Vrije Universiteit Brussel

Faculty of Science
Department of Computerscience
System & Software Engineering Lab (SSEL)

Implementation of Stateful Aspects in Padus

Dimitri Gheysels

2006-2007

Promotor: Prof. Dr. Viviane Jonckers
Co-Promotor: Dr. Wim Vanderperren

Supervisors: Mathieu Braem & Niels Joncheere

Thesis submitted in fulfilment of the requirements for the degree of Master in Applied Computer Science

Acknowledgements

The successful fulfilment of a thesis is a hard job and brings a lot of work with it. Many people are involved in this achievement, either directly or indirectly. Therefore, I find it important to express my thanks to those people.

First of all, I would like to thank my promotor, Prof. Dr. Viviane Jonckers and my co-promoter Dr. Wim Vanderperren for giving me the opportunity of doing this thesis at the System and Software Engineering Lab of the VUB.

Invaluable was the guidance of Dr. Wim Vanderperren, Mathieu Braem and Niels Joncheere during this research work. The ideas, feedback and remarks they gave were indispensable. I also appreciate very much the time they took to proof read this dissertation.

Last but not least, I would like to thank my family for giving the moral support during this hard time. The patience they must have should not be underestimated at all.

Dimitri Gheysels
June 2007

Table of Contents

Abstract

Workflow languages offer a numerous of advantages over general-purpose programming languages when it comes to business process engineering. The implementation of a business process often involves a composition of many business rules. Because it is generally known that business rules are a crosscutting concern, the aspect-oriented programming paradigm could be successfully combined with workflow languages in order to separate such crosscutting concerns from the base application. The cooperation of these two technologies should be very useful and could provide a real asset for business process engineering.

The current aspect-oriented implementations are rather limited for workflow languages. Although they are sufficient for the implementation of basic crosscutting concerns, more advanced problems cannot be solved straightforward with the basic concepts they currently provide. When facing the implementation of business protocols, where the state of the application has to be tracked, these aspect-oriented implementations fall short. A business protocol is still crosscutting if only basic aspect-oriented concepts are used for its implementation.
To handle this issue, stateful aspects are a very useful concept in aspect-oriented programming. With stateful aspects, extra advice behaviour can be executed based on the execution history of a program. This concept is already implemented in aspect-oriented extensions for general-purpose programming languages and proves to be very useful, and we claim that the domain of business process engineering also benefits from it.

Padus is a particular aspect-oriented extension for the BPEL workflow language. In this thesis, the concept of stateful aspects is added to Padus in order to control the crosscutting nature of business protocols. This extension mainly includes the design and implementation of a small language to describe a protocol, and the modification of the weaver so that the protocol is understood and woven correctly in the process.
Because some parts of the BPEL process might run in parallel, the order of the executed activities of the process is not deterministically defined anymore. This creates an additional problem on weaving business protocols because the order in which the actions of the process are executed can differ depending on the time of execution. This means that the same protocol might not match even when the same data is given as input to the same process. If it is required that the protocol matching is independent of the time the process is executed, then this should be supported too. A solution to this problem is therefore elaborated and presented in this dissertation.

Samenvatting

Ten opzichte van general-purpose programmeertalen, bieden workflow talen een waaier van voordelen voor het ontwikkelen en implementeren van bedrijfsprocessen. Meerdere bedrijfsregels worden geïmplementeerd en gecombineerd in een dergelijk bedrijfsproces en het is algemeen gekend dat dergelijke bedrijfsregels voorbeelden zijn van *crosscutting concerns*. Om deze reden is het zeer nuttig om het aspect-georiënteerde programmeerparadigma te combineren met workflow talen. Deze combinatie biedt tal van voordelen tijdens het ontwikkelen van bedrijfsprocessen.

Momenteel bieden de aspect-georiënteerde uitbreidingen voor workflow talen eerder een beperkte functionaliteit en ondersteunen ze enkel de basisconcepten van aspect-georiënteerd programmeren. Meer complexe crosscutting problemen kunnen nog altijd niet eenvoudig opgelost worden. Een voorbeeld daarvan is de implementatie van bedrijfsprotocollen, waarbij de huidige toestand van de applicatie nauwkeurig moet bijgehouden worden. Door enkel gebruik te maken van de basisconcepten van aspect-georiënteerd programmeren, blijft de implementatie van een bedrijfsproces nog steeds crosscutting.
Een geavanceerd concept binnen het domain van aspect-georiënteerd programmeren zijn *stateful aspects*. Hierbij kunnen stukken programmacode uitgevoerd worden afhankelijk van de toestanden waarin het proces zich heeft bevonden tijdens de uitvoering ervan. Dit concept heeft reeds zijn nut bewezen in tal van andere software domeinen. Dit concept biedt ook voor de implementatie van bedrijfsprocessen een toegevoegde waarde.

Padus is een specifieke aspect-georiënteerde uitbreiding voor de BPEL workflow taal. In deze tekst wordt beschreven hoe Padus uitgebreid wordt met stateful aspects zodat ook bedrijfsprotocollen op een modulaire manier geïmplementeerd kunnen worden. De uitbreiding van Padus bevat enerzijds het ontwerp en implementatie van een taal om protocollen te kunnen beschrijven en anderzijds wordt ook de weaver aangepast opdat een protocol correct geweven kan worden in het BPEL proces.
Echter, bepaalde delen van een BPEL proces kunnen simultaan uitgevoerd worden waardoor het verloop van het proces niet meer deterministisch vastgesteld kan worden. Dit zorgt voor een bijkomend probleem bij het weven van bedrijfsprotocollen omdat de volgorde van de uitgevoerde taken niet meer vastligt. Het matchen van het protocol is daardoor afhankelijk van het moment waarop het proces uitgevoerd wordt. Het staat niet vast dat eenzelfde protocol altijd matched met een identieke uitvoering van een proces waarbij dezelfde data als input gegeven is. De aspect programmeur moet de mogelijkheid hebben om ervoor te zorgen dat het protocol altijd matched, onafhankelijk van het moment waarop het proces uitgevoerd wordt. Dit probleem wordt ook besproken in deze verhandeling en een uitgewerkte oplossing wordt eveneens voorgesteld.

Chapter 1: Introduction

1 Thesis Research Context

Since the advent of the webservices technology around the turn of the century, service-oriented architectures (SOA) soon became very popular. Essentially, SOAs are a methodology of software design where the collaborating software components are independent services. This kind of software architecture promotes a flexible, transparent and maintainable implementation of distributed software systems because the different components (services) can be located everywhere. Today, service-oriented architectures are very much used within a business perspective. On the one hand, this is because these kinds of software systems have to interoperate frequently with those of the business partners. On the other hand, software systems implemented as a SOA can be easily adapted to the ever-changing business market. [55]

In service-oriented architectures, services work together and communicate with each other in order to implement some process or task. It is evident that the dataflow between the different services must be coordinated well. In other words, the services must be orchestrated with each other to perform a certain task. Therefore, workflow languages provide a huge support for SOAs to orchestrate the services and manage the coordination between them. Since business processes are easily modelled as workflows, such kinds of languages are very suitable. [41][42]. WS-BPEL [2] is a well-known workflow language and can be considered as the de-facto standard in business process engineering.

Unfortunately, the implementation of business processes with the aid of workflow languages is not always straightforward. For example, within a business process the different business rules are mostly tangled with each other and with the core business process itself: they crosscut each other [18]. This is a real disadvantage because the maintainability, reusability and evolvability of the software system, and business processes in particular, are suffering from this. Business rules are very volatile and change often and with maintainability in mind, they should be decoupled and untangled from the business process. Currently, with the aid of aspect-oriented programming [47], there is some ongoing research to tackle this problem within workflow languages.
Another important issue which is frequently encountered within business processes are protocols. It occurs often that some behaviour must be invoked after a particular sequence of events has been executed. This means that the history of these events must be traced out. Consequently, the part of the application which keeps track of this history is tangled throughout the business rule. Again, with aspect-oriented programming, solutions are available to handle this issue [61][62], but unfortunately they are all suited for general-purpose programming languages and not for workflow languages.

Aspect-oriented programming (AOP) [47] is a recent programming paradigm which provides us with new techniques and mechanisms for modularizing crosscutting concerns in software applications. Crosscutting concerns are functionalities of an application which cannot be cleanly modularized and therefore are tangled and scattered across the whole application. Well-known crosscutting concerns are logging, transaction control, security management and also business rules.
Current AOP languages are complementary to some other programming language;

mostly object-oriented languages because these are very popular to date. As a result, support for aspect-oriented programming is not well integrated into workflow languages; there exist only a few implementations [11]. Padus is a language still under development at the System & Software Engineering Lab (SSEL) at the VUB, for defining crosscutting concerns in WS-BPEL [7].

2 Problem Statement

Because aspect-oriented programming is a very recent programming paradigm, there is still much research going on in this particular domain of software engineering. Initially, AOP research was mainly focused around a static joinpoint model, which means that aspects could only be invoked on static points in the structure of the application. Later on it became also possible to invoke aspects when certain dynamic evaluated conditions were met. Current AOP implementations offer therefore a dynamic joinpoint model. This is a model where the joinpoints can be run-time events based on the current program execution.

Currently, there is very much ongoing research about event-based AOP and stateful aspects [28][29]. This domain of AOP makes it possible to define aspects based on a sequence of events or a protocol in a declarative way. Describing protocols is also possible with traditional AOP, but this is a very cumbersome task because the programmer must handle the history of events within the aspect itself. Therefore, aspects become quickly unreadable and it is also undesirable because the aspect-logic gets mixed with the applicability of the aspect.

Event-based AOP or stateful aspects are not very well supported in commercially available implementations of AOP. Nevertheless, in many application domains, protocols are frequently encountered: component-based software development and business-processes are some of them.
JAsCo [57] is one of the few AOP languages which provides decent support for stateful aspects in the domain of component-based software development. Unfortunately, there is still no implementation or any support available for stateful aspects within the domain of business-process engineering.

With the aid of Padus, AOP for workflow languages is emerging and will give business process engineering another boost. Because protocols are very much present within business process engineering, support for stateful aspects within Padus is very desirable. In this dissertation, Padus is extended with a proof-of-concept implementation of stateful aspects. The main parts in this implementation are:

- The design of a language to describe protocols declaratively and attach behaviour to it.
- Extending the weaver so that it is possible to recognize the protocol and weave the behaviour in the process correctly.
- Handling possible non-deterministic behaviour of some BPEL processes in the protocol specification.

The pointcut language of Padus is extended with a sub-language to describe protocols. The design of this language is an important part of the implementation because this is what the developer uses. It is important that a protocol could be written as declaratively

as possible which means that nothing about *how* the protocol matching should be done is mentioned in the language.
The second part involves the extension of the weaver. Padus should be able to understand a protocol and weave it in the process in such a way that the history of events is tracked and the current state of the process is recognized. Algorithms are implemented to create an internal representation of the protocol which is then used for weaving the protocol.
The last part describes the non-deterministic behaviour which might be present at some processes. This is a real issue because such behaviour heavily influences the way how protocols can be recognized properly. This problem is completely analyzed and a promising solution is also described.

3 Thesis Organisation

The first four chapters in this dissertation explain webservices, service-oriented architectures, workflow languages and aspect-oriented programming. In these chapters, the reader will remark that business processes and their implementation are frequently mentioned, although this is not an essential part in this dissertation. The reason the domain of business process applications is used as some sort of guideline is because it motivates the benefits of workflow languages and the impact of the shortcomings that these languages have.

Further, in chapter 5 some implementations of aspect-oriented programming within workflow languages are introduced and explained. The complete design and implementation of stateful aspects in Padus is presented in chapter 6. Finally, chapter 7 evaluates this implementation and concludes this dissertation.

Chapter 2: Webservices and Service-Oriented Architectures

What we need to do is learn to work in the system, by which I mean that everybody, every team, every platform, every division, every component is there not for individual competitive profit or recognition, but for contribution to the system as a whole on a win-win basis.

W. Edwards Deming (1900 - 1993)

This chapter will give a short introduction about the webservice technology. With webservices in mind, the idea of *service-oriented architectures* (SOAs) and how the business world can take advantage of it can be easily explained.

1 Webservices Technology

Since the early nineties, the World Wide Web has been expanded enormously. At the beginning, this worldwide network was especially used as a source of information: documents and multi-media were accessible for everyone connected to this network. Later on, the business world also saw the big potential of the World Wide Web. 'E-commerce' and 'e-business' became major concepts and many companies started to sell their products via the Internet.
Even after the big expansion of the World Wide Web at the end of the 20th century, new technologies are still invented. A technology which has recently gained much success are *webservices*. With this technology, companies are able to provide some form of service to their customers through the Internet. Examples of such services are 'viewing weather forecasts', 'consulting the catalogue of a bookstore' and 'consulting the exchange-rate of a particular currency'.

Webservices are a technology which enables software, or rather some application logic, to be accessible and invocable through the Internet by sending messages. The application logic implemented as a webservice is often called a *service* and is platform-independent. This means that clients can use services no matter what platform they use and there are also no restrictions whatsoever on the platform to be used when implementing the services. Figure 1 explains this in a graphical way.

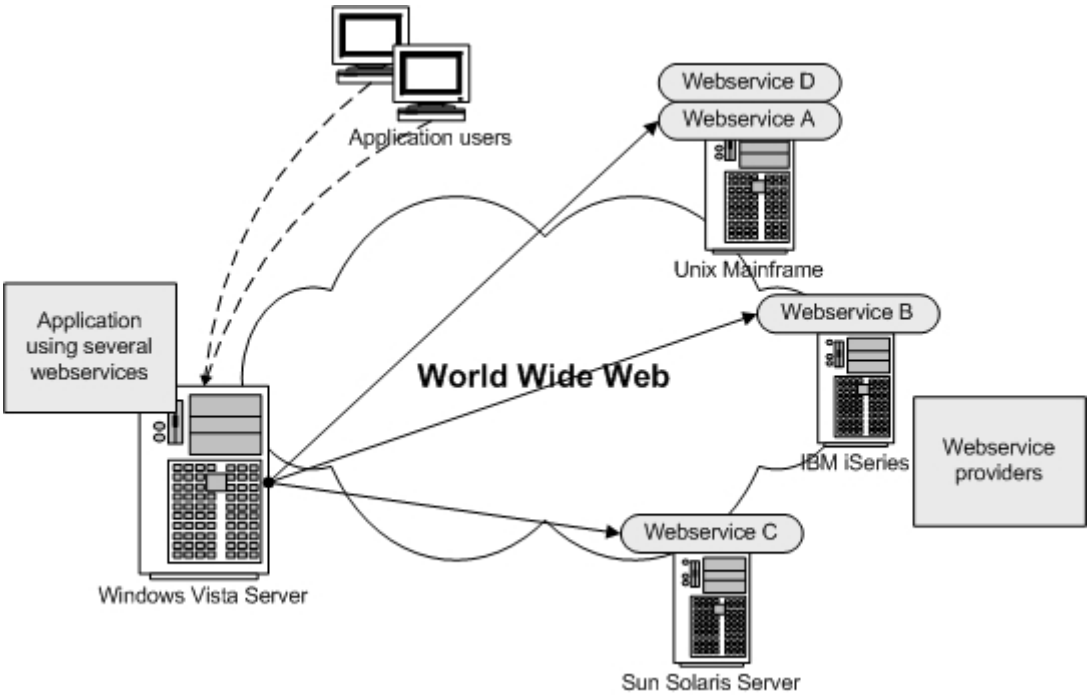

Figure 1 – webservices are fully platform independent

This platform independency is possible because a wealth of standards and protocols, together with a message-passing mechanism is at the heart of the webservice technology. The most important are XML (eXtensible Markup Language) and SOAP (Simple Object Access Protocol).

The WWW Consortium (W3C) defines a webservice as follows [8]:

> *... a software system, identified by a URI, designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machineprocessable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialisation in conjunction with other Web-related standards.*

Every service contains a description about its available operations, the messages it understands and some binding information. This description is written in an XML-based language called WSDL (Webservice Definition Language), and finally published on a UDDI (Universal Description, Discovery and Integration) repository. Clients of the service can browse this repository in order to get the WSDL description. With this description, the client knows how to invoke the operations of the service.

This interaction between the provider, publisher and client is often described as *the webservice model* (figure 2).

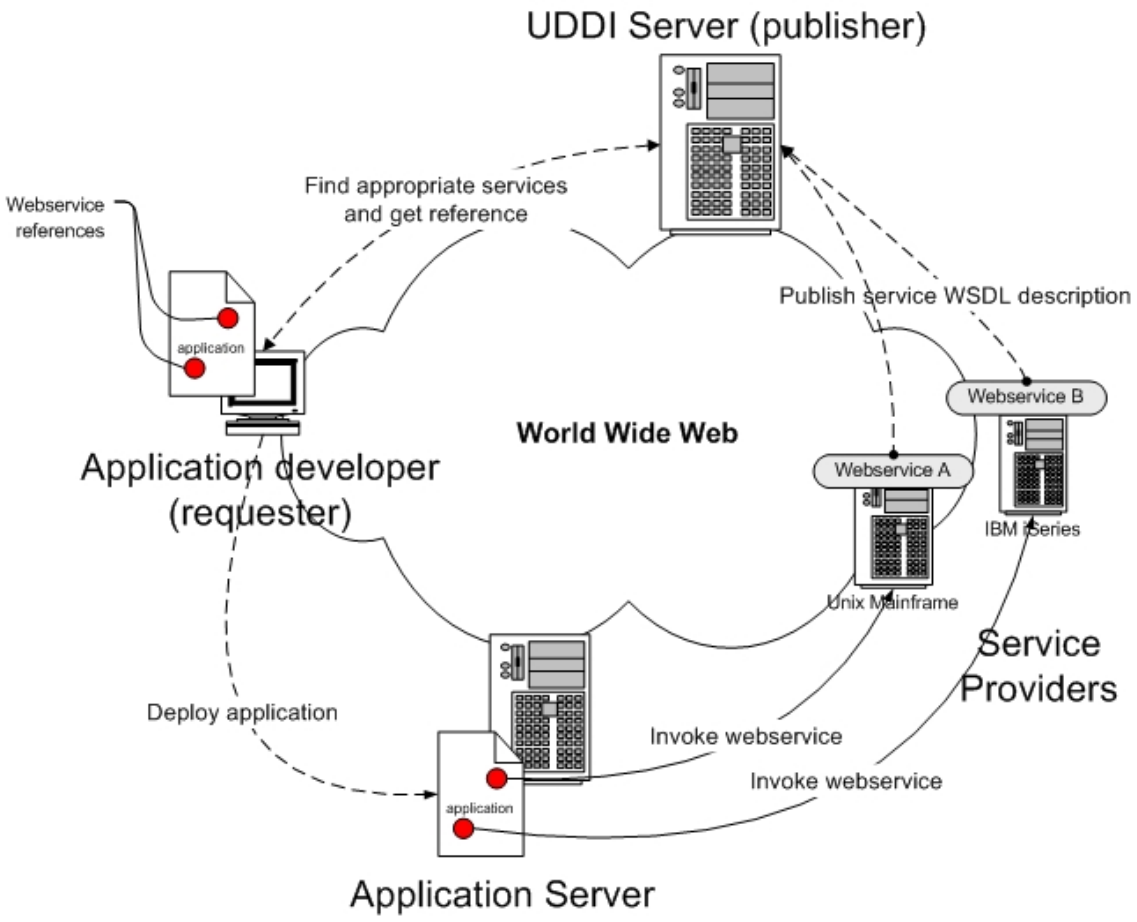

Figure 2 – the webservice model

1.1 Webservice Definition Language

Before a service can be used, the client must have some description of this service in order to know how to invoke the operations exposed by this service. The *Webservice Definition Language (WSDL)* [13] is an XML-based language invented by the World Wide Web Consortium (W3C) to describe a particular service. Such a description can be conceptually divided into four parts:

- *Interfaces* which are sets of operations available in the service.
- The *messages* and their datatype used for communication with the service and specifying the type of the exchanged information.
- *Binding information* to describe which transport protocol has to be used for information exchange.
- *Service endpoints* which is a specific address where the service is located.

The operations and messages are defined in an abstract way. The concrete details about how the messages are sent over the wire are defined in a binding.

A service can support multiple bindings for an interface, but each binding should be accessible from only one network address, identified by a URI. This URI is also referred as a service endpoint. Figure 3 below illustrates this visually:

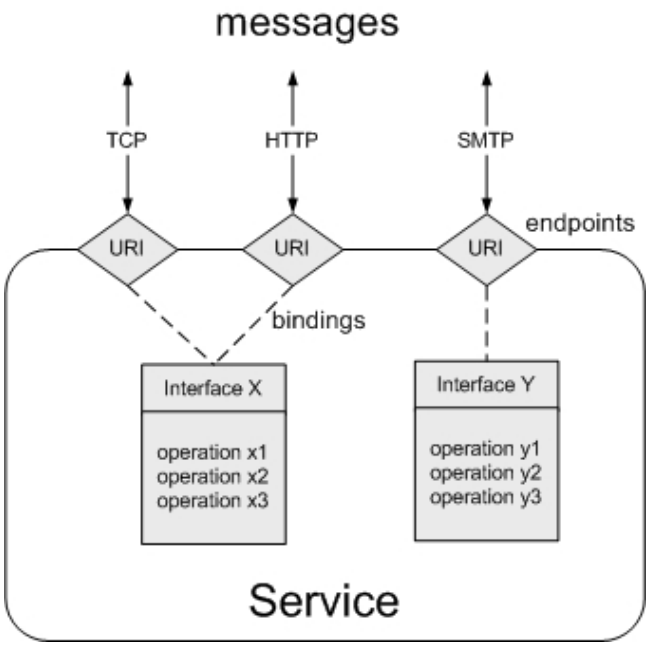

Figure 3: conceptual view of WSDL

When the client has this service description, he knows everything to locate the service and to invoke the operations. WSDL documents may be requested by browsing a UDDI repository.

The following paragraphs will dig deeper into the WSDL specification in order to know how such a service description really looks like. The description of a realistic billing service is used as an example. The whole WSDL document can be found in appendix A, but here, each part is explained individually.

1.1.1 Messages and Data-types

```
<types>
 <xsd:schema elementFormDefault="qualified"
   targetNamespace=http://systinet.com/wsdl/service/billing/
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="calculatePrice">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="p0" type="xsd:float"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
  <xsd:element name="calculatePriceResponse">
   <xsd:complexType>
    <xsd:sequence/>
   </xsd:complexType>
  </xsd:element>
 </xsd:schema>
</types>

<message name='BillingService_calculatePrice_Request_Soap'>
 <part name='parameters' element='calculatePrice'/>
</message>
<message name='BillingService_calculatePrice_Response_Soap'>
 <part name='parameters' element='calculatePriceResponse'/>
</message>
```

Codefragment 1 – WSDL types and messages

The `message` element defines the messages which serves as the input or the output of the operations. Messages consist of one or more parts with each of them having an associated type or element, depending on the particular style of the service. Message parts are interesting because they can classify the information of the message.

The types or elements of the parts of a message refer to an XML Schema type definition. It is possible to embed the type definition in the WSDL document itself. This is done using the `type` element. The type definitions define the structural details of the parts of a message.

1.1.2 Message Interface

```
<portType name='BillingService'>
 <operation name='calculatePrice'>
  <input message='BillingService_calculatePrice_Request_Soap'/>
  <output message='BillingService_calculatePrice_Response_Soap'/>
 </operation>
</portType>
```

Codefragment 2 – WSDL interfaces

A `PortType` element defines an interface of the service by grouping one or more operations and is identified with a unique name. Within each operation, a combination of either an input- or output-element or both is needed. This combination defines the *message exchange pattern*. This means that an operation can be:

- a one-way operation where the service receives a message; only one input message is defined.
- a receive-response operation where the service receives a message and responds with a correlated message; an input/output combination is defined.
- a solicit-response operation where the service sends a message and receives a response; an output/input combination is defined.
- a notification operation where the service only sends a message; only an output message is defined.

These first three elements (types, message, and portType) are abstract definitions of the service which makes up the programmatic interface.

1.1.3 Bindings

A `binding` element describes the concrete details of using a port type with a specific protocol. There can be more bindings for a particular port type as long as each uses a different protocol.
For each operation of a port type, extensibility elements are used to describe the binding details. For example, there are specific extensibility elements to describe SOAP bindings. The example in codefragment 3 illustrates a SOAP/HTTP binding for the *BillingService* port type. This means that the input and output data are encoded using SOAP and transmitted with HTTP. There exist another binding which states that the data is encoded using SOAP 1.2 and also transmitted using HTTP.

```
<binding name='BillingService' type='tns:BillingService'>
  <soap:binding transport='http://schemas.xmlsoap.org/soap/http'
                style='document'/>
    <operation name='calculatePrice'>
     <map:java-operation name='calculatePrice' signature='KEYpVg=='
                         unwrapped='true' wsiAttachments='true'
                         parameterOrder='p0' />

     <soap:operation soapAction=
        'http://URL/BillingService#calculatePrice?KEYpVg=='
        style='document'/>
     <input>
       <soap:body parts='parameters' use='literal'/>
     </input>
     <output>
       <soap:body parts='parameters' use='literal'/>
     </output>
   </operation>
</binding>

<binding name='BillingService_SOAP12' type='tns:BillingService'>
 <soap12:binding transport='http://schemas.xmlsoap.org/soap/http'
                 style='document'/>
  <operation name='calculatePrice'>
   <map:java-operation name='calculatePrice' signature='KEYpVg=='
                       unwrapped='true' wsiAttachments='true'
                       parameterOrder='p0' />

  <soap12:operation soapAction=
        'http://URL/BillingService#calculatePrice?KEYpVg=='
        style='document'/>
   <input>
    <soap12:body parts='parameters' use='literal'/>
   </input>
   <output>
    <soap12:body parts='parameters' use='literal'/>
   </output>
  </operation>
</binding>
```

Codefragment 3 – two WSDL bindings for the same operation

For each operation in the binding element, the `use` attribute defines the particular style for encoding the messages. There are two styles: *document/literal* or *rpc/encoded*. SOAP and the document styles are described in paragraph 1.3.

1.1.4 Service Endpoints

The `service` element defines a collection of ports (or endpoints) which expose a particular binding to the outside world. The specific address for the binding is specified again with extensibility elements. The example below (codefragment 4) shows a *BillingService* service that exposes two bindings at 'http://xxx.xxx.xxx.xxx/billing'. One is bound with the SOAP protocol and another with the SOAP 1.2 protocol.

```
<service name='BillingService'>
  <port name='BillingService'
       binding='tns:BillingService'>
    <soap:address location='http://xxx.xxx.xxx.xxx/billing'/>
  </port>
  <port name='BillingService_SOAP12'
        binding='tns:BillingService_SOAP12'>
    <soap12:address location='http://xxx.xxx.xxx.xxx/billing'/>
  </port>
</service>
```

Codefragment 4 – WSDL service endpoints

1.2 Publishing and Discovering Services with UDDI

Many companies or other organisations develop and provide webservices which are either used internally or exposed to the outside world. Because the WSDL document is the only essential document for the client in order to use a service, companies publish this WSDL document on a UDDI (Universal Description, Discovery and Integration) repository. For clients to use a service, they can browse this repository and receive the WSDL document of the particular service they are interested in.

UDDI is an important participant in the webservice model (figure 2). It provides the service discovery part which is the primary action prior to using a service. Many big companies might also have their own (private) UDDI repository.

1.3 Communication between Services

At the beginning of this chapter, it is mentioned that a message-passing mechanism is used in order to establish communication between services. Regardless of the origin of the services, communication should still be possible. Therefore, it is important that there is some form of standardization on the messages sent to the services so that they all use the same format and transport protocol.

The WWW Consortium designed a specification which is universally accepted as the standard protocol for communication within the webservices technology. This specification is called SOAP which stands for *Simple Object Access Protocol*.

1.3.1 Simple Object Access Protocol (SOAP)

SOAP is formally defined by the W3C as “a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment” [37]. From the definition, we understand that the main purpose of SOAP is the specification of some structured message format. The data format of the messages is defined using XML and consist of different parts.

SOAP is frequently used together with HTTP as the main transport mechanism because it works well with most of the Internet infrastructures such as firewalls. But also other transport protocols can be used such as SMTP.

1.3.2 Parts of a SOAP message:

The main part in the SOAP specification is the definition of the structure of the messages which are sent to, or received from other services. This structure is quite simple, but it has the ability to be extended and customised.

The two parts of a SOAP message are the *header* and the *body*, which are both contained into an *envelope*. In fact, an envelope is the SOAP term for a message (figure 5).

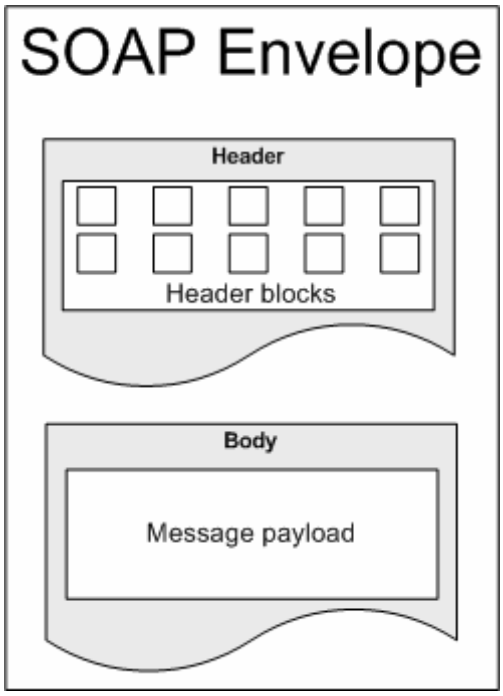

Figure 5 – the SOAP message

The first part, the header, is meant to store meta-information. Although the header is optional, it is rarely omitted in service-oriented applications. The interesting part in the header is the use of *header-blocks*. In loosely-coupled environments, such as service-oriented architecture's, it is extremely important that the SOAP messages are as independent as possible. This means that the messages should be 'intelligence heavy' and 'self-sufficient'. This can be achieved through the use of these header-blocks because they can outfit a message with all the necessary meta-information required for the services they come into contact with. Examples of meta-information are: routing information, context and transaction management information, or correlation information to associate a request message with a response message. The header might change during transmission of the message.

The body of an envelope contains the actual data. As opposed to the header information, the body should not be changed in any way during transmission of the SOAP message. However, if it is allowed, the data in the body might be used. For example, some correlation identifier stored in a header-block can be generated based on an invoice number stored in the message body.

1.3.3 Message Styles

SOAP was originally intended to replace RPC protocols, which specify how procedures can be invoked between distributed components. This is noticeable in the SOAP specification because of the support of two different styles to which the messages can be structured. These are the *RPC/encoded* style and the *document/literal style*. The document/literal style is preferred over the RPC/encoded because it is cleaner and easier to use within certain tools, but also because it enables larger payloads and thus reduces the amount of messages to be transmitted between services. The `style` attribute of the `binding` element in the WSDL document of the service specifies which style is used.

2 Webservices in a Business Perspective and Service-Oriented Architectures

An important characteristic of services is that they are modular and can be easily integrated into other applications. When services are integrated into other applications, they can be considered as *independent* software components which implements a part of the application logic of the software system. For example, an online banking application may use a service to provide the latest stock exchange rates to the customers.

The big advantage of using services as a software component, is their reusability because it reduces the overall development cost of applications substantially. A 3rd party software company may have already published a service which can be used within another software system. Today, the webservice technology is well integrated into business applications of large companies. Amazon.com for example, provides a webservice to their clients for browsing their catalogue. On the other hand, the software system of Amazon.com might use a webservice provided by the shipping company to inform the customer with the shipment cost.
It is clear that the webservice technology offers many new opportunities to companies to do their e-business. It is common practice that parts of a business processes are implemented by services provided by the company's business partners.

This kind of software architecture, where the components of the business software are in fact interacting services, is called a *service-oriented architecture (SOA)*. Formally, we can define a SOA as follows [4]:

> *A service-oriented architecture is a framework for integrating business processes and supporting IT infrastructure as secure, standardised components (services) that can be reused and combined to address changing business priorities.*

Many services within the SOA work together and communicate with each other in order to meet the functional requirements of the software system. Because a service is a stand-alone implementation of some application logic, SOAs makes it possible to develop very flexible software with minimal dependencies between the several software components. These components, implemented as services, are loosely-coupled because they have a well defined interface for communication, and they use XML as a standard for data-exchange. Webservices are certainly a means for implementing SOAs [5].

At the moment, business applications might be developed as a service-oriented architecture. This is mainly because these applications are distributed between different business-units within the company and rely on several business partners to implement the business processes. SOAs enable a better integration of the different parts of the software system because of the platform-independency of the different software components, i.e. the services. They can be deployed worldwide on different locations. Communication between those components is still possible because of the inherent characteristics of the webservice technology. However, SOAs do not simplify distributed computing and make the implementation of business software easy. Although, by thoroughly analyzing and designing the problem and by keeping the service-oriented design principles in mind, SOAs have great potential.

Another big reason which encourages the use of SOAs within businesses, is that business processes should be adjustable to the ever-changing markets. Because of the fact that the software components are loosely-coupled, this can be realised fairly easy. As a matter of fact, the adaptability and maintainability of the software increases and has a positive effect on the overall costs. IBM's Websphere software is an example of a project which gave businesses a real boost to adopt SOAs.

Some important characteristics of service-oriented architectures are:

- Open standards: a significant characteristic of SOA is that it is based on many open standards. The messages and data-exchange between services are fully standardised through the use of XML and SOAP. Furthermore, services only need the description of other services in order to communicate with them. This extensive use of open standards creates the loose coupling between services.
- Vendor diversity: building upon the previous characteristic of using open standards, corporations can choose the best environment for their specific applications. As long as the environment supports the creation of standard webservices, their applications can interoperate with each other. Here, platform independency exists because of the use of open standards.
- Architectural composability, extensibility and reusability: a very important aspect within SOA is composability. Parts of a business process can be broken down into several services which exist as independent units of logic. There is a loose-coupling between the services and this means that business processes can be highly flexible and adaptive.
- Organizational agility: because of the architectural composability characteristic of SOAs, organisations can quickly adapt themselves and respond to the ever-changing market. This is possible because services are loosely-coupled and can evolve independently from other services. Business logic can be changed without hampering other parts of the business process. There is no doubt that this characteristic gives a very significant benefit to businesses.

Applications can be assembled by combining several services due to their modularity. Using quality services and the ease of reusing them is a real benefit for companies because it greatly improves the return-on-investment of the application. The composition of services, or *service orchestration*, however, is still a difficult concern. Fortunately, workflow languages provide a real help for this issue. The next chapter explains service orchestration more in detail.

Chapter 3: Workflow Languages

"The right music played by the right instruments at the right time in the right combination: that's good orchestration,"
Leonard Bernstein (1918 – 1990)

Today, workflow technology is flourishing in its traditional application areas of business process modeling and business process coordination. Part of this technology are workflow languages. These are often used to implement processes, designed as workflows into the workflow management system of a company. WS-BPEL is a concrete example of such a workflow language and to give a better understanding about these kinds of languages, it is explained in this chapter. To conclude, the position of this workflow language against others and the main disadvantages of workflow languages are discussed.

1 Business Processes Modeled as Workflows

Business processes can be viewed as high-level descriptions of the activities and processes executed within a company. Because such business processes can be quite complex, workflows are used to model them at a conceptual level. This enables a better understanding, evaluation and redesigning of the business process because it visualizes the dataflow as a graph. Workflows are in fact a collection of well-coordinated tasks designed to carry out a well-defined complex process.

It is very important that business processes are implemented as efficiently as possible to fulfill the needs of the customers in the best way possible. The whole engineering of workflows is therefore supported by a *workflow management system* [52]. This is an information system that manages the modelling, implementation, execution and monitoring of business processes or workflows. The implementation and execution part of workflow management systems involves software which executes the workflow. This software features some high-level language, generally at a higher level than general-purpose languages (i.e. Java or C++), to describe the specification of the process structure and logic, exception handling, task duration, etc…

Previously, it is mentioned that businesses collaborate frequently with their partners to implement their business processes. Webservices and SOAs provide a solution for transparent communication with the partner's business systems in a loosely-coupled and flexible way.
Parts of a workflow might be implemented as stand-alone units of work and therefore they can be decomposed and modularised as services (see figure 6). This results in workflows which frequently refer to other services. Once the core functionalities of a business are modeled as services, then it is almost just a matter of composing and coordinating those services in the right way to implement the business processes. Service composition can easily be modeled as a workflow where the interaction with other services becomes clear. The reason why workflow languages are an important and interesting building block for implementing service-oriented architectures, is that they provide special language constructs to combine and manage the invocation of other services in a fairly straightforward way. These services should be executed in a specific sequence which must be coordinated with respect to synchronisation, concurrency, exception-handling and parallelism.

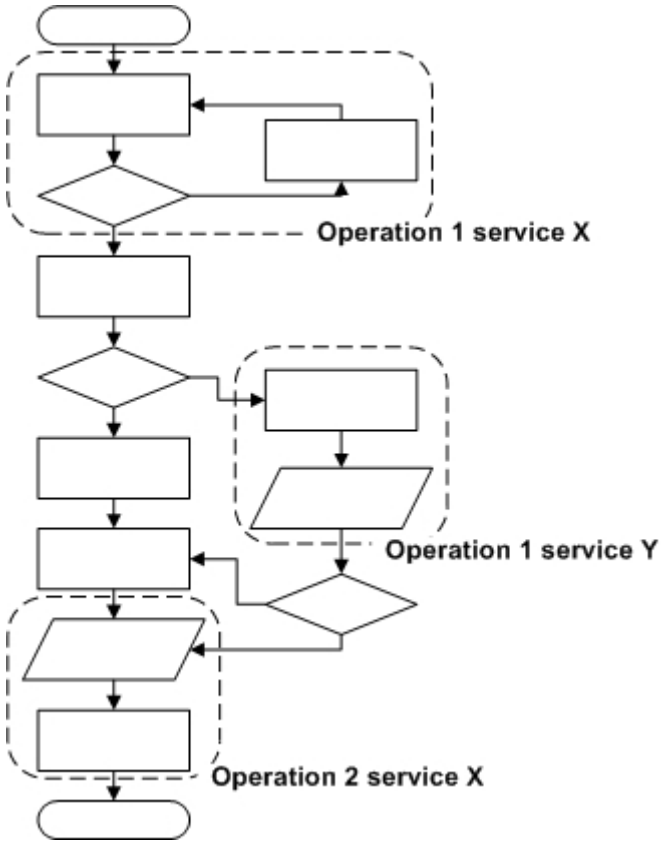

Figure 6 – a workflow as a collaboration of services

General-purpose languages can also be used for the implementation of business processes as a SOA, but then the developer himself must take the above mentioned issues of synchronization, etc… into account, which can be a real burden on the developer's work. The real benefit of using workflow languages is that they provide the necessary abstractions particularly suited for the implementation of processes, modelled as a composition of services in a workflow. They also handle the low-level aspects of service invocations and concurrency. This makes the implementation more efficient because it relieves the programmers of all this and they do not have to spend much more time on these issues anymore.

2 Service Orchestration and Choreography within Workflows

As already introduced at the end of the previous chapter, the combination of several services might be difficult and must be handled well. There are two fundamental different forms of combining services: *orchestration* and *choreography*.

Orchestration means that the combination of the services is controlled by a central process. This process, which is also a service, coordinates the execution of the services so that they play their particular role well in the larger, more complex contribution of services. All the involved services are fully unaware that they are taking part into a higher-level business process. Therefore, in a particular business process, direct communication between the individual services is not permitted and information exchange between them is not possible. Because only the central process is aware of participating in a higher-level business process, all communication goes through this

process. This central process is often called *the coordinator of the orchestration*. See the figure 7 below.

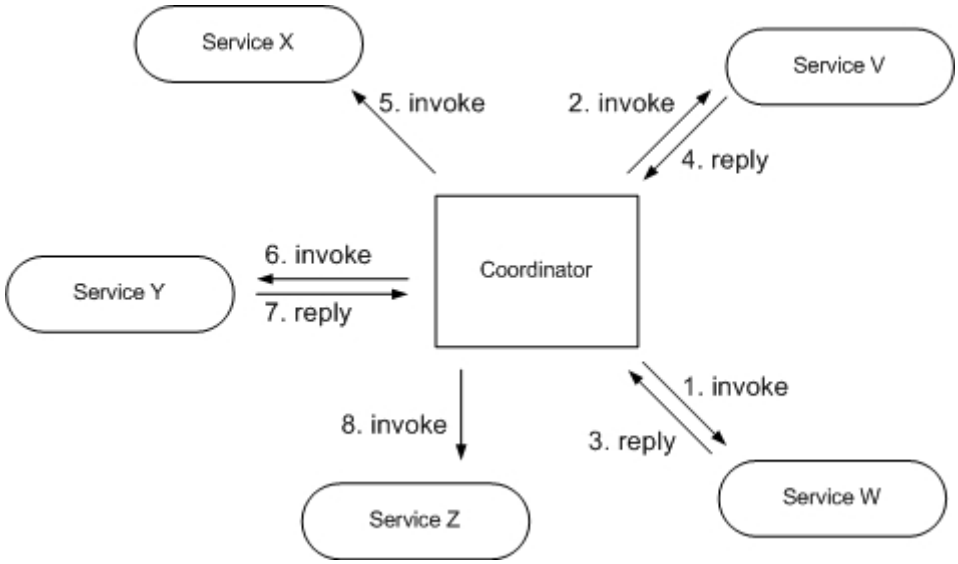

Figure 7 – orchestration between services

Because the central process is a service itself, it can contribute also to an even larger business process. We can think about orchestration as some kind of recursive composition of services.

Choreography on the other hand, can be viewed as an exchange of messages between services. Every service can be seen as a peer in the choreography and is aware that it takes part into this collaboration. Each service knows with which peer he has to interact; this implies that the services have knowledge about the messages to exchange and the operations to execute. Figure 8 illustrates this.

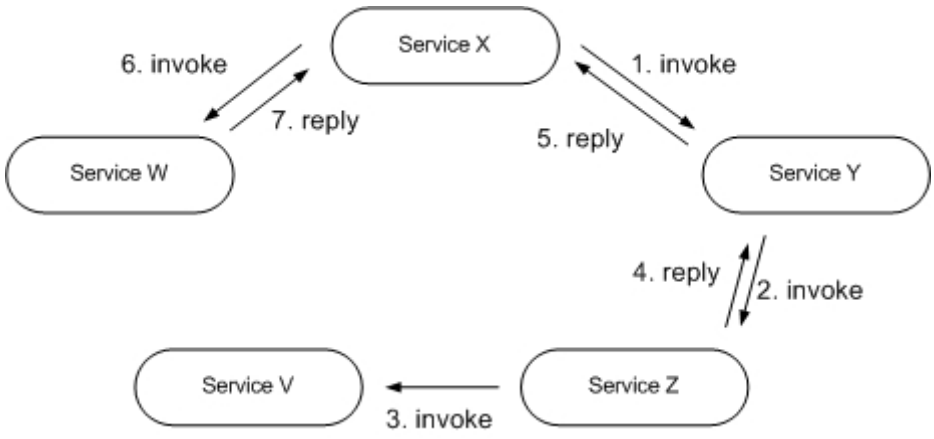

Figure 8 – choreography between services

In contrast with orchestration, choreography describes some common observable interaction between services. Languages used for orchestrating services are in itself executable, whereas choreography can be used to create some description of a behavioural contract between peers.

3 WS-BPEL

A concrete workflow language, namely *WS-BPEL* [2] (also identified as BPEL) which is an acronym for *Business Process Execution Language for Webservices*, is introduced in this section. This is by far the most popular language for implementing business process workflows. With BPEL, the adoption of service-oriented architectures in the business world has accelerated a lot because organisations are now capable of automating their business processes by composing several services with each other, and expose their process again as a service in a more straightforward manner. In fact, one of the most significant technologies which elevated the idea of SOAs to the business level is BPEL. Since the development of the original BPEL specification in 2002, several modifications and improvements have been done. Finally, the specification was submitted to OASIS (Organisation for the Advancement of Structured Information Standards) and a technical committee for BPEL has been founded which will stimulate its presence and acceptance in the industry even more. With all the support of different organisations, BPEL has good prospects for the future.

BPEL is an XML-based language which is based on WSFL (Webservices Flow Language) [49], a graph-based process language which was developed by IBM and XLANG (Webservices for Business Process Design) [59], a block structured process language which originates from Microsoft. Both the features of these two languages are combined into BPEL.

Two types of processes can be modelled in BPEL. The first type is an executable process which specifies the order of execution of the particular activities in the workflow. These processes are used for implementing the logic of a business process and follows the idea of orchestration.
The second type of process that can be modelled are abstract processes. They can be seen as a business protocol describing the interaction between the different parties without revealing their internal behaviour. In other words, a BPEL abstract process imposes sequencing rules regarding the invoked operations in the BPEL executable process. A violation of these rules may cause the BPEL orchestration engine to throw an exception. This type of process follows the choreography paradigm.

The next paragraph will introduce BPEL as an orchestration language. It will show how business processes can be built by orchestrating other services. A new, composite and executable service is obtained after deployment.

3.1 Orchestrating Processes in BPEL

Building a BPEL process involves using and combining many other services. This means that the WSDL descriptions of these services are very important. BPEL relies heavily on these descriptions in order to be able to communicate with the services. On the other hand, BPEL processes are also deployed as a service. Therefore, each BPEL process has its own WSDL description. The complete source-code of a realistic BPEL process together with the WSDL description can be found in appendix B.

A BPEL process can be divided into three main parts: *partnerlinks*, *variables* and *process logic*. These parts are described in the following sections.

3.1.1 Partnerlinks

Every BPEL process has to interact with other services in either a synchronous or an asynchronous way. A synchronous invocation means that the process invokes a service and expect an answer immediately. Asynchronous interactions are interesting if a call to some service might take a longer time. The BPEL process initiates the invocation of the service and continues the execution of the rest of the process. At a particular time, the BPEL process will be notified when the service has responded.

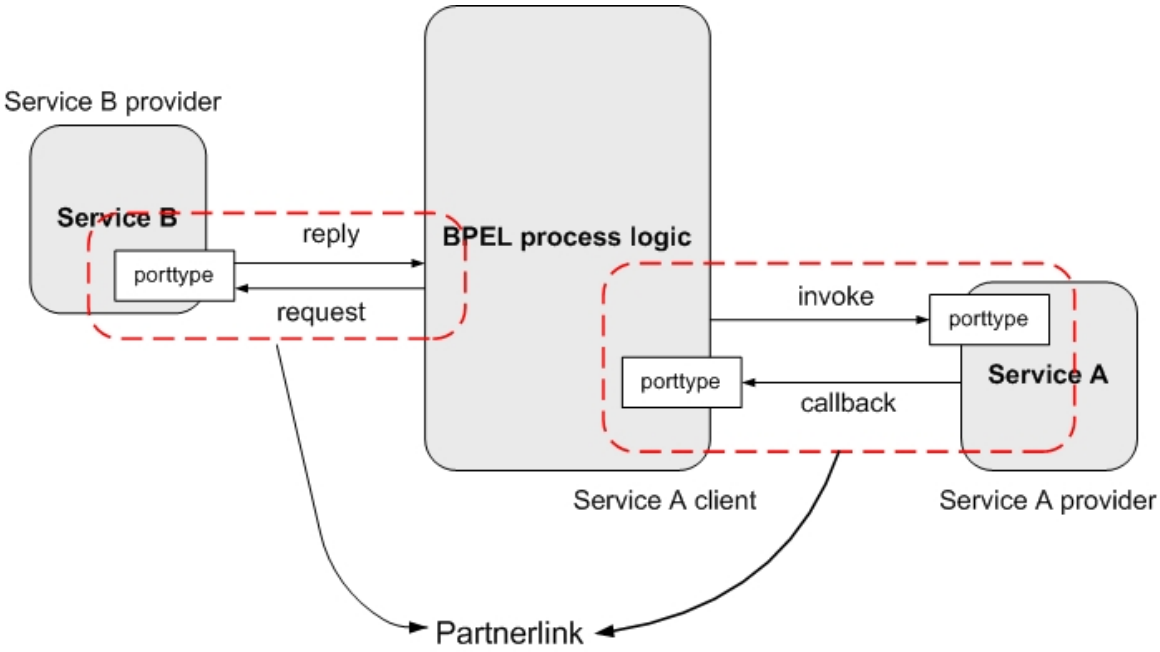

Figure 9 – partnerlinks: synchronous communication with service B and asynchronous communication with service A

Every interaction with a service is modelled as a *partnerlink*. Partnerlinks can be seen as some logical connection between the BPEL process and another service. The roles both parties play in the interaction are also specified in the partnerlink. For synchronous interactions, only the role of the service (`partner-role`) needs to be specified, while in asynchronous interactions both the role of the BPEL process (`my-role`) and the role of the service must be specified.

With each partnerlink, a *partnerlink-type* is defined to represent the interaction more concretely. A partnerlink-type will bind the roles of the partnerlink with the port type defined in the WSDL description of the service (figure 9). The following codefragments show how a partnerlink (codefragment 5) and partnerlink-type (codefragment 6) definition looks like.

```
<partnerLinks>
  <partnerLink name="AsynchPL"
               partnerLinkType="AsynchPLT"
               myRole="ServiceAClient"
               partnerRole="ServiceAProvider"/>

  <partnerLink name="SynchPL"
               partnerLinkType="SynchPLT"
               partnerRole="ServiceBProvider"/>
</partnerlinks>
```

Codefragment 5 – partnerlink definition in BPEL process

```
<partnerLinkType name="AsynchPLT">
  <role name="ServiceAClient">
    <plnk:portType name="tns:BPELProcessCallbackPT" />
  <role>

  <role name="ServiceAProvider">
    <portType name="tns:ServiceAPT" />
  </role>
</partnerLinkType>
```

Codefragment 6 – partnerlinktype defined in WSDL document of the service

3.1.2 Variables

Like in most programming languages, variables are used to store and load data and maintains the state of the application. This is not different in the BPEL workflow language. The data in BPEL processes usually originates from either the input of the client of the BPEL process or output from some invoked service.

Services interact with each other by exchanging messages. In order to catch the data coming from services, every variable is connected with a message type defined in the WSDL description. The BPEL engine will automatically copy the data of the message into the variable.
It is also possible to declare common variables to store intermediate data while processing. For such variables, the XML Schema type of the data must be defined instead of the message type.

```
<variables>
  <variable name="someVariable"
            messageType="OperationRequestMessage" />
</variables>
```

Codefragment 7 – variable declaration in BPEL

3.1.3 Process Logic

The third part in a BPEL process contains the actual process logic which specifies the order of service invocations. The process logic is modelled with *activities*. These constructs represent the basic operations found in workflows and are the dominant abstraction mechanisms for this language. For each activity, certain properties can be set. Properties are, for example, the name of an operation to invoke or a boolean-evaluated condition.

Activities are classified into three categories: service interaction, data manipulation and control-flow (table 1).

Service interaction	Data manipulation	Control-flow
<invoke>	<assign>	<sequence>
<receive>		<switch>
<reply>		<while>
		<flow>
		<pick>

Table 1 – classification of activities in BPEL

3.1.3.1 Service Interaction

<invoke>, <receive> and <reply> are activities for interacting with services. Synchronous service invocation can be done by using only the <invoke> activity, while

asynchronous invocations are realised with both the <invoke> and <receive> activity. With <invoke>, a particular operation of some service will be called. For asynchronous invocations, the <receive> activity is used somewhere in the activity flow to wait for the response of the service. Every BPEL process begins with a <receive> activity which will wait until a request from the client is received. A response to the client is sent using the <reply> activity.
Only this category of activities uses the partnerlinks and WSDL descriptions because they interact with other services.

3.1.3.2 Data Manipulation

The <assign> activity is the only one for manipulating data in variables. Clearly, this activity is extensively used because manipulating data is a common task in business processes.

An <assign> activity contains one or more copy-structures with a from-specification and a to-specification. The three most important datasources are a variable, a general expression such as a string, or literal XML data. The data is commonly copied to a variable.

The XPath language [19] is used within the assign activity to build queries and expressions for addressing the parts in a variable.

3.1.3.3 Control Flow

Control flow activities groups a collection of activities which are executed in the order defined by this control-flow activity.

The semantics of <sequence>, <switch> and <while> activities are equivalent with general-purpose programming languages. With the <flow> activity, other activities can be executed concurrently. The <pick> activity is interesting for making a choice of which activities must be executed, based on external events such as the arrival of a certain message.
Control-flow activities can be used recursively. A <flow> can be nested within a <while> for example. The <flow> activity contains for each concurrent flow a sequence of the activities which are executed in parallel. At the end of the <flow>, execution of the process is halted until the execution of all concurrent flows is correctly terminated. In this way, synchronisation between the parts of the <flow> activity is enforced. A <pick> activity can be seen as a collection of branches containing activities.

It is obvious that BPEL processes are cleanly structured because of the control flow activities. Another way of structuring activities, however, is by using *links*. A link contains a source and target activity. When the source activity is executed, the process continues by executing the target activity. By using links, the boundaries of the control-flow activities can be crossed and the process structure can be seen as a directed graph. Links should be used carefully because otherwise the source-code can evolve in spaghetti-code.

The BPEL specification contains many more advanced concepts such as *correlation*, *fault-handling*, *message-handling* and *scopes*. However, these are not mentioned here because they are less important in this dissertation. The BPEL specification gives more information on all these concepts.

Both properties of XLANG and WSFL are noticeable in this language. The control-flow structures originate from XLANG while the idea of linking activities to form a directed graph comes from WSFL.

3.2 BPEL versus other Workflow Languages

3.2.1 Comparison

Workflow language analysis and evaluation is often done using a framework of patterns. Patterns provide a standard solution to some recurring situations which can occur during software development. Equivalent with design patterns [36] in object-oriented software development, there exist also patterns for service-oriented software development: *workflow patterns* [1].

Currently, there exists a whole set of languages dedicated to workflow implementation and service composition. The evaluation of all these languages is based on these workflow patterns. For each language, it is checked if a certain workflow pattern is supported or not. Wohed et al. have analysed and evaluated BPEL using 20 workflow patterns and compared them against other workflow languages [63]. They conclude that BPEL is the better in terms of pattern support, although it doesn't support the more advanced ones. However, the complexity of the language and the unclear semantics are two negative remarks of BPEL.

3.2.2 Important Shortcomings in BPEL

If a certain workflow language supports all the possible workflow patterns, does not imply at all that this language is perfect. There are still some significant issues in workflow languages which are also important concerning business process implementation.

One issue that almost none of the current existing BPEL implementations support, is *dynamic process modification*. Most business processes are long running processes by nature and may run for hours or even days before they are completed. Imagine that a certain service this process needs becomes unavailable. The only way to modify the process is by stopping, re-designing, re-deploying and re-running it. Such a restart is not easily realized for such long running processes because already generated information can be lost. In current BPEL environments, business processes cannot be modified at run-time. Flexible workflow languages are needed in order to meet this desirable requirement. Currently, this problem of dynamic process modification is already addressed in many research projects. [39][31][10] [43]

Another issue which is important concerning software-maintenance and –evolution is the lack of support for so called *crosscutting concerns*. Crosscutting concerns are functionalities which are scattered and tangled throughout the whole application and cannot be cleanly modularised. Although aspect-oriented programming is a technology to solve this issue, it is not implemented within current workflow languages. Because business rules frequently crosscut the business process, this technology should be better supported in workflow languages.

Chapter 4: Aspect-Oriented Programming

Programs must be written for people to read, and only incidentally for machines to execute.
H. Abelson and G. Sussman

Controlling complexity is the essence of computer programming.
B. Kernighan

Because the main research in this dissertation is situated in the domain of aspect-oriented programming (AOP) [47], a whole chapter is devoted to this relatively new programming paradigm.

After a short introduction about the reason AOP is invented, we will explain the basic concepts of AOP. Furthermore, two example languages are shown together with some code examples to give the reader a practical view of AOP.

1 Introduction

In the history of computer science, many different programming paradigms are invented. *Imperative programming, functional programming*, *logic programming*, *object-oriented programming* are a few examples, but there exist many more.
When writing software, software engineers typically choose a language within the programming paradigm which suits their problem best. For example, if someone has to write a rule-based expert system, he better chooses some logic programming language rather than an imperative programming language. The reason that some paradigm is better than another with respect to the same problem, is because it offers different kinds of abstraction techniques. A logic programming language can make abstraction of rules and facts. Therefore, because expert systems are all about facts, rules and reasoning about them, this kind of language seems the best choice for the implementation of this kind of software system.
A very popular programming paradigm is o*bject-oriented programming*, where the basic form of abstraction is an object. The reason of its popularity is that an object resembles very well to real-world concepts. This characteristic really helps implementing software because the software engineer can easily determine the different parts of the software by imagining them as real-world concepts.

In order to handle the complexity of an application, the principle of *separation of concerns* [27][53] is essential. This principle states that every concern (or functionality) in the application should be implemented as a separate module and independent of other concerns. The big advantage is that a modification of a concern does not have an effect on other concerns. With this principle, the maintainability, reusability and evolvability of software increases extensively.

But there is a problem from which (almost) every programming language suffers. This problem is generally known as *The Tyranny of Dominant Decomposition* [58]. This means that using a particular programming paradigm, the different concerns of the software system can only be decomposed using the dominant abstraction mechanism – objects in an object-oriented language or rules and facts in some logic-oriented language.

It is clear that there is a direct paradox between the principle of separation of concerns and the problem of dominant decomposition. Because concerns can only be decomposed and modularized using the main abstraction technique offered by the programming paradigm, other concerns which cannot be encapsulated properly will end up scattered and tangled all over the code. These concerns are often called *crosscutting concerns* because parts of their implementation are everywhere in the application and thus crosscut other functionalities. This is a problem because this makes it impossible to reason about the concern in isolation and hence this is a violation of the principle of separation of concerns.

Recently, a new programming paradigm has emerged which tries to solve the above mentioned problem. This paradigm is known as *Aspect-Oriented Programming* [47]. One thing which should be made clear when talking about AOP is that this programming paradigm cannot be used on its own. It is rather complementary to another programming paradigm. Because object-oriented programming is one of the most advanced and certainly the most popular one to date, aspect-oriented programming goes well hand in hand with this paradigm. Therefore, we will assume in the rest of this chapter that AOP is used together with OOP. The main part of the application is written as an object-oriented program – often referred to as the base application. The crosscutting concerns are written as an aspect-oriented program.

2 Separating Crosscutting Concerns

Aspect-oriented programming is invented mainly to maintain the principle of separation of concerns. It is obvious that this can be realized by encapsulating the crosscutting concerns of the application. Some examples of such concerns are:

- Logging: where the execution of some methods must be logged.
- Security: the application must check permissions before the user can use some functionalities of the application.
- Transaction control: particular methods which talk to databases, must use transactions to keep the database consistent. The transaction mechanisms, *commit* and *rollback*, are scattered in the methods and consequently scattered throughout the whole application.

Figure 10 illustrates how a concern can crosscut a whole application. The vertical bars are the classes of the application and each small horizontal line represents a part of the implementation of this concern.

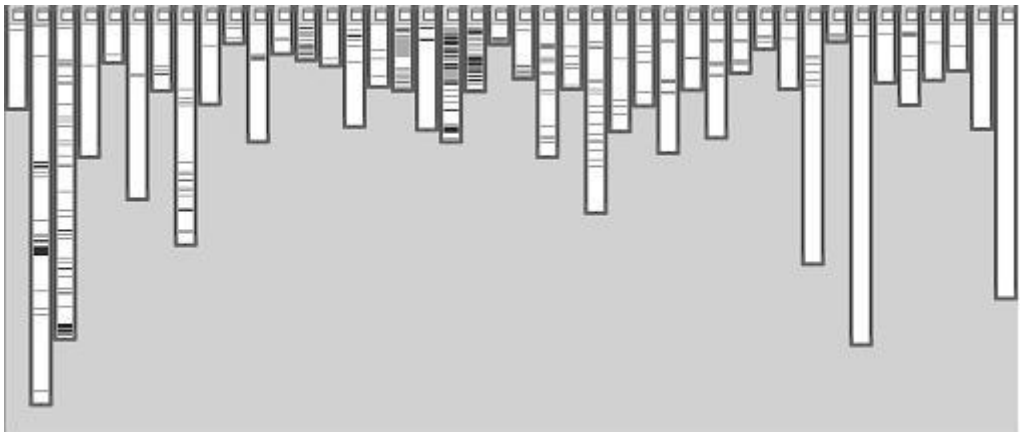

Figure 10 – a logging concern crosscutting an application

It is not hard to see that these functionalities cannot be modularized as an object. Considering the logging example, the programmer can certainly write some 'logging' object with a method which does the actual logging. Obviously, this method has to be called at the start and end of every other method to log its execution. As a consequence, the logging concern is crosscut throughout the application.
Although the logging functionality is a rather simple crosscutting concern, someone can claim that it is not a big effort to add a few lines of extra code. Consider that the application will be released soon and every piece of code that does some logging must be removed. Removing hundreds of lines of code throughout the whole application is a time-consuming task and because 'time is money', software companies wants to avoid this. When this logging concern could be implemented as a separate module, it is just a matter of removing this module from the application. This certainly does not require more effort than removing all these lines of code.

Aspect-oriented programming introduces a new, advanced modularization technique to encapsulate a crosscutting concern. The development of software systems using AOP results therefore in a better design. This has a direct impact on the quality of the software because the evolvability and maintainability will increase.

2.1 Encapsulating Crosscutting Concerns in an Aspect

The new abstraction technique which is introduced by AOP are *aspects*. Aspects are fully capable of encapsulating crosscutting concerns. All the code which was initially spread all over the application can now be modularised into one module.

The big difference between objects and aspects are situated in how their behaviour is invoked. Objects will explicitly invoke the behaviour of another object by calling its methods. On the other hand, the crosscutting behaviour of aspects will be invoked implicitly by the methods of objects. This results in the fact that objects are unaware of the crosscutting concern. The following figure 11 illustrates this.

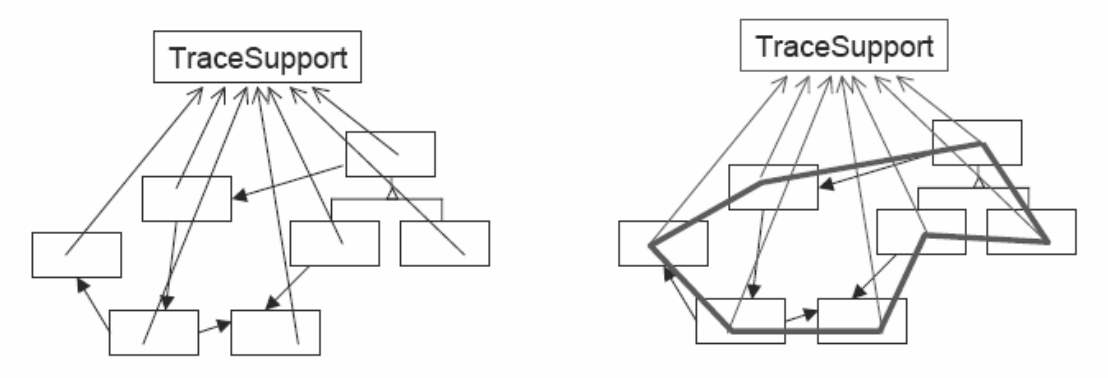

Figure 11 – difference between an object-oriented and aspect-oriented implementation of a concern.

The left-hand side of the figure above show how the tracing crosscutting concern is invoked by the different objects. The right-hand side illustrates how the aspect itself (the bold polygon) invokes its own tracing-behaviour at particular points in the objects.

The aspect is fully responsible for the invocation of its own behaviour. To be able to handle this responsibility, an aspect consists of two important parts. One part implements *what* the behaviour of the crosscutting concern is. The other part describes *where* in the base-program this behaviour must be invoked.

2.1.1 Pointcuts

An aspect will always be applied at some particular, well-defined points in the execution flow of the base-program. These points are called *joinpoints*. Examples of joinpoints in an object-oriented language are 'a method invocation' or 'throwing an exception'. Every aspect-language has its own joinpoint model which describes all the possible joinpoints that can be used in a *pointcut*. Pointcuts are constructs that describes a set of joinpoints where the behaviour of the aspect must be invoked. A particular pointcut language is used to write down the pointcut. This language can be a simple expression with predicates or even a Turing-complete language.

2.1.2 Advices

The second part of an aspect is the definition of its behaviour. This behaviour is defined in an *advice* construct and usually written in the same language as the base application extended with some specific language keywords. Advice constructs are always connected with a pointcut. When the pointcut is matched during the execution of the application, the behaviour defined in the advice construct will be executed.
There are generally three types of advice: *before*, *after* and *around* advice. Each of them specifies where the behaviour has to be inserted, relative to the pointcut. For example, a before advice will insert its behaviour just before the pointcut. Around advice, on the other hand, will replace the original behaviour with the behaviour defined in the advice construct. A special *proceed* keyword can be used in around advices to execute the original behaviour.

2.2 Weaving Aspects in the Application

It is possible to separate crosscutting concerns from other concerns by modularising their behaviour in an aspect. But after all, this behaviour should be put back at the right places in our base application. Therefore, a special kind of compiler is used: the *weaver*. A weaver will insert the aspect-behaviour at the right joinpoints defined in the

pointcuts. This process is generally known as weaving the aspect into the base application.

Currently, there exist already a whole bunch of different weavers each of them using a different weaving technology and tailored towards a specific programming paradigm. But generally, all these weavers can be separated into two groups: *static* and *dynamic* weavers.

2.2.1 Static Weaving

A static weaver will do a source-to-source code transformation. This transformation will modify the code by inserting the advice code at the right places by matching the joinpoints of the application with the pointcuts. In fact, the aspect behaviour is inlined into the source-code and this results in a new version with additional functionality. This kind of weaving gives no run-time performance drawbacks at all because the weaving is done at compile-time.

Although the performance of static weavers makes them interesting and useful, there exists also a significant disadvantage. Because the behaviour of the aspect is inlined into the source-code, it is very difficult to identify the aspects back later on. This makes the adaptation or replacement of aspects woven during run-time almost impossible. For example, it might be interesting to remove a logging aspect at run-time without halting and recompiling some critical application. The initial versions of AspectJ were released with a static weaver [48].

2.2.2 Dynamic Weaving

While static weavers do the weaving at compile-time, dynamic weavers on the other hand will weave the aspects at run-time. This gives the advantage that aspects can be woven and unwoven during the execution of the application. The main requirement for weaving and unweaving at run-time is that the aspects should be identifiable and kept separated from the base code.

When weaving an aspect, hooks are inserted at the joinpoints where the aspect behaviour should be invoked. When such a joinpoint is reached during program execution, the hook will intercept and change the normal program flow by executing the right piece of advice defined in the aspect. Using this technique, it is clear that the aspect is not inlined into the base code and can be easily identified back during run-time. Generally, custom class-loaders are used for instrumenting the base code with these hooks. Dynamic weavers provides much more flexibility for using aspects in an application. For example, a new version of an aspects can be plugged in without stopping, recompiling and restarting the application.

With dynamic weaving there is usually some performance penalty because the code is not pre-processed beforehand and thus not very optimized. Many checks must be done at run-time for determining if advice code should be invoked. But nevertheless, there exist some very performant dynamic weavers in which these performance issues are negligible. The JAsCo [57] language, for example, has a highly optimized run-time weaver which is able to compete with static weavers.

3 Characteristics of Aspect-Oriented Languages

Two important characteristics form the basis to define a language as aspect-oriented or not. These are *quantification* and *obliviousness* [35][34].

3.1 Quantification

With the explanation in the previous paragraph about the concepts of pointcuts and advice, it should be clear that programming statements can be made of the form:

In program P, whenever condition C is met, perform action A

The part …*whenever condition C is met*… tells that this is a quantified statement. This means that several non-local conditions C can be specified in the application. In AOP terms, a condition can be regarded as a certain joinpoint.

There exist two forms of quantification: *static* quantification and *dynamic* quantification. With static quantification, only static program structures such as variables, method-calls or loops can be quantified. Dynamic quantification on the other hand, makes it possible to match against conditions that happens during the execution of the application.

The quantification characteristic states that an aspect can affect any number of points in a program.

3.2 Obliviousness

According to the dictionary, the word *oblivious* means 'to be unaware of something'. When this meaning is put in an AOP context, *obliviousness* is the idea that the base application or parts of it should be unaware of the application of certain aspects. In other words, components of the application should not be aware of the fact that certain aspects crosscut them. Secondly, the programmer who writes the base application should not make any preparations or assumptions whatsoever to assure that future aspects should work properly with his code.

Obliviousness is the property which makes AOP interesting and special because it enhances separation of concerns in the application and it simplifies the analysis and design of the software components.

4 Implementations of AOP

At the moment, there exist many implementations of aspect-oriented programming. Some provide native language support by extending an existing language. AspectJ [48] and Carma [38] are examples of such language extensions of respectively Java and Smalltalk. An extended compiler is needed to do the weaving. On the other hand, AOP might also be provided through a framework. AOP concepts such as pointcuts and advices are then defined as classes. The aspects might be bound to the base application using XML, while the weaving process might be done through the use of custom classloaders. A well-known framework which provides AOP support is Spring [46].

In order to give a general idea of AOP implementations, two languages are described which are based on the Java programming language. The first one, AspectJ, is a mature language commonly regarded as *the* AOP language. The other one is JAsCo [57], a language which is more intended for research in AOP and provides some advanced features.

4.1 AspectJ

AspectJ is a simple and practical extension to the Java programming language. With only a dozen of new language constructs, crosscutting concerns can be cleanly modularised as an aspect. The AspectJ Quick Language Reference contains a list of all the possibilities of AspectJ.

4.1.1 Joinpoint Model and Pointcut Language

An important element in the design of aspect-oriented languages is its joinpoint model. AspectJ features a dynamic joinpoint model which means that the joinpoints are well-defined points in the execution of a program, and not just in the static structure of the program. For example, it is possible to capture certain joinpoints only if they are in the control flow of another joinpoint. Another possibility with dynamic joinpoints is that a certain run-time condition can be specified with the pointcut. The pointcut will only match if the condition is satisfied.

The list of pointcuts supported by AspectJ are:

Joinpoint	**Pointcut designator**
Field references	get(fieldpattern)
Field set	set(fieldpattern)
Method call	call(methodpattern)
Method execution	execution(methodpattern)
Object initialization	this
Constructor call	call(constructorpattern)
Constructor execution	execution(constructorpattern)
Handler execution	handler(typepattern)
Advice execution	adviceexecution()
	within/withincode(methodpattern)
	cflow/cflowbelow(pointcut)
	this/target/args
	if(expression)

Table 2 – overview of the supported pointcuts in AspectJ

Within a pointcut definition, some run-time context-information of the joinpoint can be exposed with the pointcut designators: `this`, `args` and `target`. For example, the values of the arguments can be caught and used within the body of the advice. With the `if` pointcut designator, a condition can be specified with the pointcut.

With intertype-declarations, members that cut across classes can now be declared within the aspect itself. This further improves the modularity of crosscutting concerns within an aspect. Declaring new instance variables, methods or even changing the parent of a class or implementing an interface are all possible using intertype-declarations.

A definition of a pointcut is illustrated below in codefragment 8. Pointcuts are defined as a member of an aspect module.

```
pointcut moves(int parValue) :
   call(public Figure setX(int)) ||
   call(public Figure setY(int)) && args(parValue)
```

Codefragment 8 – pointcut definition in AspectJ

This pointcut, named `moves`, will catch all the calls to the methods `setX` and `setY` which are members of the `Figure` class. Furthermore, the values of the actual arguments are exposed and bound to the variable `parValue`.

4.1.2 Advice Model and Language

The actual behaviour of an aspect is defined within advice constructs. The advice code is just plain Java with a few extra keywords available. There are roughly three basic types of advice in AspectJ (and generally in almost every aspect-oriented language). These are before, after and around advice for respectively defining behaviour which should be executed before, after or around the related pointcut. There are also some derived types of advices. Behaviour can also be defined when returning from a method or when an exception is thrown.
Within advice code, the keywords `thisJoinpoint` and `thisJoinpointStaticPart` makes it possible to have reflective access to the particular joinpoint where the behaviour is added. For example, the name of the joinpoint or the signature of the method can be called.

The codefragment below shows a definition of around advice for the previous defined pointcut. Advices are very much like methods, but without a name because they are implicitly invoked. A pointcut is always defined with the advice. The `proceed` keyword is used to call the original method, if necessary with new parameter values.

```
around(int parValues) : moves(parValue) {
  System.out.println(thisJoinpoint.getMethodname() +
                     "called with parameter-value: " + parValue);
  proceed();
  System.out.println(thisJoinpoint.getMethodname() + " exited";
}
```

Codefragment 9 – around advice in AspectJ

Precedence rules are applied to determine the order of execution between several advices defined in one aspect. The advice that appears earlier in the aspect has precedence.

4.1.3 Aspect Model

At runtime, aspects are just like any normal Java object. However, aspects cannot directly be instantiated but their lifecycle is controlled by the weaving process itself. As default, aspects are instantiated as a singleton. This means that only one instance of the aspects can exist at one time. However, the developer of the aspect can declare when an aspect should be instantiated and thus override the default value. Different possibilities are:

- per-object: if an aspect A is in the execution flow of an object picked out by the pointcut, a new instance of this aspect is created.
- per-controlflow: a new instance of an aspect A is created for each control-flow picked out by the pointcut.

Because aspects are modeled as java objects, they can be put together in an inheritance relationship.

When several aspects work on the same object, some form of precedence can also be defined. If an aspect A has precedence over another aspect B, then all advices of aspect A are executed before any advice of aspect B.

4.1.4 Weaving

The weaver that comes with AspectJ was first a static-weaver which could only weave aspects at compile-time or post compile-time (binary weaving). But with the release of AspectJ 5, the weaver has been extended and can now handle load-time weaving. Load-time weaving means that the weaving of aspects is deferred until the particular class is loaded in the JVM. Run-time weaving is still not supported in AspectJ.

4.1.5 Example

To give a complete view of an aspect definition in AspectJ, part of the *observer* design pattern [36] implemented as an aspect is shown in this paragraph. Many design patterns can be implemented as aspects and resulting in better modularity [40].

The part of the AOP implementation of the observer design pattern is the abstract aspect which modularises the common parts needed in all observer instances. These common parts are the mapping between subjects and observers, and the update-logic is triggered when the state of a subject is changed.

```
public abstract aspect ObserverProtocol {

  protected interface Subject { }
  protected interface Observer { }

  private HashMap perSubjectObservers;

  protected List getObservers(Subject s) {
    if (perSubjectObservers == null) {
      perSubjectObservers = new WeakHashMap();
    }

    List observers = (List)perSubjectObservers.get(s);

    if ( observers == null ) {
      observers = new LinkedList();
      perSubjectObservers.put(s, observers);
    }

    return observers;
  }

  public void addObserver(Subject s,Observer o){
    getObservers(s).add(o);
  }

  public void removeObserver(Subject s,Observer o){
    getObservers(s).remove(o);
  }

  abstract protected pointcut subjectChange(Subject s);

  abstract protected void updateObserver(Subject s, Observer o);

  after(Subject s): subjectChange(s) {
    Iterator iter = getObservers(s).iterator();
    while ( iter.hasNext() ) {
      updateObserver(s, ((Observer)iter.next()));
    }
  }
}
```

Codefragment 10 – example of an aspect in AspectJ

For each concrete instantiation of the observer pattern, a concrete aspect is defined which inherits from this abstract aspect and implements the pointcut `subjectChange` to catch changes to the subject, and the method `updateObserver` which implements the logic for updating the observer.

4.1.6 Conclusion

AspectJ is certainly a simple to use and easy to learn aspect-oriented programming language. A rich joinpoint model together with an expressive pointcut language are the main advantages.

A significant disadvantage of AspectJ, however, is that aspects are tightly-coupled to their base object. This is easy to see because the method names of the base objects to which the aspect might be applied are hardcoded in the aspect itself. The consequence is that a certain aspect cannot be reused very easily with other kinds of objects. We can conclude that AspectJ is usable for relative small projects.

4.2 JAsCo

JAsCo is an aspect-oriented extension to the Java programming language developed in the System and Software Engineering Lab (SSEL) at the VUB. As opposed to AspectJ, JAsCo is not intended for use in commercial projects but rather for research purposes.

The JAsCo programming language is tailored to component-based software engineering and it is based upon two existing aspect-oriented programming languages: AspectJ and Aspectual Components [50]. It combines the expressive pointcut language of AspectJ with the aspect reusability mechanism of Aspectual Components. Aspect reusability is a critical element for component-based software engineering.

4.2.1 Aspect Beans and Connectors

In order for JAsCo to have reusable aspects, two new concepts are introduced: *aspect beans* and *connectors*. An aspect bean is just like a regular Java bean but intended to describe crosscutting behaviour. One or more *hooks* can be defined in an aspect bean, each containing an abstract pointcut and related advices. Only abstract references to the actual methods are used in a pointcut. This results in a generic and reusable aspect, independently from a specific context. Also regular Java member variables and methods may be defined in an aspect bean.

With connectors, the generic aspects are instantiated and bound to a specific context. This is done by connecting the method references of a pointcut with actual methods. Because a connector can instantiate more than one aspect, they also handle precedence rules and combination strategies for the aspects.

Following figure 12 shows the concepts of aspect beans and connectors.

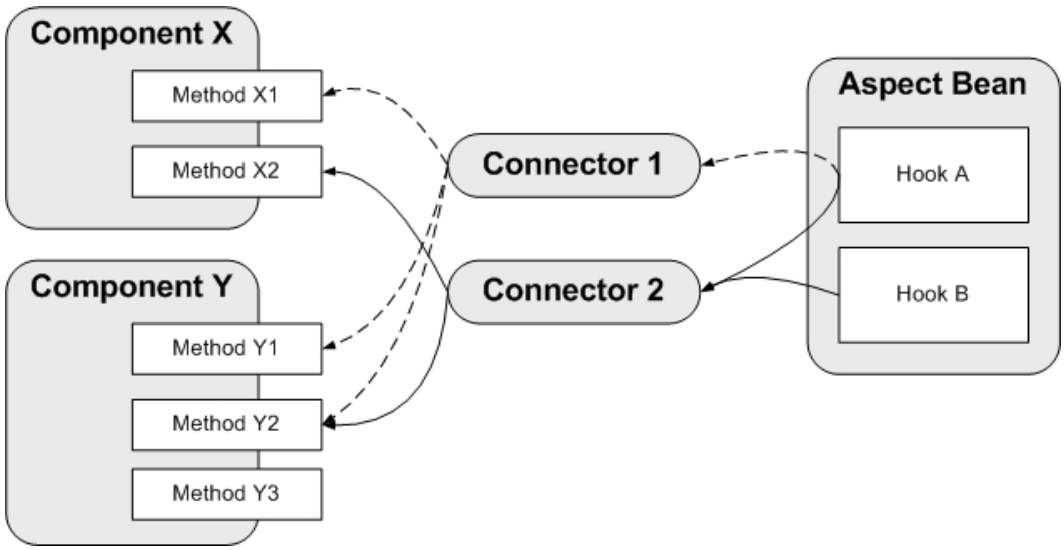

Figure 12 – aspect beans and connectors in JAsCo

Two connectors are defined. Connector 1 binds hook A to methods X1, Y1 and Y2. By using another connector hook A can be reused and bound to other methods. Connector 2 instantiates two hooks and precedence rules might be defined to control their interaction with the components.

4.2.2 Joinpoint Model and Pointcut Language

The dynamic joinpoint model and pointcut language of JAsCo is very equivalent with the current version of AspectJ, although the set of joinpoints available is not as rich as AspectJ's. Run-time context information might also be caught with the `this`, `target` and `args` pointcut designator, like in AspectJ.

Joinpoint	Pointcut designator
Method call	call(methodpattern)
Method execution	execution(methodpattern)
	within/withincode(methodpattern)
	cflow/cflowbelow(pointcut)
	this/target/args
Meta-data (Java 1.5 Annotations)	@...

Table 3 – supported joinpoints in JAsCo

The abstract pointcuts are defined in the constructor of the hook. The abstract references to the methods are defined as parameters in the hook constructor. Codefragment 11 shows how a hook constructor looks like.

```
aHook(method1(String s, int i),method2(..args) {
  execution(method1) && !cflow(method2);
}
```

Codefragment 11 – a hook constructor

Just like in AspectJ, it is also possible to attach a certain condition to the hook. The hook will trigger, i.e. the program execution will be cut, only if the condition is satisfied. This condition can be specified using the `isApplicable` construct in JAsCo. This construct can be seen as a special kind of method which returns a boolean value. The hook will only trigger if this value is true.

Because JAsCo is especially intended for research work, the pointcut language is extended with some advanced pointcut selectors, suited for following domains:

- Adaptive programming: given a certain traversal strategy of a class graph, pointcuts can be defined to cut some parts of this class graph on traversal. This helps to fulfill the *law of Demeter* [51][60].
- AWED (Aspects With Explicit Distribution): an extension to the pointcut language to define pointcuts and apply advices in a distributed environment [3].
- Stateful aspects: JAsCo is also extended to support the concept of stateful aspects. Because the thesis is about implementing stateful aspects in a workflow language, this concept is explained in detail in section 5.2 about implementations of stateful aspects.

4.2.3 Advice Model and Language

The advice model of JAsCo is very similar to that of AspectJ. Additional to the three basic advices (before, after and around), after returning, after throwing, around returning and around throwing are also supported.

The advice language is plain Java extended with some specific keywords. Just like in AspectJ, the keywords `proceed` and `thisJoinpoint` are available and have the same purposes. An extra `global` keyword is available to refer to the context of the aspect bean from a hook.

4.2.4 Aspect Model and Connectors

JAsCo supports both inheritance between aspects and hooks. Because hooks are implemented as a derived form of inner-classes, the inheritance semantics are similar to hooks.

Aspects are always defined independent from a concrete component which makes them highly reusable. A connector is used for binding an aspect to a concrete component. The instantiation and binding of a hook is explicitly done. Normally, only one aspect is instantiated for each binding. However, it is also possible to generate several instances for one binding. Following instantiations are possible:

- perobject: a unique hook for each target object.
- perclass: a unique hook for every target class.
- permethod: a unique hook for every target method.
- perthread: a unique hook for every executing thread.
- perall: a unique hook for every joinpoint.

The implementation of certain behaviour in a hook might be postponed until this hook is bound in a connector. This improves the aspect reusability even more because this behaviour can be customised when using the aspect. Behaviour can be postponed via *refinable* methods which are similar to abstract methods.

JAsCo provides a powerful and expressive combination mechanism for several aspects applied on the same application. This combination mechanism is based on *precedence rules* and *combination strategies*.
When several aspects are instantiated in a connector, the precedence of the advices can be determined by explicitly listing their order of execution. Another way of controlling the interaction between aspects is done via "combination strategies". By defining a combination strategy, it is for example possible to state that an aspect can only be applied together with another aspect. In other words, none or both aspects are applied.

4.2.5 Weaving

JAsCo features a powerful run-time weaver which makes it possible to add or remove aspects at run-time. Although weaving happens entirely at run-time, the performance of this weaver can compete with AspectJ's.

4.2.6 Example

The following example shows how a simple logging aspect is defined and deployed in JAsCo. Codefragment 12 illustrates the definition of the aspect. The binding of the aspect to a concrete context using a connector is shown in codefragment 13.

```
class Logger {
  hook LogHook {
    LogHook(method1(..args)) {
      execution(method1);
    }

    before() {
      writeLogentry("EXECUTING:" + thisJoinPoint.getSignature());
    }

    refinable void writeLogentry(String s);
}
```

Codefragment 12 – definition of an aspect in JAsCo

This aspect writes an entry into the log file before every execution of certain methods. Notice the abstract reference *method1* which is bound later to a particular method or set of methods.

```
static connector LogDeployer {
  Logger.LogHook log = new Logger.LogHook(void SomeClass.*()) {

    void writeLogentry(String s) {
      System.out.println(s);
    }
  }
}
```

Codefragment 13 – binding of the aspect to a concrete context through a connector

The connector will instantiate a hook and bind all the methods of class `SomeClass` which have no parameters to the abstract method reference.

4.2.7 Conclusion

JAsCo is a very rich featured aspect-oriented programming language. The idea of aspect reusability gives JAsCo a huge benefit over many other languages. Also the performance of the run-time weaver is a great asset to the language.
A consequence of the big number of implemented features is that the language, and in particular the pointcut language, becomes more complex than, for example, AspectJ.

5 Support for Stateful Aspects in AOP

Stateful aspects are an important part in this thesis research. Therefore, they are explained carefully in this section. Some existing implementations are also described in this section.

5.1 Definition of Stateful Aspects

Earlier in this chapter, it is mentioned that (almost) every aspect-oriented language features a dynamic joinpoint model. In those languages, joinpoints may be run-time events, which are dependent on the run-time stack or on some dynamic evaluated condition.
However, in certain applications it might be possible that some concern should only be

triggered when a particular sequence of run-time events has occurred. Such a sequence of events is sometimes called *a protocol*. For example, consider an editor which can only be closed when the modifications of the current document are already saved. This illustrates that a certain sequence of events (edit/create a document and save it) must be completed before the *close document* concern can be triggered [6].

The problem with such concerns is that the application must keep track of the history of events in order to know the state of the application. Consequently, this can be seen as a crosscutting concern because parts of the history tracking logic are encoded at several places in the application.
Unfortunately, the crosscutting nature of this concern remains when it is implemented as an aspect. This is because mainstream aspect-oriented languages rarely support *history-sensitive* pointcuts which makes it impossible to refer back to previously matched events. This means that the aspect itself should keep track of the history of events and this will end up tangled with the normal aspect behaviour.

Below is a code-fragment which illustrates this problem. It is remarkable that the main part of this aspect is needed for keeping track of the history. The history-tracking logic is written in bold.

```
public aspect PreventDirtyClose {
  private static boolean documentDirty;

  pointcut makeDocumentDirty(): call(void Editor.edit());
  pointcut makeDocumentClean():
    call(void Editor.save()) || call(void Editor.create());
  pointcut disposeDocument():
    (call(void Editor.quit()) || call(void Editor.create())) &&
    if(documentDirty);

   after(): makeDocumentDirty() {
    documentDirty = true;
  }

  after(): makeDocumentClean() {
    documentDirty = false;
  }

  void around(): disposeDocument() {
    // prevent quitting the editor or creating a new document
  }
}
```

Codefragment 14 – protocol-based concern as an aspect remains crosscutting.

From now on, it should be clear that concerns which are based on a history of events remains crosscutting, even when they are implemented as an aspect. Therefore, some new concept about aspects was needed to modularise such protocol-based crosscutting concerns in a clean way.

Douence et al., have introduced a formal model where aspects can be defined in terms of a sequence of run-time events. They call them s*tateful aspects* because they keep the history of run-time events and thus the state of the application into account [29][30]. Stateful aspects only trigger their behaviour upon successful completion of the specified sequence of events. With such kind of aspects, protocol-based concerns may now be implemented in a cleaner and modularised way.

Both JAsCo and AspectJ have already been extended with the concept of stateful aspects as part of research projects. Both implementations are described in the next section.

5.2 Implementations for Stateful Aspects

5.2.1 Stateful Aspects in JAsCo

Because protocols are frequently employed within component-based software development [33], JAsCo has also been extended with support for stateful aspects [61]. With this extension, linguistic support is provided for declaratively specifying protocols as a stateful pointcut expression. The developer is set free of the low-level implementation to recognize a protocol.

The protocol is described by a set of named transitions which corresponds to particular run-time events. With every transition, a pointcut and one or more destination transitions are specified. When the pointcut is matched during run-time, the transition fires and a destination transition is activated. The first defined transition is always considered as the start of the protocol. However, an extra `start` keyword can also be used to specify the first transition explicitly.
It is also possible to attach advice at every transition in the protocol. This advice is executed whenever the transition is fired. Codefragment 15 shows an example of a protocol specification in JAsCo.

The JAsCo extension for stateful aspects also supports some advanced features. An `isApplicable` method can be specified for each transition. This creates an extra condition to the firing of the transition. Also, protocols can be specified as being strict or non-strict (default). In contrast with a non-strict protocol, a strict protocol means that no other joinpoints may be matched between the transitions. The example in codefragment 15 illustrates a strict protocol. A stateful aspect can also be defined with a `complement` keyword which indicates that the aspect is only interested in the complement of the protocol, i.e. when the protocol is *not* executed.
A thorough explanation of these advanced features is not relevant to this thesis. The interested reader should check [61] for more information.

```
aspect DirtyCloseAspect {
  hook PreventDirtyClose {
    PreventDirtyClose(methodEdit(..args), methodQuit(..args)) {
      strict:
       Astate: execute(methodEdit) > Bstate;  // state is dirty
       Bstate: execute(methodQuit) > Astate;
    }

    after Btrans() {
      // prevent quitting the editor or creating a new document
    }
  }
}
```

Codefragment 15 – stateful aspects in JAsCo

The presence of the performant run-time weaver is a great asset for the implementation of stateful aspects in JAsCo. Because a stateful aspect is only interested in the joinpoints which are applicable for the current state (activated transition), it is unnecessary to weave code at every joinpoint required for the stateful aspect. In fact, this would give a substantial overhead at run-time.

Instead, when the state of the aspect changes, the run-time weaver unweaves the code at the joinpoints the aspect is not interested in anymore, and reweaves it at the joinpoints applicable for the current state. This implementation really creates a *jumping aspect* [24] because it literally jumps from joinpoint to joinpoint. An interesting side-effect is when the whole stateful aspect is not applicable anymore, it will be completely unwoven and does not cause any run-time overhead anymore.

In JAsCo, the specification of the protocol describes a finite-state automaton. As a result, it is only possible to specify protocols as a regular expression. Non-regular protocols such as *n times A; B* (with *n* an integer) cannot be specified.

5.2.2 Declarative Event Patterns in AspectJ

Declarative Event Patterns (DEP) [62] is an extension to AspectJ for describing non-regular protocols in a declarative way. Unlike JAsCo, which uses regular expressions to describe the protocol, DEP uses a context-free grammar. This kind of grammar gives the possibility to describe properly-nested event structures.

Two new constructs are added to the AspectJ language: a *tracecut* and a *history* primitive pointcut. The idea of a tracecut can be compared with that of a pointcut. The difference between them is that a tracecut defines a valid *pattern* of events and not a set of valid joinpoints. To define patterns of events, *primitive* tracecuts are available to capture an individual event in the execution. Two primitive tracecuts are available: entering a joinpoint is captured with the `entry` tracecut and leaving a joinpoint is capture with the `exit` tracecut. More complex execution patterns are described with *ordered* tracecuts. Each ordered tracecut is an expression involving primitive tracecuts or references to named tracecuts and is declared as a production rule which follows the rules of a context-free grammar.

The `history` primitive pointcut is needed to advice tracecuts. This pointcut evaluates to true if the provided tracecut is satisfied. Consequently, if the pointcut is matched (i.e. evaluates to true), related advice may be executed. Codefragment 16 gives an example of an advised tracecut definition.

```
aspect ComplexOrderedTracecut {
  tracecut a() ::= entry(call (void a(..)));
  tracecut b() ::= entry(call (void b(..)));
  tracecut c() ::= entry(call (void c(..)));
  tracecut nesting() ::= a() [nesting()] b() | c();
}

after() : history(nesting()) {
  // some advice
}
```

Codefragment 16 – example of a declarative event pattern

The production rules are clearly visible. Tracecut identifiers are regarded as non-terminals and are surrounded by brackets in the left-hand side of the production rule. The primitive tracecuts are the terminals of the production rule.

Certain semantic actions can also be applied to tracecuts. For example, a semantic action may reject an event occurrence dependent of the value of an argument. This is equivalent with the `isApplicable` method in JAsCo. Semantic actions can manipulate exposed state, such as parameter values, and is defined in a *semantic block*.

6 Conclusion about Aspect-Oriented Programming

Aspect-oriented programming is a recent and promising way for developing software that is better maintainable and understandable because it augments already existing paradigms with a new, advanced abstraction mechanism to modularise crosscutting concerns. Systems with ever increasing complexity can be built because of this programming paradigm. Although AOP is a young technology, there is much ongoing research to apply it within several software domains where crosscutting concerns are an important issue.

At the moment, aspect-oriented programming is especially focused on object-oriented programming. Other programming paradigms and related languages are not much supported by AOP. The next chapter will explain which implementations of aspect-oriented programming currently exist for workflow languages and what their main shortcomings are.

Chapter 5: Aspect-Oriented Programming in Workflow Languages

A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.
Frank Herbert, science fiction novelist (1920 - 1986), "Dune (First Law of Mentat)"

It is already mentioned in the dissertation that workflow languages have a lack of general support for modularising crosscutting concerns. Logging and monitoring crosscutting concerns are likely to be present in process workflows for auditing purposes, for example. As workflow languages exist mainly for implementing business processes, a business rule is an important concern within the workflow. It should not be surprising that business rules are also a typical example of crosscutting concerns [15]. Especially the connection of business rules with the domain objects becomes scattered in the core business process logic and may be tangled with other business rules [17][14]. Because business rules are volatile and evolve more frequently than the rest of the business objects, it is of extreme importance that they are modularised in order to have a maintainable and evolvable application. In the chapter about BPEL, it is explained that a business process is defined as one big sequence of activities. At the end of that chapter, in section 3.2.2, it is explained that crosscutting concerns such as business rules, cannot be specified in a clean, modularised and reusable way. Aspect-oriented support should therefore definitely be present to control this form of complexity in process workflows [12].

Computer scientists are more and more aware of this problem. With AO4BPEL as the first serious project which merges aspect technology with BPEL, research in this particular domain is emerging . Padus, for example, is another promising project which implements AOP support in the BPEL workflow language. The advantages of having a workflow language with AOP capabilities go further than only modularising crosscutting concerns. In [23] and [21] is illustrated how a business process can be dynamically adapted using an aspect-weaving engine. For example, hot-fixes to long running processes can be plugged in at run-time. This handles the problem about replacing a broken service without stopping the process.

The remaining part of this chapter shows two current research projects about AOP support within BPEL. Both language extensions are very different with each having their own specific advantages. For each language extension, the language itself is described by means of joinpoint model, pointcut language and advice language. Also the underlying structure and particular implementation decisions are also explained.

1 AO4BPEL

AO4BPEL [11] is generally accepted as the first implementation of aspect technology within the BPEL workflow language. This aspect-oriented extension of BPEL is also based completely on XML.

The main focus of AO4BPEL is to provide a solution to the most important shortcomings of BPEL:

- no support for modularising crosscutting concerns
- unable to modify the composition of a process at runtime

These shortcomings are already described previously at the end of chapter 4 about workflow languages.

Courbis and Finkelstein [22] have also presented an aspect-oriented language extension to BPEL which is very similar with AO4BPEL. Dynamic process modification by adding or removing aspects at runtime was also one of their goals for their project. In contrast with most of the aspect-oriented languages, the behaviour of the aspect is not written in the same language as the base language. Instead, Java is used to write the advices.

1.1 The Language

1.1.1 Joinpoint Model

The workflow specification in a BPEL process is composed of different activities which are the main building blocks of the program. Because activities are well-defined points in the BPEL process, it is natural to take each activity as a possible joinpoint. The joinpoints can be selected using their type (i.e. 'invoke') and attributes (i.e. 'operation'). The joinpoint model in AO4BPEL is rather limited because only the <invoke> and <reply> joinpoints are supported.

1.1.2 Pointcut Language

The pointcut language of AO4BPEL is Xpath [19]. This language is very convenient because a BPEL process is completely described with XML. With XPath, it is straightforward to select joinpoints, which are nodes in an XML file. Below is an example of a pointcut definition which selects all invocations of the *getCustomer* operation on port type *customerServicePT*.

```
<pointcut name="customerPointcut">
  //process//invoke[@portType="customerServicePT" and
                    @operation="getCustomer"]
</pointcut>
```

Codefragment 17 – a pointcut definition in AO4BPEL

Using XPath as the pointcut language has also a disadvantage. In the example of codefragment 17, it is clear that the pointcut description is tightly coupled with a specific activity in the process. This is a problem because suppose that the BPEL process definition is changed and this particular activity to which the pointcut refers to is placed somewhere else in the XML file, then the XPath expression in this pointcut definition have to be changed as well.

1.1.3 Advice Language

An advice is an activity which is inserted at the joinpoints selected by the specified pointcut. Like in most of the aspect-oriented languages, the advice language is the same as the base language, which is in this case BPEL. The three basic advice types are supported: before, after and around. An advice definition is illustrated below in codefragment 18.

```
<pointcutandadvice type="after">
  <pointcut name="…">
    …
  </pointcut>

  <advice>
    <invoke partnerLink="CarRentPortal" portType="carRentPT"
            operation="getCar" inputVariable="getCarRequest"
            ouputVariable="getCarResponse"/>
    <assign>
    …
    </assign>
  </advice>
</pointcutandadvice>
```

Codefragment 18 – advice definition in AO4BPEL

An advice is related with a pointcut by means of a parent node `pointcutandadvice` which has an attribute to define the type of advice.

1.2 Implementation

The biggest asset of AO4BPEL is its dynamic weaver which is necessary to satisfy the second requirement about dynamic process modification. By using a dynamic weaver, the process do not have to be halted in order to apply an aspect to it. For example, when a particular service becomes unavailable, an aspect can be added at run-time to inject some advice code which calls some equivalent service. The consequence of using a dynamic weaver is that a custom BPEL engine is needed. This custom engine has the capability to modify the process in such a way that an aspect can be applied at run-time. Figure 13 illustrates the architecture of the AO4BPEL execution engine.

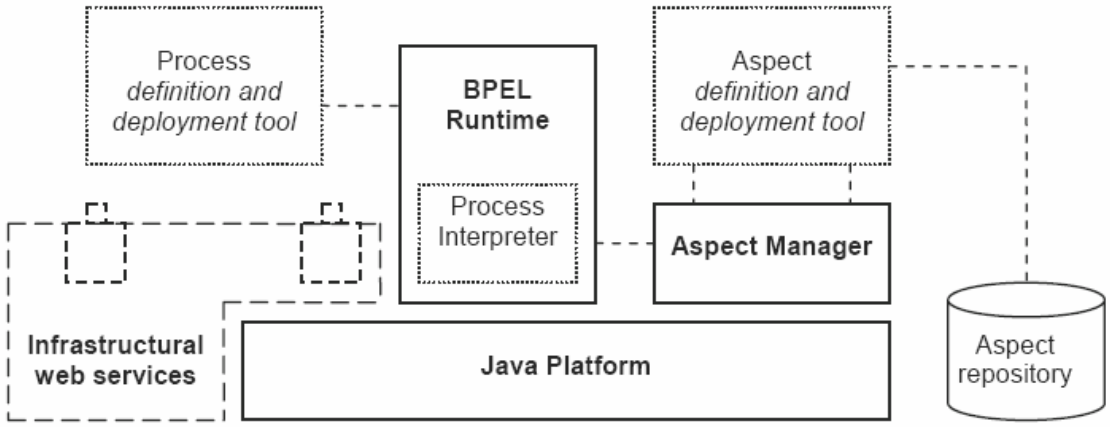

Figure 13 – architecture of the AO4BPEL execution engine

The execution engine is based upon BPWS4J [44] which is implemented in Java. The two main components are the *BPEL runtime* and the *aspect manager*. The BPEL

runtime takes the aspects into account and manages the BPEL process instances. The execution of the aspects is controlled by the aspect manager.

2 Padus

Padus [7] is another aspect-oriented extension to BPEL with some different and interesting capabilities comparing with AO4BPEL. Padus forms part of the larger WIT-CASE project and is developed in collaboration with the Distrinet research group of the KUL and Alcatel. Some design and implementation decisions of Padus are therefore influenced by the characteristics of the telecom sector. Providing decent support for modularising crosscutting concerns in BPEL is the main goal of Padus.

2.1 The Language

2.1.1 Joinpoint Model

It is obvious that the joinpoints in Padus are also related to the activities in a BPEL process. In contrast with AO4BPEL, Padus has a much richer joinpoint model in which nearly every activity can be selected. For every joinpoint, certain properties can be specified which correspond to the attributes of the appropriate activity. In this way, more specific joinpoints can be selected. The table below shows the list of supported joinpoints in Padus. The BPEL activity related to a joinpoint is easily deducible from its name.

Behavioural joinpoints		**Structural joinpoints**	
Invoking	Replying	Sequencing	Switching
Receiving	Assigning	Looping	Picking
Throwing	Terminating	Flowing	Scoping
Compensating	doingNothing (<empty>)		

Table 4 – joinpoint model of Padus

2.1.2 Pointcut Language

The pointcut language in Padus is completely based on logic meta-programming [25], which makes it fairly easy to reason about the base program, which is a BPEL process. Both the pointcut language and the implementation language of Padus are the same, i.e. Prolog [26]. A pointcut can be seen as reasoning about the base program by querying a database which contains facts about the BPEL process. The result of this query is the set of activities which match the pointcut definition. Due to the basic mechanisms of the logic programming paradigm, such as unification, context information is easily exposed through variables and may be used within the aspect.

There are many other significant advantages of using a logic language as a pointcut language. For example, if XPath is used as the pointcut language, the structure of the process is often hardcoded in the pointcut. XPath is of course a very suitable language for selecting nodes in an XML file, but the path to this particular node, or joinpoint, is absolute. When the structure of the process is changed, the XPath expression may have to be changed as well. The logic pointcut language of Padus on the contrary, has no indication at all to the structure of the process. Instead of using asterisks as wildcard mechanism, the unification process in a logic language is also a much more powerful because of the use of variables.

```
<pointcut name="invocations(Jp, Operation)"
          pointcut="invoking(Jp, 'aPartnerlink', 'aPorttype',
                             Operation)" />
```

Codefragment 19 – a pointcut definition in Padus

Logic predicates are available to select the appropriate joinpoints. For every predicate, certain variables can be set to constrain the selection of joinpoints or to expose information of the joinpoint to the aspect (for example, the name of an activity). For each activity which matches the pointcut, the `Jp` variable holds a reference to this particular activity. A pointcut definition consists of one or more predicates which might be combined with each other through the built-in Prolog operations ',' and ';', which corresponds respectively with *and* and *or*. The example below in codefragment 19 illustrates a pointcut definition in Padus. This pointcut selects all the <invoke> activities with attributes 'aPartnerlink' and 'aPorttype'. The variable `Operation` will be bound to the name of the invoked operation and may be used in the advice.
There is also an alternative notation for selecting activities in a pointcut. In this notation, the predicate accepts a list of key/value pairs in which the attribute values of the corresponding activity can be specified. This notation improves the readability of the pointcut because the name of the attributes are explicitly mentioned. In fact, the notation used in codefragment 19 is internally transformed to this notation. This notation is used in codefragment 24.

2.1.3 Advice Language

Besides the traditional types of advice (before, after and around), Padus provides an extra *in* advice. With this kind of advice, extra behaviour can be added inside an activity. For example, an additional sequence of activities can be put within a <flow> activity. This kind of advice cannot be easily emulated with one of the basic advice types and is therefore a great asset for Padus. Without this *in* advice, many duplicated code would be generated in order to obtain the same effect with after, before and around advices.

```
<around joinpoint="Jp" pointcut="invocations(Jp, Operation) ">
  <sequence>
    <assign>
      <copy>
        <from expression="begin invoking $Operation" />
        <to variable="logVar" />
      </copy>
    </assign>
    <invoke partnerLink="logging" portType="log:loggingPT"
            operation="log" inputVariable="logVar" />
    <proceed>
    <assign>
      <copy>
        <from expression="end invoking $Operation" />
        <to variable="logVar" />
      </copy>
    </assign>
    <invoke partnerLink="logging" portType="log:loggingPT"
            operation="log" inputVariable="logVar" />
  </sequence>
</around>
```

Codefragment 20 – an advice definition in Padus

Codefragment 20 illustrates the definition of an around advice. The type of an advice is specified with an appropriate XML element and the `pointcut` attribute specifies the related pointcut. The joinpoint attribute is necessary so that the weaver knows on which joinpoint the advice must be woven. Because this variable is matched with a certain activity during the weaving process, certain context information about this activity could be consulted in the advice. The values exposed by the pointcut, such as `Operation`, can be accessed in the advice body with the '$' character. An extra <proceed> activity is available to include the original behaviour.
Advice might also be defined in a separate advice element identified by a name and, if necessary, with related parameters. Actual advice definitions refer to this separate advice construct using the `<advice>` construct and its name. This allows reuse of advices within an aspect.

Process-level information of BPEL, like namespaces, variables or partnerlinks, can also be extended with the <using> construct. Some basic form of aspect inheritance can be simulated through the use of the <include> keyword. Codefragment 21 and 22 illustrates how aspect inheritance can be used for a billing concern.

```
<aspect name="GenericBilling"
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:bpel=http://schemas.xmlsoap.org/ws/2003/03/business-process/
  xmlns:bill="my.billing.uri">

  <using>
    <namespace name="xmlns:bill" uri="com.bptest.billing" />
    <partnerLink name="billing" partnerRole="Billing"
                 partnerLinkType="bill:BillingLT" />
    <variable name="billingMsg" type="bill:BillMsg" />
  </using>

  <pointcut name="confCallStarts(Jp)"
            pointcut="invoking(Jp, 'SyncConfCall',
                        'ConferenceCallPT', 'createConfCall')" />

  <pointcut name="confCallEnds(Jp)"
            pointcut="invoking(Jp, 'SyncConfCall',
                        'ConferenceCallPT', 'closeConfCall')" />

  <advice name="billService">
    <bpel:invoke partnerLink="billing" portType="bill:BillingPT"
                 operation="billService"
                 inputVariable="billingMsg"/>
  </advice>
</aspect>
```

Codefragment 21 – a generic aspect in Padus

The generic billing aspect in codefragment 21 implements the common parts of the billing concern. Pointcuts that select the start and end of the conference call to which the billing is applied and a reusable advice definition which implements an invocation of the billing service are defined in this aspect.

```
<aspect name="FixedFeeBilling"
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xmlns:bpel=http://schemas.xmlsoap.org/ws/2003/03/business-
process/>

  <include name="GenericBilling" />

  <after joinpoint="Jp" pointcut="confCallEnds(Jp)">
    <bpel:sequence>
      <bpel:assign>
        <bpel:copy>
          <bpel:from expression="string('1,5 EUR')"/>
          <bpel:to variable="billingMsg" part="payload"
                   query="/typ:Bill/typ:Price"/>
        </bpel:copy>
      </bpel:assign>
      <bpel:advice name="billService" />
    </bpel:sequence>
  </after>
</aspect>
```

Codefragment 22 – an aspect definition derived from a more generic aspect

Through the use of <include>, the aspect in codefragment 22 is derived from the generic aspect of codefragment 21. This specific aspect defines some advice which implements a fixed pricing for the conference call. The generic advice definition for invoking the billing service is used in this specific aspect.

2.1.4 Deployment Language

With aspect-oriented programming, it happens often that multiple aspects are applied to the same base application. Some languages, such as AspectJ, determine the order of application implicitly. Padus on the other hand provides a separate and powerful deployment language to let the developer himself choose the order of aspect instantiation and application.

```
<deployment>
  <!-- deploy aspects to a concrete process -->
  <aspect name="…" process="…" id="…" />
  <aspect name="…" process="…" id="…" />

  <!-- precedence rules -->
  <precedence [process="…"] />
    <aspect id="…" [advice="before|after|around|in"] />
    <aspect id="…" [advice="before|after|around|in"] />
  </precedence>
</deployment>
```

Codefragment 23 – deployment language of Padus

The structure of the deployment language in Padus is shown in codefragment 23. Two main parts can be distinguished from each other. The first part is responsible for instantiating and applying the aspects to a concrete process. Of course, it is also possible to apply the same aspect to a different process. The second part specifies the composition precedence rules of the aspects when they apply to the same joinpoint. The precedence rules might be limited to a certain process and specified per advice-type.

2.2 Implementation

Many design and implementation decisions are determined by the goals of the WIT-CASE project. Because Padus is meant to be used to describe real-time processes on a telecom service delivery platform, performance is an important requirement. In order to minimise the runtime overhead, a static weaving approach is chosen. The consequence of this decision is that aspects cannot be woven at run-time and, as opposite to AO4BPEL, no dynamic process modification is possible. On the other hand, because aspects are woven at compile-time, the resulting BPEL process can be executed on a standard BPEL execution platform. No custom execution engine is needed and standard BPEL tools might be used. Figure 14 below illustrates how aspects are woven in Padus.

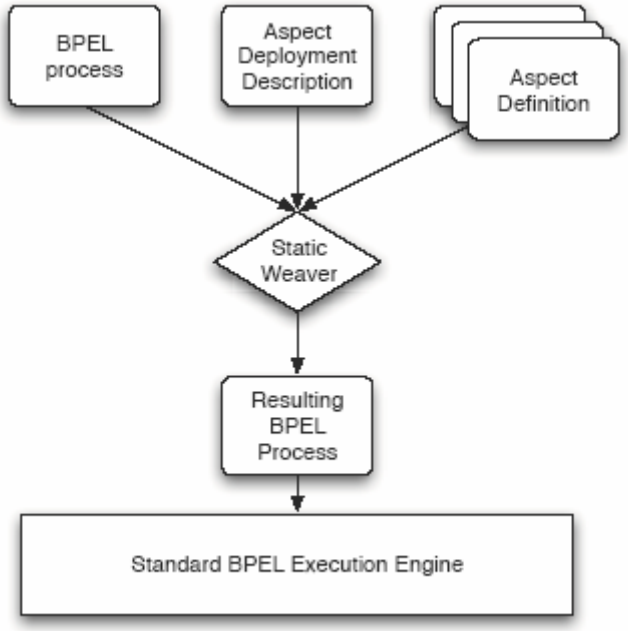

Figure 14 – Padus weaver architecture

The Padus weaver is completely built in a logic-oriented manner by using the SWI-Prolog logic engine. This makes that the implementation language is Prolog. The weaving process is broadly outlined as following:

1 First, the BPEL process and aspects are translated to facts, each representing an activity. Pointcuts defined in the aspect translated to rules.

2 The logic engine applies the rules (pointcuts) to find all the facts (activities) which are a solution to this rule. Each solution denotes a joinpoint at which the aspect can be woven.

3 Finally, an XML transformation engine is used to transform the facts back to a BPEL process. On transformation, advice is woven into the process taking the concrete advice semantics into account.

3 What about business protocols?

By introducing AO4BPEL and Padus, it is illustrated that there is ongoing research work trying to tackle the problem of crosscutting concerns in process workflows. These efforts are very promising, but nonetheless, certain issues about crosscutting business rules cannot be solved yet by these implementations. Verheecke and Cibran explain in their paper five categories of business rules whose triggering conditions are based on the dynamics of the execution of the core application [16]. They explain how to solve these crosscutting business rules with the aid of stateful aspects. Their paper certainly proves that stateful aspects are needed in AOP implementations for workflow languages. Unfortunately, none of the projects so far provides decent support for implementing business protocols in a webservice composition context.

To give at least some basic support for implementing dynamic business rules in a workflow language, Padus should be extended with at least a proof-of-concept implementation of stateful aspects.

Chapter 6: Stateful Aspects in Padus

Each problem that I solved became a rule which served afterwards to solve other problems.
Rene Descartes (1596-1650), "Discours de la Methode"

The language extension of Padus, which supports the implementation of stateful aspects, is discussed in this chapter.

First, the design of the new language is explained and motivated. Examples are given which illustrate the use of the language and how protocols are described. Thereafter, the implementation of this extension is explained. How Padus represents a protocol internally is described first. A through explanation of how a protocol is woven in the process, including history-tracking, is given next. The analysis and solution of the non-deterministic behaviour of BPEL processes and their influence on protocol weaving follows and finalizes this implementation of stateful aspects in Padus.

This chapter is concluded with a discussion about the performance and possibilities of this implementation.

1 New Language Constructs in Padus

1.1 Extending the Pointcut Language

In order to be able to declaratively describe protocols in Padus, some new language constructs and keywords are needed, which holds an extension of the pointcut. The section about stateful aspects at the end of chapter 3, illustrated that a small sub-language is typically added to the pointcut language. For this implementation of stateful aspects, it is opted to describe a protocol as a regular language. The reason for the choice of this kind of language is twofold. First, because this is a proof-of-concept implementation, a regular language is less complex and easier to implement than, for example, a context-free language. Second, in most of the cases, a regular language is sufficient to describe a protocol.

Regular expressions are often used to define a regular language because it provides a concise way for writing the grammar of the language. Three basic operations and an alphabet of symbols is needed for writing regular expressions. The operations are: concatenation, union and Kleene's star. The set of defined pointcuts in the aspects can be considered as the alphabet of symbols. By writing down a regular expression with these operations and alphabet, a particular protocol is defined. To preserve the description of protocols in Padus as simple as possible, only these three basic operations are supported.

The pointcut language of Padus is logic-based where predicates are available to select the appropriate joinpoints and combine them into a pointcut. In direct line with this logic-based nature, an extra `sequence` predicate is added to the pointcut language to indicate that the pointcut represents a protocol specification. This predicate accepts a well-formed regular expression as a parameter which represents the protocol. A regular expression is written down using three extra predicates. A predicate is available for each possible operation:

- Concatenation: concat([…])
- Union: choice([…])
- Kleene's star: loop(…)

The predicates `concat` and `choice` accept a list of regular expressions which are then combined according to the operation represented by the predicate. The `loop` predicate accepts only one regular expression as argument. Predicates may be nested within each other to get a proper defined protocol.

At the syntax level, a particular protocol is considered just like an ordinary pointcut where some extra behaviour should be added. Therefore, protocols are written like any other normal pointcut with the <pointcut> construct.

Codefragment 24 illustrates a pointcut which specifies a protocol as a regular expression with these available predicates.

```
<pointcut name="amazonSearch(Jp)"
          pointcut="invoking(Jp, [name(searchAmazon)])" />

<pointcut name="retrieveHotel(Jp)"
          pointcut="invoking(Jp, [name(retrieveHotel)])" />

<pointcut name="someProtocol(Jp1, Jp2)"
          pointcut="sequence(concat([retrieveHotel(Jp1),
                                     loop(amazonSearch(Jp2))]))" />
```

Codefragment 24 – protocol specification in Padus

The pointcut `someProtocol` specifies a protocol by using the `sequence` predicate. This protocol is satisfied if the `retrieveHotel` pointcut is matched followed with zero or more matches of the `amazonSearch` pointcut during execution of the process. When this protocol is written using the more common notation for regular expression, then it would look like `retrieveHotel(amazonSearch)*`.

1.2 Extending the Advice Language

In the previous section about the pointcut language extension, it is mentioned that the protocol specification is regarded as an ordinary pointcut. This has the consequence that the advice language does not have to undergo many modifications. Attaching additional logging behaviour after a particular protocol is matched is equivalent with attaching behaviour to some ordinary pointcut. Codefragment 25 below shows how it should be done.

```
<after joinpoint="Jp2" pointcut="someProtocol(_, Jp2)">
  <sequence>
    <assign>
      <copy>
         <from expression="'*** Logged protocol ***'" />
         <to variable="logMessage" part="parameters"
            query="/log:log/log:p0" />
      </copy>
    </assign>
    <invoke partnerLink="logPL" portType="log:LogService"
             operation="log" inputVariable="logMessage"
             outputVariable="logResponse" />
  </sequence>
</after>
```

Codefragment 25 – attaching advice on a protocol

By looking at the protocol specification illustrated in codefragment 24 and the advice defined in codefragment 26, it is clear that the variables `Jp1` and `Jp2` are bound to the activities for respectively invoking the 'retrieveHotel' operation and the 'amazonSearch' operation. This binding of variables to activities is explained earlier in chapter 5. Notice, however, that one variable (`Jp1`) is replaced by an underscore. In Prolog, an underscore is used to indicate an anonymous variable which means that this variable is not important. This is reasonable here because the advice should be attached after the whole protocol is executed, i.e. after the last pointcut in the protocol is matched. Because the `Jp2` variable represents the activity which corresponds with the last pointcut in the protocol, this is the one to use in the advice.

An additional possibility is to specify that a particular advice should be executed during the execution of the protocol. This means that it is possible to attach advices on the pointcuts which are part of the protocol. For example, consider the protocol *login – browse catalogue – add items to basket – checkout – logout*. Advice could be attached to the protocol which should be executed before the customer checks out, but only when the protocol is successfully matched so far.

To make this possible, no changes to the syntax of the advice language are needed. Instead, the advantages of logic-oriented programming, and unification in particular, comes into the spotlight here. Because each pointcut in the protocol is matched with a variable, this variable can be used to indicate where in the protocol the advice has to be woven.

```
<after joinpoint="Jp1" pointcut="someProtocol(Jp1, _)">
  …
</after>
```

Codefragment 26 – attaching advice within a protocol

By indicating now that this advice must be woven after joinpoint `Jp1` of the protocol `someProtocol`, implies that the weaver now weaves the necessary protocol trace advices to recognize the execution protocol up to the invocation of the 'amazonSearch' operation. The `Jp2` variable is not needed here and therefore it is replaced by an underscore. The aspect behaviour is executed just after the 'retrieveHotel' operation is invoked and only if the protocol is successfully matched up to this point.

2 Implementation of Stateful Aspects in the Padus Weaver

This section explains how the implementation of the Padus weaver is modified such that the specified protocol can be successfully matched during execution of the BPEL process. As previously mentioned, Padus is a logic-based aspect-weaver and hence it is implemented in a logic-oriented language, namely Prolog. Nevertheless only basic language constructs and mechanisms are used and thus it should not be very difficult to understand the implementation, a comprehensive introduction on Prolog can be found in [20].

2.1 Modeling a Protocol as a Finite State Automaton

The most substantial part which let stateful aspects work, is the fact that the history of the execution of a particular sequence of events is automatically tracked. Each execution of an event can be seen as a change of the state of the application. Keeping track of the protocol actually means knowing when such a state change has occurred. The aspect is executed only when the protocol matches the execution of the process.

The first important task is to make sure that the specification of the protocol is understood by the weaver. In the previous section about the pointcut language extension, it is explained that a protocol is specified by writing down a regular expression. To make it possible for our weaver to validate this protocol with the execution of the process, the regular expression will be translated into a finite state automaton (FSA). With a FSA, it is possible to validate a string to a particular regular language in a very efficient way.
A finite-state automaton can be formally defined as a 5-tuple $<S, S_0, E, \Sigma, \delta>$ where:

- S is a finite set of states
- $S_0 \in S$ is a the start-state
- $E \subseteq S$ is a finite set of end-states
- Σ is the input alphabet
- δ is the transition function $S \times \Sigma \rightarrow 2^S$

Informally, a FSA can be defined as a labeled, directed and cyclic graph. Each node in the graph represents some state, and a set of labeled transitions are defined to move from one state to another. There is also a designated start-state and a set of end-states. A transition is fired if its label matches with the first symbol of the string. The point now is that a string is recognized by the FSA, and thus satisfies the regular language, if it finds a path from the start-state to an end-state by reading the string symbol by symbol.

In a stateful aspect perspective, the protocol specification, which is described as a regular expression, is represented by a FSA. The string which has to be validated by the automaton is a particular execution flow of the process. The figures 15 and 16 below clarify this.

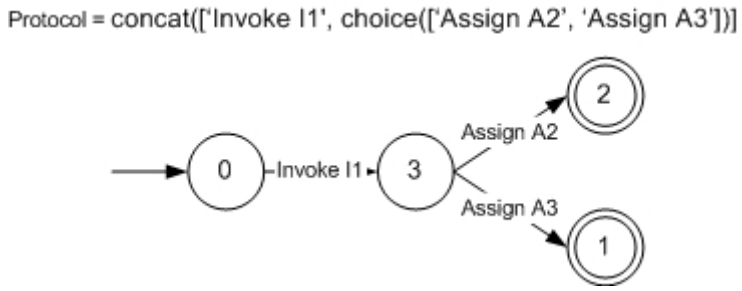

Figure 15 – a protocol with corresponding finite-state automaton

Figure 15 illustrates the finite-state automaton as a representation of the protocol, which is specified as a regular expression. This FSA accepts the strings `'InvokeI1 AssignA2'` and `'InvokeI1 AssignA3'`. Figure 16 shows how such a string is obtained from a particular execution of the BPEL process.

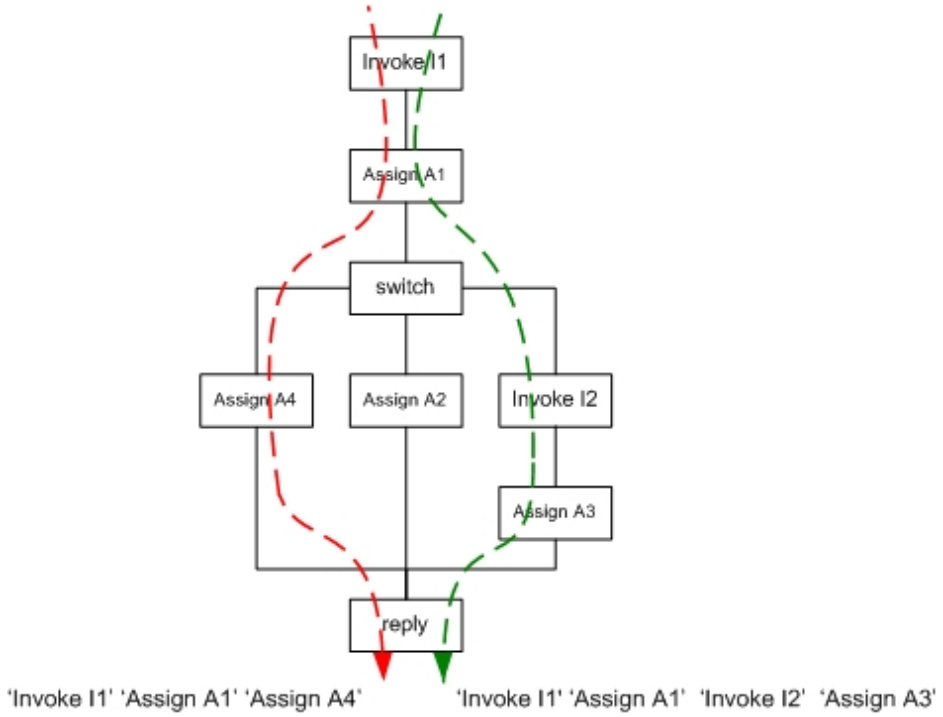

Figure 16 – obtaining a 'string' from a process execution

If such a string, which is the result of a particular execution of the process, is matched by the FSA of figure 15, then the specified protocol is executed and the aspect may be applied. By default, this implementation accepts protocols in a non-strict way. This means that other activities may be executed in the meantime which are not part of the protocol specification. This is sensible because otherwise every activity, which is possibly executed during the protocol but makes no part of it, should be added to the protocol specification. This could be very cumbersome because a <sequence> activity for example does not implement any logic and thus never adds any value to a protocol. 'InvokeI2' is an example of such an activity executed during the protocol. It is clear that the right (green) arrow is an execution which matches the protocol, while the execution flow represented by the left (red) arrow does not satisfy the protocol.

The remaining parts in this section will explain the translation of the protocol to a finite state automaton and how it is represented in Padus as logic facts.

2.1.1 Representing a Finite State Automaton in Padus

For several reasons, Padus is implemented in a logic-based language. This has the consequence that the automaton, generated from the regular expression, should be represented as a set of logic facts stored in the knowledge base of Prolog.

A finite state automaton can be represented in Prolog by listing its transitions as logic facts. Each transition is inserted into the knowledge base in the following form:

```
transitionDFA(<FSA name>, <source-state>, <symbol>, <destination-
state>)
```

To distinguish each generated FSA, a particular name is chosen which is equal to the name of the protocol pointcut. After all, more than one protocol may be defined in the aspect so it is possible that multiple FSAs are generated. The <symbol> part of each transition is the pointcut which has to be executed in order to activate the transition.

The start-state and one or more end-states are also a fact about the FSA. For each end-state, there is a separate fact in the knowledge base of Prolog. The facts have the following form:

```
startDFA (<FSA name>, <state>)
endDFA(<FSA name>, <state>)
```

Codefragment 27 gives an excerpt of the knowledge base of Prolog after generating the finite state automaton from figure 15. Remark that the symbols of each transition is a particular pointcut. The states are stored as numbers and the state [0] is always the start-state.

```
transitionDFA(someProtocol, [0], invokeI1(_), [4]).
transitionDFA(someProtocol, [4], assignA2(_), [7]).
transitionDFA(someProtocol, [4], assignA3(_), [9]).

startDFA(someProtocol, [0]).

endDFA(someProtocol, [7]).
endDFA(someProtocol, [9]).
```

Codefragment 27 – representation of a finite state automaton in Prolog

2.1.2 Translation of the Protocol to a Finite State Automaton

Because a finite state automaton is a graphical representation of a regular expression, there exists a formal algorithm to translate a regular expression to a particular FSA. This section explains how this algorithm is implemented in Padus as part of the stateful aspect extension.

The algorithm for converting a regular expression to a FSA is quite compact and rather simple to understand because it is recursively defined. Figure 17 shows that a separate FSA is generated for each operation in the regular expression (concatenation, union and Kleene's star). The final FSA is obtained by combining the partial FSAs.

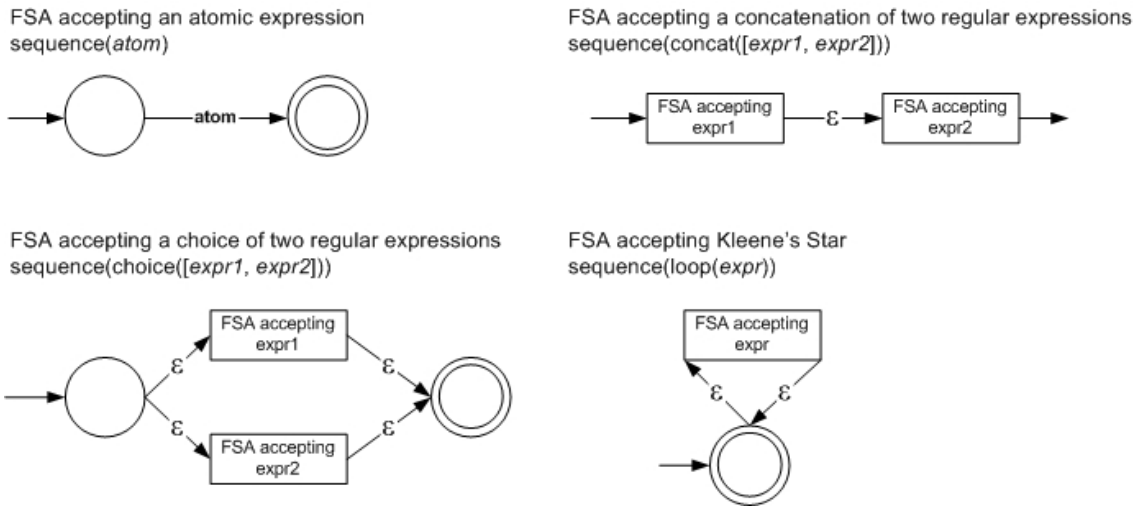

Figure 17 – composing a finite state automaton from a regular expression

This conversion algorithm is, not surprisingly, implemented in a logic-oriented manner in Prolog. As mentioned before, three separate types of FSAs have to be generated and combined appropriately afterwards. There exists one logic rule `makeSubNFA` with four different implementations. For each operation of the regular expression, there is an implementation generating its corresponding type of finite-state automaton. A fourth implementation is needed to generate an atomic FSA accepting a single symbol (i.e. pointcut).

```
makeSubNFA(MachineName, Expr, StartState, EndState) :- …
makeSubNFA(MachineName, concat(Expr), StartState, EndState) :- …
makeSubNFA(MachineName, choice(Expr), StartState, EndState) :- …
makeSubNFA(MachineName, loop(Expr), StartState, EndState) :- …
```

The subexpressions of each predicate are processed recursively and completes the generation of the (provisional) finite-state automaton.

Remark that the resulting FSA has many empty transitions. These transitions only make the FSA more complex and may be safely removed. Moreover, this algorithm results in a non-deterministic finite-state automaton. This means that there are multiple transitions accepting the same symbol that start from the same state, but with a different end-state. As a result, a choice has to be made between these transitions and therefore the path from the start-state to an end-state is not defined deterministically. Backtracking is needed when the wrong transition is chosen. The process of matching a string with a FSA is therefore much more complex and difficult to implement. Fortunately, there exist an algorithm which makes a particular non-deterministic FSA equivalent to a deterministic FSA. A deterministic finite-state automaton without any empty transition is generated by the Padus weaver.

The whole process of generating the finite-state automaton is initiated by the `generateFSA` rule which creates the start- and end-state of the FSA, makes this FSA empty-free and finally converts it to a deterministic FSA.

2.2 Weaving History-tracking Code in the Process

From now on, it is clear that the finite-state automaton is the mechanism to use for implementing stateful aspects. The best way for keeping up with the state of the process is with an FSA. This section will explain how the history of the application is traced. Some instrumental code is needed and is woven by the weaver at compile-time. In fact, tracing the history of the process boils down to following the finite-state automaton.

The logic which traces the process history is added by injecting small pieces of advice into the process. The traversal of the FSA is controlled by these advices which check and modify the current state of the process and decide whether or not the aspect has to be applied. Because a transition of the finite-state automaton is fired when the corresponding pointcut is executed, all these instrumental advice blocks are added after that pointcut. When the current state of the process become an end-state of the FSA, the aspect behaviour is executed. An extra variable is also woven into the process to store the current state of the FSA.

The protocol tracing advices are generated as ordinary <switch> activities. Figure 18 illustrates how these <switch> activities are formed and where they are placed in the process. At the very beginning of the protocol matching, the state-variable which holds the current state of the FSA is initially set to the start-state of the FSA, which is always '[0]'. This is done before each pointcut at which the protocol might start. In other words, before each transition which starts from the start-state (arrows 1a and 1b on figure 18).
Further, for transitions which do not lead to an end-state, tracing advice is woven after the corresponding pointcut (arrows 2a and 2b). When multiple transitions exist with the same label, a separate branch is made in the <switch> activity for every such transition. The state is changed in theses branches according to the transition. Similar tracing advice is generated for the transitions which do lead to some end-state, but the aspect behaviour is injected directly into the branches. Arrow 3 on figure 18 illustrates how this is done. Two transitions exist with the same label 'InvokeI1'. Consider therefore the two branches in the tracing advice which changes the state.

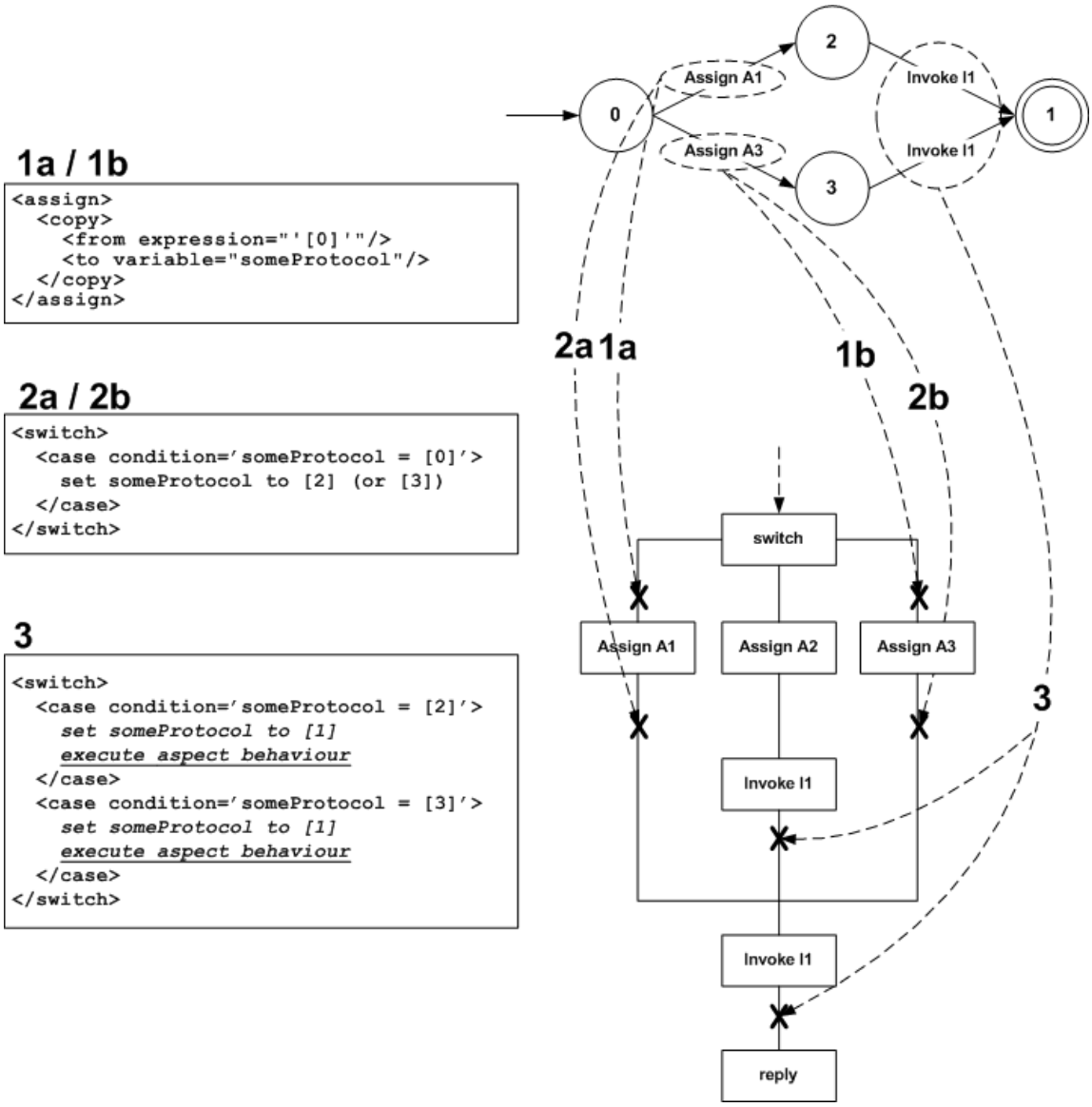

Figure 18 – tracking the traversal of the FSA with advice to keep up with the state of the process

This implementation of stateful aspects in Padus offers two ways for attaching advices to a protocol specification. One way is by specifying that behaviour should be executed after the whole protocol is matched with the execution flow of the process. The other possibility is to execute behaviour while matching the protocol. In other words, consider the protocol represented as a FSA, advice is then attached to a transition. Weaving advice for this situation is similar to weaving advice after a whole protocol is matched. Instead of injected the aspect behaviour in the tracing advices for transitions leading to an end-state, the aspect behaviour should now be added to the tracing advice of the corresponding transition. Figure 18 illustrates how behaviour is woven after the whole protocol is matched. For the other situation where behaviour should be executed while matching the protocol, consider the advice specification of codefragment 28.

```
<after joinpoint="Jp2" pointcut="someProtocol(Jp1, Jp2, Jp3)">
  …
<after>
```

Codefragment 28 – after advice specified in the context of a particular protocol

This advice should only be executed after the execution of the pointcut and only if the execution of the process has matched the protocol so far to this point. Codefragment 29 shows the code which is woven after the pointcut 'AssignA3' and should replace tracing advice 2 in figure 18.

```
<switch>
  <case condition='someProtocol=[0]'>
    set someProtocol to [3]
    insert aspect behaviour
  </case>
</switch>
```

Codefragment 29 – code for executing advices within a protocol

In fact, this is very equivalent with weaving advice when the whole protocol is matched. Instead of inserting the advice at the pointcuts which lead to an end-state, the advice is now inserted at the appropriate pointcut which is part of the protocol. The advice is also surrounded with a condition just to make sure that it is only executed when the protocol is matched so far.

An example BPEL process with a specific protocol woven into it can be found in appendix C.

2.3 Handling the Non-Deterministic Behaviour of the <flow> Activity

BPEL, and workflow languages in general, have the capability of executing parts of the process concurrently. For BPEL, the <flow> activity handles the concurrent execution of activities and hides all the details about synchronising,… for the developer. Multiple sequences of activities can be specified in separate branches in the <flow> activity which are then executed in parallel. The BPEL execution engine translates the <flow> activity into a set of separate threads, one thread for each sequence. It is common knowledge that the execution of multiple threads is not done exactly at the same time. Instead, small amounts of CPU time is given alternately to each thread very quickly and mimics in this way the concurrent execution of separate threads. Thus, the different sequences of the <flow> activity are still executed sequentially.

A particular problem arises when a BPEL process contains a section with parallel executed sequences of activities. The order of the execution of activities cannot be exactly determined anymore and hence the BPEL process is executed in a *non-deterministic* fashion. Suppose the following scenario: a particular process contains two sequences of activities which are executed in parallel. Both sequences invoke an external webservice: the first sequence uses service A, while the other sequence uses service B. Now imagine that the invocation of service A is delayed due to a congested network. This has the consequence that service B might be executed before service A. The next day, when the process is started again, the network on which service B is deployed is now congested. The immediate result is that service A might now be executed before service B. This clarifies that the behaviour of a process with concurrent sequences of activities is partially determined by external factors and thus it is impossible to exactly determine the order of execution beforehand.

Now, let's go back to stateful aspects. The definition of a protocol as a regular expression, can be seen as specifying some order in which certain activities must be executed before the aspect behaviour is executed. The above mentioned problem comes

into play when the protocol is defined on activities which reside in a branch of a <flow> activity. Because the process is executed non-deterministically, the protocol matching is not reliable anymore. One day, the protocol might be successfully matched, but on another run of the process, the same protocol might not be matched. The aspect developer does not always know that the specified protocol spans one or more <flow> activities. Therefore, the protocol matching should be independent of external behaviours and consistently match the process execution. Modifying the BPEL process itself by removing certain <flow> activities is definitely not an option because then, the obliviousness characteristic of AOP languages would not be satisfied.

If multiple <flow> activities are nested within each other, this issue might become quite complex because the different execution flows of the process increase exponentially. Some complexity analysis of BPEL processes is done in [9]. The next paragraph proposes a solution to this problem. Nested <flow> activities are not taken into account to keep everything clear.

2.3.1 Using the Shuffle Product to Enumerate all possible Flows

In the previous paragraph it is explained that a process with concurrent executing activities have different execution flows. It is unacceptable that the matching of a certain specified protocol is dependent on the time of execution of the process. The protocol should either always match or never match for the same execution flow. In the stateful aspect extension of Padus, the protocol specification is directly translated to a deterministic, optimal FSA. Unfortunately, with FSAs it is impossible to model parallelism which means that transitions can only be fired in sequential order. Still it is necessary that every possible execution flow should be accepted by the FSA and therefore, some efficient workaround is needed.

For clarity purposes, the figure 19 below shows all the possible execution flows of a process with concurrent activities. The order of the activities for each branch of the flow should be remained. For example, the execution of activity ‘invoke I1’ comes always before activity ‘invoke I2’.

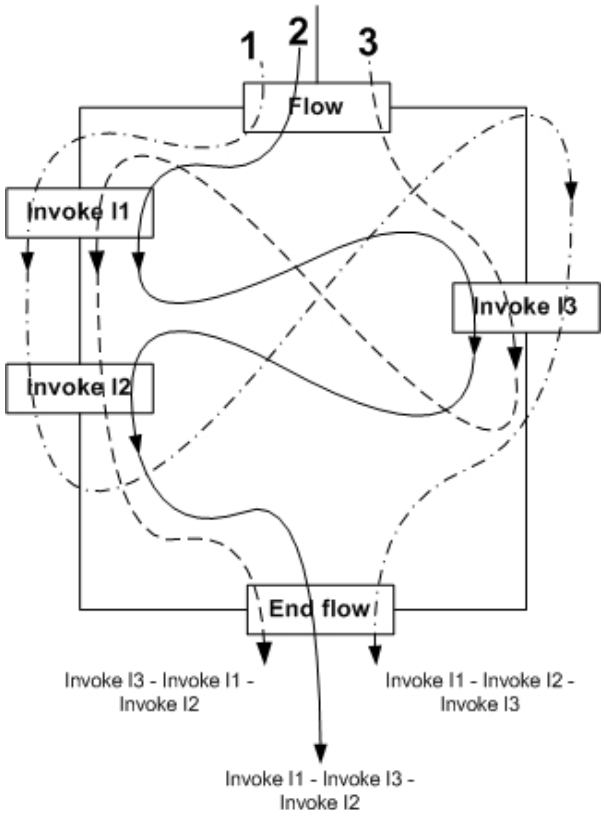

Figure 19 – all possible executions of parallel executed sequences of activities in a <flow> activity

It is obvious that every such flow should be calculated and enumerated in order to know all the different possibilities. Then, a derived FSA can be generated which accepts the protocol on every possible execution flow.

A well known operation which enumerates all the possible execution flows for a <flow> activity is the *shuffle product*. Generally, the shuffle product of two words W_1 and W_2 is obtained by inserting one word into the other word sparsely. This operation can be compared with shuffling two decks of cards. When a language L_1 and L_2 is defined for the words W_1 and W_2, the shuffle product of these languages is defined as the union of all the shuffle products of two words W'_1 and W'_2, taken from L_1 and L_2. When each language L_1 and L_2 is represented by the finite-state automata $M_1 = <S_1, S_{01}, E_1, \Sigma_1, \delta_1>$ and $M_2 = <S_2, S_{02}, E_2, \Sigma_2, \delta_2>$, the result of the shuffle product is again a finite-state automaton $M_x = <S, S_0, E, \Sigma, \delta>$ which is called the *shuffled machine*, where:

- $S = S_1 \times S_2$
- $S_0 = (S_{01}, S_{02})$
- $E = E_1 \times E_2$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $\delta : (S_1 \times S_2) \times \Sigma \rightarrow 2^{(S_1 \times S_2)}$ with:

$$\delta((s_1, s_2), \sigma) = \begin{cases} (\delta_1(s_1, \sigma), \delta_2(s_2, \sigma)) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2 \\ (\delta_1(s_1, \sigma), s_2) & \text{if } \sigma \in \Sigma_1 \setminus \Sigma_2 \\ (s_1, \delta_2(s_2, \sigma)) & \text{if } \sigma \in \Sigma_2 \setminus \Sigma_1 \end{cases}$$

Figure 20 illustrates how the shuffled machine looks like for the <flow> activity of figure 19. Each sequence of the <flow> can be mapped to a particular FSA. The shuffled machine which is the result of the shuffled product of the sequences of the <flow> activity accepts all the possible execution flows and preserves the order of the activities in a sequence.

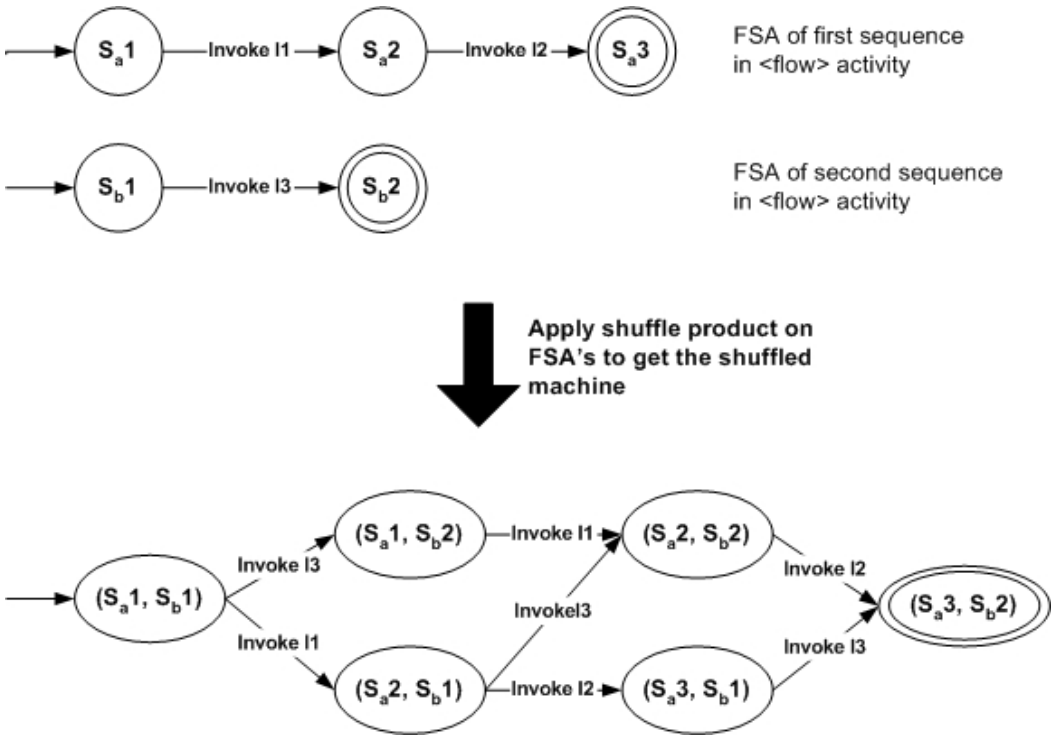

Figure 20 – the shuffled machine accepts every possible combination of two or more parallel executed sequences of activities

The following section explains how the shuffle product can be applied and implemented in the weaving of a protocol.

2.3.2 Coping with Parallelism in the Protocol Specification

For the implementation of the shuffle product in the weaver in order to handle the non-deterministic behaviour of the <flow> activity, an important choice has to be made concerning the protocol specification language. Earlier, it is stated that a developer who writes a certain business protocol does not always know if that protocol spans a <flow> activity. In other words, he might define a specific order of events as a protocol, supposing that these events are also defined in this order in the process. For example, if a protocol is defined as a concatenation of two events A and B, then he rather assumes that activity B is defined after activity A in the process flow so that activity A is always executed after activity B. When both activities are defined in a <flow> activity, then it is possible that activity B is executed before activity A due to some external events. Then, the developer expects that the protocol should also be accepted because it seems that the order of execution for both activities is not important and might change by external events. There are two possibilities available to solve this issue.

One possibility is to handle this issue in a transparent manner for the developer. This implies that the semantics of the `concat`, `choice` and `loop` predicates have to be changed so that every execution possibility of activities defined in a <flow> activity is

accepted. For example, suppose following definition of a protocol:

```
sequence(concat([getInformation, bookHotel, bookFlight]))
```

If ‘bookHotel’ and ‘bookFlight’ are activities which might be executed concurrently, then this protocol definition should also accept, for example, the execution of ‘getInformation’ followed by ‘bookFlight’ and finally ‘bookHotel’.
This illustrates that the `concat` predicate does not define just a concatenation of a set of activities, but also implicitly accepts every possible execution flow if the activities are executed concurrently. This means that the semantics of a concatenation are changed here and probably might cause some confusion. Therefore, it should be avoided as much as possible.

The second possibility, which does not require a change of the semantics of the predicates, is to let the developer himself decide whether or not the order of a part of the protocol is important. This means that it is explicitly indicated that the execution order may change. Taking back the previous example, the developer might decide that the order of the ‘bookHotel’ and ‘bookFlight’ activities does not influence a successful execution of the protocol. So if the ‘bookHotel’ and ‘bookFlight’ activities are defined in a <flow> activity and ‘bookFlight’ is executed before ‘bookHotel’, then it lays in the hands of the developer to make sure the protocol accepts this particular execution.

Although both proposals have their advantages, it is opted to implement the second one in the Padus weaver. To let the developer decide if the order is important in a particular protocol has some advantages. First of all, it might happen that the activities defined in the protocol must be executed in that specific order, regardless whether they could reside in a <flow> activity. This is impossible with the first proposal because all other possible execution flows are calculated and accepted implicitly. Another reason already mentioned is about changing the semantics. A disadvantage is that the developer himself has to take possible parallel executed activities now into consideration while defining the protocol.

Of course it is very unrealistic and not done to enumerate every possible process execution in the protocol specification just for making sure that all these executions are accepted. It is far better to introduce a new predicate, next to `concat`, `choice` and `loop` to indicate in the protocol that the order of certain activities may change. This predicate is called the `parallel` predicate and in the following section, it is explained how it works.

2.3.3 The new *parallel* Predicate

To describe sequential executions of parallel processes, the *shuffle* operation [56] is very useful. When the shuffle operation is added to the well-known regular operations, a new class of finite language is obtained which is called the *shuffle language*. The shuffle operation implements in fact the shuffle product which is explained earlier in this chapter. The shuffle operation is formally defined as follows:

Let Σ be an alphabet and ε the empty word, then the shuffle operation is defined inductively:

$u \otimes \varepsilon = \varepsilon \otimes u = \{u\}$, *for* $u \in \Sigma^*$, and

$au \otimes bv = a(u \otimes bv) \cup b(au \otimes v)$, *for* $u, v \in \Sigma^*$ *and* $a, b \in \Sigma$

The shuffle operator for between two languages L_1 and L_2 is defined as:

$$L_1 \otimes L_2 = \bigcup_{u \in L_1, v \in L_2} u \otimes v$$

Describing the parallel behaviour of a BPEL process in a protocol definition requires that this *shuffle* operator is available in the protocol specification language. Therefore, the `parallel` predicate is added to make this operator available in the language. The `parallel` predicate accepts a list of regular expressions and applies the shuffle product on it. Codefragment 30 below shows how this predicate is used in the protocol specification.

```
<pointcut name="parallelProtocol(Jp1, Jp2, Jp3)"
          pointcut="sequence(concat([getInformation(Jp1),
                                     parallel([bookHotel(Jp2),
                                               bookFlight(Jp3)])]))"/>
```

Codefragment 30 – example usage of the parallel predicate

The protocol specified in this example means that the pointcuts ‘bookHotel’ and ‘bookFlight’ can be executed in parallel and hence the order of execution of the related activities might change.

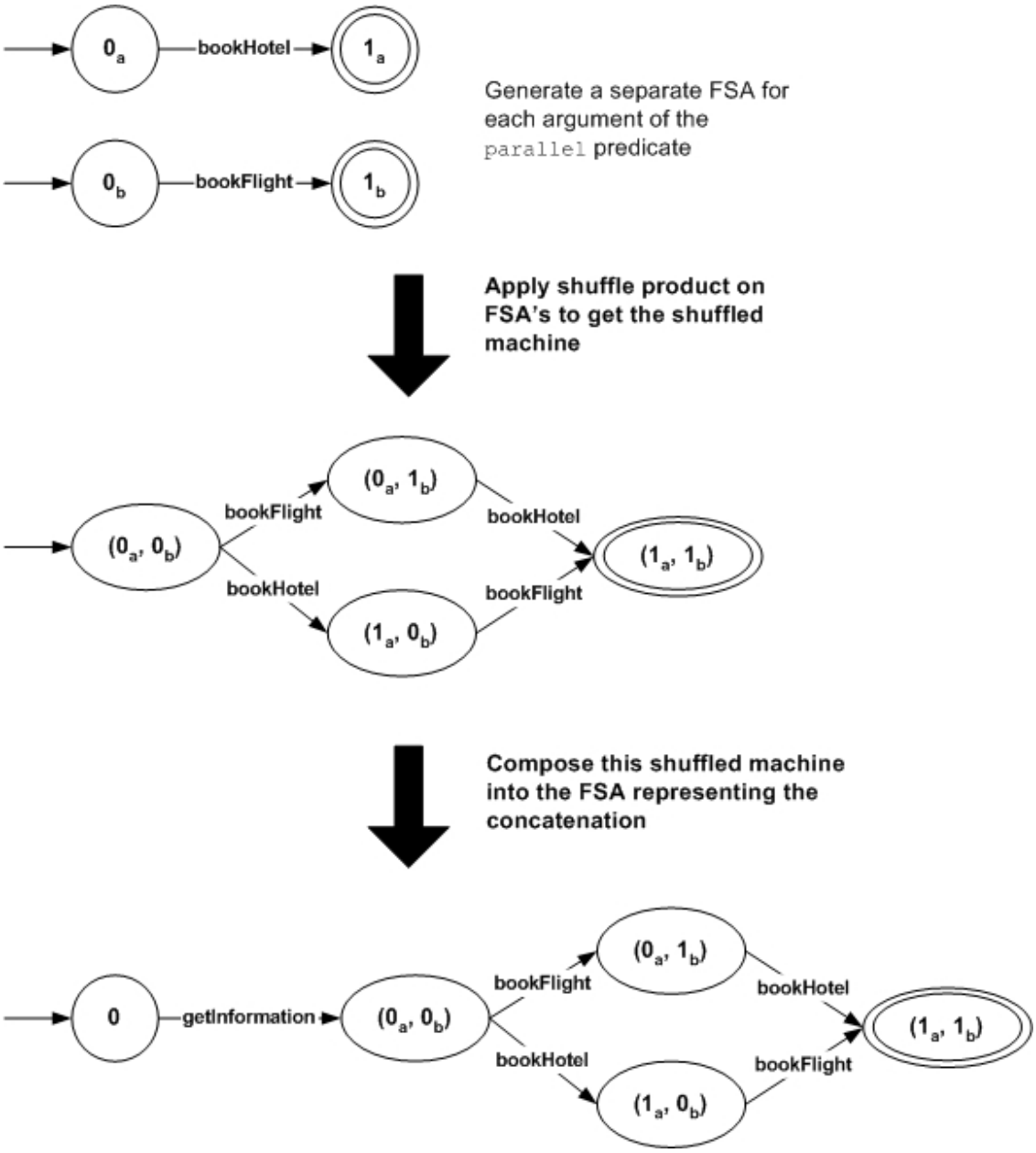

Figure 21 – The finite-state machine representing the protocol of codefragment 30

When generating the finite-state automaton from the protocol definition, the `parallel` predicate is processed as follows (figure 21):

1 For each argument of the `parallel` predicate a separate finite-state automaton is generated.

2 The shuffled machine is created from these separate automata. How such a shuffled machine is created is described earlier in section 2.3.1 of this chapter.

3 The shuffled machine is then composed with the other generated finite-state machines for the `concat`, `choice` and `loop` predicates following the rules of figure 17 in order to obtain the complete deterministic finite-state machine.

The FSA obtained after step 3 is completed is then used to weave the protocol into the process. During the weaving process, i.e. when the protocol trace advices are inserted, nothing about parallelism whatsoever has to be taken into account anymore and the protocol is woven just like any other protocol.

The advantages of using a separate predicate to explicitly indicate parallel executions of parts of the protocol are clear. When the process flow must obey the protocol exactly, then the ordinary predicates can be used.

2.3.4 Alternative solutions

Other techniques are available to model concurrency in a process. Petrinets [54] for example is a graphical language suited for modelling concurrency. While this language inherently supports concurrency, it is not used in this solution because there is no textual notation available. Although Petrinets are the state of the art for modelling concurrency, it is of little use for describing protocols in aspects. Two other solutions for modelling concurrency are described in [45][32].

In [45], it is explained how a finite-state automaton is generated from an extended form of regular expressions which also supports the shuffle operator. Although, this is a promising solution, the generation of the FSA is quite complex and the traversal is much different from a normal FSA, which creates more run-time overhead during the execution of the process.

The solution in [32] explains how the Parallel FSA is generated from a Parallel Regular Expression. This paper literally states that the parallel FSA is in fact no more than a combination of two normal FSAs and offers no more power. It is not opted to use this solution in the dissertation because a complete different algorithm for generating the Parallel FSA is needed and this only makes the weaver more complex. Instead, the solution described in this dissertation to handle parallelism reuses the algorithm for generating ordinary FSAs, and only a small modification is needed to handle the `parallel` predicate correctly.

Chapter 7: Conclusion

This chapter concludes the dissertation by summarizing its statement about the lack of support for implementing protocols in workflow languages. The contribution of this thesis is evaluated and some future work is proposed.

1 Summary of the Dissertation

In the beginning of this dissertation, an example workflow language, BPEL, is thoroughly explained. BPEL, and workflow languages in general, provides a means for composing business processes by combining several services. Currently, workflow languages provide many advantages for businesses to implement their processes. Business rules are undoubtedly one of the most important parts in the implementation of business processes. It is explained that these business rules crosscut the whole implementation of business processes and that they make the maintenance and evolvability much more complex.

Furthermore, aspect-oriented programming (AOP) is described as a recent technology which provides a solution to crosscutting concerns such as business rules. The integration of aspect-oriented programming with workflow languages is therefore interesting and two basic implementations, AO4BPEL and Padus, were explained further in the dissertation.
Further, it is illustrates that business protocols are also highly crosscutting throughout the implementation of business processes. Because some behaviour should only be executed on a successful execution of the protocol, the history of the process should be tracked. This tracking of the process history is scattered through the business logic and makes it a crosscutting concern. Unfortunately, by using basic AOP techniques, the crosscutting nature of business protocols remain and to solve this issue, it is stated that there is a need for better aspect-oriented support in workflow languages. Padus, which is an AOP language extension for BPEL based on logic programming is therefore further extended with so called stateful aspects. With stateful aspects, a protocol is described as a pattern of events which must be executed in a particular order. Stateful aspects are well implemented in a few aspect-oriented extensions for object-oriented languages, but not in workflow languages at all.

The extension of Padus involves the design of a sub-language in order to describe the business protocols declaratively as a regular expression. The extension of the underlying weaver for representing a protocol as a finite-state machine and for weaving the protocol in the BPEL process is described. By translating the protocol to a finite-state automaton, the history of the process can be traced easily by weaving pieces of instrumental code in the process itself which follow the state-changes of the FSA.
BPEL is able to execute parts of a process in parallel and therefore, the execution of the process is not deterministic anymore. This means that, for two execution flows of the process, it is not certain that the protocol will be matched, and this makes parallelism a big issue in this implementation of stateful aspects. This issue is handled also by providing a new `parallel` predicate in the protocol specification language which applies the shuffle product on the arguments which might constitute the parallel executed parts of the process. From this shuffle product, a new finite-state machine is generated which accepts every possible valid flow of the parallel executed sequences of activities. This finite-state automaton is then combined with the other FSAs to obtain

the final finite-state machine. Then, it is only a matter of weaving this new automaton into the process in order to get the protocol matching right, even with parallelisation taken into account. A new predicate is added next to the `concat`, `choice` and `loop` predicates to express these parallel automatons.

2 Contribution to Current Research

This dissertation contributes to current research work in aspect-oriented programming for workflow languages by providing a proof-of-concept implementation of stateful aspects in Padus. Three main elements were subject of this research:

1. A new language is designed to describe protocols in a declarative manner by defining a regular expression. This language is an extension to the pointcut language of Padus and preserves its logic nature by providing predicates for each operation of the regular expression.

2. In this dissertation, a complete weaving strategy is elaborated in order to weave a protocol and track the execution history of a process. This weaving strategy is based on finite-state automatons which are able to handle parallelisation.

3. Finally, Padus is extended with a proof-of-concept implementation of stateful aspects. Protocols might now be described and implemented as a separate concern and thus improves the maintainability of business software.

3 Evaluation and Future Work

This proof-of-concept implementation of stateful aspects in Padus might be a first step for further research in this particular domain of aspect-oriented programming and workflow languages. The extension of Padus with stateful aspects is evaluated in this section and some extra future work is also proposed.

About the performance of the stateful aspect extension in Padus, it can be easily stated that the protocol weaving is not as optimal as possible. Both the performance of the weaving process and the run-time performance are discussed. First, we take a closer look at the performance of the weaving process itself.
By generating a deterministic finite-state automaton from the protocol specification, many efforts are made to make the weaving process as efficient as possible. With a deterministic FSA, no backtracking is needed during the weaving of the protocol in the process which makes it more performant. However, the generation of a shuffled machine is time consuming because each argument requires the generation of a separate deterministic FSA from which the shuffled machine is generated afterwards. It might also happen that some instrumental advices are added to the process which will never be executed. This can be optimized by performing a graph-analysis on the protocol to exclude the weaving of unreachable and unexecutable protocol trace advices in the process.
The fact that Padus is implemented with a static weaver creates also some run-time performance deficiencies. During weave-time, many instrumental pieces of code are injected into the process which generates some run-time overhead. Even when it is impossible that the protocol is matched during execution, these protocol trace advices are still executed. With a dynamic weaver, the trace advices would only be inserted when needed and removed when they become needless.

Regular expressions are used for the specification of protocols because they are rather easy to understand, but they restrict the way how protocols can be described. For example, with regular expressions it is not possible to describe recursive protocols. This issue could be solved by using a context-free language instead of a regular language.
Because this implementation of stateful aspects is a proof-of-concept, some improvements and features could be further worked out. For example, the pointcut language could be extended further for defining strict protocols which does not accept intervening activities. Advice could also be defined which should be executed on the complement of the protocol. The advice is only executed if the protocol is not matched. These two features are available in the stateful aspect extension of JAsCo and have been proven to be very useful.

Appendix A: WSDL File of a Billing Webservice

```
<?xml version='1.0' encoding='utf-8' ?>
<definitions name='service.billing.BillingService'
  targetNamespace='http://systinet.com/wsdl/service/billing/'
  xmlns:tns='http://systinet.com/wsdl/service/billing/'
  xmlns:soap12='http://schemas.xmlsoap.org/wsdl/soap12/'
  xmlns:map='http://systinet.com/mapping/'
  xmlns:soap='http://schemas.xmlsoap.org/wsdl/soap/'
  xmlns='http://schemas.xmlsoap.org/wsdl/'>

<types>
  <xsd:schema elementFormDefault="qualified"
    targetNamespace="http://systinet.com/wsdl/service/billing/"
    xmlns:tns="http://systinet.com/wsdl/service/billing/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="calculatePrice">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="p0" type="xsd:float"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="calculatePriceResponse">
      <xsd:complexType>
        <xsd:sequence/>
       </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</types>

<message name='BillingService_calculatePrice_Request_Soap'>
  <part name='parameters' element='tns:calculatePrice'/>
</message>
<message name='BillingService_calculatePrice_Response_Soap'>
  <part name='parameters' element='tns:calculatePriceResponse'/>
</message>

<portType name='BillingService'>
  <operation name='calculatePrice'>
    <input message='tns:BillingService_calculatePrice_Request_Soap'/>
    <output message='tns:BillingService_calculatePrice_Response_Soap'/>
  </operation>
</portType>

<binding name='BillingService' type='tns:BillingService'>
  <soap:binding transport='http://schemas.xmlsoap.org/soap/http'
                style='document'/>
  <operation name='calculatePrice'>
    <map:java-operation name='calculatePrice' signature='KEYpVg=='
             unwrapped='true' wsiAttachments='true' parameterOrder='p0'/>
    <soap:operation soapAction=
       'http://systinet.com/wsdl/service/billing/BillingService#calculatePrice?KEYpVg=='
       style='document'/>
    <input>
     <soap:body parts='parameters' use='literal'/>
    </input>
    <output>
     <soap:body parts='parameters' use='literal'/>
    </output>
  </operation>
</binding>
```

```
<binding name='BillingService_SOAP12' type='tns:BillingService'>
  <soap12:binding transport='http://schemas.xmlsoap.org/soap/http'
                  style='document'/>
  <operation name='calculatePrice'>
    <map:java-operation name='calculatePrice' signature='KEYpVg=='
              unwrapped='true' wsiAttachments='true' parameterOrder='p0'/>
    <soap12:operation soapAction=
       'http://systinet.com/wsdl/service/billing/BillingService#calculatePrice?KEYpVg=='
        style='document'/>
    <input>
      <soap12:body parts='parameters' use='literal'/>
    </input>
    <output>
      <soap12:body parts='parameters' use='literal'/>
    </output>
  </operation>
</binding>

<service name='BillingService'>
  <port name='BillingService' binding='tns:BillingService'>
    <soap:address location='http://localhost:6060/billing'/>
  </port>
  <port name='BillingService_SOAP12' binding='tns:BillingService_SOAP12'>
    <soap12:address location='http://localhost:6060/billing'/>
  </port>
</service>
</definitions>
```

Appendix B1: A BPEL Process for Booking a Holiday

The codelisting that follows illustrates a realistic BPEL process for booking a holiday. A deployed BPEL process becomes a service with its own WSDL description which can be found in appendix B2.

The service supports two operations: `getBookingInformation` and `book`. The `getBookingInformation` operation gives information about flights and hotels to a particular city and country. Some information about the current weather is also given through an external webservice. A holiday can be booked through the `book` operation for which a particular flightnumber and hotel is given as input. When the booking is completed successfully, a webservice from Amazon.com is invoked to provide the holiday-maker with a list of books concerning the destination its holiday.

```
<?xml version="1.0" encoding="utf-8" ?>
<process name="booking"
  targetNamespace="http://localhost/booking"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:lang="http://systinet.com/wsdl/java/lang/"
  xmlns:flight="http://systinet.com/wsdl/service/flight/"
  xmlns:hotel="http://systinet.com/wsdl/service/hotel/"
  xmlns:weather="http://www.webserviceX.NET"
  xmlns:amazon="http://webservices.amazon.com/AWSECommerceService/2006-09-18"
  xmlns:tns="http://localhost/booking"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.xmlsoap.org/ws/2003/03/business-process/
http://schemas.xmlsoap.org/ws/2003/03/business-process/">

  <partnerLinks>
   <partnerLink name="clientPL" partnerLinkType="tns:clientPLT" myRole="bookingservice" />
   <partnerLink name="flightPL" partnerLinkType="tns:flightPLT" partnerRole="flightservice"/>
   <partnerLink name="hotelPL" partnerLinkType="tns:hotelPLT" partnerRole="hotelservice" />
   <partnerLink name="weatherPL" partnerLinkType="tns:weatherPLT"
                partnerRole="weatherservice" />
   <partnerLink name="amazonPL" partnerLinkType="tns:amazonPLT" partnerRole="amazonservice" />
  </partnerLinks>

  <variables>
   <variable name="bookingInformationRequest" messageType="tns:information_InputMessage" />
   <variable name="bookingInformationResponse" messageType="tns:information_OutputMessage" />
   <variable name="bookingRequest" messageType="tns:booking_InputMessage" />
   <variable name="bookingResponse" messageType="tns:booking_OutputMessage" />
   <variable name="flightInformationRequest"
             messageType="flight:FlightService_getFlights_Request_Soap" />
   <variable name="flightInformationResponse"
             messageType="flight:FlightService_getFlights_Response_Soap" />
   <variable name="flightBookingRequest"
             messageType="flight:FlightService_bookFlight_Request_Soap" />
   <variable name="flightBookingResponse"
             messageType="flight:FlightService_bookFlight_Response_Soap" />
   <variable name="hotelInformationRequest"
             messageType="hotel:HotelService_getAvailableHotels_Request_Soap" />
   <variable name="hotelInformationResponse"
             messageType="hotel:HotelService_getAvailableHotels_Response_Soap" />
   <variable name="hotelBookingRequest"
             messageType="hotel:HotelService_bookHotel_Request_Soap" />
   <variable name="hotelBookingResponse"
             messageType="hotel:HotelService_bookHotel_Response_Soap" />
   <variable name="hotelRequest" messageType="hotel:HotelService_getHotel_Request_Soap" />
   <variable name="hotelResponse" messageType="hotel:HotelService_getHotel_Response_Soap" />
   <variable name="weatherInformationRequest" messageType="weather:GetWeatherSoapIn" />
   <variable name="weatherInformationResponse" messageType="weather:GetWeatherSoapOut" />
   <variable name="amazonSearchRequest" messageType="amazon:ItemSearchRequestMsg" />
   <variable name="amazonSearchResponse" messageType="amazon:ItemSearchResponseMsg" />

   <!-- help variables -->
   <variable name="i" type="xsd:integer" />
   <variable name="arrayLength" type="xsd:integer" />
   <variable name="xpathQuery" type="xsd:string" />
  </variables>
```

```
  <sequence name="main" >
   <pick createInstance="yes">
   <!--================================================================
       REQUEST-BOOKING-INFORMATION
       ================================================================ -->
    <onMessage partnerLink="clientPL" portType="tns:clientPT"
               operation="getBookingInformation" variable="bookingInformationRequest">

     <sequence>
      <flow name="getBookingInformationFlow">
       <sequence name="getFlightInformation">
        <assign>
         <copy>
          <from expression="'Belgium'" />
          <to variable="flightInformationRequest" part="parameters"
                      query="/flight:getFlights/flight:p0" />
         </copy>
         <copy>
          <from variable="bookingInformationRequest" part="city" />
          <to variable="flightInformationRequest" part="parameters"
                      query="/flight:getFlights/flight:p1" />
         </copy>
         <copy>
          <from variable="bookingInformationRequest" part="country" />
          <to variable="flightInformationRequest" part="parameters"
                      query="/flight:getFlights/flight:p2" />
         </copy>
        </assign>
        <invoke name="requestFlightInformation"
                partnerLink="flightPL" portType="flight:FlightService"
                operation="getFlights"
                inputVariable="flightInformationRequest"
                outputVariable="flightInformationResponse" />
        <!-- prepare loop -->
         <assign>
          <copy>
           <from expression="count(
                     bpws:getVariableData('flightInformationResponse', 'parameters',
                      '/flight:getFlightsResponse/flight:response')/flight:Flight)" />
           <to variable="arrayLength" />
          </copy>
          <copy>
           <from expression="1" />
           <to variable="i" />
          </copy>
         </assign>

         <!-- loop over flights and create list of flights -->
          <while condition="bpws:getVariableData('i') &lt;=
                                bpws:getVariableData('arrayLength')">
           <sequence>

            <!-- prepare index-part of the 'xpath query' to index the Flight-array -->
            <assign>
             <copy>
              <from expression=
                         "concat('/flight:getFlightsResponse/flight:response/flight:Flight[',
                                 bpws:getVariableData('i'),
                                 ']')" />
              <to variable="xpathQuery" />
             </copy>
            </assign>

            <!-- put flights in flights-part of bookingInformationResponse variable -->
            <assign>
             <copy>
              <from expression=
                         "bpws:getVariableData('flightInformationResponse','parameters',
                        concat(bpws:getVariableData('xpathQuery'),'/flight:_flightNumber'))"/>
              <to variable="bookingInformationResponse" part="flights"query="
/tns:flightInformation/tns:contents/tns:flightEntry[bpws:getVariableData('i')]/flightNumber"/>
             </copy>
             <copy>
              <from expression="bpws:getVariableData('flightInformationResponse',
'parameters', concat(bpws:getVariableData('xpathQuery'), '/flight:_departureDate'))"/>
              <to variable="bookingInformationResponse" part="flights" query="
/tns:flightInformation/tns:contents/tns:flightEntry[bpws:getVariableData('i')]/departureDate"/
```

```
>
             </copy>
             <copy>
              <from expression="bpws:getVariableData('flightInformationResponse',
'parameters', concat(bpws:getVariableData('xpathQuery'), '/flight:_sourceCity'))" />
              <to variable="bookingInformationResponse" part="flights" query="
/tns:flightInformation/tns:contents/tns:flightEntry[bpws:getVariableData('i')]/sourceCity" />
             </copy>
             <copy>
              <from expression="bpws:getVariableData('flightInformationResponse',
'parameters', concat(bpws:getVariableData('xpathQuery'),'/flight:_destinationCity'))"/>
              <to variable="bookingInformationResponse" part="flights" query="
/tns:flightInformation/tns:contents/tns:flightEntry[bpws:getVariableData('i')]/destinationCity
"/>
             </copy>
             <copy>
              <from expression="bpws:getVariableData('flightInformationResponse',
'parameters', concat(bpws:getVariableData('xpathQuery'), '/flight:_price'))" />
              <to variable="bookingInformationResponse" part="flights"query="
/tns:flightInformation/tns:contents/tns:flightEntry[bpws:getVariableData('i')]/price" />
              </copy>
            </assign>

            <!-- update i -->
            <assign>
             <copy>
              <from expression="bpws:getVariableData('i') + 1" />
              <to variable="i" />
             </copy>
            </assign>
           </sequence>
          </while>
         </sequence>

         <sequence name="getHotelInformation">
          <assign>
           <copy>
            <from variable="bookingInformationRequest" part="city" />
            <to variable="hotelInformationRequest" part="parameters"
                        query="/hotel:getAvailableHotels/hotel:p0" />
           </copy>
          </assign>

          <invoke name="requestHotelInformation"
                  partnerLink="hotelPL" portType="hotel:HotelService"
                  operation="getAvailableHotels"
                  inputVariable="hotelInformationRequest"
                  outputVariable="hotelInformationResponse" />

          <!-- prepare loop -->
          <assign>
           <copy>
            <from expression="count(
                    bpws:getVariableData('hotelInformationResponse', 'parameters',
                    '/hotel:getAvailableHotelsResponse/hotel:response')/hotel:Hotel)" />
            <to variable="arrayLength" />
           </copy>
           <copy>
            <from expression="1" />
            <to variable="i" />
           </copy>
          </assign>

          <!-- loop over hotels -->
          <while condition=
                     "bpws:getVariableData('i') &lt;= bpws:getVariableData('arrayLength')">
           <sequence>

            <!-- prepare index-part of the 'xpath query' to index the Hotel-array -->
            <assign>
             <copy>
              <from expression=
                     "concat('/hotel:getAvailableHotelsResponse/hotel:response/hotel:Hotel[',
                             bpws:getVariableData('i'),
                             ']')" />
              <to variable="xpathQuery" />
             </copy>
```

```
            </assign>

            <!-- put hotels in hotels-part of bookingInformationResponse variable -->
            <assign>
             <copy>
              <from expression="bpws:getVariableData('hotelInformationResponse',
'parameters', concat(bpws:getVariableData('xpathQuery'), '/hotel:_name'))" />
              <to variable="bookingInformationResponse" part="hotels" query="
/tns:hotelInformation/tns:contents/tns:hotelEntry[bpws:getVariableData('i')]/hotelName" />
             </copy>
             <copy>
              <from expression="bpws:getVariableData('hotelInformationResponse',
'parameters', concat(bpws:getVariableData('xpathQuery'), '/hotel:_city'))" />
              <to variable="bookingInformationResponse" part="hotels" query="
/tns:hotelInformation/tns:contents/tns:hotelEntry[bpws:getVariableData('i')]/city" />
             </copy>
             <copy>
              <from expression="bpws:getVariableData('hotelInformationResponse',
'parameters', concat(bpws:getVariableData('xpathQuery'), '/hotel:_price'))" />
              <to variable="bookingInformationResponse" part="hotels" query="
/tns:hotelInformation/tns:contents/tns:hotelEntry[bpws:getVariableData('i')]/price" />
             </copy>
             <copy>
              <from expression="bpws:getVariableData('hotelInformationResponse',
'parameters', concat(bpws:getVariableData('xpathQuery'), '/hotel:_rating'))" />
              <to variable="bookingInformationResponse" part="hotels" query="
/tns:hotelInformation/tns:contents/tns:hotelEntry[bpws:getVariableData('i')]/rating" />
             </copy>
            </assign>

            <!-- update i -->
            <assign>
             <copy>
              <from expression="bpws:getVariableData('i') + 1" />
              <to variable="i" />
             </copy>
            </assign>
           </sequence>
          </while>
         </sequence>

         <sequence name="getWeatherInformation">
         <!-- prepare weatherinformation request -->
         <assign>
          <copy>
           <from variable="bookingInformationRequest" part="city" />
           <to variable="weatherInformationRequest" part="parameters"
               query="/weather:GetWeather/weather:CityName" />
          </copy>
          <copy>
           <from variable="bookingInformationRequest" part="country" />
           <to variable="weatherInformationRequest" part="parameters"
                 query="/weather:GetWeather/weather:CountryName" />
          </copy>
          </assign>

          <invoke name="requestWeatherInformation"
                 partnerLink="weatherPL" portType="weather:GlobalWeatherSoap"
                 operation="GetWeather" inputVariable="weatherInformationRequest"
                 outputVariable="weatherInformationResponse" />

          <!-- put weather in response -->
           <assign>
            <copy>
             <from variable="weatherInformationResponse" part="parameters"
                   query="/weather:GetWeatherResponse/weather:GetWeatherResult" />
             <to variable="bookingInformationResponse" part="weather" />
            </copy>
           </assign>
          </sequence>
         </flow>

         <reply name="returnBookingInformation" partnerLink="clientPL"
             portType="tns:clientPT" operation="getBookingInformation"
             variable="bookingInformationResponse" />
       </sequence>
      </onMessage>
```

```
<!-- ================================================================================
          BOOK
================================================================================ -->
      <onMessage  partnerLink="clientPL" portType="tns:clientPT"
            operation="book" variable="bookingRequest">
        <sequence>
          <!-- book flight and hotel -->
          <flow name="bookFlow">
            <sequence name="bookFlightSequence">
              <assign>
                <copy>
                  <from variable="bookingRequest" part="flight" />
                  <to variable="flightBookingRequest" part="parameters"
                    query="/flight:bookFlight/flight:p0" />
                </copy>
              </assign>

              <invoke name="bookFlight" partnerLink="flightPL"
                      portType="flight:FlightService" operation="bookFlight"
                      inputVariable="flightBookingRequest"
                      outputVariable="flightBookingResponse" />
            </sequence>

            <sequence name="bookHotelSequence">
              <assign>
                <copy>
                  <from variable="bookingRequest" part="hotel" />
                  <to variable="hotelBookingRequest" part="parameters"
                      query="/hotel:bookHotel/hotel:p0" />
                </copy>
              </assign>

              <invoke name="bookHotel" partnerLink="hotelPL"
                  portType="hotel:HotelService" operation="bookHotel"
                  inputVariable="hotelBookingRequest"
                  outputVariable="hotelBookingResponse" />
            </sequence>
          </flow>

          <!-- Check if flight is booked (client can only see/book available hotels)
            and generate appropriate response -->
          <switch>
            <!-- flight is booked -->
            <case condition="bpws:getVariableData('flightBookingResponse', 'parameters',
                                '/flight:bookFlightResponse/flight:response') = 1
                    and
                                bpws:getVariableData('hotelBookingResponse', 'parameters',
                                   '/hotel:bookHotelResponse/hotel:response') = 0 ">
              <assign>
                <copy>
                  <from expression="'Flight is booked, hotel is NOT booked'" />
                  <to variable="bookingResponse" part="result" />
                </copy>
              </assign>
            </case>

            <!-- flight is not booked -->
            <case condition="bpws:getVariableData('flightBookingResponse', 'parameters',
                                '/flight:bookFlightResponse/flight:response') = 0
                    and
                                bpws:getVariableData('hotelBookingResponse', 'parameters',
                                    '/hotel:bookHotelResponse/hotel:response') = 1 ">
              <assign>
                <copy>
                  <from expression="'Flight is NOT booked, hotel is booked'" />
                  <to variable="bookingResponse" part="result" />
                </copy>
              </assign>
            </case>
```

```
<!-- flight is not booked, hotel is not booked -->
<case condition="bpws:getVariableData('flightBookingResponse', 'parameters',
                    '/flight:bookFlightResponse/flight:response') = 0
        and
                bpws:getVariableData('hotelBookingResponse', 'parameters',
                    '/hotel:bookHotelResponse/hotel:response') = 0 ">
  <assign>
    <copy>
      <from expression="'Flight is NOT booked, hotel is NOT booked'" />
      <to variable="bookingResponse" part="result" />
    </copy>
  </assign>
</case>

<!-- everything went OK, search amazon for books about the city -->
<otherwise>
  <sequence>
    <assign>
      <copy>
        <from expression="'Flight is booked, hotel is booked'" />
        <to variable="bookingResponse" part="result" />
      </copy>
    </assign>

    <!-- get booked hotel, so we know the city -->
    <assign>
      <copy>
        <from variable="bookingRequest" part="hotel" />
        <to variable="hotelRequest" part="parameters"
          query="/hotel:getHotel/hotel:p0" />
      </copy>
    </assign>

    <invoke  name="retrieveHotel" partnerLink="hotelPL"
        portType="hotel:HotelService" operation="getHotel"
        inputVariable="hotelRequest" outputVariable="hotelResponse" />

    <!-- DO AMAZON SEARCH -->
    <!-- prepare amazon request variable -->
    <assign>
      <copy>
        <from expression="'***'" />
        <to variable="amazonSearchRequest" part="body"
            query="/amazon:ItemSearch/amazon:AWSAccessKeyId" />
      </copy>
      <copy>
        <from expression="'Books'" />
        <to variable="amazonSearchRequest" part="body"
            query="/amazon:ItemSearch/amazon:Request/amazon:SearchIndex" />
      </copy>
      <copy>
        <from expression="'relevancerank'" />
        <to variable="amazonSearchRequest" part="body"
            query="/amazon:ItemSearch/amazon:Request/amazon:Sort" />
      </copy>
      <copy>
        <from variable="hotelResponse" part="parameters"
            query="/hotel:getHotelResponse/hotel:response/hotel:_city" />
        <to variable="amazonSearchRequest" part="body"
            query="/amazon:ItemSearch/amazon:Request/amazon:Keywords" />
      </copy>
    </assign>

    <invoke  name="searchAmazon" partnerLink="amazonPL"
        portType="amazon:AWSECommerceServicePortType" operation="ItemSearch"
        inputVariable="amazonSearchRequest"
        outputVariable="amazonSearchResponse" />
```

```
                <!-- loop over the results (first page (10 results)) -->
                <!-- prepare loop variables -->
                <assign>
                  <copy>
                    <from expression="1" />
                    <to variable="i" />
                  </copy>
                  <copy>
                    <from expression="count(
                      bpws:getVariableData('amazonSearchResponse', 'body',
                            '/amazon:ItemSearchResponse/amazon:Items')/amazon:Item)" />
                    <to variable="arrayLength" />
                  </copy>
                </assign>

                <while condition="bpws:getVariableData('i') &lt;=
                                  bpws:getVariableData('arrayLength')">
                  <sequence>
                    <assign>
                      <copy>
                        <from expression=
                          concat('/amazon:ItemSearchResponse/amazon:Items/amazon:Item[',
                                 bpws:getVariableData('i'),
                                 ']')" />
                          <to variable="xpathQuery" />
                      </copy>
                    </assign>

                    <assign>
                      <copy>
                        <from expression="bpws:getVariableData('amazonSearchResponse',
'body', concat(bpws:getVariableData('xpathQuery'),
              '/amazon:ItemAttributes/amazon:Title'))" />
                        <to variable="bookingResponse" part="books" query="
/tns:bookInformation/tns:contents/tns:bookEntry[bpws:getVariableData('i')]/title" />
                      </copy>
                      <copy>
                        <from expression="bpws:getVariableData('amazonSearchResponse',
'body', concat(bpws:getVariableData('xpathQuery'),
              '/amazon:DetailPageURL'))" />
                        <to variable="bookingResponse" part="books" query="
/tns:bookInformation/tns:contents/tns:bookEntry[bpws:getVariableData('i')]/url" />
                      </copy>
                    </assign>

                    <assign>
                      <copy>
                        <from expression="bpws:getVariableData('i') + 1" />
                        <to variable="i" />
                      </copy>
                    </assign>
                  </sequence>
                </while>
              </sequence>
            </otherwise>
          </switch>

          <reply name="returnBooking" partnerLink="clientPL" portType="tns:clientPT"
                 operation="book" variable="bookingResponse" />

        </sequence>
      </onMessage>
    </pick>
  </sequence>
</process>
```

Appendix B2: The WSDL description of the BPEL booking-service

This is the WSDL description of the booking service, which is very similar to an ordinary WSDL description. The most important part here are the definitions of the partnerlinks at the bottom of the listing.

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions
  targetNamespace=http://localhost/booking
  xmlns=http://schemas.xmlsoap.org/wsdl/
  xmlns:plt=http://schemas.xmlsoap.org/ws/2003/05/partner-link/
  xmlns:tns=http://localhost/booking
  xmlns:xsd=http://www.w3.org/2001/XMLSchema
  xmlns:flight=http://systinet.com/wsdl/service/flight/
  xmlns:hotel=http://systinet.com/wsdl/service/hotel/
  xmlns:log=http://systinet.com/wsdl/service/log/
  xmlns:billing=http://systinet.com/wsdl/service/billing/
  xmlns:weather=http://www.webserviceX.NET
xmlns:amazon="http://webservices.amazon.com/AWSECommerceService/2006-09-18">

  <import namespace=http://systinet.com/wsdl/service/flight/
          location="FlightService.wsdl" />
  <import namespace=http://systinet.com/wsdl/service/hotel/
          location="HotelService.wsdl" />
  <import namespace=http://systinet.com/wsdl/service/log/
          location="LogService.wsdl" />
  <import namespace=http://systinet.com/wsdl/service/billing/
          location="BillingService.wsdl" />
  <import namespace=http://www.webserviceX.NET
          location="WeatherService.wsdl" />
  <import namespace=http://webservices.amazon.com/AWSECommerceService/2006-09-18
          location="AmazonService.wsdl" />

  <types>
   <xsd:schema elementFormDefault="qualified"
               targetNamespace=http://localhost/booking
               xmlns:tns="http://localhost/booking">
    <xsd:complexType name="FlightEntry">
     <xsd:sequence>
      <xsd:element name="flightNumber" nillable="true" type="xsd:string"/>
      <xsd:element name="departureDate" nillable="true" type="xsd:string"/>
      <xsd:element name="sourceCity" nillable="true" type="xsd:string"/>
      <xsd:element name="destinationCity" nillable="true" type="xsd:string"/>
      <xsd:element name="price" type="xsd:float"/>
     </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="HotelEntry">
     <xsd:sequence>
      <xsd:element name="hotelName" nillable="true" type="xsd:string" />
      <xsd:element name="city" nillable="true" type="xsd:string" />
      <xsd:element name="price" nillable="true" type="xsd:float" />
      <xsd:element name="rating" nillable="true" type="xsd:integer" />
     </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="BookEntry">
     <xsd:sequence>
      <xsd:element name="title" nillable="true" type="xsd:string" />
      <xsd:element name="url" nillable="true" type="xsd:string" />
     </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="ArrayOfFlightEntry">
     <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="0"
                   name="flightEntry" nillable="true" type="tns:FlightEntry"/>
     </xsd:sequence>
    </xsd:complexType>
```

```
  <xsd:complexType name="ArrayOfHotelEntry">
   <xsd:sequence>
    <xsd:element maxOccurs="unbounded" minOccurs="0"
                 name="hotelEntry" nillable="true" type="tns:HotelEntry" />
   </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="ArrayOfBookEntry">
   <xsd:sequence>
    <xsd:element maxOccurs="unbounded" minOccurs="0"
                 name="bookEntry" nillable="true" type="tns:BookEntry" />
   </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="flightInformation">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="contents" nillable="true" type="tns:ArrayOfFlightEntry"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
  <xsd:element name="hotelInformation">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="contents" nillable="true" type="tns:ArrayOfHotelEntry" />
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
  <xsd:element name="bookInformation">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="contents" nillable="true" type="tns:ArrayOfBookEntry" />
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 </xsd:schema>
</types>

<message name="information_InputMessage">
 <part name="city" type="xsd:string" />
 <part name="country" type="xsd:string" />
</message>
<message name="information_OutputMessage">
 <part name="flights" element="tns:flightInformation" />
 <part name="hotels" element="tns:hotelInformation" />
 <part name="weather" type="xsd:string" />
</message>
<message name="booking_InputMessage">
 <part name="flight" type="xsd:string" />
 <part name="hotel" type="xsd:string" />
</message>
<message name="booking_OutputMessage">
 <part name="result" type="xsd:string" />
 <part name="books" element="tns:bookInformation" />
</message>

<portType name="clientPT">
 <operation name="getBookingInformation">
  <input message="tns:information_InputMessage" />
  <output message="tns:information_OutputMessage" />
 </operation>
 <operation name="book">
  <input message="tns:booking_InputMessage" />
  <output message="tns:booking_OutputMessage" />
 </operation>
</portType>
```

```
  <plt:partnerLinkType name="clientPLT">
   <plt:role name="bookingservice">
    <plt:portType name="tns:clientPT" />
   </plt:role>
  </plt:partnerLinkType>

  <plt:partnerLinkType name="flightPLT">
   <plt:role name="flightservice">
    <plt:portType name="flight:FlightService"/>
   </plt:role>
  </plt:partnerLinkType>

  <plt:partnerLinkType name="hotelPLT">
   <plt:role name="hotelservice">
     <plt:portType name="hotel:HotelService"/>
    </plt:role>
  </plt:partnerLinkType>

  <plt:partnerLinkType name="weatherPLT">
   <plt:role name="weatherservice">
    <plt:portType name="weather:GlobalWeatherSoap"/>
   </plt:role>
  </plt:partnerLinkType>

  <plt:partnerLinkType name="amazonPLT">
   <plt:role name="amazonservice">
    <plt:portType name="amazon:AWSECommerceServicePortType"/>
   </plt:role>
  </plt:partnerLinkType>

  <plt:partnerLinkType name="logPLT">
   <plt:role name="loggingservice">
    <plt:portType name="log:LogService" />
   </plt:role>
  </plt:partnerLinkType>

  <plt:partnerLinkType name="billingPLT">
   <plt:role name="billingservice">
    <plt:portType name="billing:BillingService" />
   </plt:role>
  </plt:partnerLinkType>
</definitions>
```

Appendix C: A Protocol woven in a BPEL process

This appendix shows the resulting BPEL process with a protocol woven into it. The process and protocol are complete fictive in the sense that they are just used as example. A message is written in a log file when the protocol is matched with the process execution.

The BPEL process outputs a value depending on a particular cityname (Gent, Brussel, Antwerpen) given as input. A variable *i* is initialized to 0 when 'Gent' or 'Brussel' is given as input.

```
<?xml version="1.0" encoding="utf-8" ?>
<process name="test"
  targetNamespace="http://localhost/test"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:tns="http://localhost/test">

  <partnerLinks>
    <partnerLink name="clientPL" partnerLinkType="tns:clientPLT"
                 myRole="bookingservice" />
  </partnerLinks>

  <variables>
    <variable name="testInput" messageType="tns:test_InputMessage" />
    <variable name="testOutput" messageType="tns:test_OutputMessage" />
  </variables>

  <sequence name="main" >
    <receive createInstance="yes" partnerLink="clientPL" portType="tns:clientPT"
             operation="cityOperation" variable="testInput" />
    <switch>
      <case condition="bpws:getVariableData('testInput', 'city') = 'Brussel' or
                       bpws:getVariableData('testInput', 'city') = 'Gent'">
        <assign name="assign0">
          <copy>
            <from expression="0" />
            <to variable="i"/>
          </copy>
        </assign>
      </case>
    </switch>
    <switch>
      <case condition="bpws:getVariableData('testInput', 'city') = 'Brussel'">
        <assign name="assign1">
          <copy>
            <from expression="'BRUSSEL (0)'" />
            <to variable="testOutput" part="output" />
          </copy>
        </assign>
      </case>
      <case condition="bpws:getVariableData('testInput', 'city') = 'Gent'">
        <assign name="assign2">
          <copy>
            <from expression="'GENT (1)'" />
            <to variable="testOutput" part="output" />
          </copy>
        </assign>
      </case>
      <case condition="bpws:getVariableData('testInput', 'city') = 'Antwerpen'">
        <assign name="assign3">
          <copy>
            <from expression="'ANTWERPEN (5)'" />
            <to variable="testOutput" part="output" />
          </copy>
        </assign>
      </case>
    </switch>

    <reply name="return" partnerLink="clientPL"
      portType="tns:clientPT" operation="cityOperation"
      variable="testOutput" />
  </sequence>
</process>
```

The protocol is matched when the variable *i* is initialized and ‘Gent’ is given as input to the process, or when ‘Antwerpen’ is given as input.

```
<padus:aspect name="TestAspect">
  <padus:using>
    <namespace name="xmlns:log" uri="http://systinet.com/wsdl/service/log/" />
    <partnerLink name="logPL" partnerRole="loggingservice"
                 partnerLinkType="tns:logPLT" />
    <variable name="logMessage" messageType="log:LogService_log_Request_Soap" />
    <variable name="logResponse" messageType="log:LogService_log_Response_Soap" />
  </padus:using>

  <padus:pointcut name="a0(Jp)"
                  pointcut="assigning(Jp, [name(assign0)])" />

  <padus:pointcut name="a1(Jp)"
                  pointcut="assigning(Jp, [name(assign1)])" />

  <padus:pointcut name="a2(Jp)"
                  pointcut="assigning(Jp, [name(assign2)])" />

  <padus:pointcut name="a3(Jp)"
                  pointcut="assigning(Jp, [name(assign3)])" />

  <padus:pointcut name="testProtocol(Jp1, Jp2, Jp3)"
                  pointcut="sequence(choice([concat([a0(Jp1), a1(Jp2)]), a3(Jp3)]))" />

  <padus:after joinpoint="Jp1, Jp2, Jp3" pointcut="testProtocol(Jp1, Jp2, Jp3)">
    <sequence>
      <assign>
        <copy>
          <from expression="'*** Logged protocol ***'" />
          <to variable="logMessage" part="parameters" query="/log:log/log:p0" />
        </copy>
      </assign>
      <invoke partnerLink="logPL" portType="log:LogService"
          operation="log" inputVariable="logMessage" outputVariable="logResponse" />
    </sequence>
  </padus:after>
</padus:aspect>
```

The process after the protocol is woven is given below. The advices which trace the protocol are indicated in bold.

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="test" targetNamespace="http://localhost/test"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:log="http://systinet.com/wsdl/service/log/"
  xmlns:tns="http://localhost/test">
  <partnerLinks>
    <partnerLink myRole="bookingservice" name="clientPL"
                 partnerLinkType="tns:clientPLT"/>
    <partnerLink name="logPL" partnerLinkType="tns:logPLT"
                 partnerRole="loggingservice"/>
    <partnerLink name="logPL" partnerLinkType="tns:logPLT"
                 partnerRole="loggingservice"/>
  </partnerLinks>
  <variables>
    <variable messageType="tns:test_InputMessage" name="testInput"/>
    <variable messageType="tns:test_OutputMessage" name="testOutput"/>
    <variable messageType="log:LogService_log_Request_Soap" name="logMessage"/>
    <variable messageType="log:LogService_log_Response_Soap" name="logResponse"/>
    <variable name="testProtocol2" type="xsd:string"/>
  </variables>
  <sequence name="main">
    <receive createInstance="yes" operation="cityOperation"
            partnerLink="clientPL" portType="tns:clientPT" variable="testInput"/>
    <sequence>
      <assign>
        <copy>
          <from expression="'[0]'"/>
          <to variable="testProtocol2"/>
        </copy>
      </assign>
      <assign name="assign0">
        <copy>
          <from expression="0"/>
          <to variable="i"/>
        </copy>
      </assign>
      <switch>
        <case condition="bpws:getVariableData('testProtocol2') = '[0]'">
          <sequence>
            <assign>
              <copy>
                <from expression="'[6]'"/>
                <to variable="testProtocol2"/>
              </copy>
            </assign>
          </sequence>
        </case>
      </switch>
    </sequence>
    <switch>
      <case condition="bpws:getVariableData('testInput', 'city') = 'Brussel'">
        <sequence>
          <assign name="assign1">
            <copy>
              <from expression="'BRUSSEL (0)'"/>
              <to part="output" variable="testOutput"/>
            </copy>
          </assign>
          <switch>
            <case condition="bpws:getVariableData('testProtocol2') = '[6]'">
              <sequence>
                  <assign>
                    <copy>
                      <from expression="'[5]'"/>
                      <to variable="testProtocol2"/>
                    </copy>
                  </assign>
                <sequence>
                  <assign>
                    <copy>
                      <from expression="'*** Logged protocol ***'"/>
```

```
                    <to part="parameters" query="/log:log/log:p0"
variable="logMessage"/>
                  </copy>
                </assign>
                <invoke inputVariable="logMessage" operation="log"
                        outputVariable="logResponse" partnerLink="logPL"
                        portType="log:LogService"/>
              </sequence>
            </sequence>
          </case>
        </switch>
      </sequence>
    </case>
    <case condition="bpws:getVariableData('testInput', 'city') = 'Gent'">
      <assign name="assign2">
        <copy>
          <from expression="'GENT (1)'"/>
          <to part="output" variable="testOutput"/>
        </copy>
      </assign>
    </case>
    <case condition="bpws:getVariableData('testInput', 'city') = 'Antwerpen'">
      <sequence>
        <assign>
          <copy>
            <from expression="'[0]'"/>
            <to variable="testProtocol2"/>
          </copy>
        </assign>
        <assign name="assign3">
          <copy>
            <from expression="'ANTWERPEN (5)'"/>
            <to part="output" variable="testOutput"/>
          </copy>
        </assign>
        <switch>
          <case condition="bpws:getVariableData('testProtocol2') = '[0]'">
            <sequence>
              <assign>
                <copy>
                  <from expression="'[9]'"/>
                  <to variable="testProtocol2"/>
                </copy>
              </assign>
              <sequence>
                <assign>
                  <copy>
                    <from expression="'*** Logged protocol ***'"/>
                    <to part="parameters" query="/log:log/log:p0"
variable="logMessage"/>
                  </copy>
                </assign>
                <invoke inputVariable="logMessage" operation="log"
                        outputVariable="logResponse" partnerLink="logPL"
                        portType="log:LogService"/>
              </sequence>
            </sequence>
          </case>
        </switch>
      </sequence>
    </case>
  </switch>
  <reply name="return" operation="cityOperation"
         partnerLink="clientPL" portType="tns:clientPT" variable="testOutput"/>
</sequence>
</process>
```

Bibliography

[1] van der Aalst W., ter Hofstede A., Kiepuszewski B., Barros A., *Workflow Patterns*. QUT Technical report. FIT-TR-2002-02, Queensland University of Technology, Brisbane, 2002

[2] Andrews T. et al., *Business Process Execution Language for Webservices 1.1 (BPEL4WS)*. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/, July 2002

[3] Benavides Navarro L., Sudholt M., Vanderperren W., De Fraine B. and Suvee D., *Explicitly Distributed AOP using AWED.* Technical Report INRIA 5882, extended version of Benavides et al., AOSD'06, 2006

[4] Bieberstein N., Bose S., Fiammante M., Jones K., Shah R., *Service-Oriented Architecture (SOA) Compass : Business Value, Planning, and Enterprise Roadmap* (Developerworks), IBM Press, 2004

[5] Bih J., *Service-oriented architectures (SOA): A New Paradigm to Implement Dynamic E-Business Solutions*. ACM Ubiquity Magazine (Volume 7, issue 30), August 2006

[6] Bockisch, C., Mezini, M., Ostermann, K., *Quantifying over Dynamic Properties of Program execution*. Dynamic Aspects Workshop (AOSD '05), March 2005

[7] Braem M., Verlaenen K., Joncheere N., Vanderperren W., Van Der Straeten R., Truyen E., Joosen W. and Jonckers V., *Isolating Process-Level Concerns Using Padus.* In Proceedings of the 4th International Conference on Business Process Management (BPM 2006), Vienna, Austria, LNCS Springer-Verlag, September 2006

[8] Brown A., Haas H., *Web Services Glossary*. http://www.w3.org/TR/ws-gloss/, World Wide Web Consortium (W3C), February 2004

[9] Cardoso, J. *Complexity Analysis of BPEL Web Processes*. Software Process: Improvement and Practice Journal (pp. 35-49), InterScience, Wiley, 2007

[10] Casati F. et al. *Adaptive and Dynamic Service Composition in eFlow*. In Proceedings of CAiSE 2000, LNCS 1789, Springer Verlag, 2000

[11] Charfi A., Mezini M., *AO4BPEL: An Aspect-Oriented Extension to BPEL*. To appear in World Wide Web Journal: Recent Advances on Web Services (special issue), Springer-Verlag Press, 2006

[12] Charfi A., Mezini M., *Aspect-Oriented Workflow Languages*. In Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS), November 2006

[13] Christensen E., Curbera F., Meredith G., Weerawarana S., *Web Services Description Language (WSDL) 1.1*., http://www.w3.org/TR/wsdl, World Wide Web Consortium (W3C), 2001

[14] Cibran M., *Using Aspect-Oriented Programming for Connecting and Configuring Decoupled Business Rules in Object-Oriented Applications*. MSc Thesis, 2001

[15] Cibran M., D'Hondt M., *Composable and Reusable Business Rules Using AspectJ*. In the Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT) at the International Conference on Aspect-Oriented Software Development, March 2003

[16] Cibran M., Verheecke B., *Dynamic Business Rules for Web Service Composition.* In Proceedings of the 2nd Dynamic Aspects Workshop (DAW05), workshop at the International Conference on Aspect-Oriented Software Development, March 2005

[17] Cibran M., D'Hondt M., *High-Level Specification of Business Rules and Their Crosscutting Connections*. 8th International Workshop on Aspect-Oriented Modeling at the International Conference on Aspect-Oriented Programming (AOSD'06), March 2006

[18] Cibran M., D'Hondt M. *High-Level Specification of Business Rules and Their Crosscutting Connections.* 8th International Workshop on Aspect-Oriented Modeling at the International Conference on Aspect-Oriented Programming (AOSD'06), March 2006

[19] Clark J., DeRose S., *XML Path Language (XPath)*. http://www.w3.org/TR/xpath, World Wide Web Consortium (W3C), November 1999

[20] Clocksin W.F., Mellish C.S., *Programming in Prolog: Using the ISO Standard* (5th edition). Springer-Verlag, 2003

[21] Courbis C., Finkelstein A., *Towards an Aspect Weaving BPEL Engine*. Presented at 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'04), 2004

[22] Courbis C., Finkelstein A., *Towards Aspect Weaving Applications*. In Proceedings of the 27th International Conference on Software Engineering (ICSE), ACM Press pp. 69-77, 2005

[23] Courbis C., Finkelstein A., *Weaving Aspects into Web Service Orchestrations*. Presented at 3rd IEEE International Conference on Web Services (ICWS 2005), 2005

[24] De Fraine B., Vanderperren W., Suvee D. and Brichau J., *Jumping Aspects Revisited*. In Proceedings of DAW 2005, March 2005

[25] De Volder K., D'Hondt T., *Aspect-oriented Logic Meta-Programming*. In Proceedings of 2nd International Conference on Meta-Level Architectures and Reflection, LNCS 1616, pp. 250-272, Springer-Verlag, 1999

[26] Deransart P., Ed-Dbali A., Cervoni L., *Prolog: The Standard Reference Manual*. Springer-Verlag, 1996

[27] Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, 1976

[28] Douence R., Südholt M., *A Model and a Tool for Event-based Aspect-Oriented Programming (EAOP).* TR 02/11/INFO, February 2003

[29] Douence R., Fradet P., Südholt M., *Composition, Reuse and Interaction Analysis of Stateful Aspects*. In Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04), March 2004

[30] Douence R., Fradet P., Südholt M., *Trace-based Aspects*. Addison-Wesley Professional, September 2004.

[31] Edmond D., ter Hofstede A., *Achieving Workflow Adaptability by Means of Reflection*. ACM SIGGROUP Bulletin, Volume 20, Issue 3, December 1999

[32] Estrade B., Perkins L., Harris J., *Explicitly Parallel Regular Expressions*. In Proceedings of the First international Multi-Symposiums on Computer and Computational Sciences - Volume 1 (Imsccs'06) - Volume 01 (pp. 402-409), June 2006

[33] Farias A., Südholt M., *On Components with Explicit Protocols satisfying a notion of Correctness by Construction*. In Distributed Objects and Applications, October 2002

[34] Filman, R. E. & Friedman, D. P. *Aspect-Oriented Programming is Quantification and Obliviousness*. In Workshop on Advanced Separation of Concerns, Conference on Object-Oriented Programming, Systems, Languages and Applications, 2000

[35] Filman, R. E. *What is Aspect-Oriented Programming, revisited*. In Workshop on Advanced Separation of Concerns, 15th European Conference on Object-Oriented Programming, 2001

[36] Gamma, E. et al., *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994

[37] Gudgin, M., et al., *Simple Object Access Protocol (SOAP) 1.2*. http://www.w3.org/TR/soap12-part0/, World Wide Web Consortium (W3C), 2002

[38] Gybels K., Brichau J., *Arranging Language Features for more robust Pattern-based Crosscuts*, in Proceedings of the 2nd International Conference of Aspect-Oriented Software Development, 2003

[39] Han Y., Sheth A., Bussler C., *Taxonomy of Adaptive Workflow Management*. In Towards Adaptive Workflow Systems Workshop, CSCW98, Seattle, USA, 1998

[40] Hannemann J., Kiczales G., *Design Pattern Implementation in Java and AspectJ*. In Proceedings of OOPSLA 2002, ACM Press, November 2002

[41] Henkel M., Zdravkovic J., Johannesson P., *Service-based Processes – Design for Business and Technology*. In ICSOC 2004, ACM Press, November 2004

[42] Henkel M., Zdravkovic J., *Architectures for Service-oriented Processes*. Proceedings of the Nordic Conference on Web Services (NCWS'04), November 2004

[43] Houspanossian A., *Enhancing a BPEL4WS Engine: Supporting the Execution of Flexible WS-flows According to the ReFFlow Model*. MSc Thesis, April 2006

[44] IBM, *The IBM Business Process Execution Language for Web Services Java Run Time (BPWS4J)*. http://www.alphaworks.ibm.com/tech/bpws4j

[45] Jędrzejowicz J. and Szepietowski A., *Shuffle languages are in P*. Theoretical Computer Science Vol. 250 (pp 31-53), January 2001

[46] Johnson R. et al., *Spring application framework*. http://www.springframework.org/, 2004-2007

[47] Kiczales G., et al., *Aspect-Oriented Programming*. In proceedings of European Conference on Object-Oriented Programming (ECOOP'97), Springer-Verlag Press, June 1997

[48] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W., *An Overview of AspectJ*. In Proceeding of the European Conference on Object-Oriented Programming, 2001

[49] Leymann F., *Web Services Flow Language (WSFL)*. IBM Software Group, May 2001

[50] Lieberherr K., Lorenz D., Mezini M., *Programming with Aspectual Components*. Technical Report NU-CCS99-01, Northeastern University, March 1999

[51] Lieberherr, K., Orleans, D., Ovlinger, J., *Aspect-oriented programming with adaptive methods*. Communications of the ACM, 44(10), 39–41, Februari 2001

[52] Oren E., Dietz J. L. G., *Development of a DEMO based Workflow Management System*. In 1st International DEMO Workshop, May 2003

[53] Parnas, D. L., *On the Criteria to be used in Composing Systems into Modules*. Communications of the ACM, 15(12), 1053–1058, 1972

[54] Petri C.A., Fundamentals of a theory of asynchronous information flow. In Proceedings of IFIP Congress 62, (pp. 386–390), Munich, 1962

[55] Seymour M., *Enabling Adaptive Business Processes: Oracle E-Business Suite and Service-Oriented Architecture*, Oracle white paper, August 2005

[56] Shaw A.C., *Software Descriptions with Flow Expressions*, IEEE Trans. Software Engineering SE-4 (pp. 242-254), 1978

[57] Suvee D., Vanderperren W. and Jonckers V., *JAsCo: an Aspect-Oriented Approach tailored for Component Based Software Development.* In Proceedings of International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA, pp 21-29, ISBN 1-58113-660-9, ACM Press, march 2003

[58] Tarr, P., Ossher H., Harrison W., Stanley M., Sutton J. *N degrees of separation: Multi-dimensional Separation of Concerns*. In Proceedings of the International Conference on Software Engineering (pp. 107–119).: IEEE Computer Society Press / ACM Press, May 1999.

[59] Thatte S., *XLANG: Web Services for Business Process Design*. Microsoft, http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001

[60] Vanderperren W., Suvee D., Verheecke B., Cibrán M., Jonckers V., *Adaptive Programming in JAsCo*. In Proceedings of AOSD 2005, ACM Press, 2005

[61] Vanderperren W., Suvee D., Cibran M., De Fraine B. *Stateful Aspects in JAsCo*. In proceedings of SC 2005, LNCS, April 2005

[62] Walker R., Viggers K., *Implementing Protocols via Declarative Event Patterns*. In Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM Press, November 2004

[63] Wohed P., van der Aalst W., Dumas M., ter Hofstede A., *Pattern-Based Analysis of BPEL4WS*. QUT Technical report, FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002