>Wire Tap</td>
</tr>
<tr>
<td></td>
<td>Envelope Wrapper</td>
</tr>
<tr>
<td></td>
<td>Message History</td>
</tr>
<tr>
<td></td>
<td>Message Store</td>
</tr>
<tr>
<td></td>
<td>Smart Proxy</td>
</tr>
<tr>
<td></td>
<td>Test Message</td>
</tr>
<tr>
<td></td>
<td>Channel Purger</td>
</tr>
<tr>
<td>Messaging Routing:</td>
<td>Composed Message Processor</td>
</tr>
<tr>
<td></td>
<td>Scatter-Gather</td>
</tr>
<tr>
<td></td>
<td>Routing Slip</td>
</tr>
<tr>
<td></td>
<td>Process Manager</td>
</tr>
<tr>
<td></td>
<td>Massage Broker</td>
</tr>
<tr>
<td>Supporting Pattern:</td>
<td>Join Router</td>
</tr>
</tbody>
</table>

Table 3.2 Patterns parameterized in this thesis

3.4.1 System Management Patterns
In our messaging systems, components are distributed across many platforms and may reside at multiple locations. The loose coupling feature of integration solution makes testing and debugging of the system especially hard. Messages are delivered asynchronously. Besides, message senders may not know who consume the messages since in most cases the recipients do not send any reply messages to the senders. Moreover, the messaging mechanism typically only guarantees the delivery of messages, but not the delivery time. All of the facts mentioned above increase the difficulty to develop test cases that rely on the results of the message delivery.

System management patterns are designed to provide tools to keep message-based systems running. They are divided into three categories according to their functionalities: monitoring and controlling, observing and analyzing message traffic, and testing and debugging [2].

**Monitoring and Controlling**

The monitoring of a messaging system mainly focuses on the following two aspects. First, system management solution monitors the amount of messages being sent and the waiting and delivery time of messages, on the other hand. Second, sometimes it is also required to inspect the data contained in messages.

A *Control Bus* (see section 3.4.2.1) provides a single point of control to the management and monitoring of distributed systems by connecting multiple components to a central management console.

When we need to route messages through additional steps which can be dynamically switched on or off via the control bus, we can use a *detour* (see section 3.4.2.2).

**Observing and Analyzing Message Traffic**

With a *Wire Tap* (see section 3.4.2.3) messages can be copied to a secondary message flow so that their contents can be inspected without affecting the primary message flow.

The *Message History* (see section 3.4.2.4) keeps a log of all the components that the message has passed. It is stored in individual messages, when we need fully record every message that travelled through the system, we can save the records in a central *Message Store* (see section 3.4.2.5).

The three patterns mentioned above enable the analysis of message flow between components in the system. When messages are transmitted between the system and a service, the reply messages from the service to the invoking components in the system can be tracked and observed by a *Smart Proxy* (see section 3.4.2.6).

**Testing and Debugging**

In some cases observing the current messages flow is not sufficient to confirm the healthiness of the system, it is necessary to actively verify that the system is running properly by inserting a *Test Message* (see section 3.4.2.7) and analyzing the results.
When a component fails and the normal message flow interrupts, it is possible that unwanted messages are left on a channel. A Channel Purger (see section 3.4.2.8) can remove the remaining messages from a channel.

3.4.1.1 Control Bus

Administration and control are challenging in systems with distributed, loosely coupled architecture. Information of components’ status must be collected and combined from more than one machine. It is also necessary to make adjustments or change configuration settings when the system is running. Those tasks can be performed by transmitting control messages. For example, to modify the configuration of a component, a control message carrying modification information can be sent there from a management unit.

Control messages could be transported and routed just like a regular message. However, for security reasons it is possible that some control messages should not be viewed or processed like regular ones. In addition, control messages may have higher or lower priority than application messages. It is complicated to design and implement channels that can change the order of queued messages.

Control Bus is a message subsystem that supports the management of enterprise integration systems. It uses separate channels to transmit control messages that are relevant to the management of components while keeping the messaging mechanism of application messages. Therefore, each component in the system may be connected to two messaging subsystems: the application message flow and the Control Bus, as illustrated by Figure 3.4. The Control Bus does not have to reach every component that processes application messages, because it is unnecessary for some sorts of components to interact with the central management unit.

Control Bus is bi-directional. When a component sends periodic heartbeat messages, exception messages or statistics messages about the component’s running status or the application messages it processed, the control messages are transmitted from the component through Control Bus to the central management unit. When the settings of
3.4 PATTERNS

A component are to be modified, the management unit sends a configuration message through the Control Bus to that component.

The Control Bus connecting the central management unit and any single component can be regarded as a group of point-to-point sub-channels, each of them for one type of control messages. Figure 3.5 illustrates a sub-channel with its inputs, outputs and parameters.

![Sub-channel of Control Bus](image)

Similarly to the Datatype Channel, a sub-channel of Control Bus is unidirectional and transports only one type of messages. The input and output of the sub-channel are both control messages of a same message type which should be defined when creating or configuring the sub-channel.

The sub-channel can only contain a limited number of control messages in queue, thus its buffer size also need to be determined, so that it will not be overloaded with messages.

Furthermore, the logical address of the sub-channel must be defined. The address usually refers to a file, a database, or any data-storing media, through which the message sender and the message receiver can locate the sub-channel in order to retrieve messages from it or put messages into it. When a file is used as the storage of a channel, the logical address is the name and path of it. Similarly, different sorts of logical addresses work for different implementations of sub-channels.

Quality of services concerns various aspects of message delivery such as security and transaction. For example, data encryption may be required to conceal the contents of messages. It is also better to set a proper mechanism to handle extra messages when the buffer of sub-channel is full. In some cases messages need to transported within transactions. Whereas the transmission of individual messages in most messaging systems follows the ACID principle, transactions of a large group of messages must be well specified and implemented.
Figure 3.6 illustrates the *Control Bus* connecting the central management unit and a component as a set of such sub-channels. Generally the *Control Bus* achieves this functionality without any intermediate routers or transformers, supporting direct interaction between the management unit and components. Thus no parameters are needed to describe the connection topology, only every sub-channels and their directions need to be specified.

When we consider the whole system with the central management unit, the *Control Bus* and the components, it is noticeable that the messages from components are always going to one destination, while the management unit is distributing messages to sub-channels leading to diverse locations. Therefore a mapping table recording the outputs and the corresponding sub-channels must be established so that the management unit is able to find the correct channel.

The modeling of sub-channels is also influenced by different strategies of component configuration explained as follows:

1. The central management unit sends configuration messages to components to set the values of their configurable parameters, and the components only wait for the messages. When a common parameter is used by several components, it is usually duplicated and stored in each component. To synchronize the parameter, that is, to force the duplicates to always have the same value, the configuration messages must be sent to all the relevant components respectively each time the value is changed.

2. Alternatively the configuration can be started by a component by actively sending a query message to the central management unit, which then processes the request and returns a response message containing the values of relevant parameters. Since every component updates the parameter before making decisions depending on it, the management unit only have to wait for queries and communicate with individual components instead of synchronizing the parameter repeatedly.
For both strategies it is obvious that there must be sub-channels transporting configuration messages from the management unit to components. If the second strategy is adopted, channels of the opposite direction should also be created for query messages sent by the components.

**BPEL and Web Services**

In Bettina Druckenmueller’s thesis, a Datatype Channel can generally be realized with BPEL links or additional support of web services [9]. The sub-channels of Control Bus and the Datatype Channel [2] have some features in common, thus we discuss the possibility to make use of similar implementation concepts.

The flow activity of BPEL can contain a set of concurrent activities directly nested within it. The synchronization dependencies between activities are expressed with the link construct with source and target elements. When the source element of a link refers to activity a, and the target element refers to activity b, b must not start before the completion of a. A link may also have several sources or targets and a joinCondition element, which supports more complex dependencies. In brief, the BPEL links determine the order in which filters-patterns are executed.

For the application message flow, BPEL links work as control connectors, and the data connectors are implemented with intermediate variables storing messages. But if we try to simulate the Control Bus with BPEL links and variables, we face the following problem.

Suppose that the central management unit and the involved components are represented by activities within a flow like most routers or transformers, and links are used between them to define the execution order. It would be difficult to deal with sub-channels of different directions between the management unit and a single component. A direction implies a synchronization dependency which must be defined by a link, consequently there have to be links of both directions between two activities. However, the route formed by the links must be acyclic to ensure the unambiguous dependency relation. Hence this realization method does not work.

Another possibility is to use an external service to simulate the central management unit. Control messages are saved in BPEL variables of predefined message types, which are usually the inputs or outputs of operations. A component sends a message by invoking an operation provided by the service and setting the message as the input variable, while the receiving of the configuration message can be achieved by receive or pick activities. With such implementation the Control Bus is able to function properly in both directions and the direct interaction with the central management unit is supported.

Problem also arises when realizing the first strategy for the configuration of components. Since usually the changing of process model does not influence the running instances and it is also complicated to locate all these instances, the active controlling from the central management unit can only be implemented in a context-based workflow system mentioned in Section 3.3.
The second strategy makes the implementation much easier. As the components contact the service actively, all the running instances are automatically configured after the completion of the query procedure. The configuration is not accomplished at the same time in different BPEL instances, but such inconsistence does not cause any trouble, because the components are always updated with the newest setting value. This value should be stored on some persistent media (i.e. in an XML file) so that the external service is able to locate, read and modify it and the information can last after a system crash.

### 3.4.1.2 Detour

*Detour* is a *Context Based Router* controlled via the *Control Bus* which modifies the route that messages take. Its most typical application is to route a message through intermediate steps to perform validation or testing. As *Detour* is controlled by a central setting, whenever these extra steps are not required, bypassing them can be easily achieved to improve performance, and multiple *Detours* can be activated or deactivated simultaneously.

![Diagram of Detour](Figure 3.7 Parameterization of Detour)
As illustrated in Figure 3.7, the input and output of Detour is a message of the same message type, which must be specified. Generally, a Detour has one entry and two exits, but we could also extend it by adding exits for alternative validation or testing procedures. The mapping relationship between those exits and the channels connected with them should also be defined in a mapping table.

If the Detour uses the first configuration strategy, that is, passively waits for configuration messages, then the configuration message type must be specified. If the second strategy is adopted, the Detour sends a query message to the central management unit and receives a configuration message from it. In this case the types of both the query message and the configuration message should be defined. Apparently, the routing logic must be specified additionally so that different setting values are associated with their corresponding exits.

**BPEL and Web Services**

The realization of the configuration procedure must include a web service which functions as the management unit. The Detour sends a configuration query by invoking the operation provided by the service, then the web service processes the query, retrieves the setting value from an XML file and returns a configuration message containing the value.

Depending on the routing logic, the routing part of the Detour pattern can be implemented mainly with two options.

One option is to use BPEL exclusively, since BPEL supports all the basic operations needed such as extracting setting value from the configuration message and directing the application message to the right channel according to the setting value and the predefined, fixed routing logic.

Alternatively the Detour can also send the configuration message to a web service which analyzes the message and determines which exit is correct. This method is complicated and in most cases unnecessary, but it does support more flexible routing logic.

**3.4.1.3 Wire Tap**

A Wire Tap enables the inspection of messages that travel on a point-to-point channel. Normally extra listeners cannot be added to such a channel, because it consumes the message off the channel and consequently the recipient will not be able to get the message.

The Wire Tap is a fixed Recipient List [2] with two output channels, which is usually inserted into a point-to-point channel. One of its output channels leads to the intended recipient, additionally a secondary channel should be created and connected with the other exit. Each time the Wire Tap consumes a message off the input channel, it publishes one unmodified message to both output channels respectively. The
secondary channel can be turned on or off with control messages, so that the Wire Tap publishes messages to this channel only during testing or debugging.

![Diagram of Wire Tap parameterization](image)

Figure 3.8 Parameterization of Wire Tap

Figure 3.8 shows the parameterization of the Wire Tap. The inputs are application messages, and the outputs are the identical messages and possibly additional duplicates of those messages (when the secondary channel is on). As the message content is not changed within Wire Tap, the input and output messages should be of the same type. The connection from the two exits to the primary and secondary channels should also be defined in a mapping table.

The Wire Tap can be configured with either of the two strategies. If it passively waits for configuration messages, the configuration message type must be specified. If it sends a query message to the central management unit and receives a configuration message from it, then the types of the query message and the configuration message should be defined.

The routing logic is quite fixed and simple, thus there is no need to specify it. As the messages sent to both channels are identical, the Wire Tap only have to differentiate the two channels in order to close the secondary channel on the basis of the configuration setting.

**BPEL and Web Services**
The realization of configuration procedure should include a web service which functions as the central management unit. The *Wire Tap* sends a configuration query by invoking the operation provided by the service, then the web service processes the query, retrieves the setting value from an XML file and returns a configuration message containing the value.

The implementation of the rest part of *Wire Tap* pattern can be accomplished with pure BPEL code, since BPEL supports all the basic operations needed such as extracting the setting value from configuration message and directing the application message to the right channel according to the setting value and the predefined, fixed routing logic.

In case the secondary channel leads to an intermediate message processing component instead of directly to the central management unit, the *Wire tap* pattern can also be implemented with BPEL links like *Recipient List*.

### 3.4.1.4 Message History

Loose coupling is an architectural strength of message-based system. However, because usually messages are not associated with their original senders, we could have trouble with debugging and analyzing dependencies. To solve this problem, a *Message History* can be attached to the message to record all applications or components that the message passed through since its origination. Each time the message is processed by a component (including the original sender), a new entry with information of the current component is appended to the *Message History*.

![Diagram of Message History](image-url)

**Figure 3.9 Parameterization of Message History**
Figure 3.9 illustrates the parameterization of this pattern. At first we should decide which data of each component should be stored in an entry. Apart from the unique component identifiers, which is the main concern of *Message History*, extra information like timestamp may also be helpful for testing.

Details about where and how the *Message History* is saved must also be specified. Generally it is contained in message header in order to separate the system-specific control information within it from the application-specific data in the message body. Besides, the history can be realized in different data structures such as hierarchical trees or simple lists with the entries as their basic unit. Obviously the entries must be ordered as well to represent the route that messages passed.

Entry attaching mechanism describes how each component attaches its own entry to the *Message History*, which is strongly related to the history storage structure mentioned above. The history is a list of entries, however, problems arise in some scenarios, for example, when an aggregator gets several messages and aggregates their data together to a new message. Each of input messages has its own history, if we keep the *Message History* as a simple list, only the history of one incoming message can be recorded, the rest have to be abandoned. This approach can work if one incoming message plays a more important role than the others. If this is not the case, a storage structure that can combine the history of all incoming messages, such as a hierarchical tree, must be employed to ensure the completeness of *Message History*.

**BPEL and Web Services**

The BPEL implementation of *Message History* pattern is determined by implementation of the *Message* pattern [9]. Messages are usually stored in BPEL variables, and their types are defined in terms of WSDL message types or XML schema types. If the message structure is realized in such a way that BPEL variables contain both the message header and the message body, then BPEL or additional web services are enabled to access the element representing *Message History*, and the content of history can easily be changed by each component. However, BPEL cannot change the predefined message structure if it does not include a field for *Message History*.

**3.4.1.5 Message Store**

With the help of *Message History* pattern, the source of a message and the route it passed can be inspected. On the other hand, this pattern still has shortcomings. The history is only accessible during the short lifetime of messages, and the fact that the information is contained in individual messages also increases the difficulty of centralized message analyzing. A good alternative is to store message data persistently and in a central location – the *Message Store*. 
When we send a message to a channel, we send a duplicate of the message to a special channel leading to the *Message Store*, which can be achieved by the component itself or a *Wire Tap*. The application flow will not be influenced by the duplicated messages, but the network traffic does increase.

![Diagram of Message Store](image)

**Figure 3.10 Parameterization of Message Store**

As illustrated in Figure 3.10, the inputs of the *Message Store* are messages of various types from several incoming channels. It does not send any stored messages out to any component, except for the analysis of messages.

The storage capacity can be predetermined or changed dynamically in real time. The may affect other parameters of *Message Store*, such as the frequency of purging and the information saved for each message.

We want to store more details of messages for better reporting abilities. But since duplicated messages increase the network traffic and the total storage capacity is limited, it is wiser to extract the save the most important and concerned part of each message. If the extraction is not accomplished by the *Message Store* itself, a *Content Filter* [2] or a *Message Translator* [2] should be inserted into the channels that carry the duplicated messages in order to perform the same task.

Messages of different types are collected in the same *Message Store*, thus we need to consider practical storage options for them. It would make indexing and searching clear and simple, if we create a separate storage schema to match each message type’s internal data structure, but this may bring troubles to storage maintenance as well. On the other hand, it is also possible to use an XML repository, which can index the XML documents that store the message data for later retrieval and analysis.

The *Message Store* could be quickly filled with messages if we choose to keep all of them. A purging mechanism must be developed to move older message logs to a backup database or delete them altogether whenever necessary.
BPEL and Web Services

BPEL does not support direct interactions with any external storage media, so the solution is to realize this pattern with a web service.

When a component wants to save a message duplicate into Message store, it invokes the web service, which then stores the message in an external database or a file. The storage capacity is determined by the database or the file repository, and usually the purging mechanism should be manually specified.

3.4.1.6 Smart Proxy

Wire Tap can be used to track messages flowing through a component when the messages are transmitted via a fixed channel. When tracking messages flowing through a service, it is easy to capture a request message sent to the service, but intercepting reply messages is a tough problem because they are published to different channels according to the different Return Address [2] declared by the requestors.

To overcome the problem, we can use a Smart Proxy instead between the requestors and the service. It intercepts request messages, stores the Return Address of them and replaces it with the address of the Smart Proxy, then sends the modified request messages to the service. Naturally the reply messages are all returned to the Smart Proxy by the service, at this moment it may analyze the messages and report the result via the Control Bus. Afterwards it can retrieve the stored Return Address, and forward the reply messages to the original reply channels specified by Return Address.

Smart Proxy is not only helpful to message inspecting and analyzing, it is also useful when an external service can only reply to a fixed reply channel. In this case it functions as a proxy which reads messages from the fixed channel and forwards them to requestors according to the Return Address.

As illustrated in Figure 3.11, the Smart Proxy has one set of entry and exit for request messages and reply messages respectively, shown as the upper part and the lower part of the icon. The inputs and outputs of the upper part are request messages of the same type, while the inputs and outputs of the lower part are reply messages of the same type.

The address of Smart Proxy must be specified so that the service can send the reply messages to it.
In essence, the lower part of Smart Proxy functions as a message router with multiple exits. However, it is not necessary to know the total number of possible requestors, and the mapping from exits to reply channels does not need to be established, because all the reply channels are dynamically created on the basis of the Return Address just like the reply queue from an invoked service to the invoker.

After replacing the Return Address with the address of itself, the Smart Proxy can store the original reply address inside the messages in a newly created message field. When the message is returned from the service, the Smart Proxy extracts the Return Address and removes the field. This solution requires the service to accept the modification of message structure.
Alternatively the Return Address can be saved in a memory structure or a relational database. The Smart Proxy maintains the list and finds the Return Address in it when receiving a message from the service. The structure of messages is not changed with this approach. The methods to store and retrieve the Return Address should be predefined.

Two mechanisms enable the Smart Proxy to correlate the reply message back to the response message. Since most services copies the Correlation Identifier [2] from the request message to the reply message, the Smart Proxy only has to keep the identifier field untouched.

In case the requestors do not specify the Correlation Identifier or the supplied Correlation Identifier may not be unique, it would be safer for the Smart Proxy to generate a unique identifier and to use it for the interaction with the service. Pairs of the original Correlation Identifier and the new identifier are saved in the Smart Proxy, so that the identifier of reply messages can be restored.

When the purpose of the Smart Proxy is to analyze the request and reply messages, it sends report messages via Control Bus to the central management unit. Usually the analytical functions are mainly performed by the central unit, thus the Smart Proxy may have to send a large amount of information to it, unless the report criteria clarifies that only a small portion of messages have the value to be inspected. Anyway, the type of report messages should be determined.

**BPEL and Web Services**

The BPEL implementation is not so complicated as the pattern looks, actually it is usually only some additional activities based on an external service.

The requestors access the service individually from each BPEL instances, thus the function of the Smart Proxy can also be distributed in instances, not centralized managed. Since web services are always able to send reply message directly back to invokers, no proxy is needed, the only purpose to adopt a Smart Proxy is to intercept messages for analyzing.

It is easy to “intercept” messages by changing the input variable before the requestor invokes a service and changing the output variable of the service before it is used by other components, while the normal interaction between requestors and the service are not influenced. Consequently there is no need to consider the problem like the correlation mechanism or the modification and the management of Return Address.

The implementation of the reporting procedure should include a web service representing the central management unit. The Smart Proxy sends report messages by simply invoking the relevant operation provided by the service.

**3.4.1.7 Test Message**
3.4 PATTERNS

A component can inform the central management unit that it is still active by sending periodic heartbeat messages to it. However, it is insufficient to prove the component is running without any error, the Test Message mechanism should be used instead.

The Test Message pattern is an active monitoring system consisting of 4 components: Test Data Generator, Test Message Injector, Test Message Separator and Test Data Verifier.

Figure 3.12 Test Message Pattern [2]

Figure 3.12 shows the Test Message pattern as a whole system. The Test Data Generator creates messages to be sent to the component for testing. The Test Message Injector receives the test message and inserts it into the regular application flow, it also tag the test message to differentiate it from normal application messages. The component treats the test message as ordinary ones, then publishes the result message to the Test Message Separator, which directs two sorts of messages to their corresponding destinations. Finally the test message reaches the Test Data Verifier, which compares the result with expected result and reports an exception via the Control Bus when facing a discrepancy. Sometimes a duplicate of the generated message from the Test Data Generator is also needed to derive the expected result.

The parameterization of each component is introduced in the following respectively.

Figure 3.13 illustrates the Test Data Generator with its inputs, outputs and parameters. The messages sent by the Test Data Generator are all generated by itself, thus it has test messages as its outputs but no input. Naturally the test message type should be identical to the type of application messages which flow through the component to be tested.

The message generation method should be selected and configured, as test data may be constant, driven by a test data file, or generated randomly.
Sometimes test data are needed for the result analysis, therefore a duplicate of test message may also be sent to the Test Data Verifier, which requires a secondary channel that can be turned on or off via the Control Bus. In such cases the Test Data Generator has two outgoing channels, and a mapping table should be established to record the connections from the exits to these channels.

It is possible to generate test messages periodically at a fixed time interval. This may bring additional load to the network traffic and the component to be tested, thus a proper frequency of test should be found to reduce the performance burden. The Test Data Generator can also be activated only when necessary and deactivated after short usage by control messages. It must be specified how the control message functions.

If the Test Data Generator uses the first strategy for setting configuration, that is, passively waits for configuration messages, then the configuration message type must be specified. If the second strategy is adopted, the Test Data Generator sends a query message to the central management unit and receives a configuration message from it, then the types of both the query message and the configuration message should be defined.
Figure 3.14 Parameterization of Test Message Injector

Figure 3.14 illustrates the Test Message Injector with its inputs, outputs and parameters. The inputs are application messages and test messages from two channels, and the outputs are the union of these messages. All the messages are of the same type.

Since the Test Message Injector has two incoming channels, a mapping table should be established to record the connections from these channels to the entries.

The tagging of messages can be accomplished by inserting a special header field. If we are not permitted to change the message structure, we can use special values to indicate test messages (e.g., OrderID = 999999) [2], while other values indicate regular application messages.

Figure 3.15 Parameterization of Test Message Separator
Figure 3.15 illustrates the Test Message Separator with its inputs, outputs and parameters. The input is the message flow containing both application messages and result messages, while the outputs are the two sorts of messages separately to their own channels. All these messages are of the identical message type.

Since the Test Message Separator has two outgoing channels, a mapping table should be established to record the connections from the exits to the channels.

Just like the Content Based Router [2], the Test Message Separator should be able to route messages to the proper channel on the basis of the content of messages and a certain routing logic. In this case the routing logic is usually relevant to the tagging mechanism of the Test Message Injector.

![Diagram of Test Data Verifier](image)

**Figure 3.16 Parameterization of Test Data Verifier**

Figure 3.16 illustrates the Test Data Verifier with its inputs, outputs and parameters. The pattern has result messages as its inputs and no output.

When it receives duplicates of test messages from the Test Data Generator, the secondary channel should be turned on to transmit these messages. As there are two
incoming channels, a mapping table should be established to record the connections from the channels to the entries of the Test Data Verifier.

For the setting configuration (such as determining if the secondary channel is on), we can choose from two strategies as always. If the Test Data Verifier passively waits for configuration messages, the type of such messages should be specified. If it sends a query message to the central management unit and receives a configuration message from it, then the both message types must be defined.

The Test Data Verifier takes full responsibility of result verifying, therefore it must know the expected result by checking an already calculated result or deriving the result by itself according to the contents of the original test message. It must also be clarified what occasions can be regarded as an exception, for example, too long waiting time for a result message (time out). When the result is out of normal range or other exceptions occur, the Test Data Verifier reports it to the central management unit with a message describing the details.

BPEL and Web Services

Just like regular router patterns, the four components of the Test Message Pattern can be realized as BPEL activities connected with BPEL links. Actually, besides the message tagging part, the Test Message Injector and the Test Message Separator function almost like routers. The to be tested component should has them as its directly preceding and succeeding activities.

Because the Test Data Generator inserts new messages to the system, the activity representing it should be able to start new BPEL instances even without the existence of application messages in the system. The regular application flow and the test message flow are combined in the same BPEL process, therefore it may have two start activities (receive or pick with the createInstance attribute set to “yes”), one for regular messages and one for test messages. Similarly, the Test Data Verifier should be able to terminate an instance with the reply activity.

The secondary channel between the Test Data Generator and the Test Data Verifier can also be realized with BPEL links. Apparently the route formed by the links among these four components is acyclic.

The comparison of the expected result and the real test result can be performed by simple BPEL code or with the help of a web service.

To keep the Test Data Generator in a single instance sending test messages periodically at a certain time interval is no easy task when the order of activity execution is determined by links. So it is suggested to leave the control to the process or service that invokes the Test Data Generator.

In some cases we find it also possible not to mix the application flow and the test pattern together, because anyway the application messages and test messages are processed in different instances and do not disturb each other. For example, if we want to test an external service instead of a component implemented with pure BPEL
code, we can create a new BPEL process for testing alone. The Test Message Injector and the Test Message Separator are not needed any more in such a process, it only has to generate a test message, invoke (test) the service, receive the output result message and analyze it.

3.4.1.8 Channel Purger

During testing we may face such a problem: we send a test message through a channel to a component. But since the component does not function correctly, messages cannot be consumed or the processing speed is too slow, leaving the channel full of pending messages.

To make the test message reach the component as soon as possible, we have to clear the channel with a Channel Purger, which can be regarded as a filter inserted to the application flow. The channel to be purged, which is originally connected with the component to be tested, should lead to the Channel Purger instead. A new channel from the Channel Purger to the component can be created if some application messages are allowed to be transmitted further.

![Parameterization of Channel Purger](image-url)

**Figure 3.17 Parameterization of Channel Purger**
Figure 3.17 shows the parameterization of the pattern. The inputs of it are the incoming messages from the channel. A basic Channel Purger simply removes all messages from the channel and discard them, in such cases the pattern does not have any output.

On the other hand, more advanced functionalities can also be provided by the Channel Purger, such as selective purging based on contents of the messages. Then the outputs are the messages to be delivered to the next component. The message type of the inputs and the outputs are same, which is actually determined by the message type of the channel to be purged. And the purging criteria should be specified to clarify which messages can be forwarded to the following components and which must be purged.

Sometimes we may need the removed messages for later inspection or replay. For this purpose, we can save the messages in a Message Store, and eventually re-inject them when the system functions normally again. To lead messages from the Channel Purger to the Message Store, a secondary channel between them should be created and configured.

Since the message purging could be performed with above mentioned options and alternatives, we should select some of them to form a purging mechanism that is most suitable to our requirements. Usually the mechanism is represented as a set of configuration values, which is predefined or can be modified dynamically in the runtime via Control Bus.

If the Channel Purger uses the first configuration strategy, that is, passively waits for configuration messages, then the configuration message type must be specified. If the second strategy is adopted, the Channel Purger sends a query message to the central management unit and receives a configuration message from it, then the types of both the query message and the configuration message should be defined.

BPEL and Web Services

Since messages are usually distributed in individual BPEL instances, if a instance for a message is successfully created, the message will not be prevented from reaching an activity because of messages in the same channel in other instances, especially when the activity does not involve any external services. During the testing of an component, if we only want to ensure that the test message reaches the destination, it is unnecessary to purge the messages in other instances.

However, when the bottleneck is located at the interaction with an external service from the activity, we can use the Channel Purger to stop messages, so that other instances will not even try to invoke the service. The pattern may be implemented like most filters as a BPEL activity which is connected with other activities (representing components) with BPEL links. The purging function can be realized by simply setting a proper transition condition so that the succeeding links will not be activated and the messages cannot reach the activity. As a result, the instances which contain the messages stopped by the Channel Purger are suspended and need to be killed, and the external service is left free for the test message.
The implementation of the configuration procedure should include a web service which functions as the central management unit. The Channel Purger sends a configuration query by invoking the operation provided by the service, then the web service processes the query, retrieves the setting value from an XML file and returns a configuration message containing the value.

3.4.2 Message Routing Patterns

Five routing patterns discussed in the following are not simple routers. As composed routers, the Composed Message Processor and the Scatter-Gather supports parallel processing of messages, while the Routing Slip and the Process Manager control the sequential processing from a central point. Totally different from the routing patterns working under the Pipes and Filters architecture, the Message Broker describes a hub-and-spoke architectural style.

Besides the routing patterns from Hohpe’s book [2] mentioned above, we also introduce the functionality and the parameterization of Join Router, a pattern needed for the application scenario creation in next chapter.

3.4.2.1 Composed Message Processor

As a pattern composed of Splitter, Content-Based Router and Aggregator [2], the Composed Message Processor enables the separate processing of multiple elements of a message and the reaggregation of the responses.

The structure of the Composed Message Processor is shown in Figure 3.18. The input message is divided into several submessages by the Splitter, then routed to the corresponding message processors. Finally the Aggregator collects the responses from the processors and aggregates them into the output message. Apparently the input and output messages may have different message types. Messages transmitted inside the Composed Message Processor may be of various types as well, which are specified by the parameters of individual components.

To make the Composed Message Processor function correctly, all the component patterns inside it should be well configured. The Parameterization of the Splitter, the Content-Based Router and the Aggregator has already been researched [9].

The processor could be an external service, or a single or composed message processing component. Details of the involved processors should be given, including the settings of each one and the total number of processors. The latter parameter can also be passed to the Splitter to determine the amount of its exits and similarly to the Aggregator to determine the amount of its entries.
Generally we expect an output message composed of response messages from all the processors. In case some of the responses are delayed or missing, we can resend the request message or flag an exception and wait for the manual evaluation. The resending of message should only be allowed to repeat for limited times in order to avoid infinite loops.

**BPEL and Web Services**

The BPEL implementation of this pattern is primarily based on the implementation of the individual patterns contained. The connections between the components are realized with BPEL links.

A *while* activity can be used to realize the resending of messages. The aggregator or an extra service may check if the responses are complete and save the result in a status variable. The *while* activity contains the whole Composed Message Processor pattern and possibly the completeness checking service and executes the loop according to the status variable. Though the input message is resent, we can repeat without updating the corresponding variable as long as its value is kept unchanged, which means the message is still stored in it.
### 3.4.2.2 Scatter-Gather

A *Scatter-Gather* broadcasts a message to multiple recipients and reaggregates the responses into a single message with an *Aggregator*.

Two different mechanisms can be used to broadcast the request messages.

1. **Distribution** via a *Recipient List* allows the *Scatter-Gather* to control the list of recipients but requires the *Scatter-Gather* to be aware of each recipient’s message channel.

2. **Auction**-style *Scatter-Gather* uses a *Publish-Subscribe Channel* to broadcast the request to any interested participant. This option allows the *Scatter-Gather* to use a single channel but also forces it to relinquish control [2].

![Figure 3.19 Scatter-Gather](image)

Figure 3.19 Scatter-Gather

Figure 3.19 illustrates the input, output and parameters of the *Scatter-Gather* pattern. The input is a request message, which is broadcasted to all processors, and the output of *Scatter-Gather* is the output of the *Aggregator*, which is aggregated from the
3.4 PATTERNS

responses of the processors. Apparently the input and output messages may have
different message types. Messages transmitted inside the Scatter-Gather may be of
various types as well, which are specified by the parameters of individual components.

To make the Scatter-Gather function correctly, all the component patterns inside it
should be well configured. We need to choose from the two broadcasting mechanism
introduced above and configure the Recipient List or the Publish-Subscribe Channel.

The processor could be an external service, or a single or composed message
processing component. The setting of each processor and the number of them should
be given. The latter parameter can also be passed to the Aggregator to determine the
number of its entries.

According to the demand of the application scenario, the responses may be aggregated
in different ways. For example, if the processors are suppliers bidding for an order,
the Aggregator may record the bids of all the suppliers or simply choose the best
supplier. It may also wait only for a short time and take the best bid from the received
responses and neglect the other suppliers. The aggregation mechanism is a parameter
of the Aggregator.

BPEL and Web Services

The BPEL implementation of Scatter-Gather is primarily based on the
implementation of the individual patterns contained. The connections between the
components are realized with BPEL links.

Unlike the Composed Message Processor, the Scatter-Gather usually tends to select
the best response instead of resending the request message when some expected
responses are missing. In case the responses from all processors are required, the
resending of messages can be realized with the while activity similarly.

3.4.2.3 Routing Slip

Sometimes we need to route a message not just to a single component, but
consecutively through a series of processing steps. A possible solution is to attach a
Routing Slip to each message to specify the sequence of these processing steps.

As illustrated in Figure 3.20, first a Routing Slip Generator computes the Routing Slip,
a list of required steps for the input message. It then attaches the Routing Slip to the
message and forwards it to the first processor in the list. Each processor is integrated
with a router that reads the list and sends the message to the next processor in it. In
this way the message are transmitted through a predefined series of processors, while
each of them can be passed only once.
The inputs and outputs of the *Routing Slip* are messages of possibly different types. Messages transmitted among the processors may be of various types as well, which are specified by the parameters of each processor.

All processors involved in the *Routing Slip* must be listed along with their functionalities and interfaces. The *Routing Slip Generator* takes the responsibility of computing the *Routing Slip* on the basis of this list and a generation algorithm.

In order to keep the processor unmodified, we need a router for each processor. The input messages of the processor can be delivered to it directly, but the output messages must be intercepted by its router, because only the router is able to tell the address of the next processor from *Routing Slip*.

When the message has passed through all the processors in the list, the message should be sent out of the *Routing Slip* system to the next component in the whole application message flow. A practical solution is to append the address of this
component at the end of the *Routing Slip*, so that the router of the last processor can forward the message directly to the component.

### BPEL and Web Services

It may be possible to include the *Routing Slip Generator* and all involved processors in a single *flow* and connect them with BPEL links. But since each processor must have outgoing links to any other processors, the connection network composed of these links could be a chaos.

A possible solution is to encapsulate each processor and its router into a BPEL process and define a process for the *Routing Slip Generator* as well. The routers and the *Routing Slip Generator* retrieve information of the BPEL process representing the next processor in the *Routing Slip* and invoke that process.

#### 3.4.2.4 Process Manager

After the computing the *Routing Slip*, the sequence of processors must be fixed and cannot be changed afterwards. Moreover, the processor steps must be executed sequentially. The *Process Manager* pattern gives us more flexibility. It has a central processing unit, the *Process Manager*, which determines the next processing step based on intermediate results.

Figure 3.21 illustrates how the *Process Manager* works. It sends a incoming message (1) to the first processor A based on the predefined rules. After unit A finishes its task, it returns a reply message to the *Process Manager*, which then forwards a message (2) to the next processor B.

The inputs and outputs of the *Process Manager* are messages of possibly different types. Messages transmitted among the processors may be of various types as well, which are specified by the parameters of each processor.

All processors involved must be listed along with their functionalities and interfaces. With the help of this information we can manually create process definitions. The *Process Manager* locates the current state in the process definition and decides which message should be handled by which processor in the next.

In order to decide the processing steps dynamically, the *Process Manager* must be able to remember the state of each message, that is, where the message is located in its processing sequence. A table should be established to record the states and updated whenever the state of a message is changed.

It is noticeable that the message sent to a processing step may not be the reply message of the last processing step. Actually, the *Process Manager* is able to store and maintain intermediate results for later usage. This mechanism and the memory of message state together enable the *Process Manage* to maintain non-sequential and flexible processes.
The Process Manager is dealing with messages from multiple process instances simultaneously. To differentiate instances from each other, we associate an incoming message with a unique Correlation Identifier, and processors always copy the identifier to its reply message. As a result, the messages in the same process instance possess the identical Correlation Identifier.

**BPEL and Web Services**

Actually, BPEL is a language used to define processes, and the BPEL engine works just as a Process Manager.

It supports concurrent running of multiple instances of any process predefined with BPEL. For each instance, the engine performs the routing of messages, management of message state, the storage of intermediate results and any other tasks specified by the process definition. It also provides correlation mechanism to differentiate instances.
3.4.2.5 Message Broker

*Message Broker* is an architecture pattern as opposed to individual design pattern. It chains components in a hub-and-spoke style.

All components are directly connected with the *Message Broker* and exchange messages through this central point, which may turn it into a bottleneck. This problem can be solved with a *Message Broker Hierarchy* that resembles a network composed of subnets. A local *Message Broker* and the components connecting to it form a subnet, and the central *Message Broker* is in charge of the message transmission between local brokers.

![Diagram of Message Broker Hierarchy](image-url)

Figure 3.22 Parameterization of Local Message Broker

Figure 3.22 illustrates a local *Message Broker*. Each processor may have different incoming and outgoing message types, firstly the *Message Broker* has to record the processors and their message types. It may also be connected with a central *Message Broker*, through which it can communicate with other local *Message Brokers*. It is possible that they all accept different message type, as a result, the various types should be clarified as well.
As the local *Message Brokers* usually possess many interfaces for channels between itself and the processors or between itself and the central *Message Broker*, a mapping table is needed to describe the mapping relationship among interfaces and channels. Since messages in the subnet are sent only through one router and the destinations are always clearly specified, the routing logic is simple.

A problem while routing with the *Message Broker* is the various message types. Each processor only accepts one message type, which in most cases cannot be changed. As the only intermediate component between two processors, the *Message Broker* must accomplish the transformation of messages. A *Canonical Data Model* [2] is suggested by Hohpe as a helpful tool.

![Diagram of Message Brokers](image)

**Figure 3.23 Parameterization of Central Message Broker**

The central *Message Broker* is very similar to the local ones, except that it only has access to brokers, not the processors. Naturally the parameterization of it also includes the specification of connections, message types and how the message should be transformed to be accepted by the receiver.

**BPEL and Web Services**
Usually the *Message Broker* is supposed to be stateless, thus for every single message processed by it, the *Message Broker* only has to receive the message from one incoming channel, transform the message if necessary and send the message to the correct outgoing channel. Both the local *Message Broker* and the central *Message Broker* function in a similar way.

A possible implementation of *Message Broker* is illustrated in Figure 3.24. We divide the function of any processor into two parts, represented by a *receive* activity and a message processing activity. All these activities and the *Message Broker* activity are connected by BPEL links in a *flow*. The *joinCondition* of the *receive* activities is defined so that any incoming link can activate the *Message Broker*, which analyzes the message, transforms it if needed and then activates the link to the proper message processing activity.

If all the processors are encapsulated in individual BPEL processes. The functionality of *Message Broker* can also be realized with a *flow* activity containing only the *receive* activities and the logic of *Message Broker*. However, the logic should be modified so that it invokes the proper BPEL process instead of activating the links.

### 3.4.2.6 Join Router

In some cases we need to combine multiple message streams to one stream. When several channels contain messages of an identical type and merge at the *Join Router*, all the messages can be mixed and forwarded to the outgoing channel of it.
3.5 COMBINATION OF PATTERNS

*Join Router* is simple but useful. For example, the *Test Message Injector* can be replaced by a *Join Router* if the message tagging is performed during the creation of test messages. And the outgoing routes of a *Detour* should also converge at a *Join Router*.

![Diagram of Join Router](image)

Figure 3.25 Parameterization of Join Router

Figure 3.25 shows the input, output and parameters of the *Join Router* pattern. Both inputs and the outputs are messages of the same type. The number of the incoming channels should be specified as well. In addition, a mapping table must be established to describe the relationship between the incoming channels and the entries.

**BPEL and Web Services**

It is quite easy to realize the *Join Router* pattern as a BPEL activity connected to other activities via BPEL links. The messages are forwarded by copying any output variable of the preceding activity (representing an incoming message) to the input variable of the succeeding activity (representing an outgoing message).

### 3.5 Combination of Patterns

According to the principle of EAI patterns, individual patterns should be combined to build a messaging system which represents an application scenario in the real world. We should be able to choose patterns and combine them freely to meet the requirements of the application. Nevertheless, some patterns cannot be connected together without any intermediate processing steps since their inputs and outputs do not match. Some patterns cannot be integrated to a complicated pattern because of the special function mechanism of it.
3.5 COMBINATION OF PATTERNS

In this section we will discuss the possibility to combine the patterns parameterized in section 3.3 with other patterns. Afterwards some combination examples are also introduced.

3.5.1 Combination Possibilities

Bettina Druckenmueller has investigated the combination possibilities of several pattern types in her thesis [9]. Therefore in the following we do not place emphasis on the patterns that behave mainly as ordinary filters, only the special features of the pattern combination or integration are our interest.

3.5.1.1 System Management Patterns

**Control Bus**
The *Control Bus* can be combined with any filter pattern if it provides a corresponding interface to read the contents of control messages and form its own control messages such as an exception report. It is recommended, but not necessary that the filters use a uniform control message format, since theoretically the *Control Bus* is able to transmit messages of any type, and the central management unit can perform different message parsing logics for different filters.

**Detour**
The *Detour* pattern can be treated as an ordinary message router, which should be connected with one incoming channel and two outgoing channels of the identical data type.

**Wire Tap**
The *Wire Tap* pattern can be treated as an ordinary message router, which should be connected with one incoming channel and two outgoing channels of the identical data type.

**Message History**
The *Message History* pattern is neither a filter nor a pipe, but rather a modification to the message structure. Thus when a messaging system adopts the *Message History* pattern, every message processor that appends an entry on the history list must extend its process to support the appending operation.

**Message Store**
The *Message Store* pattern is relatively independent of the other filters. Because of the loose coupling principle of EAI, we do not want to change the logic of message processors so that they directly send duplicate messages to the *Message Store*, in most cases a *Wire Tap* is used instead to perform such function.

Besides the secondary channel (essentially a datatype channel) from the *Wire Tap*, the *Message Store* may also be connected with other channels such as the *Invalid*
**Message Channel** and the **Dead Letter Channel**, if we choose to store those special messages for later usage or analysis.

**Smart Proxy**
The **Smart Proxy** pattern is not used between filters, but between a service and the invokers of the service. Apparently it can only be combined with other patterns which access or control the request queue or the reply queue.

**Test Message**
The **Test Message** pattern can be integrated with any single filter with one incoming channel and one outgoing channel by inserting the **Test Message Injector** and the **Test Message Separator** into the application flow before and after the filter. A section of the whole messaging system composed by several patterns may also be tested by the same means as long as it possesses the same number of incoming and outgoing channel and therefore can be regarded as a single filter.

When the to be tested filter or combination of patterns has multiple exits or entries, most likely it will lead to incorrect test result if we merely focus on one pair of channels. Therefore a variant of **Test Message** pattern should be designed according to the requirements of individual application scenarios.

A simple example is an **Aggregator** aggregating three input messages from three channels respectively to one single output message. It is suggested to place one **Test Message Injector** on each incoming channel. The **Test Data Generator** may send test messages to a **Content Based Router** first, which delivers the messages to the corresponding **Test Message Injector**. In such a way the **Aggregator** is able to receive all the test messages.

**Channel Purger**
The **Channel Purger** pattern should be used exclusively on various sorts of messaging channels.

### 3.5.1.2 Message Routing Patterns

**Composed Message Process**
The **Composed Message Process** pattern is a combination of patterns that appears like a simple filter to the rest of the system, naturally it can be treated as a filter with one entry and one exit during the pattern integration as well.

The processing step inside the pattern between the **Content Based Router** and the **Aggregator** could be a single filter or a combination of patterns with one incoming channel and one outgoing channel.

**Scatter-Gather**
The **Scatter-Gather** pattern is a combination of patterns that appears like a simple filter to the rest of the system, naturally it can be treated as a filter with one entry and one exit during the pattern integration as well.
3.5 COMBINATION OF PATTERNS

The processing step inside the pattern between the broadcasting pattern and the Aggregator could be a single filter or a combination of patterns with one incoming channel and one outgoing channel.

Routing Slip
The Routing Slip pattern is a combination of patterns that appears like a simple filter to the rest of the system, naturally it can be treated as a filter during the pattern integration as well.

Since the message is processed sequentially, the message processing component inside the pattern should be a single filter with one incoming channel and one outgoing channel. Therefore routers are usually not included, unless as the last step of the Routing Slip. Another constraint is that the output message type of a processor must match the input message type of the next processor, as a result, the Routing Slip pattern may often contain some message transformers.

Process Manager
The Process Manager pattern is a combination of patterns that appears like a simple filter to the rest of the system, naturally it can be treated as a filter during the pattern integration as well.

The constraints inside the pattern between the components are similar to the ones of Routing Slip pattern. The difference lies in the possibility to integrate routers into the Process Manager pattern, because messages are not simply passed to the next processor, but can also be stored temporarily by the manager.

Message Broker
The message processing components connected with the Message Broker by messaging channels are single filters. The Message Broker takes over the responsibility of message routing, thus pure routers are not needed for this pattern. On the other hand, though in most cases the Message Broker itself is able to transform messages to a format accepted by the next component, message transformers which change the contents of messages rather than merely the structure of them could still be useful components in a system connected with the Message Broker.

Join Router
The Join Router is an ordinary message router, which should be connected with multiple incoming channels and one outgoing channel of the identical data type.

3.5.2 Combination Examples

In the following we are going to present two typical examples to illustrate how patterns are integrated together, one for the system management purpose, the other with two message routing patterns.
3.5 COMBINATION OF PATTERNS

3.5.2.1 Channel Purger and Message Store

Figure 3.26 Combination of Channel Purger and Message Store

Figure 3.26 shows an example of a subsystem composed of the Channel Purger, the Message Store and the central management unit. As mentioned in section 3.4.1.8, sometimes the Channel Purger does not discard the messages, but rather sends them through its secondary channel to the Message Store, which is able to save them permanently for later usage.

Nevertheless, when the testing is finished and the channel is set back to the normal status, the Message Store cannot accomplish the task of retrieving those messages and re-injecting them back into the channel, since according to the original design it only receives messages and manages the storage. To solve the problem, we also integrate the central management console into our subsystem, which accesses the Message Store via the Control Bus, searches for the saved messages and forwards them to the Channel Purger.

3.5.2.2 Composed Message Processor and Scatter-Gather

In the example illustrated by Figure 3.27 we combine the Scatter-Gather with the Composed Message Processor, which is achieved by setting the Scatter-Gather as the message processing component of the Composed Message Processor.
3.5 COMBINATION OF PATTERNS

The input message of the Composed Message Processor, an order containing several items, is divided into submessages that each contains an individual item. As the input of Scatter-Gather, every submessage is broadcasted to various vendors for a bid, then the Aggregator of Scatter-Gather aggregates all bid responses for each item into a complete quote and returns it to the Composed Message Processor, which waits for the quotes for all items and combines them together into a validated order using its Aggregator.

Figure 3.27 Combination of Composed Message Processor and Scatter-Gather

We notice the fact that the Content Based Router should be a component of the Composed Message Processor but is not used in the example. It is designed to route split messages to different processors, but since the processor of the submessages is the identical Scatter-Gather, the router is redundant here.
Chapter 4

BPEL Implementation of Patterns

In the previous chapter we have discussed theoretically how the parameterized EAI patterns can be realized with BPEL and web services. To prove the correctness of the implementation idea, we are going to build a simple application scenario utilizing some system management functionalities, generate BPEL code automatically from it and run the process to simulate the patterns.

Florian Schebelle and Bettina Druckenmueller have developed a system “EAI to BPEL” [9] as a practical tool for the construction of the EAI messaging system and the generation of the corresponding BPEL/WSDL code. Before building our application scenario with this tool, the system has to be extended to support the involved patterns that were not parameterized previously.

Firstly we will give a short introduction on how to make use of the system, then describe the implementation details about the extension and the modification of the system, finally we focus on the scenario construction and the deployment and execution of the generated BPEL processes.

4.1 System Usage Instructions

In this section we explain how to create an EAI messaging system with various patterns, to store it in an EAI file and to generate BPEL and WSDL code from the file using the “EAI to BPEL” system.
4.1 SYSTEM USAGE INSTRUCTIONS

4.1.1 The User Interface

After starting the system, the user can convert to the EAI to BPEL perspective in this way: select Window → Open Perspective → Other from the menu bar, then choose EAI to BPEL in the Open Perspective dialog. Under this perspective, we can see the menu bar, the toolbar and four panes.

![Figure 4.1 Interface of the system “EAI to BPEL”](image)

In the upper left is the Navigator pane, which gives an overview of the projects and files in the workspace.

The Outline pane in the lower left corner displays an graphical outline of the EAI messaging system. The user can locate a section of the system by simply clicking in the corresponding part of the Outline pane.

The messaging system itself is displayed in the Editor pane in the upper right. The Editor pane is composed of two parts: in the left area the user can draw the system with components and configure them by setting values in configuration dialogs, the palette on the right contains various types of components and tools used to build the system.

There are two tabs in the lower right pane. When the user saves the EAI file, the program checks the system and lists errors within the Problems tab. Double-clicking on the error entry can direct the user to the file containing the error, and the entry fields Description, Resource, Path and Location provide more details of the error.
4.1 SYSTEM USAGE INSTRUCTIONS

The other one is the Properties tab, which displays the properties of the currently selected object such as filters, channels or the connections between them. If no object is selected, the properties of the massaging system is shown. We can change values of some parameters in the Properties tab, but sometimes it does not support the access to all parameters of the component, in such cases we can use the configuration dialogs, which will be described in a later section.

4.1.2 Creation of an EAI File

First of all an EAI project should be created to hold the EAI files. Select File → New → Project from the menu bar to open the New Project dialog, choose EAI to BPEL Project under the folder Other, click Next, fill in the project name and location, then the project is created.

To create an EAI file in this project, right-click on the project name and select New → Other from the menu, a dialog appears. Similarly the user can choose EAI to BPEL under the folder Other, give the name and location of the EAI file and confirm with the Finish button.

4.1.3 Construction of the Message System

Now that we have an EAI file, we can build a message system, assign proper values to parameters of the patterns and save the system in this file. Bettina Druckenmueller had a detailed description of the system and the previously implemented patterns [9], providing enough information for the involved components, thus we do not have to repeat it in this thesis. After a brief introduction of the basic operations and functionalities, we concentrate on the newly implemented patterns.

4.1.3.1 Overview

The drawing and configuring of the messaging system is mainly accomplished in the Editor pane illustrated in Figure 4.2.

For a new EAI file the white editing area on the left is empty, the user can click on an entry from the palette on the right to choose a component and then click on the editing area to add it to the messaging system.

There are two ways to select objects in the editing area. One is the Select mode, under which the user is able to select any single object by clicking on it. The other is the Marquee mode, under which the user can drag a rectangle around multiple objects to select them together, but only the nodes like filters and channel, the connections between them are not included.
4.1 SYSTEM USAGE INSTRUCTIONS

When an object is selected, it can be configured by right-clicking on it and choose **Configuration Dialog** from the menu. Examples of configuration dialogs will be given in the following subsections. Alternatively, the properties of an object can also be modified in the **Properties** tab introduced in section 4.1.1.

As illustrated in Figure 4.3, the components are classified into several categories in the palette. Each component is represented by an icon followed by the component’s name. An “Eclipse icon” (the blue sphere) indicates that the component is only listed here but not implemented yet thus cannot be used to construct the messaging system.

![Figure 4.2 Example of a message system](image)

![Figure 4.3 Palettes for various types of components](image)
According to the EAI pattern’s architecture, the filters are directly connected to the channels. Nevertheless, in the system “EAI to BPEL” we have to draw the connections between filters and channels. Though they are all represented by arrows in the editing area, there are various sorts of them. For instance, the Filter to Channel Connection are used when the filter has only one outgoing channel. If it has multiple outgoing channels, the filter is regarded as a router and we should choose the Router to Channel Connection instead. Furthermore, the type of the message transmitted in a channel must be specified twice as the property of the connections that are connected with the channel.

4.1.3.2 System Configuration

![System Configuration Dialog](image)

Figure 4.4 System configuration dialog

By right-clicking on anywhere in the white background of the editing area and selecting Configuration Dialog from the popped up menu, the user can start to edit the properties of the messaging system in the dialog shown in Figure 4.4.

Since the system generates a BPEL file and a WSDL file, the namespaces of the two files should be specified respectively, including the target namespace and the additional namespaces. In both files we are allowed to define multiple additional
namespaces, which can be added, edited or deleted with the corresponding buttons. All namespaces used as the prefix of port types or message types in the EAI messaging system must be defined here.

In order to use a correlation set in individual patterns when interacting with external web services, it must be configured in advance in this dialog. The attribute Name, Prefix for Property and Property can be edited for each correlation set, and multiple sets are supported.

4.1.3.3 Join Router

Join Router is a simple pattern that does not modify the messages passing through it or invoke any web services. Since no parameter is to be configured for this pattern, the user only have to connect the component with correct channels.

4.1.3.4 Detour

The Detour is connected with two outgoing channels and routes messages to one of them controlled by the management web service. Therefore the parameters of the service such as Prefix for Porttype, Porttype and Operation should be defined in the dialog. Moreover, the Detour receives the configuration information from the output
message of the service, thus the configuration message type and its prefix must be given as well.

The content of the configuration message indicates an outgoing channel, usually by specifying its name. To support the mechanism we should give each channel a name in advance by mapping the channel numbers onto the channel names. The channel numbers are generated automatically by the system according to the order in which the channels were connected with the Detour.

4.1.3.5 Test Data Generator

![Test Data Generator configuration dialog]

Figure 4.6 Test Data Generator configuration dialog

The Test Data Generator does not generates test messages itself, but usually starts a new BPEL instance by receiving a test message from outside. The prefix of the relevant port type is of the default value “home”, but the port type and operation should be defined.

To configure the correlation of the service, the user can click on the Edit Correlation button to edit the attributes Set-Name, initiate and pattern in the popped up dialog. Note that the correlation set used must be defined for the messaging system.
4.1 SYSTEM USAGE INSTRUCTIONS

In case the Test Data Generator is sending a duplicate test message to the Test Data Verifier as a reference for the verification, the user should check the check box **sends duplicated message to test data verifier**. Then the Test Data Generator is allowed to be connected with two outgoing channels, one leading to the to be tested processor and one to the Test Data Verifier.

4.1.3.6 Test Data Verifier

The Test Data Verifier is a relatively complicated pattern with several working modes. The differences lie in whether it receives the duplicate test message from the Test Data Generator and whether it uses BPEL or a web service to verify the test result.

![Test Data Verifier configuration dialog (without duplicate message)](image)

Figure 4.7 Test Data Verifier configuration dialog (without duplicate message)

Figure 4.7 shows the setting for a verifier without the duplicate message that checks the result with an external verification service. Since the verifier is the receive-reply partner of the Test Data Generator, the relevant port type, operation and correlation must be configured to match those of the generator.

If the test result does not meet the expectation, the verifier will send an error message to the management console by invoking a service. For the error reporting, the user should define the following parameters: the port type, the prefix of the port type, the name of operation, the error message type and its prefix.
Since there is no duplicate test message, the check box **receives duplicated message from test data generator** is unchecked and the related fields under the check box on the dialog are disabled.

To configure the verification web service, the user should check the check box **verify test message with web service** and fill in the three available text boxes. The port type of the service and the prefix are always needed to be defined for any working mode. As the verifier has to send only the test result but not the duplicated test message to the web service, it is sufficient to define one operation, the **Verification Operation**.

![Test Data Verifier configuration dialog (with duplicate message)](image)

Figure 4.8 illustrates the dialog for a verifier that receives the duplicate test message and verifies the result with an external service. Now that the verifier has two incoming channels, it must be specified which channel is for the test result and which for the duplicate message, therefore a mapping table for the channel numbers and the channel names is also required.

The Test Data Verifier sends two messages to the verification service one by one using different operations and retrieves the verification result from another operation. Thus the text boxes for these three operations are enabled, and the text box **Verification Operation** disabled.
When the verification logic is relatively simple, it is more resource-saving to check the test result within the BPEL process using XPath expressions [15]. The user can define multiple conditions comparing an element of the test result with a value, which are combined with a logical “and” relationship. The example indicates that the acceptable test result should have a value between 300 and 900 for its element “creditResponse”.

### 4.1.3.7 Saving the EAI File

The messaging system can be saved at any time by simply selecting **File → Save** from the menu or clicking the **save** icon in the toolbar and reloaded later for further development. If the **save** option is disabled, just move any object a bit and retry, as only a modified EAI file can be saved.

The EAI file is in essence an XML file recording all the objects in the messaging system along with their properties and positions.

### 4.1.4 Generation of BPEL and WSDL Code
After saving the EAI file, errors in the system are displayed in the Problems Tab. Errors occur typically when a filter’s connections does not match the number of entries/exits or when required parameters are missing.

As long as there are no errors, the BPEL file and the WSDL file can be generated for the messaging system. To start the generation, just select **EAI to BPEL → Generate BPEL** from the menu or click on the Generate BPEL icon in the toolbar. After selecting OK in two confirmation dialogs, the user can specify the name and location of both files. Then two messages will pop up to inform the user whether the generation was successful or not.

### 4.2 Implementation

This section explains how the previous implementation was modified and extended to support the new patterns. Before that we will firstly give a brief description of the previous work, including the implementation environment, the system model and important classes.

#### 4.2.1 Implementation Environment

The implementation environment of the system “EAI to BPEL” is Eclipse 3.2.1 with the plugins EMF 2.2.0 and GEF 3.2.0. First we introduce the fundamentals of Eclipse and the concept of plugin, then the functions of the plugins used in the system.

#### 4.2.1.1 Eclipse

![Figure 4.10 Eclipse-Platform architecture](image)

Figure 4.10 Eclipse-Platform architecture [16]
4.2 IMPLEMENTATION

The Eclipse Platform's principal role is to provide tool providers with mechanisms to use, and rules to follow, that lead to seamlessly-integrated tools. These mechanisms are exposed via well-defined API interfaces, classes, and methods. The Platform also provides useful building blocks and frameworks that facilitate developing new tools [16].

One of the key benefits of the Eclipse Platform is realized by its use as an integration point. Building a tool or application on top of Eclipse Platform enables the tool or application to integrate with other tools and applications also written using the Eclipse Platform. Although the Eclipse Platform has a lot of built-in functionality, most of that functionality is very generic. It takes additional tools to extend the Platform to work with new content types, to do new things with existing content types, and to focus the generic functionality on something specific [16].

A plugin is the smallest unit of Eclipse Platform function that can be developed and delivered separately. Usually a small tool is written as a single plugin, whereas a complex tool has its functionality split across several plugins. Except for the kernel application, the Platform Runtime, all of the Eclipse Platform's functionality is located in plugins [16].

Plugins are coded in Java. A typical plugin consists of Java code in a Java Archive (JAR) library, some read-only files, and other resources such as images, web templates, message catalogs, native code libraries, etc. A single plugin's code libraries and read-only content are located together in a directory in the file system, or at a base URL on a server [16].

Each plugin has a plugin manifest declaring its interconnections to other plugins. The interconnection model is simple: a plugin declares any number of named extension points, and any number of extensions to one or more extension points in other plugins [16].

On start-up, the Platform Runtime discovers the set of available plugins, reads their manifests, and builds an in-memory plugin registry. The Platform matches extension declarations by name with their corresponding extension point declarations. Plugins can also be added, replaced, or deleted after startup. A plugin is activated when its code actually needs to be run [16].

4.2.1.2 EMF

The Eclipse Modeling Framework (EMF) is designed to ease the design and implementation of a structured model. The Java framework provides a code generation facility in order to keep the focus on the model itself and not on its implementation details [17].

When talking about modeling, we generally think about things like Class Diagrams, Collaboration Diagrams, State Diagrams, and so on. UML (Unified Modeling Language) defines a standard notation for these kinds of diagrams. Using a
combination of UML diagrams, a complete model of an application can be specified [18]. In EMF, the model can be created in three different ways, including using tools like the Omondo EclipseUML plugin [19]. The class diagram of the EMF model is established similarly to an UML diagram and saved in a file with the extension .ecd.

Once an EMF model is specified, the EMF generator can create a corresponding set of Java implementation classes for each class in the model. Each generated interface contains getter and setter methods for each attribute and reference of the corresponding model class. Each generated implementation class includes implementations of the getters and setters defined in the corresponding interface, plus some other methods required by the EMF framework [18].

The user can edit these generated classes to add methods and instance variables and still regenerate from the model as needed. The additions will be preserved during the regeneration. If the code added by the user depends on some modified model, the code needs to be updated manually to reflect those changes; otherwise, it is completely unaffected by model changes and regeneration [18].

### 4.2.1.3 GEF

The Graphical Editing Framework (GEF) allows us to easily develop graphical representations for existing models. All graphical visualization is done via the Draw2D framework, which is a standard 2D drawing framework based on SWT from eclipse.org [18].

The editing possibilities of GEF allow the user to build graphical editors for nearly every model. With these editors, it is possible to do simple modifications to the model, like changing element properties or complex operations like changing the structure of the model in different ways at the same time. All these modifications to the model can be handled in a graphical editor using very common functions like drag and drop, copy and paste, and actions invoked from menus or toolbars [18].

EditParts are the central elements in GEF applications. They are the controllers that specify how model elements are mapped to visual figures and how these figures behave in different situations. Usually an EditPart class should be created for every model element class so that the class hierarchy for the EditParts are similar to the hierarchy for the model.

EditPolicies are those GEF parts which bring the editing functionality into EditParts. An EditPolicy defines what can be done with an EditPart, it is also responsible for feedback management and is allowed to delegate work (forward requests) to other EditParts. To execute the operations such as creating, modifying or deleting objects in the model, a command class should be created for any set of \(<\text{operation, object}\>\). This class is able to change the attributes of objects and define the to be executed actions for the “undo” and “redo” commands of the editor.
4.2 IMPLEMENTATION

4.2.2 Previous Work

At first we inspect the model of the previous system. The class diagram is too large, therefore we only describe its important elements in words, the diagram itself can be found in file model.ecd under the root directory of the project and viewed with the EclipseUML plugin [19].

The class Messagesystem is the main class of model, which contains all the other components (like filters) and parameters relevant to the BPEL and WSDL files (like correlation sets).

The superclass of all the channel classes is Pipe, it has InvalidMsgChannel, DeadLetterChannel, DatatypeChannel and PubSubChannel as its subclasses.

The superclass of all the filter classes is Filter, which contains the classes for external service, message transformers and routers. The routers are divided into three types according to the number of their inputs and outputs and sorted under different subclasses of Filter. Besides, the Filter class is associated with the messaging endpoint classes since they are additional components based on filters.

The model also contains other supporting classes such as correlation information for each filter and the mapping between channel number and channel name.

The interface classes and the implementation classes of the EMF model are generated into the package de.unistuttgart.iaas.eaiparam.model and the package de.unistuttgart.iaas.eaiparam.model.impl.

![Figure 4.11 Packages relevant to the editor](image)

The package de.unistuttgart.iaas.eaiparam.editors and its subpackages contain classes that enable the user to edit the messaging system in the editor. For example, the configuration dialog classes are placed in the dialogs package, the editParts classes in the editParts package and the classes defining the actions needed for creating, deleting and editing objects in the commands package.

The class BPELWriter and WSDLWriter accomplish the task of generating BPEL and WSDL. They generate BPEL code for system parameters like namespaces directly and the code for individual patterns by invoking methods of the pattern classes in the package de.unistuttgart.iaas.framework.xformtobpel.patterns and its subpackages. The pattern classes define the BPEL activity of each pattern with the BPEL element
4.2 IMPLEMENTATION

classes provided by the BPEL framework (such as If, Assign etc.) developed by Schebelle [12].

4.2.3 Modifications to the Model

![Diagram of Test Data Generator and Detour](image)

The class \textit{Routing\_1\_to\_many} is a subclass of the class \textit{Filter}. It is the superclass of routers with one input and multiple outputs like the Content Based Router. When we consider the functionality, the Detour pattern and the Test Data Generator pattern should both belong to the system management category. However, the EMF model class hierarchy is classified by the number of inputs/outputs of the filter, since extra mapping information may be needed for multiple channels connected with the filter. Therefore we decide to add the \textit{TestDataGenerator} and the \textit{Detour} class under the class \textit{Routing\_1\_to\_many}. The attribute \textit{MsgDuplication} of the class \textit{TestDataGenerator} indicates whether the generator sends a duplicate test message.

In the previous system, the class \textit{Aggregator} was a direct subclass of the class \textit{Filter}, because the Aggregator pattern was the only implemented router which has multiple incoming channels. Now we have more routers with several entries, therefore we choose to add a class \textit{Routing\_many\_to\_1} between the class \textit{Filter} and \textit{Aggregator} (shown in Figure 4.13), which plays a similar role as \textit{Routing\_1\_to\_many}.

Besides setting the \textit{Aggregator} class as the subclass, we move the associations originally linked with \textit{Aggregator} to \textit{Routing\_many\_to\_1}. One is the \textit{RoutingRecipient} class specifying the entries of the router, the other is the \textit{Name\_to\_Entry} class, which deals with mapping of the incoming channel’s number onto the channel name.
The new class *JoinRouter* is added as a subclass of *Routing_many_to_1*, because a Join Router has multiple inputs and one output. The pattern itself is quite simple, so it has no extra parameters.

Another new subclass of *Routing_many_to_1* is *TestDataVerifier*, because a Test Data Verifier may have two incoming channels when it receives the duplicate message from the Test Data Generator. *TestDataVerifier* has a long list of attributes. *WSForVerification* indicates whether the verification is performed by a service or by the BPEL code. *PTForVerification*, *prefixPTForVerification*, *callbackOperation*, and *WSForVerification*.
4.2 IMPLEMENTATION

testMsgOP, resultMsgOP and verificationOP are all parameters relevant to the verification service. The attribute channelForDuplicatedMsg specify the name of the channel that transmits duplicate messages, and MsgDuplication is similar to the same attribute of TestDataGenerator.

![Diagram of ConfigInfoRetrieval and MsgSendingToControlBus](image)

Figure 4.15 ConfigInfoRetrieval and MsgSendingToControlBus

In order to enable any filter to interact with the Control Bus, two classes, ConfigInfoRetrieval and MsgSendingToControlBus are created and associated with Filter, which is shown in Figure 4.15. As the interaction with the Control Bus is achieved by invoking the management service, both classes contain attributes describing the service such as portType, prefixForPortType, prefixForMsgType, configMsgType (msgType) and operation.

4.2.4 The Patterns

4.2.4.1 Join Router

The original idea to implement the pattern Join Router was to use a BPEL activity with multiple incoming BPEL links and one outgoing link, and to copy the variable through the activated link from the source activity. In most cases the Join Router activity in a BPEL process only receives one message and forwards it, that is, only one BPEL link is activated., no message has passed through the other paths.

Figure 4.16 illustrates how problem arises with this design. Suppose the BPEL links come from two activities respectively, one of them uses Variable1, the other
4.2 IMPLEMENTATION

Variable2. The task of the Join Router is to copy either Variable1 or Variable2 to Variable3 (when the corresponding link is activated). However, it is hard to judge which link is activated using the transition condition of the IF activity, which means that the Join Router does not know which variable is ready to be copied and which is null. In this occasion, if we try checking the value of the variable or copying any of them, we may face an exception.

To solve the problem we decided to implement the Join Router pattern with several activities, standing for several simple 1-to-1 routers and a merger, as illustrated in Figure 4.17. The copy operation is moved to the routers, and the merger just activates the BPEL link to the next activity when any of its incoming BPEL links is activated. The separation of the copy operation and the control of flow ensures that only the initialized variable will be accessed and copied, thus no risk of an exception.

The Router activity is implemented in JoinRouterPattern.java and the Merger activity in JoinRouterMergerPattern.java.

4.2.4.2 Detour

The Detour pattern is implemented in DetourPattern.java. BPELWriter invokes the method setupDetour to add BPEL code for the Detour. The method has a input parameter of type ExternalServicePattern, which is the service provided by the central management unit.
Detour compares the channel name retrieved from the configuration message and the name of both outgoing channels with the IF activity, then copies the incoming variable with an Assign activity.

### 4.2.4.3 Test Data Generator

The Test Data Generator pattern is implemented in `TestDataGeneratorPattern.java`. `BPELWriter` invokes the method `setupTestDataGeneratorNormal` to add BPEL code for a generator without the duplicate test message, or invokes the method `setupTestDataGeneratorDuplicated` to add BPEL code for a generator with the duplicate test message. Both methods have a input parameter of type `ExternalServicePattern`, which represents the service starting the test instance.

The Test Data Generator without the duplicate message functions as a Receive activity that creates a new instance when receiving a message. The extra BPEL code for the duplicate message is merely the copying of variables using Assign.

### 4.2.4.4 Test Data Verifier

The Test Data Verifier pattern is implemented in `TestDataVerifierPattern.java`. `BPELWriter` invokes the method `setupTestDataVerifierNormal` to add BPEL code for a verifier without the duplicate test message, or invokes the method `setupTestDataVerifierDuplicated` to add BPEL code for a verifier with the duplicate test message. Both methods have a input parameter of type `ExternalServicePattern`, which is the service provided by the central management unit.

The main procedure of the `setupTestDataVerifierNormal` method is as follows: if the test result is verified by a service, the verification service is configured and invoked at first. Then the method `sendErrorMsg` is called, with two input parameters, the management service and the result of the verification. The method adds an IF-ELSE activity, executing an Empty activity when the test result meets the expectation or sending an error report otherwise. At last the activity reply is performed to reply a status message to the invoker of the instance.

The method `setupTestDataVerifierDuplicated` has a similar procedure, except that multiple operations of the verification service are invoked and both the test message and the test result should be sent to the service.

### 4.2.5 BPEL Generation

To support the invocation of the management console service, the methods `generateBPELforNode` and `generatePartnerLinks` in the class `BPELWriter` was modified. The method `generateBPELforNode` contained only the configuration of one
external service for each filter, now some filters are invoking various services for different purposes, therefore the configuration of additional services is inserted to the method. Meanwhile, the partner links related to these services should be generated in the BPEL code, thus the method `generatePartnerLinks` is modified as well.

The class `WSDLWriter` is not affected.

### 4.2.6 Checklist for Developers

If a developer of the system wants to add some patterns, the following checklist could be a good reference.

1. modify the model and regenerate the code from model
2. the icon must be added in the class `EAIPaletteRoot`
3. the class `ModelCreationFactory` must be modified
4. a configuration dialog of the pattern must be inserted into the package `de.unistuttgart.iaas.eaiparam.editors.dialogs`
5. the corresponding EditPart class must be completed
6. the Father-EditParts or EditParts that are associated with the new Editpart may require modifications
7. the related Figure class must be modified so that the class Figure can be used (or/and create a .jpg figure)
8. Create- and Edit-commands must be added into the package `de.unistuttgart.iaas.eaiparam.editors.editParts.commands`
9. Delete-commands may be required
10. the class `EditAction` must be modified
11. the class `EAIEditPartFactory` must be modified
12. the class `MsgSysXYLayoutEditPolicy` must be modified
13. the class `NodeComponentEditPolicy` may require modification
14. the class `NodeEditPolicy` may require modification
15. the Pattern-class must be created in a sub-package of `de.unistuttgart.iaas.framework.xform2bpel.patterns`
4.3 Deployment and Execution of BPEL

This section explains how to deploy and run a BPEL process on ActiveBPEL engine [20]. BPEL processes were created using the system “EAI to BPEL” to simulate the system management functions in a “loan broker” scenario. Firstly a short introduction of the ActiveBPEL [20] and the application scenario, then we describe the steps of the deployment and execution.

4.3.1 Environment

A BPEL engine is needed to run BPEL processes. Nowadays there are several commercial BPEL engines such as IBM Websphere Process Server [21], Oracle BPEL [22] and many open source BPEL engines. We choose to deploy our BPEL processes on an open source engine, the ActiveBPEL engine [20], because of its high availability.

The ActiveBPEL Designer is a plugin to the Eclipse integrated development environment. It can be used to visually create BPEL process definitions by choosing partners, services and operations, and defining how data flows among those entities or to read BPEL process definitions (and other inputs such as WSDL files) and to create representations of BPEL processes [23].

Once the BPEL processes is defined, ActiveBPEL allows the user to visually simulate their execution using sample data to find errors in the BPEL process. Afterwards the ActiveBPEL Designer can automatically generate endpoint references for services used in the BPEL process as well as all of the supporting artifacts needed to deploy those processes to the ActiveBPEL engine. Processes are automatically deployed to the appropriate server location within a package that contains all required files [23].

The ActiveBPEL engine runs in any standard servlet container such as Apache Tomcat [24]. It is possible to remotely debug a process running on the server and suspend a process on an uncaught fault to perform exception management. Besides, the engine takes care of persistence, queues, alarms, and many other execution details [23].

4.3.2 The Application Scenario
4.3 DEPLOYMENT AND EXECUTION OF BPEL

Our BPEL processes are created for the application scenario “Loan Broker”, which is illustrated in Figure 4.18. The patterns in the figure compose the loan broker application in a Scatter-Gather style.

The loan broker gets loan requests from clients and returns the best loan quotes back to them. Inside the loan broker, the steps of message processing are listed as follows:

1. the input loan request is forwarded to a credit bureau service, which checks the credit score of the customer
2. the loan request along with the credit score is sent through a Recipient List to various bank services
3. the banks make loan quotes according to the credit score and send the quotes to the Aggregator
4. the Aggregator selects the best offer and passes it to the customer

If any error occurs with the credit bureau service, the bank services may face wrong information. To make the credit score gathering process more reliable, we add a secondary credit bureau service to the loan broker.
Figure 4.19 shows the system with two credit bureau services. A Detour is inserted before the credit services to intercept the loan request. The central management console configures the detour, informing it about where the loan request should be processed, at the primary credit bureau or the secondary one. A Join Router is also needed after the credit services in order to router the message from any credit service to the Recipient List.

![Diagram](attachment:image.png)

Figure 4.20 Testing the primary credit bureau

The primary credit service is the default service, whereas the secondary service is a backup one. To keep the management console updated with the running status of the primary credit service, the system can periodically send test messages to it and analyze the test result. The test loan request has the identical message type as the normal loan request, and the credit bureau treats it like normal requests. If an error is found, the Test Data Verifier sends an error report to the management console, then the console will lead the request message flow to the secondary credit service via controlling the Detour.

According to the original description, a Test Message Injector and a Test Message Separator are also the components of the Test Message pattern. However, it is already explained in section 3.4.1.7 that the Test Data Generator and the Test Data Verifier are sufficient for a BPEL process to perform the testing, thus we only see these two components in Figure 4.20.

To simulate the requests routing mechanism described above, we created two messaging system in two EAI files. One contains the process in Figure 4.19 expect the Recipient List, which means the Join Router is the last activity of the main flow. The other contain the process in Figure 4.20.

A BPEL file and a WSDL file was generated for each process respectively. Now we have the files LoanCredit.bpel, LoanCredit.wsdl, TestLoanCredit.bpel and TestLoanCredit.wsdl, which can be used after small modifications by the ActiveBPEL Designer for deployment and execution of the two BPEL processes.

### 4.3.3 Deployment

Before the deployment we have to modify the generated files and create WSDL files specifying the message types and the port types used in our BPEL processes. In
addition, to eliminate all errors, it is suggested to simulate the processes using the
ActiveBPEL Designer before trying to run them. Since the simulation itself does not
require information like endpoint references or service binding, it can be added either
before or after the simulation.

4.3.3.1 Preparation for Simulation

We take the file LoanCredit.bpel as the example, the process for the test message
should be handled in a similar way.

Some namespace such as msg and port are used in the BPEL file and the relevant
WSDL file, we created two files message.wsdl and port.wsdl to define the message
types and the port types under these namespaces. The files are displayed in the
appendix.

The messages types in the BPEL process are all using the prefix msg, thus
message.wsdl defines all the types: detourConfiguration, detourConfigRequest,
loanRequest and loanRequestWithCredit.

The loanRequest type contains the parts for the Social Security Number (SSN), the
loan amount and the loan term. It is for the input message of the credit service, and the
type loanRequestWithCredit for the output of the service. This type should have
extra parts on the basis of loanRequest to reflect the credit score and other
information. However, due to the problem of the serialization/deserialization errors
during the execution of BPEL processes, we had to set the type
loanRequestWithCredit with only one part indicating the credit score.

In the file port.wsdl we defined the port type consolePT for the service of the
management console, the port type creditPT1 and creditPT2 for the two credit
bureau services.

The newly created WSDL files must be imported into both LoanCredit.wsdl and
LoanCredit.bpel.

ActiveBPEL Designer accepts WSDL elements with a wsdl prefix and BPEL
elements with a bpel prefix. It can read the bpel files generated by the system “EAI to
BPEL” and add the prefix to their elements automatically. Nevertheless, it cannot do
the same to WSDL files, thus the wsdl prefix must be manually added.

Some small changes were performed additionally, most of them were caused by
syntax inconsistence between the generated files and the accepted standards, which
can be solved by modifying the generation procedure of the editor.

As shown in Figure 4.21, ActiveBPEL provides a Web References pane, where the
user can add all WSDL files and XSD schemas as web references so that the BPEL
process is able to recognize the elements defined in these files during the simulation.
When the BPEL process is free of errors, the simulation can be started.
4.3 DEPLOYMENT AND EXECUTION OF BPEL

4.3.3.2 Simulation

In essence, the simulation is an execution of the BPEL process with sample messages and overridden outputs for involved web services. Figure 4.22 is the message view of the **Web References** pane, which presents the message types defined in WSDL files. The user can add sample values for any part of the message type. The output messages of the web services can be overridden with a fixed value.

The user can view both the text and the graphical representation of a BPEL file. The simulation is started under the graphical view, from which the user can see what activity is currently being executed. During the simulation the contents of the BPEL variables can also be inspected. The simulation may terminate due to errors, or stop after executing the last activity in the process, indicating the BPEL process is ready to be deployed.
4.3 DEPLOYMENT AND EXECUTION OF BPEL

4.3.3.3 Deployment

Before deployment the endpoint references must be specified. We start with defining bindings and services for the three involved web services in LoanCredit.wsdl.

By selecting File → New → Deployment Descriptor in the menu, the user can use the wizard dialogs to configure the generation and then let ActiveBPEL generate the deployment descriptor file (.pdd file) LoanCredit.pdd automatically.

The main part of the descriptor is the endpoint definition for partner links. For services invoked by the BPEL process, the user can set the endpoint type of the corresponding partner link to *static*, then the endpoint reference will be shown in the text box in the wizard dialog. If the user tries to generate endpoint references when some information is missing, ActiveBPEL will fill in the file with place holding texts.

For the each partner link of *myrole* type, the user has to choose a binding type, such as *RPC Encoded* or *Document Literal*. We used *RPC Encoded* for our process, meaning that the body of a SOAP request has an outer element which matches the operation name and contains inner elements each of which maps to a parameter of the operation.

To deploy the process to an ActiveBPEL server, all relevant files should be added to a process deployment package (.bpr file), which is similar to a Web archive file. The archiving process is automated in the ActiveBPEL Designer. The .bpr file is generated in the project by default, but the user may also choose to deploy it directly under the server folder `/bpr`. When the ActiveBPEL engine is running, it scans this folder for new, changed, and deleted files.

4.3.4 Execution
In order to execute BPEL processes, the services invoked by the process must be deployed on the server. We also developed a simple interface to invoke the BPEL processes and to display the execution results.

### 4.3.4.1 Web services

So far we have only defined the endpoints of services, obviously the services should be implemented and deployed before being invoked.

```java
public class CreditService1 {
    RuntimeParams rp;

    public CreditService1() {
    }

    public BigInteger getCredit(String SSN, BigInteger amount, BigInteger term) throws Exception {
        rp = new RuntimeParams("eaitobpel_default.xml");
        BigInteger creditScore = new BigInteger(rp.getText("/rundata/credit1/score"));
        System.out.println(creditScore.toString());
        return creditScore;
    }
}
```

Above is the JAVA implementaion of the primary credit service. Using a supporting class that enables the reading and writing of XML files, the method `getCredit` receives three input parameters indicating the SSN, the loan amount and the loan term, reads a predefined credit score value from an XML file and returns it. The secondary credit service functions identically.

After the JAVA source files are built, the package of .class files should be copied into the server folder `bpr\LoanCredit` and `webapps\LoanCredit\WEB-INF\classes`. In addition, a Web Service deployment descriptor file (.wsdd file) is required under the folder `bpr\LoanCredit\META-INF`, which contains the specification of the web services and relevant information such as the operations and their parameters. The `service.wsdd` is manually written, it can be found in the appendix. The deployment of the web services is accomplished by archiving the .class files and the .wsdd file into a package (.wsr file) and placing it under the server folder `bpr`.

### 4.3.4.2 Execution Interface

The interface for the execution was implemented in the `index.jsp` file and the `testcredit.jsp` file under `webapps\LoanCredit`.
4.3 DEPLOYMENT AND EXECUTION OF BPEL

Figure 4.24 shows the interface. The user can enter values for the SSN, the loan amount and the loan term. The URL and operation used to invoke the BPEL process must be given as well. After filling in all text boxes, just click the button **Apply for a loan** to execute the process.

![Loan Credit Client Interface](image)

**Figure 4.24 The interface for execution**

After a successful execution of the BPEL process, a message like **“The process has been successfully executed. The reply from the process is: "800"”** is displayed. In case error occurs during the execution, the exception message is displayed.

To save the typing and keep the interaction as simple as possible, we save a set of values for all the five parameters in `eaitobpel_default.xml` and load them as default values each time the page is refreshed.

The interface for the execution of `TestLoanCredit.bpm` is quite similar, except that the result displayed is not the credit score, but the currently working credit service.

### 4.3.4.3 Execution Result

So far we have deployed two BPEL processes and three web services. Firstly we clarify how they function.

When receiving a loan request, a credit service just reads the predefined value from `eaitobpel_default.xml` as the credit score. The two services `CreditService1` and `CreditService2` have different entries in the XML file, thus may return different results.

Another XML file `eaitobpel_console.xml` is used by `ManagementService`. The Detour controlled by the management console has two exits, one to the primary credit bureau (named creditA), the other to the secondary credit bureau (named creditB). The Detour has an entry in `eaitobpel_console.xml`, specifying which exit should be used. When the Detour invokes the `configDetour` operation of the management service, the
service reads the data in this entry and sends it the Detour. The default credit service is the primary credit bureau (creditA).

The other operation of ManagementService, sendTestResult, is invoked by the Test Data Verifier when it finds out that the test result is out of the expected range. When the management service receives this error report, it changes the exit of Detour to “creditB” in eaitobpel_console.xml, so that loan requests are sent to the backup credit service.

We execute the following steps to simulate the system management functionalities:

- Use the link http://localhost:8080/LoanCredit/index.jsp to run the process LoanCredit. The result is the credit score from the primary credit service, which should be more than 300 and lower than 900.

- Modify eaitobpel_default.xml manually, change credit score of the primary credit service to an invalid value, such as 100.

- Run the process LoanCredit. The result should be 100 now.

- Use the link http://localhost:8080/LoanCredit/testcredit.jsp to run the process TestLoanCredit. It detects the error and asks the management service to change the configuration of Detour. When this is done, a message indicates that now the secondary credit service is in use instead of the primary one.

- Run the process LoanCredit. The result provided by the secondary service is in the normal range.

In the reality the test messages are sent periodically, thus most loan requests can be directed to the backup service in time when the primary one does not function correctly. The Test Data Verifier should also be able to report it to the management console when it finds out the primary service has recovered and is free of errors again, the implementation is left for further work.
Chapter 5

Summary

The main objective of this work is to parameterize selected EAI patterns from the categories Message Routing and System Management and to find the possible ways to map the EAI patterns onto BPEL processes. The characteristics of totally 14 patterns were studied, then we specified the inputs, outputs and the parameters for each pattern to reflect their characteristics. We also provided ideas for the BPEL representation of these patterns, some of them can be simulated by pure BPEL code, whereas some need the help of external services.

In order to prove that the design of the BPEL simulation works well in the reality, we generated BPEL processes for an application scenario, the “Loan Broker”. Two processes were used to realize the functionalities of three system management patterns. They were deployed and executed on the ActiveBPEL engine, and the result was as expected.

The System “EAI to BPEL” is an editor implemented by Florian Schebelle [12] and Bettina Druckenmueller [9], which is used to create EAI messaging systems and is able to automatically generate BPEL implementation from them. We modified and extended the system so that the patterns used for our simulation can be supported.

Further researches may have several directions. So far only a portion of the parameterized patterns are implemented in the system “EAI to BPEL”, and some patterns from Hohpe’s book [2] are still to be researched.

The BPEL processes were only deployed on the ActiveBPEL engine, we can study the deployment possibilities on other engines and try to increase the level of automation in deployment.

It will also be interesting to find an alternative implementation of the Control Bus and some management related functions such as configuration of processors in a context based environment.
Bibliography

[1] Prof. Dr. Frank Leymann: Lecture Material “Message Based Applications”, University of Stuttgart, Winter Semester 2005/06


Appendix A

A.1 BPEL file

LoanCredit.bpel

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- BPEL Process Definition
Edited using ActiveBPEL(r) Designer Version 4.1.0 (http://www.active-endpoints.com)-->

<bpel:process
xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
xmlns:corr="http://message.com/correlationSet"
xmlns:ele="http://example.com/messageelement"
xmlns:home="http://example.com/wsd1"
xmlns:msg="http://example.com/message"
xmlns:ns1="http://example.com/complextype"
xmlns:port="http://example.com/port"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
name="LoanCredit"
suppressJoinFailure="yes"
targetNamespace="http://LoanCredit">

<bpel:import
importType="http://schemas.xmlsoap.org/wsdl/
location="WSDL/message.wsdl" namespace="http://example.com/message"/>
<bpel:import
importType="http://schemas.xmlsoap.org/wsdl/
location="WSDL/port.wsdl" namespace="http://example.com/port"/>
<bpel:import
importType="http://schemas.xmlsoap.org/wsdl/
location="WSDL/LoanCredit.wsdl" namespace="http://example.com/wsd1"/>

<bpel:partnerLinks>
<bpel:partnerLink myRole="MyRole1" name="PartnerLink1"
partnerLinkType="home:PartnerLinkType1"/>
<bpel:partnerLink name="PartnerLink2"
partnerLinkType="home:PartnerLinkType2"
partnerRole="PartnerRole2"/>
<bpel:partnerLink name="PartnerLink3"
partnerLinkType="home:PartnerLinkType3"
partnerRole="PartnerRole3"/>
<bpel:partnerLink name="PartnerLink4"
partnerLinkType="home:PartnerLinkType4"
partnerRole="PartnerRole4"/>
</bpel:partnerLinks>

<bpel:variables>
<bpel:variable
messageType="msg:loanRequest" name="Variable1"/>
<bpel:variable
messageType="msg:loanRequest" name="Variable2"/>
<bpel:variable
messageType="msg:loanRequest" name="Variable3"/>
<bpel:variable
messageType="msg:loanRequestWithCredit"
name="Variable4"/>
<bpel:variable
messageType="msg:loanRequestWithCredit"
name="Variable5"/>
<bpel:variable
messageType="msg:loanRequestWithCredit"
name="Variable6"/>
<bpel:variable
messageType="msg:detourConfiguration"
name="Detour1Variable"/>
<bpel:variable
messageType="msg:detourConfigRequest"
```
APPENDIX

```xml
<bpel:sequence name="ExternalService1">
  <bpel:sources>
    <bpel:source linkName="Link0"/>
  </bpel:sources>
  <bpel:receive createInstance="yes" name="Receive1" operation="getLoanCredit" partnerLink="PartnerLink1" portType="home:customerPT" variable="Variable1"/>
</bpel:sequence>

<bpel:sequence name="Detour1">
  <bpel:targets>
    <bpel:target linkName="Link0"/>
  </bpel:targets>
  <bpel:sources>
    <bpel:source linkName="Link3">
      <bpel:transitionCondition>
        $Detour1Variable.detourConfiguration = "creditA"
      </bpel:transitionCondition>
    </bpel:source>
    <bpel:source linkName="Link4">
      <bpel:transitionCondition>
        $Detour1Variable.detourConfiguration = "creditB"
      </bpel:transitionCondition>
    </bpel:source>
  </bpel:sources>
  <bpel:assign name="Assign1">
    <bpel:copy>
      <bpel:from>"Detour1"</bpel:from>
      <bpel:to part="component" variable="Detour1ConfigInput"/>
    </bpel:copy>
  </bpel:assign>
</bpel:sequence>

<bpel:invoke inputVariable="Detour1ConfigInput" name="Invoke1" operation="configDetour" outputVariable="Detour1Variable" partnerLink="PartnerLink2" portType="port:consolePT"/>

<bpel:if name="If1">
  <bpel:condition>$Detour1Variable.detourConfiguration = "creditA"</bpel:condition>
  <bpel:assign name="Assign2">
    <bpel:copy>
      <bpel:from variable="Variable1"/>
      <bpel:to variable="Variable2"/>
    </bpel:copy>
  </bpel:assign>
</bpel:if>
```

<bpel:condition>$Detour1Variable.detourConfiguration = "creditB"</bpel:condition>
<bpel:assign name="Assign3">
  <bpel:copy>
    <bpel:from variable="Variable1"/>
    <bpel:to variable="Variable3"/>
  </bpel:copy>
</bpel:assign>
</bpel:elseif>
</bpel:if>
</bpel:sequence>

<bpel:sequence name="ContentEnricher0">
  <bpel:targets>
    <bpel:target linkName="Link3"/>
  </bpel:targets>
  <bpel:sources>
    <bpel:source linkName="Link1"/>
  </bpel:sources>
  <bpel:sequence name="ExternalService3">
    <bpel:invoke inputVariable="Variable2" name="Invoke3" operation="getCredit" outputVariable="Variable5" partnerLink="PartnerLink3" portType="port:creditPT1"/>
  </bpel:sequence>
</bpel:sequence>

<bpel:sequence name="ContentEnricher1">
  <bpel:targets>
    <bpel:target linkName="Link4"/>
  </bpel:targets>
  <bpel:sources>
    <bpel:source linkName="Link2"/>
  </bpel:sources>
  <bpel:sequence name="ExternalService4">
    <bpel:invoke inputVariable="Variable3" name="Invoke5" operation="getCredit" outputVariable="Variable6" partnerLink="PartnerLink4" portType="port:creditPT2"/>
  </bpel:sequence>
</bpel:sequence>

<bpel:sequence name="JoinRouter1">
  <bpel:targets>
    <bpel:target linkName="Link2"/>
  </bpel:targets>
  <bpel:sources>
    <bpel:source linkName="Link6"/>
  </bpel:sources>
  <bpel:assign name="Assign8">
    <bpel:copy>
      <bpel:from variable="Variable6"/>
      <bpel:to variable="Variable4"/>
    </bpel:copy>
  </bpel:assign>
</bpel:sequence>

<bpel:sequence name="JoinRouter2">
  <bpel:targets>
    <bpel:target linkName="Link1"/>
  </bpel:targets>
  <bpel:sources>
    <bpel:source linkName="Link7"/>
A.2 WSDL file

**Message.wsdl**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://example.com/message"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:message name="detourConfiguration">
    <wsdl:part name="detourConfiguration" type="xsd:string"/>
  </wsdl:message>

  <wsdl:message name="detourConfigRequest">
    <wsdl:part name="component" type="xsd:string"/>
  </wsdl:message>

  <wsdl:message name="loanRequest">
    <wsdl:part name="SSN" type="xsd:string"/>
    <wsdl:part name="loanAmount" type="xsd:integer"/>
    <wsdl:part name="loanTerm" type="xsd:integer"/>
  </wsdl:message>

  <wsdl:message name="loanRequestWithCredit">
    <wsdl:part name="SSN" type="xsd:string"/>
    <wsdl:part name="loanAmount" type="xsd:integer"/>
    <wsdl:part name="loanTerm" type="xsd:integer"/>
  </wsdl:message>
</wsdl:definitions>
```
```xml
<wsdl:part name="creditResponse" type="xsd:integer"/>
</wsdl:message>

<wsdl:message name="testResult">
  <wsdl:part name="testResult" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="testStatus">
  <wsdl:part name="testStatus" type="xsd:string"/>
</wsdl:message>

</wsdl:definitions>

Port.wsdl

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://example.com/port"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:msg="http://example.com/message"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:import namespace="http://example.com/message"
location="message.wsdl"/>
  <wsdl:portType name="creditPT2">
    <wsdl:operation name="getCredit">
      <wsdl:input message="msg:loanRequest"/>
      <wsdl:output message="msg:loanRequestWithCredit"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:portType name="creditPT1">
    <wsdl:operation name="getCredit">
      <wsdl:input message="msg:loanRequest"/>
      <wsdl:output message="msg:loanRequestWithCredit"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:portType name="consolePT">
    <wsdl:operation name="configDetour">
      <wsdl:input message="msg:detourConfigRequest"/>
      <wsdl:output message="msg:detourConfiguration"/>
    </wsdl:operation>
    <wsdl:operation name="sendTestResult">
      <wsdl:input message="msg:testResult"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>

LoanCredit.wsdl

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://example.com/wsdl"
xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
APPENDIX

<wsdl:import namespace="http://example.com/port"
location="port.wsdl"/>
<wsdl:import namespace="http://example.com/message"
location="message.wsdl"/>

<wsdl:portType name="customerPT">
  <wsdl:operation name="getLoanCredit">
    <wsdl:input message="msg:loanRequest"/>
    <wsdl:output message="msg:loanRequestWithCredit"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="ManagementBinding" type="port:consolePT">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="configDetour">
    <wsdl:input>
      <soap:body use="encoded"
        namespace="http://example.com/message"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="encoded"
        namespace="http://example.com/message"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="Management">
  <wsdl:documentation>System Management Service</wsdl:documentation>
  <wsdl:port name="SOAPPort3" binding="home:ManagementBinding">
    <soap:address location="http://localhost:8080/active
    bpel/services/ManagementService"/>
  </wsdl:port>
</wsdl:service>

<wsdl:binding name="CreditBinding1" type="port:creditPT1">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="getCredit">
    <wsdl:input>
      <soap:body use="encoded"
        namespace="http://example.com/message"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="encoded"
        namespace="http://example.com/message"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="Credit1">
    <wsdl:documentation>Credit Service 1</wsdl:documentation>
    <wsdl:port name="SOAPPort4" binding="home:CreditBinding1">
        <soap:address location="http://localhost:8080/activebpel/services/CreditService1"/>
    </wsdl:port>
</wsdl:service>

<wsdl:service name="Credit2">
    <wsdl:documentation>Credit Service 2</wsdl:documentation>
    <wsdl:port name="SOAPPort5" binding="home:CreditBinding2">
        <soap:address location="http://localhost:8080/activebpel/services/CreditService2"/>
    </wsdl:port>
</wsdl:service>

<plnk:partnerLinkType name="PartnerLinkType1">
    <plnk:role name="MyRole1" portType="home:customerPT"/>
</plnk:partnerLinkType>

<plnk:partnerLinkType name="PartnerLinkType2">
    <plnk:role name="PartnerRole2" portType="port:consolePT"/>
</plnk:partnerLinkType>

<plnk:partnerLinkType name="PartnerLinkType3">
    <plnk:role name="PartnerRole3" portType="port:creditPT1"/>
</plnk:partnerLinkType>

<plnk:partnerLinkType name="PartnerLinkType4">
    <plnk:role name="PartnerRole4" portType="port:creditPT2"/>
</plnk:partnerLinkType>

</wsdl:definitions>

A.3 PDD file

LoanCredit.pdd

<?xml version="1.0" encoding="UTF-8"?>
<process xmlns="http://schemas.active-endpoints.com/pdd/2006/08/pdd.xsd"
xmlns:bpelns="http://LoanCredit"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
location="bpel/LoanCredit/LoanCredit.bpel"
name="bipelns:LoanCredit"
persistenceType="full">

<partnerLinks>

<partnerLink name="PartnerLink1">
  <myRole allowedRoles="" binding="RPC"
            service="LoanCreditService"/>
</partnerLink>

<partnerLink name="PartnerLink2">
  <partnerRole endpointReference="static">
    <wsa:EndpointReference xmlns:s="http://example.com/wsdl"
                          xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing">
      <wsa:Address>http://localhost:8080/active-bpel/services/ManagementService</wsa:Address>
      <wsa:ServiceName PortName="SOAPPort3">s:Management</wsa:ServiceName>
    </wsa:EndpointReference>
  </partnerRole>
</partnerLink>

<partnerLink name="PartnerLink3">
  <partnerRole endpointReference="static">
    <wsa:EndpointReference xmlns:s="http://example.com/wsdl"
                          xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing">
      <wsa:Address>http://localhost:8080/active-bpel/services/CreditService1</wsa:Address>
      <wsa:ServiceName PortName="SOAPPort4">s:Credit1</wsa:ServiceName>
    </wsa:EndpointReference>
  </partnerRole>
</partnerLink>

<partnerLink name="PartnerLink4">
  <partnerRole endpointReference="static">
    <wsa:EndpointReference xmlns:s="http://example.com/wsdl"
                          xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing">
      <wsa:Address>http://localhost:8080/active-bpel/services/CreditService2</wsa:Address>
      <wsa:ServiceName PortName="SOAPPort5">s:Credit2</wsa:ServiceName>
    </wsa:EndpointReference>
  </partnerRole>
</partnerLink>

</partnerLinks>

<references>

<wsdl location="project:/LoanCredit/WSDL/message.wsdl"
      namespace="http://example.com/message"/>

<wsdl location="project:/LoanCredit/WSDL/port.wsdl"
      namespace="http://example.com/port"/>

<wsdl location="project:/LoanCredit/WSDL/LoanCredit.wsdl"
      namespace="http://example.com/wsdl"/>

</references>
</process>
A.4 WSDD file

Service.wsdd

```xml
<deployment xmlns="http://xml.apache.org/axis/wsdd/
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
xmlns:msg="http://example.com/message">

<service name="ManagementService"
    provider="java:RPC"
    xmlns:ns="http://example.com/wsdl">
    <namespace>http://example.com/wsdl</namespace>
    <parameter name="className"
        value="de.unistuttgart.iaas.eaitobpel.ws.ManagementService"/>
    <parameter name="allowedMethods" value="*"/>
    <operation name="configDetour"
        returnQName="msg:detourConfiguration"
        returnType="xsd:string">
        <parameter name="component" type="xsd:string"/>
    </operation>
    <operation name="sendTestResult">
        <parameter name="testResult" type="xsd:string"/>
    </operation>
</service>

<service name="CreditService1" provider="java:RPC"
    xmlns:ns="http://example.com/wsdl">
    <namespace>http://example.com/wsdl</namespace>
    <parameter name="className"
        value="de.unistuttgart.iaas.eaitobpel.ws.CreditService1"/>
    <parameter name="allowedMethods" value="getCredit"/>
    <operation name="getCredit"
        returnQName="msg:loanRequestWithCredit"
        returnType="xsd:integer">
        <parameter name="SSN" type="xsd:string"/>
        <parameter name="loanAmount" type="xsd:integer"/>
        <parameter name="loanTerm" type="xsd:integer"/>
    </operation>
</service>

<service name="CreditService2" provider="java:RPC"
    xmlns:ns="http://example.com/wsdl">
    <namespace>http://example.com/wsdl</namespace>
    <parameter name="className"
        value="de.unistuttgart.iaas.eaitobpel.ws.CreditService2"/>
    <parameter name="allowedMethods" value="getCredit"/>
    <operation name="getCredit"
        returnQName="msg:loanRequestWithCredit"
        returnType="xsd:integer">
        <parameter name="SSN" type="xsd:string"/>
        <parameter name="loanAmount" type="xsd:integer"/>
        <parameter name="loanTerm" type="xsd:integer"/>
    </operation>
</service>
</deployment>
```
Authors Statement

I certify that I have realized this work on my own, and that all sources that I have used or consulted are duly noted herein.


Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.


____________________________
(Xin Yuan)