Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
2006

ACTA DEPENDENCIES AS A UNIFIED MECHANISM FOR COMPENSATION, ACTIVITIES SYNCHRONIZATION AND FUNCTIONAL REPLICATION IN BPEL4WS

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Sergio Castro Mejia

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Advisor: Johan Fabry (Vrije Universiteit Brussel)

To all my teachers, especially those who taught me how to read

Contents

List of Tables

List of Figures

Abstract

Advanced transaction management in loosely coupled environments is not a trivial task. Web Services are the current state of the art for interoperability for such kind of systems, and the BPEL4WS language is nowadays the de facto standard for web services composition in industry. However, the BPEL4WS strategy for expressing transactional properties is limited, non extensible, and relies on different unrelated constructs for managing the concerns of activity synchronization, functional replication and activity compensation. We believe that these concerns should be managed using a more simpler, extensible and unified set of constructs, relying on well proved existing formal models for advanced transaction management. In this work we introduce such a composition framework called DBCF (Dependency Based Composition Framework). DBCF demonstrates the viability of our conceptual model in service composition languages. In addition, we present an extension for BPEL4WS that adds these features, and suggest ways of describing declaratively transactional properties expressed with our proposed extension using aspect oriented technologies.

Chapter 1

Introduction

Large scale distributed systems are based in a client-server architecture of a number of conceptual layers. Basically, what is proposed in this architecture is a separation in tiers of the different components of a distributed system. For example, a separation is performed between the client of the system, the interface of the application, the application logic and the data sources accessed by the application.

The infrastructure of services located between the client and the resources manager is called middleware, and resource managers maintain the data sources that the system needs for working properly. An interesting detail to mention here is the fact that resource managers could be databases, file systems, or even other information systems. This can be considered a recursive definition, since these new information systems could also have other information systems as part of their resource managers.

Therefore, we can see two possible settings here: in the most simple case, no one resource manager is an information system itself, all the components of the system are deployed in a homogeneous environment and there is only a single middleware platform. This is called a *tightly-coupled environment*. In this kind of systems, often components cannot be used independently.

In the second case, one or more resource managers of the system are information systems themselves, some of them even deployed outside the boundaries of the company where the original system was deployed, and the execution environments are completely heterogeneous. This is called a *loosely-coupled environment*. In these systems, some of the individual components may be used independently of the others (since they are also information systems). [ACKM04].

How the components present in a loosely-coupled environment can interact is a very old issue in computer science, and the most recent strategy for solving this problem is called

Web Services.

Web Services basically are a technology that proposes mechanisms for standardizing the interfaces of such components, with the objective of solving their interoperability and integration problems. One additional advantage of Web Services technology, is that it seems well suited for accomplishing automated integration of cross-enterprise applications, therefore a lot of standardization efforts have been targeted in this direction.

One of the reasons for the success of Web Services as a common interface for different heterogeneous systems, is the use of popular and widely used open standards, such as XML [XML] as a base for structuring data [1], and HTTP [HTT] for transporting them. Since the transport of Web Service messages is based on HTTP, it has the advantage that these messages can be easily sent over the Internet, since this protocol has already solved most of the associated communication problems on this platform.

Another strong advantage of this technology, is the possibility of easily defining a Web Service as a composition of other Web Services. As we will see, the existence of specialized languages for accomplishing this work, open the door to more automated interactions between heterogeneous systems, since a description of such interactions can be formally written using these languages.

1.1 The problem of transaction management in loosely coupled systems

A transaction is a protected form of program execution which has a collection of properties commonly identified as *ACID* properties [GR93]. These properties are:

- *Atomic*.- A transaction is an atomic sequence of actions: either all actions are executed, or none of them is executed.
- *Consistent*.- A transaction takes the database from one consistent state to another consistent state.
- *Isolated*.- The intermediate effects of a running transaction are isolated from the other transactions.
- *Durable*.- The results of a transaction are kept in persistent storage.

[1] Messages are sent over the net using the SOAP [SOA] protocol, a XML based protocol that provides all the abstractions for defining Web Service messages.

This definition works well in tightly-coupled systems, but in loosely-coupled systems, where transactions could last a great amount of time and involve a lot of distributed components, the enforcement of the ACID properties can create performance penalties or even worse problems. As an example, a long lived transaction composed for a certain number of sub-transactions, cannot choose to block the resources in all the involved hosts until a global commit or abort be issued, since this situation could lead to deadlocks if a lot of transactions are interested in the same group of resources. This create the necessity of relaxing the isolation requirement of traditional acid transactions, in order to avoid the blocking of resources, and employ an alternative model, such as *Sagas* [GMS87] or *Mixed Transactions* [ELLR90] (both described in chapter 3). In general, since the traditional concept of transactions is not applicable anymore, some of the ACID constraints must be relaxed. These alternative models that relax the traditional ACID properties of transactions, are known as Advanced Transactions Mechanisms (ATMS).

1.2 Transaction Management in Web Services. Problems and proposed solutions

In the jungle of Web Services related specifications, two kind of specifications related to transaction management can be found.

- Collections of specifications that define frameworks for signalling and distributing transactional messages [CCF+05c] [CCF+05b] [CCF+05a] [BCC+04] [BBC+06b] [BBC+06a] [BCH+03a] [BCH+03b] [BCH+03c] [BTP].
- Composition language specifications that incorporate transaction management concerns [ACD+03] [BPM] [Ley01].

The first category is related to frameworks that only send transactional related messages to the interested participants in a distributed interaction. These specifications do not define how the transaction management will be accomplished in any of the individual participants, instead a general coordinator and its communication protocol are defined. Such coordinator is in charge of receiving and dispatching the transactional messages to the components. Nowadays there is no clear leader from the available specifications in this category, therefore defining a state of the art is not an easy task. In addition, the license agreements of most of them are not well defined, which could cause problems for a full adoption of these technologies in the Web Services community.

In the second category we have a different situation. The BPEL [ACD+03] language is the state of the art for Web Services composition nowadays. Therefore, during this thesis,

most of our efforts will be devoted to discuss the way BPEL realizes advanced transaction management, and what can be done for improving this in BPEL like languages.

As we will demonstrate, the BPEL constructs for transaction management, in certain cases demand a lot of work for expressing very simple ideas, and do not define extensible points for dealing with new kind of transactional concerns. As a solution of these problems, we present DBCF, a framework for Web Services composition that overcomes the BPEL limitations. DBCF can be considered as a tool for implementing applications that compose Web Services, and can be used as a framework for implementing new services composition languages that follow the DBCF transactional model. We also present an extension to BPEL that makes use of the DBCF concepts, and suggest ways of employing advanced mechanisms for separation of concerns, such as aspect oriented programming techniques, for declaratively specifying the transactional properties of activities in a BPEL process.

1.3 Contributions

The contributions of this work are:

- An exploration of the current industry state of the art language for Web Services composition, and a discussion of its limitations related to advanced transaction management, have been presented.
- We have realized a study of relevant academic proposals for advanced transaction management, and from this research we have extracted commonalities that can be applied to Web Services composition.
- An unified conceptual model for transaction management in Web Services composition, based on the transactional strategies researched before, has been developed.
- We have demonstrated the correctness of our model with the design and implementation of DBCF, a framework that is based on it. This framework can be used for implementing single composition applications or Web Services composition languages that follows our model.
- As an additional application of our model, an extension to BPEL has been proposed. The constructs of our approach provide a simpler and more extensible proposal in comparison to the BPEL constructs related to transaction management.
- As a final step, we issue recommendations about how to take advantage of separation of concerns, using aspect oriented programming for expressing transactional concerns in our proposed BPEL extension.

1.4 Overview of this thesis

This dissertation begins with an overview of the main concepts related to Web Services in chapter 2. It is presented as the state of the art for achieving interoperability between components and automating interaction in loosely coupled systems. We also discuss in that chapter relevant technologies and standards related to Web Services that will be revisited later in the development of this work, particularly BPEL4WS [ACD+03] [2], the current state of the art for web services composition that has become a de facto standard in industry and PADUS [BVJ+06], an aspect oriented language for modularizing crosscutting and tangled concerns in BPEL.

In chapter 3 we present a revision of relevant general purpose models and languages for advanced transaction management. Later, chapter 4 discusses the analysis requirement of a solution for advanced transaction management in Web Services composition, that incorporates relevant insights from the models discussed before and makes a special emphasis in a discussion of the BPEL proposal, in order to establish later a transactional model that surpass the one used in BPEL. The result of this work, a model mainly inspired in the ACTA framework [CR92] and concepts from KALA [Fab05], is described in chapter 5, where a reference implementation is also provided, called DBCF.

The ideas presented in that chapter differ from the way BPEL manages these issues, and we demonstrate the advantages of DBCF over BPEL in chapter 6, where a detailed comparison of these two proposals is presented.

We claim also that our proposal can bring extensibility and simplification (related to transaction management) to BPEL, and we concretize this idea in chapter 7, where some new constructs for BPEL are proposed, in such a way that this extension to the language can express straightforwardly the relevant ideas of our model.

Chapter 8 shows a case study, where first a problem is developed using only standard BPEL constructs and after the same problem is presented using our proposed extension. As a further step, the last section of this chapter shows the solving of the same problem using PADUS for modularizing some transactional concerns present in the case study, using aspect oriented technologies.

Finally chapter 9 will show a summary of our work, our conclusions and the future work to accomplish.

[2] In the development of this thesis, *BPEL4WS* will be simply referred as *BPEL*.

Chapter 2

Web Services

This chapter will introduce the Web Services technology and related specifications. We begin defining this technology, its building block standards and discussing how Workflow Management systems are related to it. After, we present advanced transaction management specifications for Web Services, and we discuss Web Services composition languages. Finally, We conclude with an exposition of how aspect oriented programming techniques can be used in the domain of Web Services composition languages, for modularizing crosscutting and tangled code. All these topics will provide the basic technological background required for further discussion in the rest of this dissertation.

2.1 What are Web Services?

Existing definitions for Web Services range from the very generic to the very specific [ACKM04]. Often, a Web Service is seen as an application accessible to other applications over the Web [Fis02] [MA01]. According to this perspective, anything that has a URL in the Web is a Web Service.

Another more precise definition, and the one that we will use for our purposes, is the offered by the World Wide Web consortium (W3C) [w3c]: "a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web Service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols" [ws002].

According to this definition, Web Services should be "Services" in the sense that in addition to be up and running, they should be described and advertised so that it is possible to write clients that bind and interact with them. In other words, Web Services are components that can be integrated into more complex distributed applications [ACKM04].

In addition, they can be seen as an attempt to standardize the interfaces of middleware platforms in order to solve the interoperability problems that have been always present in application integration efforts. They are envisioned to be the basis for a seamless and almost completely automated infrastructure for cross-enterprise application integration.

Figure 2.1 shows an example of the utility of Web Services. In this example we show a travel agency system, which permits the query of different flight schedules to its clients. Each time a request for flight schedules arrives, it has to communicate with different flight companies, query their corresponding schedules, and send a summary of this information to the client. As showed in the example, one of the biggest problems to solve is the automated communication between all the heterogeneous systems that are involved in the process. Without Web Services, in order to automatically query the systems of the partners flight companies, the travel agency would need to install and maintain different clients for each possible technological platform that its business partners use. Conversely, the use of Web Services as a standard technology for exposing the functionality of information systems, reduces heterogeneity and facilitates application integration, since that with all the business partners providing the same kind of interface, the amount of work needed for interacting with them get significantly decreased.

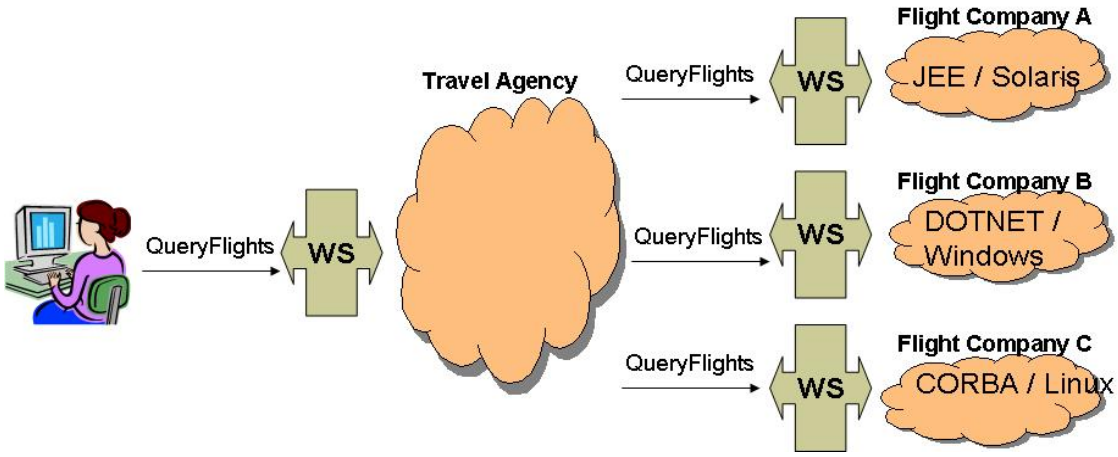

Figure 2.1: Web Services

2.2 Building block standards

One reason for the acceptation of Web Services, is their use of well known and open standards. At a low level, messages are transported over the Internet via *HTTP* [HTT], this creates a lot of flexibility since internet protocols have already solved most communication problems between distributed systems. The request and response messages, are sent across

the net in the form of SOAP messages [SOA]. SOAP is a XML [XML] based protocol that provides all the abstractions for defining Web Service messages with or without binary attachments. Once defined the transport protocol (SOAP over HTTP), the next step is to define a standard for describing Web Services interfaces, this problem is solved by the *WSDL* [WSD] specification.

Over these basic standards, additional Web Services related specifications have been defined, for example: specifications for enabling the deployment and discovery of Web Services, such as *ebXML* [ebX] or *UDDI* [UDD]. The lack of agreement in the Web Services community for defining common standards instead of competitive ones that try to solve exactly the same thing, is one of the biggest current problems of the technology. In the example that we mentioned before, the ebXML and the UDDI specifications share the same objectives, but ebXML is sponsored by the OASIS [oas] group and the U.N./CEFACT [cef] [1] and UDDI is sponsored by an important vendor consortium [BCE+06]. We will refer again to this problem later, but in the domain of advanced transaction management for Web Services.

2.3 Relationship between Web Services and Workflow Systems

2.3.1 Overview of Workflow Management Systems

Workflow Management Systems [2] have their origin in office automation systems. They were used first for automating administrative process based on paper documents, where all the participants of the process were human. When the technology matured, they began to be used also as a tool for facilitating the definition and maintenance of the business logic necessary to integrate heterogeneous and distributed systems. This kind of workflows received the name of *Production Workflows*.

Workflow Management Systems can be defined as the software platform that supports the design, development, execution and analysis of a business process. A business process is a collection of activities performed by human users or software applications. All these activities constitute the different steps to be completed in order to achieve a business objective. The runtime execution of a workflow is called a *Workflow Instance*.

Most of the time, workflow models are specified with some kind of graphic notation, like a representation of a direct graph with distinct nodes types such as: *Work* nodes, *Routing* nodes and *Begin* and *End* nodes. The following is a short description of these typical

[1] An United Nations body with standardization purposes in the fields of trade and Electronic Business.
[2] Unless stated the contrary, most of the information of this section has been adapted from [ACKM04].

nodes:

- *Work nodes.*- Represent a basic unit of work to be performed by a human user or a system.
- *Routing nodes.*- Define the order in which work items should be executed.
- *Begin and End nodes.*- Denote the start and completion points of the workflow.

2.3.2 Programming in large

Workflow systems are used to compose large software modules, which typically are entire applications. Therefore, they embody the idea of *megaprogramming* [WWC92]. This kind of programming can be viewed as the composition of large modules (sometimes called *megamodules*) rather than programming from scratch. Therefore, workflows play an important role as the programming language for enterprise application integration [ACKM04].

Most of the time workflow languages are similar to third generation languages. Basically, they specify the invocation of several functions in an order described by a workflow logic. As in programming languages, a workflow can have data items (variables) that can be used as input/output parameters when invoking work activities. This data can be used for evaluating execution conditions or for interchanging information between nodes.

Workflow languages however, need a special emphasis on dynamic resource selection, transaction management and exception handling. These requirements are a consequence of the loosely-coupled environment in which workflow systems are executed, where continuously new resources can be available and others can disappear, activities in a process can last a no determined amount of time, and improved strategies for managing errors and problems, associated with such long-lasting activities, are needed.

2.3.3 Standardization efforts in Workflow Systems and Web Services

Many standardization efforts have been targeting workflow management systems. The best well known effort is performed by the Workflow Management Coalition [wmc], beginning at the mid-90 having as objective the standardization of workflow definition languages and their APIs for accessing workflow management systems. However, nowadays the interest in this organization is fading, and only few vendors currently support the proposed standards. Now, the standardization efforts are being revived in the context of Web Services composition, and in this area BPEL [ACD$^+$03] has emerged as the widely accepted proposal. This language will be described later in this chapter.

2.4 A note about transaction management and Web Services composition

The great number of uncoordinated proposals related to transaction management and web services composition, makes the appropriate selection of standards for our discussion a difficult task. Currently there is no guarantee that most of these specifications will survive a significant amount of time, that will make them an essential part of the Web Services technology, citing [ACKM04]: "In many cases, existing specifications ignore the complexities of implementing the functionality they describe and build upon other specifications that have not yet been standardized." Therefore, we have chosen in this chapter to make a selection of representative proposals. We will discuss the following specifications:

- WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity.- These are the main specifications of one of the existing collection of specifications related to advanced transaction management with Web Services.
- BPEL.- A popular specification for Web Services composition that has become the de facto standard superseding other similar specifications.

We have not chosen to present all the existing specifications because the extension of such description would be excessively lengthly. As stated before, currently there is not a clear leader among the specifications for advanced transaction management, differently to what happened with composition related specifications. Therefore, we have centered our efforts on Web Services composition, where the BPEL specification is currently considered the state of the art in industry.

At this point, it is important to say that although our focus in this work is centered in a proposal for Web Services composition, this does not mean that transaction management issues are not present in such compositions. As we will see soon, BPEL provides specialized constructs for dealing with concerns related to advanced transaction management, these constructs will be of particular interest in chapter 7.

2.5 Transaction management in the Web Services world

As mentioned previously, the specifications present here are a selection from the broad range of similar proposals. We believe that the chosen specifications share most of the same objectives of similar ones, and therefore they are representative enough for giving a clear general idea of the strategies that are followed by other ATMS related specifications in the Web Services domain.

These specifications constitute a strategy for implementing transaction concerns, that consist solely in the signaling and distribution of messages related to transaction management between registered participants. The specifications do not provide any information about how individual participants should implement the receiving of messages, since one of the main objectives of this model is to to preserve the lightweight nature of the Web Services architecture, being as minimalist and unintrusive as possible.

2.5.1 WS-Coordination

The WS-Coordination specification [3] defines an extensible framework for coordinating activities using a coordinator and set of coordination protocols.

The coordinator service (or coordinator) is an aggregation of various services:

- Activation Service.- This service allows a new coordination context to be created.
- Registration Service.- Defines a register operation that allows a Web Service to register to participate in a coordination protocol.
- A set of coordination protocol services for each supported coordination type.- These services are defined in the specification that defines the coordination type.

Every time that a participant needs to send a transactional message, it is sent to the appropriate coordinator. This coordinator will communicate the received message to the participants in the transaction, based on the coordinator context of the message .

In case a participant wants to initiate a new transaction, a message of activation is sent to the coordinator. The coordinator will answer a coordination context, and the caller will be able to propagate this context to other participants of the transaction. When other participants receive the coordination context, they can manifest their desire for registration at the coordinator, and a register request is sent to it.

This is the core of the communication protocol, a lot of additional details are provided in the specification, like a complex model for doing the same with more than one coordinator. For additional description of this specification see [CCF+05c].

[3] This section is adapted from [CCF+05c].

2.5.2 WS-AtomicTransaction

The WS-AtomicTransaction specification [4] defines a two-phase commit protocol that can be controlled by the coordinator described in the previous section. This coordination protocol defines how multiple participants reach agreement on the outcome of an atomic transaction. It has two variants: Durable 2PC and Volatile 2PC.

Volatile Two-Phase Commit Protocol

Upon receiving a Commit notification, the coordinator begins the prepare phase of all participants registered for the Volatile 2PC protocol. All participants registered for this protocol must respond before a Prepare is issued to a participant registered for Durable 2PC. New participants may register with the coordinator until the coordinator issues a Prepare to any durable participant. In addition, a volatile recipient is not guaranteed to receive a notification of the transaction's outcome.

Durable Two-Phase Commit Protocol

After receiving a Commit notification and upon successfully completing the prepare phase for Volatile 2PC participants, the root coordinator begins the Prepare phase for Durable 2PC participants. All participants registered for this protocol must respond Prepared or ReadOnly before a Commit notification is issued to a participant registered for either protocol. If that is the case, the Commit notification is sent to all the participants than answered Prepared, and the participants that answered ReadOnly will be ignored. Otherwise, if a participant answered Aborted, the coordinator will send the message Rollback to all the participants. Afterwards, the participants will commit or abort according to the message received.

The WS-AtomicTransaction specification and its protocols Volatile Two-Phase and Durable Two-Phase Commit do not define all the features required for a complete solution. They are complemented with other specifications, such as the WS-Coordination specification discussed before.

[4] This section is adapted from [CCF+05a].

2.5.3 WS-BusinessActivity

The WS-BusinessActivity specification [5] defines protocols that enable existing business process and workflow systems to wrap their proprietary transactional mechanisms and interoperate across each other. Here, actions are applied immediately and are permanent. Also, compensating actions may be invoked in the event of an error.

The coordination protocols for business activities are summarized below:

- *BusinessAgreementWithParticipantCompletion.*- A participant registers for this protocol with its coordinator, so that its coordinator can manage it. A participant must know when it has completed all work for a business activity.
- *BusinessAgreementWithCoordinatorCompletion.*- A participant registers for this protocol with its coordinator, so that its coordinator can manage it. A participant relies on its coordinator to tell it when it has received all requests to perform work within the business activity.

Similar to the WS-AtomicTransaction specification, WS-BusinessActivity is not a complete solution, but should be used with other specifications, such as WS-Coordination, for providing a complete solution.

All the WS-* specifications described above, are not recognizable as universally accepted standards, because quite similar specifications are maintained by other groups (e.g. [BCH+03a] [BCH+03b] [BCH+03c] [BTP]). Since it is likely that these equivalent specifications eventually will merge, giving place to a truly accepted standard in the whole community, we do not go into more detail here.

2.6 Web Services composition with BPEL

A Web Service execution can involve the invocation of operations offered by other Web Services. Services implemented by combining the functionality provided by other services are usually referred as *Composite Services*. The process of developing a composite Web Service is referred as *Service Composition* [ACKM04].

In this section we will present BPEL (*Business Process Execution Language for Web Services*)[ACD+03][6], the current state of the art in industry for developing distributed systems based on Web Services composition.

[5] This section is adapted from [CCF+05b].

[6] This section is mainly a summary of [ACD+03]

2.6.1 BPEL overview

BPEL is a XML based language layered on top of several XML specifications (WSDL [WSD], XML Schema [Sch] and XPath [XPA]). Its objective is to provide a notation for specifying business process behaviour based on Web Services. Processes in BPEL export and import functionality exclusively by using Web Service interfaces.

Business process in BPEL can be described both as *Abstract Business Process* and *Executable Business Process*. Abstract processes specify the mutually visible message exchange of each of the parties involved in the protocol, without revealing their internal behaviour. In the other hand, executable business processes model the behaviour of a participant in a business interaction.

For example, lets consider three different roles in a business interaction: a customer, a supplier and a warehouse. In our scenario, the customer asks for buying specific supplies, sending a purchase order to the supplier. Afterwards, the supplier asks for inventory data to the warehouse before answering to the client. Therefore, the abstract process of the supplier can be informally described as:

- The receiving of a message from a customer.
- The invocation of a request to a warehouse.
- The response of the original message to the customer.

As we can see, the abstract business process does not include any internal business logic. It is the executable business process the one in charge of defining these details. In the context of our example, the executable business process of the supplier, must answer questions like what to do if the warehouse responds that the requested supplies are not in stock, or any other specific business condition that need to be implemented.

Executable business processes in BPEL are built from abstract business processes, in the sense that they should reflect the public message interchange defined in the abstract process, but as stated above they also add implementation details such as the management of message parameters and control flow data. We will focus in the rest of this dissertation on executable business process, since they provide more elements for discussion.

2.6.2 Language requirements of executable business processes

In order to provide precise descriptions of service behaviour for cross-enterprise business protocols, BPEL needs a rich process description notation with many features reminiscent

of an executable language. Some of the main concerns that should be considered by such language are:

- Business protocols invariably include data-dependent behavior.
- They need to consider conditional execution of activities and time-out events.
- The ability to specify exceptional conditions and their consequences, including recovery sequences.
- Long-running interactions include multiple, often nested units of work (hierarchical composition of activities), each with its own data requirements.
- Business protocols frequently require cross-partner coordination of the outcome (success or failure) of units of work at various levels of granularity.

2.6.3 Component Model

A BPEL process is basically composed of activities, which can be *basic* or *structured*. Basic activities represent the actual components of the composition, and correspond to the invocation or the receiving of a Web Service request (e.g. the *invoke*, *request* and *response* activities). Structured activities are used for establishing a hierarchy of activities and the execution control flow (e.g. the *sequence*, *switch*, *pick*, *while* and *flow* activities). In addition to them, it can also be defined activities for other purposes, like assigning data to variables (e.g. the *assign* activity) or timing management (e.g. the *wait* activity). In the next section we will offer a description of all these activities.

2.6.4 Language elements

BPEL provides a great amount of language constructs. We omit a detailed description, as this is outside of the scope of this work. An extensive discussion is provided in [ACD+03]. Instead, we offer a summary of the constructs for expressing BPEL activities, and additional discussions about the constructs related to transaction management will be given in chapter 4.

The activities that can be defined in BPEL are expressed with the following constructs:

- *receive*.- Allows the business process to do a blocking wait for a matching message to arrive.

- *reply*.- Allows the business process to send a message in reply to a message that was received through a *receive* activity. The combination of a *receive* and a *reply* forms a request-response operation.

- *invoke*.- Allows the business process to invoke a one-way or request-response operation.

- *assign*.- Can be used to update the values of variables with new data.

- *throw*.- Generates a fault from inside the business process.

- *terminate*.- Can be used to immediately terminate the behaviour of a business process instance within which the terminate activity is performed. All currently running activities must be terminated as soon as possible without any fault handling or compensation behavior.

- *wait*.- Allows you to wait for a given time period or until a certain time has passed.

- *empty*.- Represents a dummy, empty activity.

- *sequence*.- Allows you to define a collection of activities to be performed sequentially in lexical order.

- *switch*.- Allows you to select exactly one branch of activity from a set of choices.

- *while*.- Allows you to indicate that an activity is to be repeated until a certain success criteria has been met.

- *pick*.- Allows you to block and wait for a suitable message to arrive or for a time-out alarm to go off. When one of these triggers occurs, the associated activity is performed.

- *flow*.- Allows you to specify one or more activities to be performed concurrently. This activity permits the definition of synchronization *links*, this construct is used in BPEL for synchronizing activities nested in a flow, that otherwise would be executed concurrently. We will discuss the link construct with more detail in chapter 4.

- *scope*.- Allows you to define a nested activity with its own associated variables, fault handlers, and compensation handler.

- *compensate*.- Is used to invoke compensation on an inner scope that has already completed normally. This construct can be invoked only from within a fault handler or another compensation handler.

This concludes our introduction to BPEL, next section will explore enhancements to this language with the use of aspect oriented programming.

2.7 Aspect oriented programming in Web Services Composition

2.7.1 Overview of Aspect Oriented Programming

Aspect oriented programming [KLM+97] is a proposed solution for the software engineering problem of separation of concerns [HL95]. Ease of development and maintainability is increased when more appropriate techniques of modularization are used, since it takes into account the well known concept of "divide and conquer". In this way software is decomposed in smaller pieces, which individually are easier to understand and manage, and ideally each of these pieces address a specific concern of the application. [Par72]

The main objective of AOP is to provide techniques for the modularization of concerns that crosscut and are tangled with other concerns, facilitating in this way the development and maintainability of programs where other techniques (e.g. procedural or object oriented programming) are not enough for capturing clearly, in well defined modules, all the design decisions.

2.7.2 Aspect Oriented Programming and BPEL

Existing standards for web services composition (e.g. BPEL [ACD+03], BPML [Ark02]) are not an exception to the modularization problems described before, since they exhibit limitations regarding their modularization capability for crosscutting and tangling concerns [CM04]. Some proposals related to the possibility of applying aspect oriented programing to these composition languages have been made, but we have decided to focus our research on aspect oriented proposals for BPEL; AO4BPEL [CM04], PADUS [BVJ+06].

We will present PADUS [BVJ+06][7] as a sample of the state of the art on aspect oriented languages for web services composition that nowadays are available. Additional reasons for this choice will be given in chapter 7

In the rest of this section, we will give a short introduction to this aspect language and present the main components of it. In chapter 7, we will discuss again this aspect language, this time with a focus on how we can use it for expressing declaratively transactional concerns in BPEL.

[7] Unless stated the contrary, all the information related to PADUS is summarized from [BVJ+06], since at the moment of writing this thesis it was the only formal paper about this language and no testing implementation was yet available.

```
1  <aspect name="requestShippingTransaction">
2   <using>
3     ...
4     <partnerLink name="shipping" ... />
5     <partnerLink name="shippingAlt" ... />
6     ...
7   </using>
8   <pointcut ... />
9   <advice ... >
10    ...
11  </advice>
12  <in ... >
13    ...
14  </in>
15 </aspect>
```

Listing 2.1: Example of a PADUS aspect

2.7.3 Overview of PADUS

PADUS [BVJ+06] is an aspect oriented extension to BPEL with a higher-level pointcut language and a rich joinpoint model. One of the main objectives of PADUS is to be compatible with existing BPEL engines and at the same time minimize the runtime overhead produced for including aspect modules in a BPEL process (it has followed an "industry-applied" approach). In order to accomplish these objectives, PADUS follows a static weaving strategy. Therefore a BPEL process is merged with PADUS aspects using an aspect deployment descriptor, in such a way that the result of this transformation is a fully compliant BPEL process that can be executed in any engine that follow the standard BPEL specification.

Another interesting detail of PADUS is that analogously to BPEL, it is also a XML-based language. This facilitates the inclusion of standard BPEL elements in PADUS aspects. The listing 2.1 shows an example of a PADUS aspect and some of its mains constructs: Lines 2 to 7 shows the *using* element, inside of it we declare all the BPEL constructs required later by our aspect (e.g. partner links declarations), line 8 shows the declaration of a *pointcut*, lines 9 to 11 the declaration of an *advice* and lines 12 to 14 an *aspect module*.

In the following sections we will briefly describe PADUS along the five dimensions it is described in [BVJ+06]: its joinpoint model, pointcut language, advice language, aspect modules and aspect deployment language. We will not give here a full detailed description for each of these elements, for such exposition we refer to [BVJ+06].

```
1 ...
2 <pointcut name="shippingInvocation(Jp, activityName, inputVariable)"
3   pointcut="invoking(Jp, activityName, 'shipping', 'shippingPT', 'requestShipping',
4   inputVariable, _)"/>
5 ...
```

Listing 2.2: A PADUS pointcut

Joinpoint model

Joinpoints in aspect oriented programing are well defined points in the execution of a program where additional functionality can be added. They can be implicit or explicit, where implicit joinpoints are described around existing constructions of the base language and their semantics and explicit joinpoints are marked using labels [DB05].

According to this definition, PADUS uses an implicit joinpoint model, since joinpoints are related to the activities that are provided in BPEL. As mentioned in [BVJ+06], the PADUS joinpoint model allows in addition to " behavioural joinpoints" (related to basic activities, e.g. the invocation of a web service) structural joinpoints, which contain one or more activities themselves (e.g. the iteration activity). Joinpoints are also associated with the properties that are relevant to that particular joinpoint, an example of such properties are the attributes and elements related to a corresponding BPEL activity.

Pointcut language

A pointcut is a predicate on the runtime state. The elements that may appear in this predicate are "atoms" (e.g. a web service invocation), operators of boolean algebra, inspection of the dynamic or static type of an object and stack inspection matched with regular expressions [DB05].

As stated in [BVJ+06], the pointcut language of PADUS is based on logic meta-programming. A pointcut can be written as a collection of constraints on the type and properties of allowed joinpoints. Pointcuts in PADUS, similarly to other well known pointcut languages (e.g. the pointcut language of AspectJ [Asp]) are able to expose information related to a referred joinpoint in such a way that an advice can exploit it. Also, the PADUS pointcut language defines a different predicate for each type of joinpoint (e.g. invocation of web services, iteration, etc).

The listing 2.2 shows an example of a PADUS pointcut. In this particular example, the joinpoints to match are all the occurrences of an invoking activity with certain properties (partner link, port type and operation), and the exposed joinpoint information are the activity name, and the name of the input variable used in the invocation activity.

```
1 ...
2 <advice name="shippingTransaction(activityName, inputVariable)">
3   ...
4 </advice>
5 ...
```

Listing 2.3: A PADUS advice

Advice language

The advice language is mostly the target language [DB05] (in our case, a piece of XML representing BPEL activities). It is used to specify how the behaviour of certain joinpoints should be altered after the application of the aspect. Advices can be added to the original behaviour (typically *before* or *after* advices) or can replace it (*around* advices). For around advices, the *proceed* element can be added to include the original behaviour at the matched joinpoint. The pointcut attributes exposed to the advice can be referenced by prefixing their name with the *$* character.

In addition to these typical kind of advices, PADUS permits also the definition of *in* advices. This advice type performs the inclusion of activities nested inside other activities referenced in the pointcut where the *in* advice is applied. It is also possible to customize the behaviour of certain BPEL activities (adding flow links, variables, etc). We will see in chapter 7, how this kind of advice is particularly useful for our purposes, and since this cannot be simulated in a straightforward way using *before*, *after* or *around* advices, this feature makes PADUS more powerful than other proposals (e.g. AO4BPEL [CM04]).

Listing 2.3 shows an advice definition. This advice is parametrized with two values (*activityName* and *inputVariable*), which can be referenced inside the advice prefixing them with $ (*$activityName* and *$inputVariable*).

Aspect modules

Formally, an aspect is almost always an association between a pointcut and its advices [DB05]. Aspects in PADUS can contain several *before*, *after*, *in* and *around* advices. The listing 2.4 shows a fragment of an aspect module where a relationship between a specific pointcut (*shippingInvocation*) and an advice (*shippingTransaction*) is established. Here, two of the exposed values of the pointcut are passed as parameters to the advice (*activityName* and *inputVariable*), in order that the last one can use this information later. Another interesting detail here is how we are declaring where the advice *shippingTransaction* should be applied. Since the *in* construct is used in our example, the code of *shippingTransaction* will be nested inside the activities matched by the pointcut *shippingInvocation*.

```
1 ...
2 <in joinpoint="Jp" pointcut="shippingInvocation(Jp, activityName, inputVariable)">
3   <advice name="shippingTransaction(activityName, inputVariable)"/>
4 </in>
5 ...
```

Listing 2.4: An aspect module in PADUS

Aspect deployment language

Deployment in PADUS consists of two parts. First it treats when to instantiate and apply an aspect to a particular process. Processes can be referenced using their name, or selected in a pattern-based manner (e.g. all the process that invoke a particular service). The second part of the deployment treats aspect composition. The composition is responsible for specifying the aspect precedence in case that more than one aspect should be applied to the same join point. As stated in [BVJ+06]: "In case no precedence is specified, the advice is executed in the order in which their corresponding aspects are specified. A precedence declaration overrides this default and is able to specify precedence on a per-advice-type basis".

2.8 Conclusions

In this chapter we presented Web Services as the current proposal for solving the problem of interaction amongst components in loosely-coupled heterogeneous distributed systems. We have seen that this technology aims to provide a solution to the interoperability problems by defining a standard for the interfaces of the communicating components, and basing this standard in well known open specifications such as XML.

After, we discussed that one of the main problems of Web Services is the lack of an agreement from the distinct groups that produce different standards for solving similar problems. This creates confusion in the community and delay the adoption of the technology in industry, this is particularly true with advanced transaction management related standards.

We have also presented Web Services Composition, as a solution for automated cross-enterprise application integration, and BPEL was discussed as the current state of the art technology in industry for implementing Web Services composition. We also presented the BPEL component model and a review of its composition activities, since this language will be of our particular interest in the rest of this thesis.

Finally, a discussion was done about how the paradigm of aspect oriented programming

could be applied to composition languages like BPEL, bringing in this way all the advantages of the separation of concerns to the services composition domain.

Chapter 3

Some general purpose models and languages for ATMS

The previous chapter has presented concepts related to Web Services standards, and has given an overview of some state of the art proposals in the web services domain related to: Web Services composition, advanced transaction management with Web Services and aspect oriented programming applied to Web Services.

This chapter will give an overview of different formal models for advanced transaction management. The objective is to apply in the next chapters the relevant insights of these proposals to the state of the art described before. Therefore, we are establishing here a common base of knowledge for future discussion.

3.1 The InterBase project

The InterBase project [ELLR90] [1] is a model created for multi database systems. The main objective of the project was creating a common model for heterogeneous database communications. Therefore, most of its ideas can be applied to the Web Services domain. The projects claims that the traditional requirements of atomicity, consistency, isolation and durability [Gra81] may bee inappropriate when multiple databases are involved, and that the only use of serializability as the correctness criterion is not always appropriate, specially when more complex variables are present, such as timing. Also, a fundamental issue when designing multiple database systems, is the autonomy of all participants. This autonomy has a lot of implications related to performance, or when trying to support atomic transactions.

[1] Unless stated the contrary, this section is a summary of [ELLR90].

The main proposals of the project are:

- The definition of flexible transactions which can tolerate failures, by taking advantage of the fact that a given functionality can be accomplished by more than one system.
- The definition of mixed transactions as a group of transactions composed by compensable and non compensable transactions.
- The concept of timing as a fundamental issue in distributed programming.

In the rest of this section we will develop these ideas with more detail.

3.1.1 Functional Replication

Distributed systems are composed of independently administered systems, and most of the time users are interested in doing operations that involve more than one single system. Problem arises if a local system choses to refuse the execution of a transaction, or is unavailable. Therefore, it is important to have additional available local systems that can supply the same functionality (from the invocator of the request point of view), this concept is referred as *functional replication* [LER89]. Since this idea is common in distributed transaction management, a model that provides constructs for expressing it is needed. In this way, flexibility is increased by allowing the user to specify alternative nodes that provide the same task or provide equivalent data. In this sense, global transactions that are composed of subtransactions that are functionally replicated, are fault-tolerant and can survive local failures of components.

3.1.2 Mixed Transactions

As mentioned before, the traditional transactional properties of atomicity, consistency, isolation and durability (or ACID properties) may become too restrictive in distributed loosely coupled systems. One of the main problems is that transactions in such environments are potentially long lived, which could case serious performance problems. The main issue here is that these transactions can block resources in different components of the system, for a non determined amount of time (releasing them only when the entire transaction commit or aborts). This can lead to deadlocks [Gra81], if more clients are interested in accessing the same resources. Away of dealing with this problem is the definition of compensable transactions. These are transactions that define an associated compensating transaction, that should be invoked if it is necessary to compensate semantically the work previously done.

In this way, long lived transactions can partially commit each subtransaction. In case that a general abort is needed, the invocation process will begin invoking compensating activities to undo the work which has been performed. As we can see, this strategy is incompatible with the isolation requirements of ACID transactions (the global transaction has to reveal partial results before it commits), therefore this constraint has to be relaxed.

A global transaction that groups only subtransactions that can be compensated is called a *saga* [GMS87]. However, in most real life situations not all the subtransactions can be compensated. For example, transactions associated to real actions are not compensable (e.g. a product was manufactured as part of a transaction).

Mixed transactions are defined as such transactions that are composed of both compensable and non-compensable transactions. In these transactions, the subtransactions that are compensable can be allowed to commit before the global transaction has committed, but the non-compensable subtransactions should wait for a global agreement before they can commit. If a global transaction aborts, all the executing subtransactions, and the already executed non-compensable subtransactions are aborted, and the compensable subtransactions are compensated. In this sense, mixed transactions are different from the Sagas model described above, since this model allows only compensable transactions.

3.1.3 Timing

The response times of different components in a distributed system may differ greatly, since they can have distinct processing speed or depend on different hardware resources. Therefore, a traditional distributed transaction is forced to proceed according to the rate of the slowest system, and its execution can take an undetermined amount of time. This can have direct implications on the real value of a transaction.

For example, a transaction for buying airplane tickets in a travel agency that offers discount if the tickets are bought before the end of the week, should not be executed after this time. Or the same client of the system would like to rent a car after two days of the airplane ticket buying, since then it will get another significant discount.

These examples illustrate the importance of involving temporal issues when validating the value of a transaction, and therefore the definition of constructs that support temporal predicates are convenient for programming languages that manage this kind of issues. The main proposal of InterBase related to timing, is the definition of such constructs, in a way that a transaction only can be executed at the time its associated temporal predicates are true. The detailed description of this temporal predicates are out of the scope of this work, this discussion is developed in [ELLR90].

```
1  CONTRACT Business_Trip_Reservations
2    CONTEXT_DECLARATION
3      cost_limit, ticket_price: dollar;
4      from, to: city;
5      ok: boolean;
6      ...
7
8    CONTROL_FLOW_SCRIPT
9      ...
10     S3: Flight_Reservation(in_context: airline, flight_no, date, ...);
11     S4: Hotel_Reservation(...)
12     IF(ok[S4])
13       THEN S5: Car_Rental(...)
14       ELSE S6: Cancel_Flight_Reservation(...)
15     S7: Print_Documents(...);
16   END_CONTROL_FLOW_SCRIPT
17 END_CONTRACT Business_Trip_Reservations
```

Listing 3.1: A ConTract Script

3.2 The ConTract model

The ConTract model [WR92] [2] provides a formal basis for defining and controlling long-lived, complex computations. A ConTract is defined as a *consistent and fault tolerant* execution of an arbitrary sequence of predefined actions (*steps*) according to a control flow description (*script*). In this model, scripts describe the structure (control flow) of a complex activity, while steps implement its algorithmic parts. The consequence of this, is that there are at least two levels of programming in this model.

As an example of these two levels of programming, we show a ConTract script in listing 3.1 and a ConTract step in listing 3.2. Listing 3.1 shows the control flow execution of a certain number of steps (from line 8 to line 16). The implementations of the steps are not showed in the control flow script, since as stated above they are developed in other independent modules. The listing 3.2 shows the implementation of the step *Flight_Reservation*, where an access to a database is performed (from lines 13 to 17). In this way, steps can be developed independently of the script where they will be composed, favouring reuse.

One fundamental contribution of the ConTract model in our research domain, that is not discussed in other ATMS proposals, is that this model defines clearly the consistency and fault tolerance requirements that a ConTract like engine should have. This includes additional fault tolerance considerations that a composition engine must take into account, in order to be able to accomplish a compensation process successfully.

[2] Unless stated the contrary, this section is a summary of [WR92].

```
STEP Flight_Reservation
  DESCRIPTION: Reserve a seat of a flight ...
  IN airline: STRING;
        flight_no: STRING;
        date: DATE;
  ...


flight_reservation()
{ char* flight_no;
  int seats;

  EXEC SQL
       UPDATE Reservations
       SET seats_taken = seats_taken + 1
       WHERE ...
  END SQL
  ...
}
```

Listing 3.2: A ConTract Step

Since in ConTract a script is basically a composition of steps, a Web Services composition engine can be considered in this way a ConTract engine. Therefore, the same consistency and fault tolerance requirements of ConTract, can also apply in the domain of Web Services composition.

In the ConTract model, a ConTract Manager provides the application independent system services, which exercises control over the execution of the scripts. We will describe in more detail the ConTract manager, the ConTract script and the ConTract steps in the remaining of this section.

3.2.1 The ConTract manager

The ConTract Manager is in charge of administering the runtime services of a distributed system that follows the ConTract model. It provides the execution control of the application, implementing an event oriented flow management by using some sort of predicate transition net to specify activation and termination conditions for the steps in a script.

This manager is able to dynamically managing the control flow between transactions. Therefore, a script execution can be suspended, migrated to other machine, and resumed later when needed.

Management of failures

The ConTract manager is aware of possible exceptional cases and problems during the execution of scripts, and therefore it provides a failure model where:

- A system failure may not destroy an entire computation.
- An application as a whole is forward recoverable (e.g. the global data set can be reconstructed by applying forward recovery log records, to an earlier copy of this data set).
- Not only the database should be reconstructed, but also the local state of the application (the computation itself is a recoverable object).
- In case a node fails completely, a secondary ConTract manager on another node has to take over the interrupted execution.

Management of Compensation

Compensating activities in a distributed environment can have a lot of side effects. Some of the issues that has to be taken into account for the ConTract manager are:

- The global effects of a canceled ConTract cannot be undone by simple restoring before images, rather they have to be compensated by a semantic undo.
- Since under certain circumstances this compensation may affect other ConTracts, a ConTract step should be able to specify its isolation requirements.

Therefore, since the compensation of steps can be a cumbersome task itself, the use of entire scripts for compensation can be a useful alternative to single compensation steps only.

Also, in order to provide strong compensation capabilities, the following actions should be considered when executing a ConTract script:

- After the execution of a step, all the input data needed for its compensation step should be computed and saved.
- All the global objects accessed for a step and relevant to its compensation have to exist until ConTract's termination. (e.g. accessed database files).

- In case of a failure in a compensation step (after a n number of intents), the ConTract manager should notify a human system administrator.

- If neither automatic nor manual correction does help, compensation continues with an error message, but without any further recovery attempts.

- If a compensation step originates that a step executed in other ConTract becomes invalid, the affected ConTract has to backtrack his history until the mentioned step and execute it again.

- For doing that, the system has to keep track of all the steps which have used a compensated object after the original update of this object and before its compensation.

3.2.2 The ConTract script

As stated at the beginning of this section, in this model the coding of steps is separated from defining an application's control flow script. The reason for this separation is that, since large applications are usually defined by combining existing steps, an appropriate programming model has to support code reusability, and this reusability is accomplished in ConTract if the scripts can be potentially composed from different predefined steps.

The scripts are responsible of both providing a logical representation of the local computational state of the application, and for administering its control flow. The local state is represented by a set of private data called *context*, and the application control flow can be modeled using elements like *sequence* (sequential execution), *parallel* (concurrent execution) or *loop* (repetition) amongst others.

In addition, the control flow description has to specify what should be done when a resource conflict occurs, since sometimes the options of waiting or doing an entire rollback are not appropriate choices.

3.2.3 Steps

Steps are basic units of execution in a ConTract. They involve the execution of a short ACID transaction. Steps should be coded without worrying about things like managing asynchronous computations, distribution, synchronization or failure recovery, since this concerns are managed by the control flow script.

Since a ConTract is a long lived transaction that is not in itself an ACID transaction, in the context of steps execution the term *commit* should be used carefully, instead *externalization* is preferred. Externalization is the fact of showing partial updates of subtransactions

although the long lived transaction that composes them has not yet committed. Therefore, the changes are visible to external entities as soon as the subtransaction commits. A step represents a subtransaction that although can update the state of a database, this work could be revoked later (e.g. with a semantic undo) if the process defined in the ConTract script aborts.

3.3 The ACTA formal model

The objective of ACTA [CR91][CR91] [3]is to provide a formal model that allows us to specify a large number of ATMS models. In ACTA, databases are considered as repository of objects and transactions are considered as invocations of operations on these objects (also called *object events*) or on transaction management primitives (also called *significant events*).

In the model, these events can be executed concurrently and such executions are logged in the sequence in which they are occurred, creating a history. ACTA can reason about this event history for defining the properties of an ATMS. This is possible since the definition of axioms declaring certain constraints on this history are enough for specifying most ATMS. Cited from [CR91]: "The correctness properties of different transaction models can be expressed in terms of the properties of the histories produced by these models".

Three kind of constraints can be declared, they are:

- An event X can be constrained to only occur after an event Y.
- A particular event can only occur in a history if a certain condition is satisfied.
- A particular event has to occur in a history if a certain condition is satisfied.

Transactions models specified only in terms of axioms that declares these constraints, although powerful for specifying a great amount of ATMS, have an important drawback: the related specifications are very large and complex. To address these problems, a number of abstraction mechanisms have been defined [CR91] [CR92]: *dependencies*, *views* and *delegation*. In the rest of this section we will explore these abstractions in more detail.

3.3.1 Dependencies

Different transactional models express constraints between different significant events of their transactions. The abstraction of *dependencies* is one of the elements of ACTA for

[3]Unless stated the contrary, this section is a summary of [CR92] and [Fab05].

abstracting the definition of such constraints. Twelve kind of dependencies have already been defined in [CR91] and [CR92], we will describe as an example the followings:

Begin-on-Commit Dependency.- A transaction *Tj* has a Begin on Commit Dependency with a transaction *Ti* (or: *"Tj BCD Ti"*) when: *Tj* cannot begin executing until *Ti* commits.

Begin-on-Abort Dependency.- A transaction *Tj* has a Begin on Abort Dependency with a transaction *Ti* (or: *"Tj BAD Ti"*) when: *Tj* cannot begin executing until *Ti* aborts.

Compensation Dependency.- A transaction *Tj* has a Compensation Dependency with a transaction *Ti* (or: *"Tj CMD Ti"*) when: if *Ti* aborts, *Tj* must commit.

3.3.2 Views

In a number of ATMS, the isolation between different transactions is relaxed, in such a way that a particular transaction can see the results of one or more transactions that are still executing. For expressing this visibility, two kind of abstractions are defined in ACTA: the view and access set.

The view set is an abstract repository of all the objects that are visible to the transaction. The access set contains all the objects that have been accessed or created by the transaction.

Using these abstractions, we can easily manipulate the content that is visible for a particular transaction, providing in this way a useful mechanism for ATMS. One well known example is *Nested Transactions*, in this model a parent transaction has a certain number of child transactions. Each child transaction need to view the intermediate work performed by its parent, recursively all the way up to the root transaction.

3.3.3 Delegation

Delegation is the capability of a particular transaction to delegate its committing, or aborting of a particular number of its objects (called a *Delegate Set*), to another transaction. For demonstrating the utility of this abstraction we will refer to the Nested Transaction model again. In This ATMS, when a child transaction commits, it delegates the final commit of its data to its parent. Only when the parent commits, the data originally modified by the child transaction is committed.

```
1 org.tree.ChildNode.childMethod(float, String){
2 }
```

Listing 3.3: Static naming in KALA

3.4 The KALA language

KALA [Fab05] [4] is an aspect oriented language based on the ACTA formal model. It stands for Kernel Aspect Language for ATMS and was created as a tool for specifying separately Java methods and their transactional properties. Since KALA is based on ACTA, it assures that a wide variety of ATMS can be implemented using this language.

One of its main objectives is to provide an easy way for defining and modifying ATMS code. This is accomplished since this code is modularized in an aspect, and it brings all the advantages of separation of concerns.

3.4.1 Naming and grouping

Static Naming

KALA defines static naming support, for specifying which are the Java methods that are advised by a KALA aspect. This name is written using the class name and the complete signature of the method, separated by a dot.

Listing 3.3 shows an example of an empty block of declarations for a method *childMethod* declared in the class *org.tree.ChildNode*.

Dynamic Naming

A dynamic naming is also required, since some transactional properties can only be determined at runtime. For example, in nested transactions, a child transaction has certain transactional relationships with its root transaction, but the identity of this root transaction is only known at execution time. In KALA, the *name* construct can be used for creating a new name associated to a transactional id. The *alias* construct can be used to locally make a reference to an already existing transaction id at the naming service.

Listing 3.4 shows an example of dynamic naming. Line 2 associates the transaction id of the method *fatherMethod* (denoted by *self*) with a logical name, in this case the thread

[4] Unless stated the contrary, this section is a summary of [Fab05].

```
1 org.tree.RootNode.fatherMethod(int){
2   name(self <Thread.currentThread()>);
3  }
4  org.tree.ChildNode.childMethod(float, String){
5   alias(father <Thread.currentThread()>);
6  }
```

Listing 3.4: Dynamic naming in KALA

identifier. Line 5 declares a binding between the logic name *father* and the transaction already defined in the name service, with the id of the thread. For additional discussion of naming issues in KALA see [Fab05].

Group Support

The transactional constructs of KALA can have as parameters entire group of transactions, instead of only individual ones. In addition, the members of these groups can be determined at runtime, this is required since certain operations take as arguments groups of transactions, that cannot be determined statically (e.g. in the context of Nested Transactions: the number of child transactions of a parent transaction could be dependant on the application logic). For additional discussion of groups and its related constructs see [Fab05].

3.4.2 Dependencies, Views and Delegation

A KALA aspect specifies declaratively the transactional properties of a Java method, using constructs that directly abstract concepts from ACTA. Therefore, the ACTA concepts of dependencies, views and delegation can be easily associated with a method (considered as a transaction) using KALA. The declarations of the transactional properties of methods, can be also set to coincide with any of the significant events of a transaction (e.g. begin, commit and abort).

Therefore, a declaration of transactional properties for a given method contains begin, commit and abort statements, and each of these statements contains a nested block of KALA constructs that specify dependencies, views and delegations of the method.

3.4.3 Secondary transactions

Multiple ATMS require to run some transactions, that conceptually are separated from the main control flow of the application, when certain constraints are satisfied. These transactions receive the name of *secondary transactions*.

KALA eases the use of secondary transactions, since the *autostart* construct is provided for indicating that a particular method should be invoked as a secondary transaction. This element can also be associated with transactional specifications, this properties will be applied when the secondary transaction is executed.

3.4.4 ATPMos

Before finishing the discussion of KALA, we would like to briefly discuss one implementation detail of this language: the underlying transactional processing monitor used, named *ATPMos*. This is because this tool will be used for us later in chapter 5, when implementing an ACTA based Web Services composition engine using this TP monitor.

ATPMos is a TP monitor that in addition to support traditional transactions, is well suited for different ATMS, since it implements support for the ACTA primitives. ATPMos has been built for monitoring Java Beans objects. Since it is assumed that all the state accesses in these objects is realized using setters and getters methods, ATPMos can act as a layer between these methods and the invocator, enforcing the constraints of the transactional model used. Textually from [Fab05]: "calling a getter or setter on an Entity Bean [5] when in a transaction requires that first a call is made to ATPMos. These calls inform ATPMos that a read or write will be performed, and contain as parameters the current transaction identifier, the object and the field on which the read or write occurs."

As we will see in chapter 5, the capability of ATPMos for supporting ACTA abstractions, specially dependencies, will make this TP monitor a fundamental keystone on the design of our solution.

3.5 Conclusions

We have presented a review of some relevant proposals for advanced transaction management. During this discussion, we have presented some of the fundamental concerns that

[5] The first version of ATPMos was conceived for working with Entity Beans, the last version works with the more general Java Beans.

need to be solved for advanced transactional models, and we have reviewed some alternative solutions.

The *ConTract* model gave us a clear vision of strategies for composing atomic transactions in long lived ones, it also presented the fundamental services that a middleware implementing this strategy should provide, putting a special emphasis in the consistency and fault tolerance requirements.

The model proposed by *Interbase* made an emphasis on some critical concerns in distributed systems, and give us important clues about how to solve them.

The *ACTA* formal model showed us the power of a general mechanism for expressing ATMS, and emphasized the importance of an extensible model as a tool, since possible future problems can depend on new non-predictable business situations.

Finally, *KALA* proposed a way of implementing the concepts of ACTA in an aspect language, contributing both with the expressiveness of the ACTA model and the advantages of the separation of concerns. It also provided an implementation of a powerful transactional monitor with support for ACTA that will be used for us in chapter 5.

Chapter 4

A requirement analysis of a proposal for ATMS in Web Services Composition

4.1 Motivations

The main objective of this thesis is to propose an enhancement to the way advanced transactional concerns are currently managed in web services composition languages. Since BPEL (described in chapter 2) is the state of the art for Web Services composition, our transactional model should be able to surpass the one used in BPEL. Therefore, in order to facilitate comparisons, we need a model that from a composition point of view has a lot of similarities with BPEL, but that employs improved transaction management strategies. This chapter realizes an analysis of the requirements that this model should fulfill.

The steps we have followed for accomplishing it are:

- We first describe the generalities of our composition architecture.
- We define the infrastructural and transaction management requirements that our model should provide.
- For each one of the defined transactional requirements, we will justify why they are relevant in our domain and how the proposals discussed in previous chapters have tried to solve these problems.
- For each concern, we always present the BPEL solution, since this is the model that we want to improve. The main ideas are abstracted from the language details, in order to use a common terminology for all the discussed solutions.

Once concluded our analysis, we will use later in chapter 5 the ideas resulting from this research, for defining and implementing a solution that is able to express the mentioned transactional concerns, incorporating relevant insights from the advanced transactional models discussed before and including enough extension points for working in the future with new kind of transactional requirements.

4.2 Overview of our conceptual model for advanced transaction management

4.2.1 General composition architecture

One of the objectives of our model is to be able of expressing the main concerns that can be currently expressed in BPEL. The starting point for accomplishing it is providing a model that is equivalent, from a functional point of view, with the composition model of this language. The reason for this objective is that if we want to be able later, to propose improved ways for managing transactional concerns in BPEL like languages, we have to take into account that the way transactions are related between each other is dependent on the composition model used. Therefore, we need to choose a composition model that defines relationships between transactions in a similar way than BPEL.

BPEL follows a hierarchical composition of activities. It is possible to define atomic activities that perform simple tasks (e.g. web service invocations), but also to define structural activities that contain or compose more than one activity, forming in that way a hierarchy. These structural activities define the order (control flow) in which the nested activities will be executed, or the conditional execution of such nested activities. Examples of structural activities are BPEL *sequence* or *flow* activities. Structural activities can also be used as if they were simple activities, for example, they can be nested inside other structural activities.

Therefore, our proposal will also follow a model based on hierarchical composition of activities where, similar to BPEL, structural and atomic activities can be defined. Once we have defined the way in which the activities in our model will be composed, we have to define which are the basic infrastructural services that it needs in order to be functional, and the transactional concerns that it should implement.

In the rest of this section we will detail both the main infrastructure and transaction management requirements of web services composition languages. Since our interest is on transaction management, we will not focus on the infrastructure requirements (although an implementation of such requirements is provided in the prototype presented in chapter 5). Rather, in the next section, we will center our attention in how different proposals

solve our transactional requirements, with the intention of using that information later in chapter 5, for defining and implementing model that is able to satisfy them, and that has enough flexibility to be extended in the future, in order to deal with new transactional requirements. As mentioned before, BPEL will be always present in this exploration, since our final objective will be defining ways for improving the management of transactional concerns in this language.

4.2.2 Infrastructure and transaction management requirements

consistency and fault tolerance

Service composition frameworks can have a big amount of particularities and features, for example it could be desirable the implementation of the fault tolerance mechanisms specified in the ConTract model [WR92] discussed in chapter 3, or that the binding between an invocation activity and a concrete service provider definition is solved at execution time. However, we have focused our survey of requirements along the two dimensions that are more relevant for our purposes:

- *Infrastructure support requirements.*- The fundamental services that should be provided by a framework in order to administer workflow instances, manage instance state, handling events and the appropriate correlation of external messages with executing workflow instances.
- *Transaction Management requirements.*- The definition of the transactional concerns that are needed to be administered by our framework.

We will expand these requirements in the rest of this section.

Infrastructure support requirements

The infrastructure support requirements define the kernel of a framework for services composition. They specify the basic functionality that should be provided by it.

These requirements are:

- Management of workflow instances life cycle.- A workflow manager should be responsible of creating and eliminating workflow instances as needed (recall we discussed workflow instances in chapter 2).

- Management of events.- Once created, the workflow instances should be notified of events, like the arrival of messages or the timeout of waiting activities.

- Management of correlation issues.- In object oriented systems, every message is always sent to a specific, well defined object. Working with workflow systems with web services interfaces has a fundamental difference with this assumption. Messages are first received by a workflow manager, and this manager has the responsibility of determining which is the correct workflow instance that should receive the message. How to accomplish this correlation is an important issue, since cannot always be assumed the existence of a sophisticated conversational infrastructure [ACD+03].

- Management of instances state.- Workflow instances need some way of storing state. Since they manage state, their response to external events is most of the time dependent on it. State can be also used for determining how the control flow of the process should be executed.

Transaction management requirements

Once our composition architecture is described (hierarchical composition) and the basic infrastructure requirements our framework should meet, we will define the transactional related concerns currently managed by BPEL like languages. A summary of these features is presented here:

- Synchronization of activities.- This concern is about the necessity of synchronizing, at a fine granular level, the activities that are part of a business process. This is: what is the order in which these activities should be executed. Frequently, synchronization dependencies between activities are a direct consequence of related data dependencies between them.

- Compensation of activities.- The definition of compensating activities is an essential concept in long lived transactions. They allow us for example, to make early commits on subtransactions (and therefore releasing their resources) belonging to a long lived transaction. In case that the main transaction eventually fails, the compensating transactions of all the already invoked subtransactions should be executed. This concept was discussed in more detail in chapter 3.

- Functional replication of activities.- When the request for a service fails, frequently an alternative service provider can be selected. From the point of view of the service invocator, both alternatives are functionally equivalent. This is a common concept in a service-oriented architectures, and it has been already discussed in chapter 3.

- Exception handling.- Workflow languages should offer appropriate constructs for dealing with exceptional cases or irrecoverable problems. Also, mechanisms for capturing

events that can occur asynchronously with respect to the control flow of the process are important (e.g. a client sending a message canceling the entire process).

- Management of timing.- In distributed systems the management of timing is a fundamental issue. A component has to react appropriately if a request is not answered in a determined amount of time (timeout configuration), or we would like to be able of scheduling an activity for beginning in a particular date or after a specific amount of time.

4.3 Analysis of the transaction management requirements

In this section we will discuss in detail the transactional requirements defined before. After justifying the relevance of each requirement, our strategy will be to examine the different solutions from relevant proposals, keeping in mind our objective of defining and implementing later in chapter 5, a common model for solving these problems. In the rest of this section, we will develop this strategy:

4.3.1 Synchronization requirements

A business process can involve the concurrent execution of a large number of transactions. Some of these transactions can have data dependencies with other transactions, for example, the data produced for a particular transaction is needed as an input for another. It is possible also that although a data dependency is not present, the business rules determine that one or more transactions cannot begin until another one has successfully finished (for example, an authorization requirement).

Some of these problems can be solved using control flow structures for specifying the order in which activities should be executed (e.g. a *sequence* activity in BPEL), but in other cases, particularly where high concurrency is present, more sophisticated synchronization mechanisms are needed.

Different solutions have been proposed, some of them based in expressing these dependencies in terms of the significant transactional events of the activities that participates in the dependency. However, other models contribute with additional ideas, such as the inclusion of boolean logic and state variables in the predicates that defines the dependencies.

In the rest of this section we describe such models in more detail.

Dependencies in terms of significant events

Some models propose the definition of dependencies amongst pairs of activities (transactions) in terms of the significant events of these transactions. Significant events are them that occurs when a transaction management primitive is invoked (e.g. *begin*, *commit*, *abort*) [CR92]. One model that follows this strategy is the ACTA framework (already discussed in chapter 3), which also has a special focus in presenting extensibility properties. This extensibility is provided because although some ACTA dependencies (twelve) have been already defined in [CR91] and [CR92], new kind of dependencies can be always added to the model.

We present only the ACTA proposal here, since other proposals that include dependency mechanisms (such as the one presented in [ELLR90], describing only *negative* and *positive* dependencies) are surpassed by the dependency model of ACTA. In addition, the ACTA framework is one of the best well known models in the ATMS community.

As an example of the properties of the ACTA dependency model, we briefly discuss an example.

The *Commit Dependency* is described in [CR92] with the following statement:

A transaction *Tj* has a *Commit Dependency* with a transaction *Ti* (or: "*Tj CD Ti*") when: if *Ti* and *Tj* commit, *Ti* must commit before *Tj*.

The significant events that are mentioned in this definition, are the commit of transactions *Ti* and *Tj*, and what it is basically said is that in the history of significant events, the commit of *Ti* should precede the commit of *Tj*.

We can summarize the properties of the ACTA dependency model in the following way:

- Dependencies are established amongst pairs of transactions.
- Dependencies are established in terms of significant events of such transactions.
- The number of dependencies that can de defined is virtually unbounded, providing in this way an extensible model.

Dependencies in terms of logic expressions

Now we will add more expressiveness power to the way transactional dependencies can be defined. Basically, we will offer a different perspective to the ACTA dependency model, where dependencies can be established only amongst pairs of activities. Using logic expressions, we could employ more than a pair of dependencies in a dependency predicate. One

```
1  ...
2  <flow suppressJoinFailure="yes">
3    <links>
4      <link name="buyToSettle"/>
5      <link name="sellToSettle"/>
6    </links>
7    ...
8    <receive name="getBuyerInformation">
9      <source linkName="buyToSettle"/>
10   </receive>
11   ...
12   <receive name="getSellerInformation">
13     <source linkName="sellToSettle"/>
14   </receive>
15   ...
16   <invoke name="settleTrade"
17     joinCondition="bpws:getLinkStatus('buyToSettle')
18       and bpws:getLinkStatus('sellToSettle')">
19     <target linkName=" buyToSettle "/>
20     <target linkName=" sellToSettle "/>
21   </invoke>
22   ...
23 </flow>
24 ...
```

Listing 4.1: Synchronization dependencies using boolean logic expressions

example of this is the model employed by BPEL, where synchronization dependencies for BPEL activities can be expressed using boolean logic expressions in terms of the *outcomes* of other activities, established using the standard attribute *joinCondition*.

Listing 4.1 shows a BPEL fragment, where the activity *settleTrade* (line 16 to 21) expresses a synchronization dependency with both *getBuyerInformation* (line 8 to 10) and *getSellerInformation* (line 12 to 14). This synchronization dependency is established using a logic boolean expression (lines 17 and 18) in terms of the outcomes of synchronization links. In our particular example, the activity *settleTrade* will wait until for the execution of *getBuyerInformation* and *getSellerInformation*. After that, if the outcomes of the links associated with these two activities are positive (since the *joinCondition* attribute is an AND expression) it will be executed, if this is not true, the activity will be ignored or a fault will be thrown, according to the configuration of the BPEL process. BPEL synchronization links will be discussed with more detail in chapter 7. Since the semantic of the other BPEL elements present in the listing have been already described in chapter 2, we will not repeat this discussion here.

We can summarize the properties of this proposal in the following way:

- Activities can express synchronization dependencies in terms of the outcomes of other activities.
- When a synchronization dependency is not fulfilled, in certain cases abandoning the activity is correct, but in others the signaling of a error is appropriate.

In the next topic we will continue expanding the BPEL transactional model, now discussing the possible outcomes of the transactions involved in a dependency.

Dependencies in terms of state variables

This idea goes one step further, since now we will define the possible transactional outcomes of activities not only in terms of the transactional result of executing such activity, but also we will include the possibility of assigning a logical outcome based on the values of state variables of the process. This concept is also present in BPEL, where source links can use the *transitionCondition* attribute in order to declare a specific outcome, which is a boolean predicate in terms of state variables. The default value of the *transitionCondition* attribute is *true*.

Listing 4.2 shows an example of transactional outcomes in terms of state variables. The *receive* activity (lines 9 to 14) declare two source links: *receive-to-assess* (lines 10 and 11) and *receive-to-approval* (lines 12 and 13). Each of these source links declare different outcomes expressed with boolean conditional expressions in terms of state variables of the process.

One particularity of the BPEL model based on source and target links, is that each link must be used exactly once as a source and exactly once as a target. Therefore, if a group of activities have a synchronization dependency with a common activity, different links must be defined for each of these dependencies, although the transition condition is the same for all of them.

Another point of interest is that, although state based dependencies could be considered a powerful expression mechanism, it is also true that it is redundant. The same can be expressed with traditional alternative constructs. Listing 4.3 shows the same example of listing 4.2, but dependencies are written in a different way, with the expression of conditional dependencies on data values written using *switch* activities.

In this new version, we can see that the source links nested in the receive activity (lines 9 to 12) do not declare *transitionCondition* attributes anymore. Instead, *switch* activities (lines 14 to 19 and 21 to 26) express the same idea. In this example, the two *switch* activities will not begin until the outcome of the links *receive-to-assess* and *receive-to-approval* are determined, after the execution of the *receive* activity. Since no transition

```
<process ... suppressJoinFailure="yes">
  ...
  <flow>
    <links>
      <link name="receive-to-assess"/>
      <link name="receive-to-approval"/>
    </links>
    ...
    <receive ...>
      <source linkName="receive-to-assess"
        transitionCondition="bpws:getVariableData('request','amount') &lt; 10000"/>
      <source linkName="receive-to-approval"
        transitionCondition="bpws:getVariableData('request','amount')>=10000"/>
    </receive>
    ...
    <invoke ...>
      <target linkName="receive-to-assess"/>
    </invoke>
    ...
    <invoke ...>
      <target linkName="receive-to-approval"/>
    </invoke>
    ...
  </flow>
  ...
</process>
```

Listing 4.2: Synchronization dependencies in terms of state variables

```
<process ... suppressJoinFailure="yes">
  ...
  <flow>
    <links>
      <link name="receive-to-assess"/>
      <link name="receive-to-approval"/>
    </links>
    ...
    <receive ...>
      <source linkName="receive-to-assess"/>
      <source linkName="receive-to-approval"/>
    </receive>
    ...
    <switch>
      <target linkName="receive-to-assess"/>
      <case condition="bpws:getVariableData('request','amount') &lt; 10000">
        <invoke .../>
      </case>
    </switch>
    ...
    <switch>
      <target linkName="receive-to-approval"/>
      <case condition="bpws:getVariableData('request','amount') >= 10000">
        <invoke .../>
      </case>
    </switch>
    ...
  </flow>
  ...
</process>
```

Listing 4.3: An alternative to synchronization dependencies expressed in terms of state variables

conditions are specified, they will take the default value of *true*, causing that both switch activities be executed after the finishing of the receive activity. The switch activities will evaluate a boolean predicate in order to determine if the nested invoke activities should be executed. This conditional execution has the same effect observed in listing 4.2 when transition conditions were used instead.

Both alternatives have a lot of similarities from a functional point of view, however there is one minor semantic difference, related to the time the conditional boolean predicates are evaluated: When using transition conditions, the boolean predicate will be evaluated after the execution of the activity that declares the source links. Conversely, if switch activities replace transition conditions, the boolean predicate will be executed after the dependencies have been resolved, and the switch activity has begun.

We have finished the discussion about the different properties of this model, we end this section with a summary of them:

- The transactional outcomes of activities can be a function of state variables of the process.
- Activities can have more than one outcome, possibly defined using different logic expressions.
- Each outcome of an activity can can have a logical name. This name is used later for establishing synchronization dependencies.
- A dependency expressed in terms of state variables of a process can be refactored as a dependency without this constraint, with minor semantic alterations.

4.3.2 Compensation requirements

The definition of compensating activities is an essential concept in long lived transactions. For example, in the Sagas model discussed in chapter 3, they allow us to make early commits on subtransactions (and therefore releasing their resources) belonging to a long lived transaction. In case that the long lived transaction eventually fails, the compensating transactions of all the already committed subtransactions should be executed. In addition, when the compensation process has begun, the compensating transactions must commit, since an abort of compensation could lead to an inconsistent state of the process. The compensation concepts are discussed with more detail in chapter 3.

In this section we revise two strategies for defining compensation activities. First we demonstrate how using a certain number of ACTA dependencies a compensation relationship can be described, and after we offer the review of a proposal with a single construct for expressing the whole idea.

Defining compensation activities with dependencies

As reviewed in chapter 3, ACTA dependences are well suited for expressing the concept of compensating activities. [CR92] has already defined a compensation dependency in the following way:

A transaction *Tj* has a *Compensation Dependency* with a transaction *Ti* (or: "*Tj CMD Ti*") when: if *Ti* aborts, *Tj* must commit.

However, this dependency is useful for establishing only one of the constraints that are present when defining a compensation transaction: the concept that the compensating activity should always commit if the parent transaction aborts.

The other additional two constraints that this dependency does not express, and that are related to the time the compensating activity should begin, are:

- The compensating transaction should begin only after the compensated transaction has committed (an ACTA *Begin on Commit Dependency*).
- The compensating transaction should begin only if the parent long-lived transaction has aborted (an ACTA *Begin on Abort Dependency*).

These dependencies have been already mentioned in chapter 3, therefore we do not present their definitions here in detail.

One additional requirement of this model is that the language that implements it, should provide a mechanism for defining compensation transactions, in such a way that this transaction is not part of the normal flow of a particular process. Instead, they should be executed only when their associated begin dependencies are satisfied.

The requirements for establishing a compensation transaction using ACTA dependencies can be summarized as:

- Establishing appropriately the three dependencies described above: the ACTA dependencies *Compensation*, *Begin on Commit* and *Begin on Abort*.
- A mechanism for specifying that the compensating transactions are not part of the normal execution flow of the application.

Defining compensation activities with specialized constructs

Until now we have presented separately all the concepts related to compensation. However, dealing with all of them can be cumbersome in real applications, since compensation is a concept that is applied frequently in most distributed systems. Therefore, specifying each time a compensation activity in terms of its basic related concepts can be a tedious and error prone work.

In order to avoid it, some proposals provide dedicated constructs for expressing straightforwardly this idea. BPEL for example, provides the element *compensationHandler* for expressing this idea. This construct is showed in listing 4.4 (example taken from the

```
1 <scope>
2   <compensationHandler>
3     <invoke operation="CancelPurchase" .../>
4   </compensationHandler>
5   <invoke operation="SyncPurchase" .../>
6 </scope>
```

Listing 4.4: A dedicated construct for compensation handler in BPEL

BPEL specification [ACD+03]), lines 2 to 4. The scope showed in the listing, has only one activity (line 5), in the form of an invocation operation. A compensating activity is declared in the *compensationHandler* element (line 3). In this code, it is clear that the compensating activity is not part of the business execution of the process, but it should be invoked as part of a compensation process. This process is invoked implicitly, by default, each time that a fault is thrown and not caught in a BPEL scope. In this case, the compensation handlers of all the nested scopes will be invoked in the inverse order of execution of such scopes, before the fault is rethrown to the enclosing scope.

Another interesting feature of the BPEL approach, is the fact that the compensation process can be invoked explicitly, though with certain limitations. For example, such invocation can only be done inside a compensation or fault handler that immediately encloses the scope to be compensated. This option allow us to overwrite the default compensation behaviour, introducing changes such as the modification of the order in which compensation activities belonging to nested scopes will be invoked.

For additional discussion about the BPEL compensation process we refer to the BPEL specification [ACD+03].

Summarizing the main proposals discussed here:

- A special construct for expressing the entire concept of compensation can be useful, since compensating concerns are very common in distributed applications.
- Once a single way for defining compensating activities is defined, an implicit compensation behaviour is provided, but with the possibility of explicitly overwriting this behaviour.

4.3.3 Functional replication requirements

In the context of loosely coupled distributed systems, when the request for a service fails, frequently an alternative service provider can be selected. From the point of view of the service invocator, both alternatives are functionally equivalent. This is a common situation

in a service-oriented architectures, and it is highly desirable to be able to capture this idea in transactional constructs.

In this section we will discuss two common approaches for doing this, they are:

- The definition of functional replication using dependencies.
- The definition of functional replication using exception handlers.

Defining functional replication with dependencies

Previous works have considered the definition of transactional dependencies as a mechanism for expressing functional replication concerns. [ELLR90] mentions the use of *negative* dependencies, for indicating that an activity should wait until another transaction has been executed and failed in order to start. Also, the ACTA framework provides the *Begin On Abort* dependency for specifying the same concept, this dependency has been talked about previously (in the section about compensation), therefore we will not repeat its definition here.

Similar to the compensation concern, another requirement of this model is that the language that implements it, should provide a mechanism for defining functional replication transactions, in such a way that this transaction is not part of the normal flow execution of a particular process. Instead, they should be executed only when their associated replicated transactions fail. Note the similarity with the compensation concern, where a mechanism for specifying that the compensating transaction is not part of the normal flow execution is also needed.

Summarizing the requirements for managing functional replication concerns using dependencies, we need:

- Establishing appropriately the *Begin on Abort* dependency described above.
- A mechanism for specifying that the functional replication transactions are not part of the normal execution flow of the application.

Managing functional replication with exception handlers

Other proposals, like the one used in BPEL, manage functional replication concerns as traditional exception handling. They do not create dedicated constructs for this concern or define a specialized mechanism. The problem with this strategy is that the functional

```
1 <scope>
2   <invoke partnerLink="shipping" operation="requestShipping" ... />
3   <faultHandlers>
4     <catchAll>
5       <invoke partnerLink="shippingAlt" operation="requestShipping" ... />
6     </catchAll>
7   </faultHandlers>
8 </scope>
```

Listing 4.5: Functional replication in BPEL using fault handlers

replication concern will be tangled with other error management concerns, obstructing in this way development and maintainability of the system.

Listing 4.5 shows the BPEL strategy for implementing functional replication. A particular activity is executed, in this case the invocation activity showed in line 2, and an associated fault handler is defined (lines 3 to 7). This fault handler defines an activity that should be executed as an alternative to the failed one. In this particular example, a general *catchAll* fault handler is defined, but handlers for specific faults could also be defined.

Summarizing this approach, we have:

- Functional replication can be always described as a case of exception handling.
- However, this tangling of functional replication with other exception management concerns could difficult the development and maintainability of the application.

4.3.4 Timing management requirements

In distributed systems the management of timing is an important issue. Frequently, it is not realistic to assume that all the component systems are operational at the same time. Or even when all of them are available, it could be necessary to postpone the execution of a transaction to a different time (e.g. scheduling an activity for beginning in a particular date or after a specific amount of time). Also, a component has to react appropriately if a request is not answered in a determined amount of time (timeout configuration). This is important since usually, transaction management is only based on serializability (see chapter 3) for evaluating the correctness of a transaction, and the question if whether the transaction has accomplished its objectives in time is frequently ignored. Also, the definition of temporal expressions for expressing the timing concerns described before is a basic starting point for the management of these problems [ELLR90].

Some proposals (e.g. [ELLR90]) define the use of temporal predicates as a medium of dealing with these issues. For example: a transaction can be executed only when its

```
1  <pick>
2    <onMessage operation="inputLineItem" ...>
3      ... <!-- an activity -->
4    </onMessage>
5
6    <!-- set an alarm to go after 3 days and 10 hours -->
7    <onAlarm for="'P3DT10H'">
8      ... <!-- handle timeout -->
9    </onAlarm>
10 </pick>
```

Listing 4.6: Timeout handling in BPEL

```
1  <sequence>
2    <invoke operation = "reserve" ... />
3    <wait until="'2002-12-24T18:00+01:00'"/>
4    <invoke operation = "confirmReservation"... />
5  </sequence>
```

Listing 4.7: Delaying execution in BPEL

associated temporal predicate becomes true.

BPEL also provides constructs related to timing, like the *pick* and *wait* activities or the *eventHandlers* construct. The *pick* activity groups a collection of activities associated with events. Only one of these activities will be executed, and it will be determined for which of their associated events arrive first. These messages can be either a web service message or an alarm (in terms of time) message. This make this construct well suited for handling timeout configurations.

Listing 4.6 shows an example of timeout handling in BPEL using the *pick* construct: the *onMessage* construct declares that a message is waiting (lines 2 to 4), and the *onAlarm* construct declares that a timeout is also configured [1] (lines 7 to 9). Since both activities are nested in a pick activity, only one of them will be executed, and it will depend on which event arrives first.

As mentioned above, another BPEL activity related to timing is the *wait* activity. This activity allows a business process to specify a delay for a certain period of time or until a certain deadline is reached [ACD+03]. Listing 4.7 shows an example of the utility of this activity. In the example, three activities are executed inside of a sequence. Therefore, the *wait* activity showed in line 3, will delay the execution of the *invoke* activity defined in line 4, until the date indicated by its time expression.

Finally, the *eventHandlers* element, similarly to the *pick* activity, can execute activities

[1] The description of the specific temporal predicates used in BPEL is out of the scope of this work.

```
1  <process ...>
2    ...
3    <eventHandlers>
4      <onAlarm for="bpws:getVariableData(orderDetails,processDuration)">
5        ... <!-- a scheduled activity -->
6      </onAlarm>
7      ...
8    </eventHandlers>
9    ...
10 </process>
```

Listing 4.8: Scheduling activities in BPEL

related to events. The difference with the *pick* activity is that this selects one activity from a set, and only executes the selected activity. The *eventHandlers* construct just listens for events, and will execute them as long as they occur. This construct is well suited for scheduling activities in a process. Listing 4.8 shows an example of scheduling using the *eventHandlers* construct. An alarm is defined (line 4) in terms of a temporal expression (*processDuration*) extracted from a state variable (*orderDetails*). The activity defined inside the construct (line 5) will be executed when the time event of the alarm arrives.

This concludes our discussion of timing, summarizing our exposed ideas, we have:

- Management of timing should include the definition of temporal predicates as a base for expressing timing concerns.
- Timing should include the schedule of activities, both in terms of dates or after a specified amount of time.
- Timing should include the possibility of establishing relationships based on time amongst different activities, such as a delay between the execution of them.
- Timing should include management of timeout concerns, since it cannot be assumed that all the components in a loosely distributed system will accept a request or will be always available.

4.3.5 Exception handling requirements

Workflow languages should offer appropriate constructs for dealing with exceptional cases or irrecoverable problems. Also, mechanisms for capturing events that can occur asynchronously with respect to the control flow of the process are important, and frequently

```
1  <faultHandlers>
2    <catch faultName="x:foo">
3      ...
4    </catch>
5    <catch faultVariable="bar">
6      ...
7    </catch>
8    <catch faultName="x:foo" faultVariable="bar">
9      ...
10   </catch>
11   <catchAll>
12     ...
13   </catchAll>
14 </faultHandlers>
```

Listing 4.9: Special constructs for exception handling in BPEL

they are related with the management of exceptional cases (e.g. a client sending a message canceling the entire process).

BPEL provides the dedicated construct *faultHandlers* for managing exceptional cases. Before describing an example, we would like to recall that in BPEL the functional replication concerns are part of the exception handling, therefore, a lot of situations expressed with BPEL fault handlers, can be also expressed in a model that is based on the ACTA framework, since we have already demonstrated the utility of it for describing concerns related to functional replication.

Listing 4.9 shows an example of this construct, with four different kinds of fault handlers. The first fault handler (lines 2 to 4) specifies a fault using the logic name of the fault. The second fault handler (lines 5 to 7) specifies a fault using the name of the fault variable associated with it, the third fault handler (lines 8 to 10) combines the two precedents, and specifies a fault using both the fault logic name and the name of the fault variable associated with it and finally the fourth fault handler (lines 11 to 13) specifies a default fault handler (with the construct *catchAll*) that will be activated if a fault is thrown and none of the other handlers catches it. For additional discussion on BPEL fault handlers see the BPEL specification [ACD+03].

For the case of the management of asynchronous messages, in order to detect exceptional situations, the *eventHandlers* construct is provided. Listing 4.10 shows an example of this construct. This event handler defines an activity (line 5) that should begin if and only if a *cancel* message (line 4) arrives.

Summarizing the concepts of exception handling discussed here, we have:

- Special dedicated constructs for exception handling are important when writing dis-

```
<process ... >
  ...
  <eventHandlers>
    <onMessage operation="cancel" ... >
      <terminate/>
    </onMessage>
    ...
  </eventHandlers>
</process>
```

Listing 4.10: Asynchronous messages in BPEL

tributed systems. Since a lot of different systems are involved, exceptional situations could occur frequently, and dedicated constructs facilitate the handling of such situations.

- An additional important aspect of exception handling in distributed systems, is the listening for asynchronous messages that can report the occurrence of exceptional situations. This message could be sent by one of the components of the system or a human actor.

4.4 Conclusions

In this chapter, we have presented the infrastructural services and the transactional requirements that a framework for Web Services composition should fulfill. We have defined the basic infrastructural services as: management of workflow instances life cycle, management of events, management of correlation issues and management of instances state. The transaction management requirements have been classified in the following concerns: synchronization concerns, compensation concerns, functional replication concerns, timing concerns and exception handling concerns. We have focussed in the analysis of the transactional management requirements, and the main points of this research were:

Synchronization concerns

- Synchronization Dependencies can be established in terms of significant events between transactions, using the ACTA framework.

- The number of synchronizations dependencies that can de defined in this way is virtually unbounded, providing an extensible model.

- Extensions can be developed over the dependency model of ACTA, for example the dependencies can be logic boolean expressions in terms of the outcomes of other

activities.

- When a synchronization dependency is not fulfilled, in certain cases abandoning the activity is correct, but in others the signaling of a error is appropriate.
- The outcomes of transactions could be a function of state variables of the process.
- In this case, activities can have more than one outcome, possibly defined using different logic expressions.
- Each outcome of an activity can can have a logical name. This name is referenced later for establishing synchronization dependencies.
- A dependency expressed in terms of state variables of a process can be refactored as a dependency without this constraint, with minor semantic alterations.
- Therefore, dependencies based on outcomes of activities that are logic expressions in terms of state variables is a redundant mechanism.

Compensation concerns

- A compensation concern can be expressed using three ACTA dependencies: *Compensation*, *Begin on Commit* and *Begin on Abort*.
- For expressing compensation, a mechanism for specifying that the compensating transactions are not part of the normal execution flow of the application is needed.
- A special construct for expressing the entire concept of compensation can be useful, since compensating concerns are very common in distributed applications.

Functional replication concerns

- A functional replication concern can be expressed using the ACTA dependency *Begin on Abort*.
- Similar to compensation, a mechanism for specifying that the replication transactions are not part of the normal execution flow of the application is needed.
- Functional replication can be always described as a case of exception handling.
- However, this tangling of functional replication with other exception management concerns could difficult the development and maintainability of the application.

Timing concerns

- Management of timing should include the definition of temporal predicates as a base for expressing timing concerns.
- Timing should include the schedule of activities, both in terms of dates or after a specified amount of time.
- Timing should include management of timeout concerns, since it cannot be assumed that all the components in a loosely distributed system will accept a request or will be always available.

Exception handling concerns

- Special dedicated constructs for exception handling are important when writing distributed systems. Since a lot of different systems are involved, exceptional situations could occur frequently, and dedicated constructs facilitate the handling of such situations.
- An additional important aspect of exception handling in distributed systems, is the listening for asynchronous messages that can report the occurrence of exceptional situations. This message could be sent by one of the components of the system or a human actor.

The main focus of this chapter has been discussing transactional requirements and showing different perspectives for solving them. We have not yet presented a solution or made an analysis of the BPEL shortcomings for transaction management, because this work is done in chapters 5 and 7.

Chapter 5

DBCF

5.1 Defining our model for transaction management

From the requirement analysis done in chapter 4, we can observe that the dependency model of ACTA [CR92] is an option in most of the problems discussed, in addition, from the models revised in chapter 3, ACTA has the following advantages over other proposals:

- It is a well known formal model, therefore anybody familiarized with it can easily cope with related specifications.
- It is simple but very powerful. The concept of dependencies between transactions is useful for expressing most of the transactional concerns currently managed by BPEL, and it is not limited to only them.
- It is absolutely extensible. New kind of dependencies can be added to the model, facilitating in this way the use of new transactional models.

This fact qualifies ACTA as a good starting point, however this model also has limitations, for example:

- Dependencies can be established only amongst pairs of activities, therefore expressing complex concepts (e.g. compensation) using only ACTA dependencies can be a cumbersome task.
- The model itself does not provide any mechanism for specifying that certain transactions should not be considered as part of the main control flow of the application.

- The model does not include any support for dependencies in terms of state variables of the process.
- Since dependencies are established amongst pairs of activities, this also creates the limitation that complex expressions (based in boolean logic for example) cannot be written when specifying dependencies.
- There is no support for including temporal predicates when establishing dependencies.

From these limitations, we think that the lack of mechanisms for expressing dependencies in terms of state variables of the process is not a serious issue, since as we described in the previous section, this problem can be easily rewritten using structural activities. For the other limitations, some of them will be solved with proposals for extensions to the dependency model of ACTA, and others will be left as a future work to accomplish.

Therefore, our model can be defined as an *extension to the ACTA formal model*, the general concepts of this extension will be described in the rest of this section, and additional issues will be discussed later when describing its implementation.

5.1.1 Extensions to the ACTA dependency model

In this section we will describe the extensions we propose for solving some of the expressiveness limitation of the ACTA dependency model applied to our problem domain. The problems that we want to solve here are:

- How to use dependencies for expressing complex concepts that involve more than a pair of activities.
- Defining a mechanism for differentiating the activities pertaining to the main business concern of the application (e.g. the main control flow), from such activities that are part of the transactional concerns.

Redefining the concept of dependencies

ACTA is a general framework for expressing a great number of transactional models. Since its objective is being able to define a great amount of models, its concepts are not related to a specific model. However, our domain is the composition of Web Services, where a hierarchical composition of activities is the model to be used. This fact can play as an advantage for us, since the constraint of a specific model allows us to make assumptions

that cannot be done in the ACTA formal model. For example, we always know that every activity in the process has a parent activity (even a root activity has as a parent the process itself), and that possibly more than one activity ancestor is present (since a parent activity could also has a parent).

We can use this knowledge for changing the concept of dependencies used in the ACTA framework. Instead of defining a dependency as a relationship that should be satisfied amongst a pair of transactions in terms of the significant events of such transactions, we will define that:

- Dependencies will be established amongst pairs of transactions, as in the ACTA framework.
- The constraints enforced by the dependency would not be restricted to the significant events of such transactions.
- Instead, additional significant events of transactions no mentioned explicitly in the dependency can be considered.
- Such other transactions can be inferred implicitly, since we know that the relationships amongst transactions follow a hierarchical composition model. Therefore, additional transactions that are present in the hierarchy of activities can participate in the dependency constraints.

In the next chapter, we will see an example of such concept, where a *Complete Compensation Dependency* is defined. This dependency is a variation of the ACTA *Compensation Dependency*, and as we will see in that chapter, it includes all the possible constraints that are involved in the idea of compensation.

Defining secondary activities

Secondary transactions have already been proposed in [Fab05], the following citation describes their objectives: "multiple ATMS require at certain points, conceptually separate from the main control flow of the application, secondary transactions to run automatically when some constraints are satisfied". This concept fits with our necessity of separating the activities in charge of implementing concerns like functional replication or compensation, from the ones that are part of the main business case of the process. Once the conceptual idea is defined, all that is needed is a construct or language element that can express it. We will define, when discussing the implementation of our framework, the strategy for specifying secondary transactions on it. In addition, chapter 7 will define a special construct for specifying a similar concept in the BPEL language.

5.1.2 Future work

We have considered as future work an extension to the ACTA dependency model that includes complex expressions (such as boolean logic expressions) or temporal predicates. We think that such work will bring additional abstraction capabilities to BPEL like languages. In the case of the possibility of using complex expressions in dependencies, we could mention as an example:

In the context of nested activities, we can have the case where a parent activity has to abort if one of its child transactions aborts. This can be expressed with a disjunction, since what we are saying is that the parent should abort if *child1* OR *child2* OR ... *childN* aborts. This could be written as an *Abort Dependency* in terms of the mentioned disjunction.

As an example of the utility of temporal expressions in dependencies, we could add to the dependency model ways for defining that a particular activity should not begin after one hour has passed since the commit of another activity. This could be written as a *Begin On Commit Dependency* that also considers time as an element of the dependency.

5.2 Overview of DBCF

In order to evaluate the correctness of our model and as a tool for additional discussion of it, we provide the implementation of a framework for composition of web services according to this model. Our framework is called DBCF, which stands for *Dependency Based Composition Framework*, and composition applications implemented with it will have, in addition to the capability of expressing transactional properties of BPEL like languages, the extensibility and simplicity from our conceptual model.

Furthermore, our framework can also be considered as an extensible tool for implementing service composition languages, for example, we could use it for implementing a BPEL engine, where the XML code that defines a BPEL process could be converted to elements of our framework using a translation engine. Later, since our framework implements the basic infrastructural requirements that a Web Service composition engine needs in order to appropriately receive and dispatch remote requests, the main services of DBCF could be started in order to administer instances of the previously transformed BPEL process.

We will discuss the relevant features of DBCF here, exploring why it is fully compatible with BPEL like languages, and why it goes beyond them.

DBCF follows the same hierarchical composition of activities architecture of BPEL, and employs the ideas for advanced transaction management that were developed in the previous section. These are the expression of transactional concerns using the ACTA formal

model and the extensions to this model defined before, they are:

- The definition of dependencies that implicitly can involve more than a pair of activities.
- The semantic classification of activities in *primaries* and *secondaries*.

In our model, the control flow will be determined by the way activities are composed and for the use of dependencies. Therefore, a specific order of execution in a structured activity that groups two or more activities cannot be assumed. Instead, such order will be given strictly by dependencies amongst them.

In the rest of this section, we will present first the technology involved, a general review of the architecture of the system, and a description of how the infrastructural and transaction management requirements discussed in the previous section were implemented. Additionally, fragments of code showing solutions with our framework and BPEL will be showed.

5.3 Technological issues

We have developed our implementation in Java [Sun]. The main reason for this is that we already had a well proven transaction monitor with support for dependencies amongst transactions written in this language (ATPMos, described in chapter 3), and since dependency management is a fundamental part of our research, this option was well suited for us.

Since Java is our chosen language, a natural selection for programming Web Services is JAX-WS [JAX]. This technology is the state of the art for web services programming in the Java language. All our tests have been realized in Tomcat [Tom], version 5.0. In addition, we have used AspectJ [Asp] for programming concerns that otherwise will crosscut and tangle other concerns of our code.

Figure 5.1 shows a representation of the architecture of our framework.

At the lowest level, ATPMos is in charge of the dependencies management that constitutes the core of our model. Over it is built our framework, which basically is composed by a Workflow Manager, an Event Manager, a collection of classes that define the different activities of the application, and an aspect library (implemented in AspectJ) that is in charge of doing most of the work present in our framework. On top of our framework, specific activities are implemented (extending the provided for us), and a workflow composition model is provided, in order to define how the business activities should be combined. Finally, on top of this layer are the objects related to JAX-WS: the Web Service interface of our workflow system and a collection of remote and local stubs.

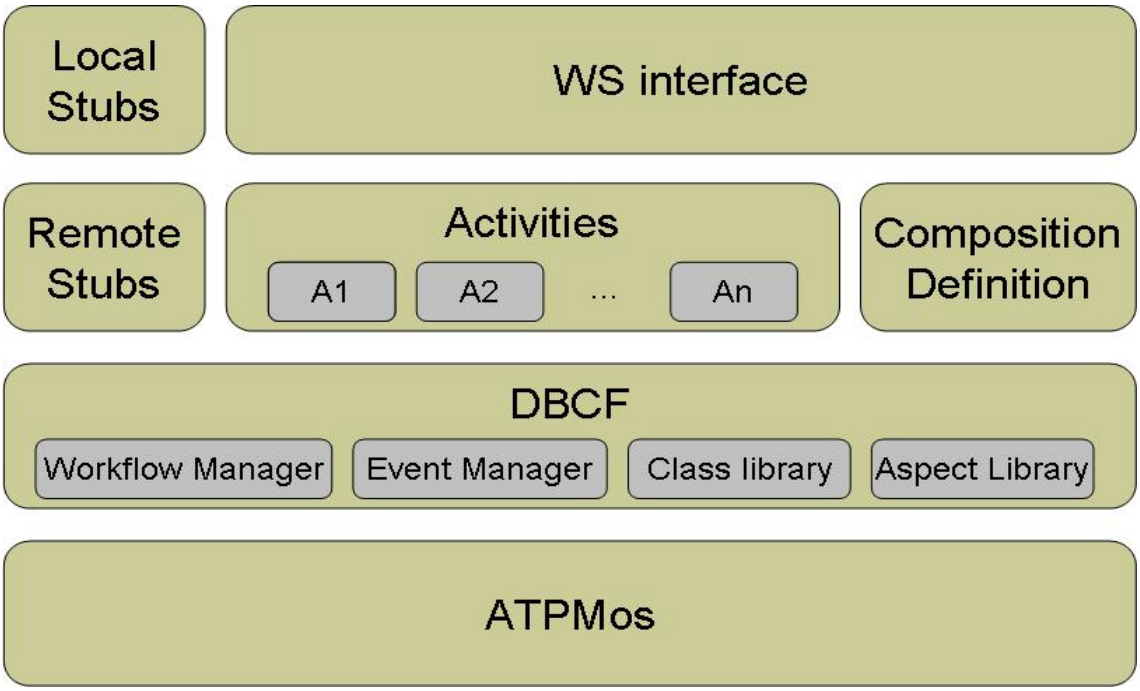

Figure 5.1: The DBCF architecture

5.4 Simplifying development using Aspect Oriented Programming

As mentioned before, we used aspect oriented techniques for abstracting the programmer some tasks that otherwise he would have to do at hand. Here we will present one of the main examples of this simplification, and this is the linking amongst a Web Service interface definition and DBCF.

Using JAX-WS, the programmer can implement a Java class that defines the implementation of various Web Service methods. According to this specification, the class should be annotated with the *WebService* annotation, and each of the interface method should have the *WebMethod* annotation.

Listing 5.1 shows an example of a JAX-WS class. Line 3 specifies that we are defining the class *ReservationImpl* as a Web Service called *ReservationService*. Line 5 specifies that we are defining the *queryFlightSchedules* method as part of the Web Service interface. Inside this method, the programmer should write the code that makes the link amongst the Web Service class and DBCF. However, this task is tedious, repetitive and error prone. Therefore, we have abstracted this work using an AspectJ aspect.

```
1  //import statements
2  ...
3  @WebService(serviceName="ReservationService", name="Reservation")
4  public class ReservationImpl {
5    @WebMethod()
6    public List<FlightInfo> queryFlightSchedules(...) {
7      //Web Service implementation
8    }
9    ...
10 }
```

Listing 5.1: Definition of a Web Service interface using JAX-WS

```
1  public aspect WorkflowDelegation {
2    pointcut workflowDelegation():
3        @this(WebService) && execution(@WebMethod * *(..));
4
5    Object around() : workflowDelegation() {
6      MethodSignature ms = (MethodSignature)thisJoinPoint.getSignature();
7      WorkflowManager wm = WorkflowManager.getWorkflowManager();
8      ...
9      String methodName = ms.getName();
10     Class[] parameterTypes = ms.getMethod().getParameterTypes();
11     Object[] parameterValues = thisJoinPoint.getArgs();
12
13     MethodCallEvent event =
14         new MethodCallEvent(methodName, parameterTypes, parameterValues);
15     MethodCallEventResponse response = wm.onEvent(event);
16     return response.getReturnValue();
17   }
18 }
```

Listing 5.2: Linking a Web Service interface with DBCF using aspects

Listing 5.2 shows the *WorkflowDelegation* aspect. This aspect defines the pointcut *workflowDelegation* (lines 2 and 3) that intercepts all the joinpoints produced when invoking a method with the annotation *WebMethod* inside of a class with the annotation *WebService*. The advice declared in lines 5 to 17, gathers the name of the method, its parameter types and parameter values (lines 9 to 11), and send them to the Workflow Manager (line 15) that represents a DBCF engine.

In this way, the programmer only has to write empty methods in the Web Service interface, and the linking code will be injected automatically.

5.5 Programming based on activities

Activities are the basic constructs of our hierarchical composition model, therefore we have developed a wide range of activities that supply different behaviour and that combined fulfill our requirements. The following is a summary of the main activities implemented in our model. Some of them will be referred later in the remain of this chapter.

- *Activity.*- Base abstract activity from which all the other activities inherit. Defines the default activity life cycle.
- *ScopeActivity.*- Base activity for all the structural activities (activities composed by other activities). Defines the default life cycle for structured activities. The BPEL equivalent is the *scope* element.
- *WorkflowActivity.*- Represents a Workflow Instance. The BPEL equivalent is the *process* element.
- *InvokeActivity.*- Represents an abstract invocation of a service (e.g. a web service invocation, or an email sent).
- *InvokeMethodActivity.*- Represents an abstract invocation of a remote method.
- *InvokeWSMethodActivity.*- Represents the invocation of a Web Service using a Remote Procedure Call (RPC) style. The BPEL equivalent is the *invoke* element.
- *RepetitionActivity.*- Represents an iteration activity. The BPEL equivalent is the *while* element.
- *ChooseOneConditionalActivity.*- Represents a multiple choice conditional activity. The BPEL equivalent is the *switch* element.
- *ConditionalActivity.*- Represents a single conditional option in a *ChooseOneConditionalActivity.*
- *ChooseOneEventActivity.*- Represents a multiple choice event activity. The BPEL equivalent is the *pick* element.
- *OnEventActivity.*- Represents an activity that waits for the occurring of an event in order to begin.
- *OnAlarmActivity.*- Represents an abstract activity that waits for an alarm (in terms of time) for beginning.
- *OnDateActivity.*- Represents an activity that waits for a specific date for beginning.

- *OnTimeOutActivity.*- Represents an activity that waits for a specific amount of time for beginning.

- *OnExternalEventActivity.*- Represents an abstract activity that waits for an external event (e.g. a web service reception) in order to begin.

- *OnMethodCallActivity.*- Represents an activity that waits for an external event in the form of a RPC web service invocation in order to begin.

5.6 Implementing the infrastructural requirements

We have discussed in the previous section, that the infrastructural requirements of our model are: management of workflow instances life cycle, management of external events, management of correlation issues and management of instances state.

5.6.1 Management of workflow instances life cycle

Management of workflow instances life cycle involves the answering of questions like: when a new instance should be created, what happens when this instance is created or when the life cycle of it is finished? We will answer these questions here.

Workflow instances in our model are created when a message arrives that fulfills certain requirements:

- The message is not directed to an already existing workflow instance.

- A message handler activity capable of managing the message is configured in the workflow definition.

- The mentioned message handler activity does not have any kind of begin dependency with other activity.

If all this conditions become true, a new instance is created and the ACTA dependencies amongst all its activities are established. Immediately after this, all the activities that do not have any begin dependency with another one will start their own individual life cycle.

A workflow instance is finished when all its primary activities and the started secondary activities have finished. As we discussed before, in our model primary activities are such activities belonging to the main business case of the process. Instead, secondary activities are such activities that are related to transactional concerns. Therefore, it is possible that

some secondary activities never begin during the execution of a process (e.g. functional replication activities, that do not begin if the replicated activity was executed successfully, without triggering the execution of the alternative activities), and waiting for them will block the process ad infinitum. This is the reason why a workflow instance can finish although secondary activities have not begun.

5.6.2 Management of events

We have defined two kind of events: External events and alarm events.

An external event is created by a component outside the workflow management system. For our purposes, we have implemented the management of web services calls, although the listening for additional external events could be implemented. As mentioned above, each time that a web service call arrives to the system, the workflow manager determines if an already started instance has registered as being interested in the event. If that is true, the message is delegated to it. If not: if it is determined that the message can be managed by a new instance (the message is a starting event), a new one is created. It the message is not a starting event, an exception is thrown and a fault is sent to the remote invocator. Every time that a message is accepted for a workflow instance, it is received by an *OnMethodCallActivity* in the form of a Remote Procedure Call.

Listing 5.3 shows an example of an activity that extends *OnMethodCallActivity*. This activity, as a normal Java class, can have an unbounded number of methods and instance variables, but the method that will the request for answering the web service invocation, is determined by the annotation *MethodHandler* (line 13). This Java annotation has to provide in the parameter *methodHandled*, the name of the web service method that the activity is interested in. Since more than one Web Service method can have the same name, the appropriate match is done transparently by the framework using the signature of the method that declares the *MethodHandler* annotation (lines 14 to 16).

As mentioned before, alarm events manage the occurrence of time related events. These events can be configured as a *DateEvent* or *TimeOutEvent*, according if the event occurs at a specific date or after a specific amount of time. When a time related event arrives to an instance, it is managed for an activity that inherits from *OnAlarmActivity*.

5.6.3 Management of correlation issues

As discussed before, one complication in workflow systems is the mapping from an arrived message to the adequate workflow instance. This is solved in our framework with the implementation of a correlation method in the *OnMethodCallActivity* activities. This method

receives the invocation parameters and it should return a boolean value indicating if there is a match or not. A match indicates that this is the target activity for the message. The invocation of this method is done automatically on all the *OnMethodCallActivity* activities, every time that a web service message arrives. The only activities that do not provide an implementation for the correlation method, are such *OnMethodCallActivity* activities that manages web service calls that creates new workflow instances. Since the instance is new, there is no state to correlate.

In listing 5.3, we can see a class that extends *OnMethodCallActivity* and that provides a boolean *correlation* method (lines 8 to 11). This method receives the parameters of the web service invocation, and uses it for determining if the correlation is or not successful.

5.6.4 Management of instance state

In an architecture based on a hierarchical composition of activities, state variables can be declared at different levels in the hierarchy, and this state should be visible to all the activities that are nested in the scope where such state is declared. Particularly, this is the case of BPEL, and we have decided to follow this model. Since all the activities in our model are defined as separate modules, (or in implementation terms: different Java classes), the sharing of variables is not a straightforward issue. We cannot use inheritance, since the activities already inherit from base classes (such as the abstract class *Activity*), therefore a manual lookup of state variables had to be implemented. We abstract all the details of this variable lookup to the programmer, providing a special Java annotation (*WorkflowState*) for denotating that certain instance variables of the activity, should be considered also state variables of the process, and therefore, all nested activities should have visibility and modification rights to this state.

In the example showed in listing 5.3, we can see two state variables (line 3 and 6). They are considered state variables of the process instead of simple instance variables of the class, since both are preceded by the *WorkflowState* annotation (lines 2 and 5). The first variable has been already declared in a upper activity, therefore it does not initialize to true the *create* parameter of the *WorkflowState* annotation. The second variable initializes it, since the variable is declared the first time. The mapping between the value of a state variable declared in a upper activity and referenced in a nested one (in this case, the variable *clientInfo*, in line 3) is done transparently by the framework.

```
1  public class AReserveHotelAndCar extends OnMethodCallActivity{
2    @WorkflowState
3    ClientInfo clientInfo;
4
5    @WorkflowState(create=true)
6    XMLGregorianCalendar from;
7    ...
8    public boolean correlation(MethodCallEvent event) {
9      String id = ((ClientInfo)event.getParameterValues()[0]).getId();
10     return clientInfo.getId().equals(id);
11   }
12
13   @MethodHandler(methodHandled="reserveHotelAndCar")
14   public void onReserveHotelAndCar(ClientInfo clientInfo,
15       XMLGregorianCalendar from, XMLGregorianCalendar to) {
16     this.from = from;
17     ...
18   }
19 }
```

Listing 5.3: A method handler activity example

5.7 Implementing the transaction management requirements

With the implementation of the infrastructural requirements of our framework discussed, we will cover in this section the implementation of the transactional requirements.

5.7.1 Synchronization of activities

Listing 5.4 shows an example of a synchronization concern using DBCF. Two invocation activities, *rh* (lines 4 and 5) and *rc* (lines 7 and 8) are defined, and a *Begin on Commit Dependency* amongst them is declared in line 10.

The method *setDependency* can be invoked by any activity, and receive three parameters: the source of the dependency, the dependency identifier and the destination of the dependency. The dependency invoked by this method will be applied when the declaring activity starts its life cycle (in this case: *rc*). It is possible also to apply dependencies at *begin*, *commit* or *abort* times. Every activity can extend the methods *onBegin onCommit* and *onAbort*, that will be transparently invoked by the framework when such events occur.

In our particular example, the result of applying the mentioned dependency, is that the activity *rc* will not begin until activity *rh* has begun. This dependency can be particu-

```
1 ...
2 ScopeActivity reserveHotelAndCar = new ScopeActivity(...);
3 ...
4 AReserveHotelWithPartner rh = new AReserveHotelWithPartner(reserveHotelAndCar, ...);
5 ...
6 AReserveCarWithPartner rc = new AReserveCarWithPartner(reserveHotelAndCar, ...);
7 ...
8 rc.setDependency(rc, "BCD", rh);
9 ...
```

Listing 5.4: Synchronization handling in DBCF

larly useful for specifying sequential execution order amongst a collection of activities. It could be argued that the use of this dependency as the only mechanism for implementing sequential execution could be cumbersome, since for each activity should be necessary to write a new dependency. However, since it is possible the definition of a sequential activity, that automatically insert the appropriate Begin on Commit dependency to all the nested activities, this problem is not considered a fundamental issue, since its solution can be considered as "syntactic sugar" for our framework.

5.7.2 Compensation of activities

Compensation of activities involves the employment of one additional concept: the employment of secondary transactions. Listing 5.5 shows an activity that plays the role of nesting transaction (*reserveHotelAndCar* in line 2) and two activities that plays the role of nested transaction (*rh* in line 4 and *ch* in line 6). This nesting is specified at the moment the two last activities are instantiated. They both send *reserveHotelAndCar* as a parameter in the invocation of their respective constructors. This first parameter in our framework defines the parent activity.

Once solved the nesting issues, the next step is to establish the compensation dependency. Since *ch* compensates *rh*, the former should be defined as a secondary activity. For this reason, line 7 invokes the method *setExecutionMode* and pass the constant value *SECONDARY_ACTIVITY*. Now the problem consists in establishing the adequate dependencies amongst the compensated and compensating activity. Line 8 declares that the compensating activity has a *Begin On Commit Dependency* with the compensated activity, therefore, the former will not be invoked if the latter has not committed. In addition, a *Begin On Abort Dependency* and a *Compensation Dependency* is established amongst the compensating activity and the nesting activity, therefore the compensating activity will begin only if the nesting activity aborts and, if the nesting activity aborts, the compensating activity should commit.

```
1  ...
2  ScopeActivity reserveHotelAndCar = new ScopeActivity(...);
3  ...
4  AReserveHotelWithPartner rh = new AReserveHotelWithPartner(reserveHotelAndCar, ...);
5  ...
6  ACancelHotelWithPartner ch = new ACancelHotelWithPartner(reserveHotelAndCar, ...);
7  ch.setExecutionMode(ExecutionMode.SECONDARY_ACTIVITY);
8  ch.setDependency(ch, "BCD", rh);
9  ch.setDependency(ch, "BAD", reserveHotelAndCar);
10 ch.setDependency(ch, "CMD", reserveHotelAndCar);
11 ...
```

Listing 5.5: Compensation handling in DBCF with traditional ACTA dependencies

```
1  ...
2  ScopeActivity reserveHotelAndCar = new ScopeActivity(...);
3  ...
4  AReserveHotelWithPartner rh = new AReserveHotelWithPartner(reserveHotelAndCar, ...);
5  ...
6  ACancelHotelWithPartner ch = new ACancelHotelWithPartner(reserveHotelAndCar, ...);
7  ch.setExecutionMode(ExecutionMode.SECONDARY_ACTIVITY);
8  ch.setDependency(ch, "CPD", rh);
9  ...
```

Listing 5.6: Compensation handling in DBCF with a customized dependency

As discussed before, establishing all these dependencies for each compensation concern could be a cumbersome work, therefore we defined a new kind of dependency for compensation that automatically implies all the dependencies mentioned before. We will call it *CPD* in order to distinguish it from the ACTA compensation dependency *CMD*. This goes away because of the hierarchy of the transactions that define our process (recall that we use a hierarchical composition of activities), thanks to it is possible to infer all the transactions that are not mentioned in the dependency, but that have implicit dependencies with the activities defined on it (in our example, the *reserveHotelAndCar* activity can be inferred as the parent of *rh*).

Listing 5.6 shows an example of this dependency, where the dependency declaration in line 8 is enough for substituting the three dependencies of the previous listing.

5.7.3 Functional replication of activities

The implementation of a functional replication concern demands, as the previous case, the definition of a secondary transaction. We show in listing 5.7 an example of this concern, where the activity *reserveHotelAndCarA* is replicated by the activity *reserveHotelAnd-*

```
1 ...
2 ScopeActivity reserveHotelAndCarA = new ScopeActivity(reserveHotelAndCar, ...);
3 ...
4 ScopeActivity reserveHotelAndCarB = new ScopeActivity(reserveHotelAndCar, ...);
5 reserveHotelAndCarB.setExecutionMode(ExecutionMode.SECONDARY_ACTIVITY);
6 reserveHotelAndCarB.setDependency(reserveHotelAndCarB, "BAD", reserveHotelAndCarA);
7 ...
```

Listing 5.7: Functional Replication handling in DBCF

CarB. Since *reserveHotelAndCarB* is an activity that implements a functional replication concern, it should be declared as secondary. The same explanation detailed in the compensation case is applicable here. Line 5 declares that the activity is secondary, and line 6 establishes a *Begin On Abort Dependency* with the primary activity, in such a way that the secondary activity will be executed only if the primary activity fails.

5.8 Conclusions

This chapter has presented a model for transaction management in web services composition that improves over similar proposals. Particularly, we have demonstrated the flexibility and extensibility of the ACTA formal model for expressing a lot of these concerns. We have also seen that ACTA itself is not enough for dealing with all of them, and that for obtaining additional expressiveness and simplification in this domain, additional ideas could be developed over ACTA in order to provide a major significant improvement. Among these ideas, we distinguished the necessity of establishing dependencies implicitly and between more than pairs of activities, since it can bring more abstraction to the programmer, and reduce the probabilities of error when maintaining the application. Also, we have emphasized the necessity of a mechanism for specifying transactions that are not part of the main concern of the application, and we have used the concept of secondary transactions, already defined in [Fab05], as a mechanism for expressing this idea. After that, we have proposed a model for web services composition that follows our directions, and we have implemented both its infrastructural and transactional requirements.

We also mentioned that other possible use of our framework could be the implementation of service composition languages. We took as an example the BPEL language, where a XML document that defines a BPEL process could be converted to DBCF modules using a transformation engine, and once transformed, the infrastructural services that are implemented in DBCF could be started in order to deploy the process originally defined using BPEL. The mentioned transformation engine is not implemented in this work, since it is part of our future work to accomplish.

Next chapter will present a more detailed comparison of BPEL and DBCF, with a focus on how their respective constructs implement the infrastructural and advanced transaction management requirements of Web Services composition.

Chapter 6

A comparison between BPEL and DBCF

In chapter 5 we have discussed how transactional concerns can be managed in DBCF and in chapter 4 we made an analysis of how the same concerns are managed by BPEL and other models. In this chapter we will summarize both the approaches of BPEL and DBCF, in order to present a comparison of their strategies. We have included in this survey constructs that implement the infrastructure services defined in chapter 4, since these features will give us a more broad perspective of the style of both proposals.

6.1 Overview of the differences

The table 6.1 presents an overview of the differences between BPEL and DBCF that are related both to transaction management and infrastructure services. In the rest of this chapter we will present a detailed comparison of how these concerns are managed by the two proposals. We have not presented an extensive description of concerns that have already been discussed in previous chapters, instead we have preferred to recall the main ideas of the topic and refer to the chapter where they are discussed.

6.2 Detailed comparison of each concern

In this section we will present a more detailed comparison for each concern presented in the table 6.1.

FEATURE	**BPEL**	**DBCF**
Scoping	*scope* activity	*ScopeActivity* activity
State management	Scoped variables and message properties	Scoped variables using annotations
Concurrent activities	*flow* activity.	All the activities at the same scope are concurrent by nature.
Sequential activities	*sequence* activity.	Sequential execution is created using dependencies.
Other control flow constructs	*switch*, *while* activities.	*ConditionalActivity*, *ChooseOneConditionalActivity*, *RepetitionActivity*
Time constructs	*wait* activity. Parametrized with Deadline-Valued or Duration-Valued attributes. The *pick* activity and the *eventHandlers* construct are also related to timing.	*OnDate* and *OnTimeOut* activities. The *ChooseOneEvent* activity is also related to timing.
Non deterministic choice based on external events	*pick* activity.	*ChooseOneEvent* activity.
Creation of workflow instances	With a web service message managed by a *receive* or a *pick* activity that has the *createInstance* attribute set to *yes*	With a web service message managed by an external event activity that does not have any begin dependencies
Correlation	Correlation sets	Definition of correlation methods in *OnMethodCallActivity* activities.
Exception Management	The exception can be propagated (and thus managed) to upper activities	An exception leads to the abortion of the activity
Synchronization of activities	Through the notion of links	Through ACTA dependencies
Compensation of activities	Compensation handlers	Through secondary activities and ACTA dependencies
Functional Replication	Fault Handlers	Through secondary activities and ACTA dependencies

Table 6.1: Comparison between BPEL and DBCF

```
1  <scope>
2    <variables>
3    ...
4    </variables>
5
6    <correlationSets>
7    ...
8    </correlationSets>
9
10   <faultHandlers>
11   ...
12   </faultHandlers>
13
14   <compensationHandler>
15   ...
16   </compensationHandler>
17
18  <eventHandlers>
19   ...
20   </eventHandlers>
21
22   <sequence>
23     <invoke ... />
24     ...
25   </sequence>
26 </scope>
```

Listing 6.1: A scope definition in BPEL

6.2.1 Scoping

Scoping in BPEL

A BPEL scope provides the behaviour context and a namespace where state variables for activities in a BPEL process can be defined. A scope can define the following constructs: fault handlers, event handlers, a compensation handler, data variables, and correlation sets. All these elements are syntactically optional and will be discussed with more detail later in this chapter. A BPEL scope can directly nest only one activity. However, this activity can be a structured activity that contains other activities, and in this way, more than one activity can be nested and therefore executed in the context of a BPEL scope. In BPEL, scopes are represented with the *scope* element. Listing 6.1 shows an example of a typical structure of a BPEL scope. Lines 2 to 20 show the possible constructs that can be nested inside of it, and lines 22 to 25 show a BPEL activity. In our example, it is a *sequence* activity that nests other activities inside of it.

```
1 ...
2 ScopeActivity reserveHotelAndCarA = new ScopeActivity(reserveHotelAndCar, ...);
3 ...
4 AReserveCarWithPartner rcA = new AReserveCarWithPartner(reserveHotelAndCarA, ...);
5 ...
6 ACancelCarWithPartner ccA = new ACancelCarWithPartner(reserveHotelAndCarA, ...);
7 ccA.setExecutionMode(ExecutionMode.SECONDARY_ACTIVITY);
8 ...
```

Listing 6.2: A scope activity in DBCF

Scoping in DBCF

In DBCF, all the activities can be considered scopes, in the sense that all of them can be namespaces in which state variables are declared , as we will see in the *State Management* section, this feature provides a simplified composition model, since the additional declaration of a scope activity for defining a new namespace each time it is required is not compulsory for a programmer.

In addition to this, the *ScopeActivity* activity is defined in DBCF as a base class for all the structured activities. As we will see with more detail in the *Concurrent Activities* section, the default behaviour of a ScopeActivity is to execute all its nested activities concurrently. However, other activities that inherit from it, like the *ChooseOneConditionalActivity* or the *RepetitionActivity* activities, can choose to execute only one of its activities based in certain conditions, or to repeat them a certain number of times. We will come back to these activities in the *Other control flow constructs* section.

Another interesting feature of the *ScopeActivity* activity, is that it can compose both primary and secondary activities (as we discussed in chapter 5, secondary activities are not considered part of the main control flow of the application, since they implement transactional concerns).

As an example, listing 6.2 shows a scope activity (line 2). In this example, two activities are nested inside of it. The activity *rcA* (line 4) and the secondary activity *ccA* (line 6). The nesting is declared since these two activities when instantiated, use *reserveHotelAndCarA* as first parameter of their constructor, and in DBCF this parameter is used for establishing the parent of an activity. In a scope, all the activities where the *setExecutionMode* method is invoked with the parameter *SECONDARY_ACTIVITY*, are considered nested secondary activities, in the sense discussed in chapter 5.

Summary

- In BPEL, scopes define an entire behaviour context for execution of activities.

- They also define a namespace where state variables can be defined.
- Only one activity can be directly nested in a BPEL scope, if more than one is needed, a structured activity must be used.
- In DBCF, all the activities define a namespace where state variables can be defined.
- The *ScopeActivity* serves as a base class for all the structured activities . It can contain both primary and secondary activities.
- The composition model of DBCF is simpler than the BPEL one, since in the first not explicit scopes are needed for declaring namespaces for variables, and more than one activity can be directly nested in a scope.

6.2.2 State management

State management in BPEL

As we discussed in the last section, BPEL scopes can include the *variables* element in which state variables can be defined and associated with a particular scope. These variables will exist only during the execution of their declaring scope [ACD+03]. Once declared, the variables can be used inside the scope as parameters of Web Service invocations, for determining the control flow execution of the process (e.g. a BPEL *switch* activity can define its *case* conditions as a boolean expressions in terms of the state variables), or simply as a repository of data received in Web Service messages.

In addition, BPEL provides special constructs for managing data, such as the *assign* activity. The listing 6.3 shows a BPEL scope where three state variables are declared (lines 3 to 5). After that, an invocation activity (lines 11 and 12) makes use of the *sendPO* variable as an input parameter, and the *getResponse* variable as an output parameter. Lines 15 to 20 show an *assign* construct, where one of the components of the *getResponse* variable is copied into the *address* variable. For additional discussion of the *assign* activity and other data related expressions, see the BPEL specification [ACD+03].

State management in DBCF

As we mentioned previously, all the activities in DBCF serve as namespaces for declaring variables. State management is accomplished using the Java annotation *WorkflowState*. This annotation, when used without parameters in a Java class representing a DBCF activity, implies that the annotated instance variable represents part of the state of the process, and that such process variable has already been declared in an upper activity.

```
<scope>
  <variables>
    <variable name="sendPO" messageType=... />
    <variable name="getResponse" messageType=... />
    <variable name="address" element=.../>
    ...
  </variables>

  <sequence>
    ...
    <invoke ... operation="SyncPurchase"
        inputVariable="sendPO" outputVariable="getResponse" >
      ...
    </invoke>
    <assign>
      <copy>
        <from variable="getResponse" part = "address"/>
        <to variable="address"/>
      </copy>
    </assign>
    ...
  </sequence>
</scope>
```

Listing 6.3: State management in BPEL

The framework will be in charge of doing all the synchronization work with the variable declared in the upper activity and referenced in the nested one.

If the annotation *WorkflowState* is parametrized with the *create* parameter set to *true*, this implies that a new state variable must be created, and associated with the scope of the DBCF activity where such state is declared. Similar names can be used inside distinct nested scopes, since the framework will resolve the names using the first namespace where the variable can be solved, beginning the search from the scope of the DBCF activity that declares the variable, and finishing at the scope of the entire workflow process. If in this search path the variable cannot be found, a *NoSuchVariableException* exception will be thrown.

Listing 6.4 shows an example of state management in DBCF. Lines 2 and 3 show an annotated instance variable that is part of the process state. Since the *WorkflowState* annotation is used without parameters in this case, it implies that this variable is declared somewhere in an upper activity, and therefore it can be used without having been locally initialized, as it is done in line 10. Lines 5 and 6 show the declaration of other instance variable, also annotated with the *WorkflowState* annotation, but this time the annotation is parametrized with the *create* parameter set to *true*. This implies that a new entry in the variables dictionary associated with the scope of this activity will be created, containing

```
1  public class AReserveHotelAndCar extends OnMethodCallActivity{
2    @WorkflowState
3    ClientInfo clientInfo;
4
5    @WorkflowState(create=true)
6    XMLGregorianCalendar from;
7    ...
8    public boolean correlation(MethodCallEvent event) {
9      String id = ((ClientInfo)event.getParameterValues()[0]).getId();
10     return clientInfo.getId().equals(id);
11   }
12
13   @MethodHandler(methodHandled="reserveHotelAndCar")
14   public void onReserveHotelAndCar(ClientInfo clientInfo,
15       XMLGregorianCalendar from, XMLGregorianCalendar to) {
16     this.from = from;
17     ...
18   }
19 }
```

Listing 6.4: State management in DBCF

the name of the annotated variable. Later, the variable can be initialized (as it is done in line 16) and used as a normal Java object.

Summary

- BPEL variables are declared within the *variables* construct.
- The declaration of variables in BPEL must be associated with a BPEL scope. Therefore, every time that a new variable is declared, a new scope must be created.
- Additional manipulation of state variables, is done in BPEL with special constructs (such as *assign*) and other expressions.
- DBCF relies on the *WorkflowState* annotation for declaring the state variables of a process.
- The *WorkflowState* annotation can be parametrized with the *create* parameter.
- If the *create* parameter has the value of *true*, a new variable definition will be created in the scope of the DBCF activity where such variable is declared.
- If the *create* parameter has the value of *false* or it is omitted, the variable has already been declared in an upper scope. Therefore, a new entry will not be created on the variable dictionary of the activity scope, and the variable can be used without having been locally initialized.

```
1 <flow>
2   <activity1 />
3   <activity2 />
4   .
5   .
6   .
7   <activityN />
8 </flow>
```

Listing 6.5: Concurrent activities in BPEL

```
1 ...
2 ScopeActivity reserveHotelAndCar = new ScopeActivity(...);
3 ...
4 AReserveHotelWithPartner rh = new AReserveHotelWithPartner(reserveHotelAndCar, ...);
5 ...
6 AReserveCarWithPartner rc = new AReserveCarWithPartner(reserveHotelAndCar, ...);
7 ...
```

Listing 6.6: Concurrent activities in DBCF

6.2.3 Concurrent activities

Concurrent activities in BPEL

In BPEL, concurrency is established using the *flow* activity. This activity basically indicates that all the activities nested inside of it will be executed concurrently. The flow activity will be considered finished when all its nested activities have finished. In addition, the flow construct can declare control links elements for management of synchronization concerns amongst its nested activities. Listing 6.5 shows an example of a BPEL flow construct.

Concurrent activities in DBCF

The execution of the activities nested inside a structural activity will be determined by the dependency constraints that they declare. Therefore, if they are not affected for any kind of begin dependencies, they will be executed concurrently.

Listing 6.6 shows a parent activity called *reserveHotelAndCar* (line 2) and two nested activities: *rh* (line 4) and *rc* (line 6). In this example, *rh* and *rc* will be executed concurrently, since no dependencies are declared.

```
1 <sequence>
2   <activity1 />
3   <activity2 />
4   .
5   .
6   .
7   <activityN />
8 </sequence>
```

Listing 6.7: Sequential activities in BPEL

Summary

- BPEL needs a special construct for declaring concurrency: the *flow* construct.
- DBCF does not need a dedicated construct. It uses the scope construct for declaring a nesting of activities. If no dependencies are declared the nested activities will be executed concurrently.

6.2.4 Sequential activities

Sequential activities in BPEL

Sequential execution is established using the *sequence* construct. This construct indicates that all the activities nested inside of it will be executed sequentially. The sequence activity will be considered finished when all its nested activities have finished. Listing 6.7 shows an example of a BPEL sequence construct.

Sequential activities in DBCF

As stated previously, the order in which activities are executed in a scope is only dependant of the dependencies that are configured amongst such activities. Listing 6.30 shows an example of sequential execution where the activity *rc* (line 6) will be executed only after the activity *rh* (line 4) has successfully finished. This is specified in the ACTA *Begin On Commit dependency* declared in line 8. Since this dependency has already been discussed in chapter 3, we will not repeat its explanation here.

The manual writing of begin on commit dependencies between sequential activities is not considered a serious issue in DBCF, since a new kind of activity can be easily defined that automatically creates this dependency amongst its nested activities. This work is not implemented since it is considered just as syntactic sugar for our framework.

```
1 ...
2 ScopeActivity reserveHotelAndCar = new ScopeActivity(...);
3 ...
4 AReserveHotelWithPartner rh = new AReserveHotelWithPartner(reserveHotelAndCar, ...);
5 ...
6 AReserveCarWithPartner rc = new AReserveCarWithPartner(reserveHotelAndCar, ...);
7 ...
8 rc.setDependency(rc, "BCD", rh);
9 ...
```

Listing 6.8: Sequential activities in DBCF

Summary

- BPEL needs a special construct for declaring sequential execution: the *sequence* activity.
- DBCF does not need a special construct since the activities nested in a scope will be executed sequentially if begin on commit dependencies are declared amongst them.
- DBCF could implement a special construct for expressing sequencing, like the one used in BPEL, if it is determined that the declaration of begin dependencies is a cumbersome work for expressing sequential execution.

6.2.5 Other control flow constructs

Other control flow constructs in BPEL

BPEL implements other control flow constructs in addition to sequencing or concurrent execution. They are the *switch* and *while* activities.

Listing 6.9 shows a *switch* activity in BPEL. A number of *case* constructs are nested inside of it, each of them associated with a logical boolean condition. In our example, only one case is shown (lines 2 to 6). In addition to the case constructs, an *otherwise* construct represents the option that must be executed if no one of the cases was executed.

In addition to the *switch* activity, iteration can be expressed in BPEL with the *while* activity. Listing 6.10 shows a *while* activity in BPEL. This activity will be executed until the associated condition (line 4) becomes false.

```
<switch>
  <case condition="bpws:getVariableData(
      'shippingInfoOption1', 'numberOfDays', ...)
      <= bpws:getVariableData('shippingInfoOption2', 'numberOfDays', ...)">
    ...
  </case>
  ...
  <otherwise>
    ...
  </otherwise>
</switch>
```

Listing 6.9: A switch activity in BPEL

```
...
<variable name="orderDetails" type="xsd:integer"/>
...
<while condition="bpws:getVariableData(orderDetails) > 100">
  ...
</while>
```

Listing 6.10: A while activity in BPEL

Other control flow constructs in DBCF

Similar to BPEL, DBCF implements other control flow constructs in addition to sequencing or concurrent execution. They are the *ConditionalActivity*, *ChooseOneConditionalActivity* and *RepetitionActivity* activities.

Listing 6.11 represents a *ConditionalActivity* activity. All the classes that extends *ConditionalActivity* must implement the *condition* method (lines 13 to 15). This method will be evaluated when the activity begins (in other words, when the activity satisfies its begin dependencies), and it will return a boolean value that will determine if the activities nested in the ConditionalActivity will be executed (in case of true) or ignored (in case of false). In listing 6.12, a conditional activity is defined in line 1 (*cfrA*). Line 3 defines an activity nested into this conditional activity (*rfA*) (since it declares as a first parameter of its constructor the conditional activity). Therefore, the life cycle of *rfA* will be executed only if the *condition* method of *cfrA* return true.

Listing 6.13 shows the definition of a *ChooseOneConditionalActivity* (lines 4 and 5). This activity is defined composing a number of *ConditionalActivity* activities (lines 6 to 11). The behaviour of this activity is equivalent to the BPEL *switch* activity. When started, it will evaluate the conditions of all its nested conditional activities, and it will only start the lifecycle of the first activity which has an associated condition with value of true at the moment of the evaluation. In case that a default activity has to be defined (the activity

```
1  ...
2  public class AConditionalFlightReservation extends ConditionalActivity {
3    ...
4    @WorkflowState
5    FlightInfo flightInfo;
6    String id;
7
8    public AConditionalFlightReservation(..., String id) {
9      super(...);
10     this.id = id;
11   }
12
13   protected boolean condition() {
14     return id.equals(flightInfo.getId());
15   }
16 }
```

Listing 6.11: A conditional activity in DBCF

```
1 AConditionalFlightReservation cfrA = new AConditionalFlightReservation(...);
2 ...
3 AReserveFlightWithPartner rfA = new AReserveFlightWithPartner(cfrA, ...);
```

Listing 6.12: Defining conditional execution in DBCF

that must be executed if the conditions of all the other conditional activities were evaluated to false), this can be solved simply declaring a conditional activity with a dummy *condition* method, that always return true.

Iteration is implemented in DBCF with the *RepetitionActivity* activity. This activity is defined in terms of a conditional activity. It will instantiate the conditional activity (and therefore, the activities nested in the conditional activity) as long as its associated condition is true. When false, the iteration will stop.

Listing 6.14 shows an example of the use of the *RepetitionActivity* activity. The activity receives as a parameter an *ActivityFactory* object (lines 3 to 14), that is in charge of instantiating a *ConditionalActivity* when required (lines 4 to 13). The conditional activity defines its *condition* method (lines 9 to 11), in terms of the instance variables *counter* and *limit*. Since these instance variables are annotated with the *WorkflowState* annotation, they are also state variables of the process, and this implies that these two variables are defined in an upper activity. In the case of the *counter* variable, it is automatically defined in the *RepetitionActivity* activity, and represents an iteration counter. Since the conditional activity is nested into the iteration activity, the former is able of accessing the state declared on the latter.

Listing 6.15 shows a short fragment of the *RepetitionActivity* implementation. As we can

```
...
AReserveFlight reserveFlight =
    new AReserveFlight(workflow, ...);
ChooseOneConditionalActivity chooseReservation =
    new ChooseOneConditionalActivity(reserveFlight, ...);
AConditionalFlightReservation cfrA =
    new AConditionalFlightReservation(chooseReservation, ...);
AConditionalFlightReservation cfrB =
    new AConditionalFlightReservation(chooseReservation, ...);
AConditionalFlightReservation cfrC =
    new AConditionalFlightReservation(chooseReservation, ...);
...
```

Listing 6.13: A switch activity in DBCF

```
...
RepetitionActivity ra = new RepetitionActivity(...,
  new ActivityFactory<ConditionalActivity>() {
    public ConditionalActivity createActivity(ScopeActivity parent) {
      return new ConditionalActivity(parent, ...) {
        @WorkflowState int counter;
        @WorkflowState int limit;

        protected boolean condition() {
          return counter == limit;
        }
      };
    }
  }
);
...
```

Listing 6.14: Defining repetition activities in DBCF

```
1  // import statements
2  public class RepetitionActivity extends ScopeActivity {
3    ...
4    @WorkflowState(create=true) int counter;
5    ...
6    protected void doWork() {
7      counter = 0;
8      while(...) {
9        ...
10     }
11     ...
12   }
13
14 }
```

Listing 6.15: A while activity in DBCF

see in line 4, the instance variable *counter* is annotated with the *WorkflowState* annotation. Since this annotation is parametrized with the *create* parameter equal to *true*, the *counter* state variable will be created in the scope of the *RepetitionActivity* activity.

Summary

- Both BPEL and DBCF provide constructs for expressing conditional execution and iteration.
- BPEL uses the *switch* activity for expressing conditional execution, and the *while* activity for expressing iteration.
- DBCF provides the *ConditionalActivity* activity for expressing simple conditions, the *ChooseOneConditionalActivity* activity is useful for defining more complex conditions, and is composed of one or more *ConditionalActivity* activities. Finally, the *RepetitionActivity* activity can express iteration, and is also based on a *ConditionalActivity* activity for defining how many iterations it must accomplish.

6.2.6 Time constructs

Time constructs in BPEL

The BPEL language elements for timing management are the activities *pick* and *wait* and the construct *eventHandlers*, always associated with a BPEL *scope*.

```
1  <pick>
2    <onMessage operation="inputLineItem" ...>
3      ... <!-- an activity -->
4    </onMessage>
5
6    <!-- set an alarm to go after 3 days and 10 hours -->
7    <onAlarm for="'P3DT10H'">
8      ... <!-- handle timeout -->
9    </onAlarm>
10 </pick>
```

Listing 6.16: Timeout handling in BPEL

```
1  <sequence>
2    <invoke operation = "reserve" ... />
3    <wait until="'2002-12-24T18:00+01:00'"/>
4    <invoke operation = "confirmReservation"... />
5  </sequence>
```

Listing 6.17: Delaying execution in BPEL

As discussed in chapter 4, the *pick* activity groups a collection of activities associated with events. Only one of these activities will be executed, and it will be determined for which of their associated events arrives first. These messages can be either a web service message or a time related event. This makes this construct well suited for handling timeout configurations.

Listing 6.16 shows an example of a BPEL *pick* activity used for expressing timeout management. In this example, the message defined in lines 2 to 4, will be valid only if it arrives before the timeout configured in lines 7 to 9.

The *wait* activity allows a business process to specify a delay for a certain period of time or until a certain deadline is reached [ACD+03]. Listing 6.17 shows an example of how it can be used for delaying the execution of the *invoke* activity defined in line 4.

Finally, the *eventHandlers* element, similarly to the *pick* activity, can execute activities related to events. The difference with the *pick* activity is that this selects one activity from a set, and only executes the selected activity. The *eventHandlers* construct just listens for events, and will execute them as long as they occur. One example of the utility of this construct is the scheduling of activities in a process.

Listing 6.18 shows an example of scheduling using the eventHandlers construct. In our example, the schedule activity defined in line 5, will be executed only when defined in the timing condition expressed in line 4.

All the examples of this section have been discussed with more detail in chapter 4. There-

```
<process ...>
  ...
  <eventHandlers>
    <onAlarm for="bpws:getVariableData(orderDetails,processDuration)">
      ... <!-- a scheduled activity -->
    </onAlarm>
    ...
  </eventHandlers>
  ...
</process>
```

Listing 6.18: Scheduling activities in BPEL

```
1 ...
2 ChooseOneEventActivity chooser = new ChooseOneEventActivity(...);
3 OnTimeOutActivity timeOut = new OnTimeOutActivity(chooser, "", 60, TimeUnit.SECONDS);
4 InputLineItemActivity input = new InputLineItemActivity (chooser, ...);
5 ...
```

Listing 6.19: Timeout handling in DBCF

fore, we have only presented a brief overview of the code.

Time constructs in DBCF

DBCF provides the following activities for timing management: *OnDateActivity*, *OnTimeOutActivity* and *ChooseOneEventActivity*. Both *OnDateActivity* and *OnTimeOutActivity* extend the more general *OnAlarmActivity*. The objectives of these activities is to express a delay of time. *OnDateActivity* can delay the execution of certain activity to a particular date, and *OnTimeOutActivity* can delay its execution a determined amount of time.

The *ChooseOneEventActivity*, similar to the *pick* BPEL activity, can group activities that wait for events in order to begin. The *ChooseOneEventActivity* activity will administer the lifecycle of its nested activities, and will start the first activity in receiving its associated message. All the other activities will be canceled.

Listing 6.19 shows an example of Timeout management in DBCF. A *ChooseOneEventActivity* (line 2) is nesting two activities based on events. The first activity (line 3) is an *OnTimeOutActivity* activity and the second (line 4) represents an activity waiting for a Web Service arrival (in DBCF, this means that it inherits from the *OnMethodCallActivity* activity). If the Web Service invocation required by the latter, does not arrive before the timeout configured in the former, the *ChooseOneEventActivity* activity will execute the *OnTimeOutActivity* activity and will cancel the *OnMethodCallActivity* activity.

```
1 ...
2 ScopeActivity reservation = new ScopeActivity(...);
3 ReserveActivity reserve = new ReserveActivity(reservation, ...);
4 OnTimeOutActivity delay = new OnTimeOutActivity(reservation, "", 60, TimeUnit.SECONDS);
5 timeOut.setDependency(delay, "BCD", reserve);
6 ConfirmReservation confirm = new ConfirmReservation(reservation, ...);
7 confirm.setDependency(confirm, "BCD", delay);
8 ...
```

Listing 6.20: Delaying execution in DBCF

Listing 6.20 provides other example of timing management in DBCF. Here, we declare a *ScopeActivity* activity (line 2) and three nested activities (lines 3, 4 and 6). The activities are executed sequentially, since the second declares a begin on commit dependency with the first (line 5) and the third declares a begin on commit dependency with the second (line 7).

In our example, the second activity is a *TimeOutActivity* activity. Therefore, the delay configured in this activity will happen between the execution of the first and third activity.

The *TimeOutActivity* and *OnDateActivity* activities can be used for scheduling activities for a future execution. However, in DBCF once an activity is scheduled, it must be executed, because it is considered as a started activity. This creates a semantic difference with the scheduling of activities in BPEL that we presented before. The BPEL scheduling example discussed in listing 6.18, indicates that the scheduled activity (line 5) must be executed only if the scope where the event handlers are defined (lines 3 to 8) has not yet finished its execution. If the scope has finished, the *onAlarm* activity will not be active anymore, and the scheduled activity discarded.

Summary

- BPEL provides the activities *pick* and *wait* activities and the construct *eventHandlers* for managing timing concerns.
- The timing concerns that can be expressed with these constructs involve scheduling of activities, timeout management, and delay of the execution of activities.
- DBCF provides the activities *TimeOutActivity*, *OnDateActivity* and *ChooseOneEventActivity* activities for managing timing.
- DBCF can manage the same timing concerns of BPEL: scheduling of activities, timeout management, and delay of the execution of activities.

```
1 <pick>
2   <onMessage operation="operation1" ...>
3     ... <!-- an activity -->
4   </onMessage>
5   <onMessage operation="operation2" ...>
6     ... <!-- an activity -->
7   </onMessage>
8   ...
9 </pick>
```

Listing 6.21: Choice of activities based on events in BPEL

- BPEL can define schedules that are valid only during the execution time of an associated scope, while DBCF enforces that once an activity has been scheduled, it has to be executed.

6.2.7 Non deterministic choice based on events

Non deterministic choice based on events in BPEL

We already discussed the BPEL *pick* activity as a tool for expressing timeout concerns. This activity can also be used for choosing between various activities that wait for external events, such as the arrival of Web Service messages. Only one of the activities will be activated, and it will be determined by which associated message arrives first.

Listing 6.21 shows an example of this use of the *pick* activity. In our example, the activity defined in line 3, will begin if the message *operation1* arrives before message *operation2*. Conversely, the activity defined in line 6, will be executed only if the message *operation2* arrives before message *operation1*.

Non deterministic choice based on events in DBCF

Similar to the BPEL *pick* activity, the DBCF *ChooseOneEventActivity* activity can be used for choosing between activities waiting for external events. The listing 6.22 shows an example of this situation. The *ChooseOneEventActivity* defined in line 2, composes two activities that wait for events (line 3 and 4). These activities extend from the *OnMethodCallActivity* activity (this is not showed in the listing), therefore they wait for specific Web Service messages. The message that arrives first, will determine which activity will be executed.

```
1 ...
2 ChooseOneEventActivity chooser = new ChooseOneEventActivity(...);
3 InputOption1 option1 = new InputOption1(chooser, ...);
4 InputOption2 option2 = new InputOption2(chooser, ...);
5 ...
```

Listing 6.22: Choice of activities based on events in DBCF

Summary

- Both BPEL and DBCF provides constructs for managing non deterministic choice based on events.
- BPEL uses the *pick* activity and DBCF the *ChooseOneEventActivity* activity. They both are semantically equivalent, and select an activity based on which associated message arrives first.

6.2.8 Creation of workflow instances

Creation of workflow instances in BPEL

BPEL defines certain activities that wait for Web Service messages, as activities that can originate the creation of a new BPEL process instance. In other words, each time that a Web Service message destined to one of these activities arrives, a new process instance will be created. These starting activities must have the following features:

- They must be a *receive* or a *pick* activity.
- They must have set the *createInstance* attribute to *yes*.
- They must be initial activities in the sense that there are no basic activities that logically precede them in the behaviour of the process.

Listing 6.23 shows an example of an starting activity (lines 7 to 11). In addition to set the *createInstance* attribute to *yes*, this *receive* activity also initiates a *correlation set* (lines 8 to 10). As we will see in section *Correlation*, correlation sets are the strategy of BPEL for managing conversational state amongst two parties that send and receive Web Service messages. For additional discussion on correlation sets refer to the BPEL specification [ACD+03].

```
<process ... >
  ...
  <correlationSets>
    <correlationSet name="auctionIdentification" properties="as:auctionId"/>
  </correlationSets>
  ...
  <receive ... operation="provide" variable="buyerData" createInstance="yes">
    <correlations>
      <correlation set="auctionIdentification" initiate="yes"/>
    </correlations>
  </receive>
  ...
</process>
```

Listing 6.23: Starting activities in BPEL

Creation of workflow instances in DBCF

In DBCF, a *starting message* is defined as an external message that when arrives, will create a new workflow instance. Every time that a new message arrives to the DBCF workflow engine, the following happens:

- The active instances will be queried about if they are interested in the messages (we will extend on this in the section *Correlation*).
- If an instance is interested, the message will be delegated to it.
- If no instance is interested, and the message is found to be a starting message, a new instance is created and the message is delegated to it.
- If no instance is interested, and the message is not a starting message, an exception is thrown.

For determining all the possible starting messages in a Workflow, its composition activities are examined, and all the messages will be selected that can be managed for activities that have all these conditions:

- The activities must be directly nested in the workflow activity.
- They do not have any kind of begin dependency.
- They must be of type *OnExternalEventActivity* or *ChooseOneEventActivity*.

```
1 ...
2 AReserveFlight reserveFlight = new AReserveFlight(workflow, ...);
3 ...
4 AReserveHotelAndCar reserveHotelAndCar = new AReserveHotelAndCar(workflow, ...);
5 reserveHotelAndCar.setDependency(reserveHotelAndCar, "BCD", reserveFlight);
6 ...
```

Listing 6.24: Starting activities in DBCF

Listing 6.24 shows an example of some activities directly nested in the workflow activity. The *reserveHotelAndCar* activity (line 4) has a begin on commit dependency (line 5). Therefore, this activity cannot be considered a starting activity. Conversely, the activity *reserveFlight* does not have any kind of begin dependencies, and extends *OnMethodCallActivity* (this is not showed in the listing). Therefore, the event managed for this activity (the reception of a particular Web Service message) is considered as a starting event.

Summary

- BPEL and DBCF define strategies for establishing which activities are considered starting activities.
- In BPEL, starting activities can be any *receive* or *pick* activities that have the *createInstance* attribute set to *yes*, and that are initial activities in the sense that no other activities precede them in the execution of the process.
- In DBCF, starting activities can be any *OnExternalEventActivity* or *ChooseOneEventActivity* activities that are directly nested in the workflow activity and that do not have any kind of begin dependency with other activities.

6.2.9 Correlation

Correlation in BPEL

In BPEL, the correlation between a Web Service message and a specific process instance is established using correlation sets. A correlation set is represented as a set of properties related to the process state. These properties can be initialized only once in the execution of a process, and once initialized they can be used for mapping incoming messages to appropriate process instances.

Listing 6.25 shows an example of a *receive* activity associated with the correlation set (lines 4 to 6) *auctionIdentification* (line 5).

```
1 <process ... >
2   ...
3   <receive ... operation="answer" variable="auctionAnswerData">
4     <correlations>
5       <correlation set="auctionIdentification"/>
6     </correlations>
7   </receive>
8   ...
9 </process>
```

Listing 6.25: Correlation in a BPEL process

For additional discussion on how the correlation sets are mapped to properties of the message, see the BPEL specification [ACD+03].

Correlation in DBCF

As discussed in chapter 5, correlation issues are managed in DBCF with the implementation of a *correlation* method in the *OnMethodCallActivity* activities. This method receives the invocation parameters related to the reception of a Web Service message, and it should return a boolean value indicating if there is a match with the process instance or not. A match indicates that this is the target process instance for the message.

The only *OnMethodCallActivity* activities that do not provide an implementation for the correlation method, are such activities that manage web service calls that create new workflow instances. Since the instance is new, there is no state to correlate.

Listing 6.26 shows an activity that extends *OnMethodCallActivity* and that provides a boolean correlation method (lines 8 to 11). This method receives the parameters of the web service invocation, and uses it for determining if the correlation is or not successful.

Summary

- Correlation management is done in BPEL using correlation sets.
- These correlation sets are set of properties that associate a Web Service message with a particular process instance.
- Correlation management is done in DBCF with the implementation of the *correlation* method in all the *OnMethodCallActivity* activities.
- The *correlation* method receives the parameters of the invocation, and must return a boolean value indicating if there is a match between the Web Service message and

```
1  public class AReserveHotelAndCar extends OnMethodCallActivity{
2    @WorkflowState
3    ClientInfo clientInfo;
4
5    @WorkflowState(create=true)
6    XMLGregorianCalendar from;
7    ...
8    public boolean correlation(MethodCallEvent event) {
9      String id = ((ClientInfo)event.getParameterValues()[0]).getId();
10     return clientInfo.getId().equals(id);
11   }
12
13   @MethodHandler(methodHandled="reserveHotelAndCar")
14   public void onReserveHotelAndCar(ClientInfo clientInfo,
15       XMLGregorianCalendar from, XMLGregorianCalendar to) {
16     this.from = from;
17     ...
18   }
19 }
```

Listing 6.26: Correlation in DBCF

the process instance.

- The only *OnMethodCallActivity* activities that do not need to implement a *correlation* method, are such activities classified as starting activities. As discussed in section *Creation of workflow instances*, starting activities are associated with the creation of a new process instance. Since the instance is new, there is no state to correlate.

6.2.10 Exception Management

Exception Management in BPEL

Exception management in BPEL is accomplished with the *faultHandlers* construct. Fault handlers can be nested in a BPEL scope or directly in a BPEL process. Inside a *fault-handlers* element, BPEL faults can be referenced with a fault variable, a fault name, or both. If a fault occurs in a scope and it is not managed by the scope fault handlers, the compensation handler of the scope will be invoked and the fault will be rethrown to the upper scope.

Listing 6.27 shows an example of this construct, with four different kinds of fault handlers. Since this example has been extensively discussed in chapter 4, we will not repeat such discussion here.

```
<faultHandlers>
  <catch faultName="x:foo">
    ...
  </catch>
  <catch faultVariable="bar">
    ...
  </catch>
  <catch faultName="x:foo" faultVariable="bar">
    ...
  </catch>
  <catchAll>
    ...
  </catchAll>
</faultHandlers>
```

Listing 6.27: Exception handling in BPEL

Exception Management in DBCF

In DBCF there is not a special construct for exception management. Currently the error handling support is based on the mechanism of exception management of Java.

Summary

- In BPEL, exception management is accomplished with the *faultHandlers* construct.
- Fault handlers are associated with BPEL scopes or with an entire BPEL process.
- In BPEL, when an activity throws a fault, it is managed for the fault handler of the scope where the fault is defined. If it is not managed there, the compensation handler of the scope will be invoked and the fault will be rethrown to the next enclosing scope.
- DBCF does not provide any special construct or mechanism for exception handling, but uses the mechanism provided by the Java language.

6.2.11 Synchronization of activities

Synchronization of activities in BPEL

As discussed in chapter 4, synchronization dependencies for BPEL activities can be expressed using synchronization links. Synchronization links can be declared inside a *flow* activity, and can be used for coordinate the execution of activities nested on it. Each link in a *flow* activity must be related with exactly one *source* link construct and exactly one

```
1  ...
2  <flow suppressJoinFailure="yes">
3    <links>
4      <link name="buyToSettle"/>
5      <link name="sellToSettle"/>
6    </links>
7    ...
8    <receive name="getBuyerInformation">
9      <source linkName="buyToSettle"/>
10   </receive>
11   ...
12   <receive name="getSellerInformation">
13     <source linkName="sellToSettle"/>
14   </receive>
15   ...
16   <invoke name="settleTrade"
17     joinCondition="bpws:getLinkStatus('buyToSettle')
18       and bpws:getLinkStatus('sellToSettle')">
19     <target linkName=" buyToSettle "/>
20     <target linkName=" sellToSettle "/>
21   </invoke>
22   ...
23 </flow>
24 ...
```

Listing 6.28: Activities Synchronization in BPEL

target link construct. This implies that each synchronization concern needs to be expressed with at least these three constructs, which are tangled in the process code.

Synchronization links basically express begin dependencies, activities that nest *target* link constructs, cannot be executed until the outcomes of all these target links be known. These outcomes will be solved when the activities that nest the corresponding *source* link constructs have finished.

In addition, activities that nest *target* links can express their synchronization dependencies with a boolean logic expression in terms of the outcomes of the *source* links associated with the *target* links, using the standard attribute *joinCondition*.

Listing 6.28 shows a BPEL fragment, where the activity *settleTrade* (line 16 to 21) expresses a synchronization dependency with both *getBuyerInformation* (line 8 to 10) and *getSellerInformation* (line 12 to 14). This synchronization dependency is established using a logic boolean expression (lines 17 and 18) in terms of the outcomes of the *buyToSettle* and *sellToSettle* links. Since this example was discussed in chapter 4, we will not give more details here.

We also discussed that the transactional outcomes of activities could be determined with

```
1  <process ... suppressJoinFailure="yes">
2    ...
3    <flow>
4      <links>
5        <link name="receive-to-assess"/>
6        <link name="receive-to-approval"/>
7      </links>
8      ...
9      <receive ...>
10       <source linkName="receive-to-assess"
11         transitionCondition="bpws:getVariableData('request','amount') &lt; 10000"/>
12       <source linkName="receive-to-approval"
13         transitionCondition="bpws:getVariableData('request','amount')>=10000"/>
14     </receive>
15     ...
16     <invoke ...>
17       <target linkName="receive-to-assess"/>
18     </invoke>
19     ...
20     <invoke ...>
21       <target linkName="receive-to-approval"/>
22     </invoke>
23     ...
24   </flow>
25   ...
26 </process>
```

Listing 6.29: Using transition conditions for activities Synchronization in BPEL

logical expressions in terms of state variables of the process, using the *transitionCondition* attribute in source links. Using *source links*, activities can have more than one possible named outcome. However, we mentioned in chapter 4, that dependencies based on state variables are a redundant mechanism, since similar synchronization dependencies can be expressed with traditional control flow constructs.

Listing 6.29 shows an example of the utilization of *transitionCondition* expressions. As in the last example, we will not repeat the discussion of the code here, since it is explained in detail in chapter 4.

Synchronization of activities in DBCF

As discussed in chapter 5, in DBCF synchronizations are expressed using ACTA dependencies. Listing 6.30 shows an example of a synchronization concern in DBCF. Line 2 declares a *ScopeActivity* activity that nests two activities: *rh* (line 4) and *rc* (line 6). As we discussed in the section *Concurrent Activities*, all the activities nested in a *ScopeActivity*

```
1 ...
2 ScopeActivity reserveHotelAndCar = new ScopeActivity(...);
3 ...
4 AReserveHotelWithPartner rh = new AReserveHotelWithPartner(reserveHotelAndCar, ...);
5 ...
6 AReserveCarWithPartner rc = new AReserveCarWithPartner(reserveHotelAndCar, ...);
7 ...
8 rc.setDependency(rc, "BCD", rh);
9 ...
```

Listing 6.30: Synchronization handling in DBCF

will be executed concurrently. However, an ACTA *begin on commit dependency* is defined amongst these two transactions (line 8), and as we discussed in chapter 3, the semantic of an ACTA *begin on commit dependency* implies that the source activity of the dependency (*rc*) cannot begin until the destination activity (*rh*) has successfully finished. Therefore, a synchronization is established, since *rc* will block its execution until the finishing of *rh*.

We discussed also in chapter 5, that the advantage of using a model based on ACTA, is that in addition to have a broad range of transactional dependencies already defined, it is also possible the definition of new dependencies, according to the particularities of a specific problem.

Summary

- BPEL employs synchronization links as a strategy for managing synchronization concerns.
- This strategy has the tradeoff that for each synchronization concern, at least three constructs must be used, tangled with the main concern of the application: a *synchronization link* declaration, a *source* link and a *target* link.
- Synchronization concerns can be expressed as logic boolean expressions in terms of the outcomes of synchronization links, using the *joinCondition* attribute of the target link elements.
- An activity can have more than one named outcome, using source link elements.
- The outcomes of these source link elements can be boolean expressions in terms of state variables of the process, using the *transitionCondition* attribute of the source link elements.
- DBCF uses a strategy based on ACTA dependencies for expressing synchronization concerns.

```
1 <scope>
2   <compensationHandler>
3     <invoke operation="CancelPurchase" .../>
4   </compensationHandler>
5   <invoke operation="SyncPurchase" .../>
6 </scope>
```

Listing 6.31: Compensation handling in BPEL

- The *begin on commit dependency* can be used for expressing the basic synchronization concern that one activity cannot begin until another one has successfully finished.
- However, other kind of dependencies can be defined, according to the specific business problem that has to be solved.

6.2.12 Compensation of activities

Compensation of activities in BPEL

BPEL provides the *compensationHandler* construct for expressing the idea of compensation. This element can be associated with a BPEL scope or be directly nested inside a BPEL process. In BPEL, the compensation handler can be invoked explicitly, or it will be implicitly invoked if a fault is thrown and not managed by a fault handler in a particular scope.

Listing 6.31 shows an example of compensation handling in BPEL. In this example, the invoke activity defined in line 3, provides a semantic compensation for the activity defined in line 5. For additional discussion of this construct see chapter 4.

Compensation of activities in DBCF

As discussed in chapter 5, DBCF employs in addition to ACTA dependencies, the concept of secondary activities for defining compensation handling. Secondary activities are defined as all these activities that are not part of the main concern of the application, but to the concern of transaction management [Fab05].

Listing 6.32 shows an example of compensation handling. In this example, the activities *rc* (line 4) and *ch* (line 6) are nested inside the *reserveHotelAndCar* activity. The *ch* activity is configured as a secondary activity in line 7, using the method *setExecutionMode* and passing as a parameter the constant value SECONDARY_ACTIVITY. In line 8, the *CPD* dependency is established amongst the *ch* and *rc* activities.

```
1 ...
2 ScopeActivity reserveHotelAndCar = new ScopeActivity(...);
3 ...
4 AReserveHotelWithPartner rh = new AReserveHotelWithPartner(reserveHotelAndCar, ...);
5 ...
6 ACancelHotelWithPartner ch = new ACancelHotelWithPartner(reserveHotelAndCar, ...);
7 ch.setExecutionMode(ExecutionMode.SECONDARY_ACTIVITY);
8 ch.setDependency(ch, "CPD", rh);
9 ...
```

Listing 6.32: Compensation handling in DBCF with a customized dependency

We defined in chapter 5 the *CPD* dependency as a variation of the ACTA compensation dependency *CMD*. Our dependency implies by itself all the constraints related to compensation. Therefore, *ch* must start only if *rc* has committed and the upper scope activity has aborted. In addition, the *ch* activity cannot abort, instead a commit must be enforced, since an abort of this activity will leave the process instance in an inconsistent state. These implicit constraints can be established amongst more than two activities using a single dependency, since the composition model of the transactions is known (a hierarchical composition model), and this implies that certain transactions can be inferred implicitly (such as the parent of the transactions participating in the dependency). For additional discussion on this example and compensation management in DBCF, see chapter 5.

Summary

- BPEL uses a special construct for expressing compensation management.
- The compensation activities in BPEL, can be invoked explicitly, or implicitly if a fault is thrown and it is not managed by a fault handler in a particular scope.
- In DBCF, compensation management is done with the concepts of secondary activities and implicit dependencies.
- Secondary activities are such activities that are not part of the main concern of the application, but to the transactional concern.
- Implicit dependencies are a variation of ACTA dependencies. They can enforce constraints between more than a pair of activities.
- Implicit dependencies are possible if the composition model of the transactions participating in the dependencies is known.

```
1 <scope>
2   <invoke partnerLink="shipping" operation="requestShipping" ... />
3   <faultHandlers>
4     <catchAll>
5       <invoke partnerLink="shippingAlt" operation="requestShipping" ... />
6     </catchAll>
7   </faultHandlers>
8 </scope>
```

Listing 6.33: Functional replication in BPEL

6.2.13 Functional Replication

Functional Replication in BPEL

As we discussed in chapter 4, functional replication in BPEL is just a case of exception management. This can has the disadvantage that the functional replication concern could be tangled with other exception management concerns.

Listing 6.33 shows an example of functional replication in BPEL. A fault handler is declared in a scope where a particular activity is defined (line 2). In case that the activity thrown a fault when executed, an alternative activity (line 5), that provides an equivalent functionality will be executed. In our example, the generic fault handler *catchAll* is used, but more specific ones could be employed.

Functional Replication in DBCF

As in compensation management, functional replication in DBCF is expressed with the concept of secondary transactions, for denoting that the functional replication activity is not part of the main concern of the application, but to the transaction management concern.

Listing 6.34 shows an example of functional replication handling in DBCF. In our example, the *reserveHotelAndCarA* is a primary activity, and *reserveHotelAndCarB* is a secondary activity, since in line 5 the *setExecutionMode* of *reserveHotelAndCarB* is invoked passing as parameter the constant value *SECONDARY_ACTIVITY*. In addition, line 6 declares a *begin on abort dependency*, taking as a source of the dependency the *reserveHotelAndCarB* activity and as a target, the *reserveHotelAndCarA* activity. As discussed in chapter 3, this ACTA dependency implies that *reserveHotelAndCarB* can only begin if *reserveHotelAndCarA* has aborted, implementing in this way a functional replication concern.

```
1 ...
2 ScopeActivity reserveHotelAndCarA = new ScopeActivity(reserveHotelAndCar, ...);
3 ...
4 ScopeActivity reserveHotelAndCarB = new ScopeActivity(reserveHotelAndCar, ...);
5 reserveHotelAndCarB.setExecutionMode(ExecutionMode.SECONDARY_ACTIVITY);
6 reserveHotelAndCarB.setDependency(reserveHotelAndCarB, "BAD", reserveHotelAndCarA);
7 ...
```

Listing 6.34: Functional Replication handling in DBCF

Summary

- Functional replication management is implemented in BPEL as a particular case of exception management.
- The problem of considering functional replication as error management, is that this concern will be tangled with other error management concerns.
- DBCF uses ACTA dependencies and the concept of secondary transactions for management of functional replication concerns.

6.3 Summary of differences between BPEL and DBCF

In this section we will present a summary of the differences discussed in the previous section.

6.3.1 Scoping

- In BPEL, scopes define an entire behaviour context for execution of activities.
- They also define a namespace where state variables can be defined.
- Only one activity can be directly nested in a BPEL scope, if more than one is needed, a structured activity must be used.
- In DBCF, all the activities define a namespace where state variables can be defined.
- The *ScopeActivity* serve as a base class for all the structured activities . It can contain both primary and secondary activities.
- The composition model of DBCF is simpler than the BPEL one, since in the first not explicit scopes are needed for declaring namespaces for variables, and more than one activity can be directly nested in a scope.

6.3.2 State management

- BPEL variables are declared within the *variables* construct.
- The declaration of variables in BPEL must be associated with a BPEL scope. Therefore, every time that a new variable is declared, a new scope must be created.
- Additional manipulation of state variables, is done in BPEL with special constructs (such as *assign*) and other expressions.
- DBCF relies on the *WorkflowState* annotation for declaring the state variables of a process.
- The *WorkflowState* annotation can be parametrized with the *create* parameter.
- If the *create* parameter has the value of *true*, a new variable definition will be created in the scope of the DBCF activity where such variable is declared.
- If the *create* parameter has the value of *false* or it is omitted, the variable has already been declared in an upper scope. Therefore, a new entry will not be created on the variable dictionary of the activity scope, and the variable can be used without having been locally initialized.

6.3.3 Concurrent activities

- BPEL needs a special construct for declaring concurrency: the *flow* construct.
- DBCF does not need a dedicated construct. It uses the scope construct for declaring a nesting of activities. If no dependencies are declared the nested activities will be executed concurrently.

6.3.4 Sequential activities

- BPEL needs a special construct for declaring sequential execution: the *sequence* activity.
- DBCF does not need a special construct since the activities nested in a scope will be executed sequentially if begin on commit dependencies are declared amongst them.
- DBCF could implement a special construct for expressing sequencing, like the one used in BPEL, if it is determined that the declaration of begin dependencies is a cumbersome work for expressing sequential execution.

6.3.5 Other control flow constructs

- Both BPEL and DBCF provide constructs for expressing conditional execution and iteration.

- BPEL uses the *switch* activity for expressing conditional execution, and the *while* activity for expressing iteration.

- DBCF provides the *ConditionalActivity* activity for expressing simple conditions, the *ChooseOneConditionalActivity* activity is useful for defining more complex conditions, and is composed of one or more *ConditionalActivity* activities. Finally, the *RepetitionActivity* activity can express iteration, and is also based on a *ConditionalActivity* activity for defining how many iterations it must accomplish.

6.3.6 Time constructs

- BPEL provides the activities *pick* and *wait* activities and the construct *eventHandlers* for managing timing concerns.

- The timing concerns that can be expressed with these constructs involve scheduling of activities, timeout management, and delay of the execution of activities.

- DBCF provides the activities *TimeOutActivity*, *OnDateActivity* and *ChooseOneEventActivity* activities for managing timing.

- DBCF can manage the same timing concerns of BPEL: scheduling of activities, timeout management, and delay of the execution of activities.

- BPEL can define schedules that are valid only during the execution time of an associated scope, while DBCF enforces that once an activity has been scheduled, it has to be executed.

6.3.7 Non deterministic choice based on events

- Both BPEL and DBCF provides constructs for managing non deterministic choice based on events.

- BPEL uses the *pick* activity and DBCF the *ChooseOneEventActivity* activity. They both are semantically equivalent, and select an activity based on which associated message arrives first.

6.3.8 Creation of workflow instances

- BPEL and DBCF define strategies for establishing which activities are considered starting activities.
- In BPEL, starting activities can be any *receive* or *pick* activities that have the *createInstance* attribute set to *yes*, and that are initial activities in the sense that no other activities precede them in the execution of the process.
- In DBCF, starting activities can be any *OnExternalEventActivity* or *ChooseOneEventActivity* activities that are directly nested in the workflow activity and that do not have any kind of begin dependency with other activities.

6.3.9 Correlation

- Correlation management is done in BPEL using correlation sets.
- These correlation sets are set of properties that associate a Web Service message with a particular process instance.
- Correlation management is done in DBCF with the implementation of the *correlation* method in all the *OnMethodCallActivity* activities.
- The *correlation* method receives the parameters of the invocation, and must return a boolean value indicating if there is a match between the Web Service message and the process instance.
- The only *OnMethodCallActivity* activities that do not need to implement a *correlation* method, are such activities classified as starting activities. As discussed in section *Creation of workflow instances*, starting activities are associated with the creation of a new process instance. Since the instance is new, there is no state to correlate.

6.3.10 Exception Management

- In BPEL, exception management is accomplished with the *faultHandlers* construct.
- Fault handlers are associated with BPEL scopes or with an entire BPEL process.
- In BPEL, when an activity throws a fault, it is managed for the fault handler of the scope where the fault is defined. If it is not managed there, the compensation handler of the scope will be invoked and the fault will be rethrown to the next enclosing scope.
- DBCF does not provide any special construct or mechanism for exception handling, but the provided by the Java language.

6.3.11 Synchronization of activities

- BPEL employs synchronization links as a strategy for managing synchronization concerns.
- This strategy has the tradeoff that for each synchronization concern, at least three constructs must be used, tangled with the main concern of the application: a *synchronization link* declaration, a *source* link and a *target* link.
- Synchronization concerns can be expressed as logic boolean expressions in terms of the outcomes of synchronization links, using the *joinCondition* attribute of the target link elements.
- An activity can have more than one named outcome, using source link elements.
- The outcomes of these source link elements can be boolean expressions in terms of state variables of the process, using the *transitionCondition* attribute of the source link elements.
- DBCF uses a strategy based on ACTA dependencies for expressing synchronization concerns.
- The *begin on commit dependency* can be used for expressing the basic synchronization concern that one activity cannot begin until another one has successfully finished.
- However, other kind of dependencies can be defined, according to the specific business problem that has to be solved.

6.3.12 Compensation of activities

- BPEL uses a special construct for expressing compensation management.
- The compensation activities in BPEL, can be invoked explicitly, or implicitly if a fault is thrown and it is not managed by a fault handler in a particular scope.
- In DBCF, compensation management is done with the concepts of secondary activities and implicit dependencies.
- Secondary activities are such activities that are not part of the main concern of the application, but to the transactional concern.
- Implicit dependencies are a variation of ACTA dependencies. They can enforce constraints between more than a pair of activities.
- Implicit dependencies are possible if the composition model of the transactions participating in the dependencies is known.

6.3.13 Functional Replication

- Functional replication management is implemented in BPEL as a particular case of exception management.
- The problem of considering functional replication as error management, is that this concern will be tangled with other error management concerns.
- DBCF uses ACTA dependencies and the concept of secondary transactions for management of functional replication concerns.

6.4 Analysis of the differences

In this section, we want to make an analysis of the differences that we discovered in our survey. We have preferred to include only the concerns that presented significative differences, and that can contribute with the outline of future research directions. We have classified the relevant differences in three categories:

- Advantages of DBCF over BPEL.
- Advantages of BPEL over DBCF.
- Problems related to transaction management of the two proposals.

In the rest of this section we will discuss each of these topics.

6.4.1 Advantages of DBCF over BPEL

In this section we outline the main contributions of the DBCF transactional model to the one used in BPEL.

- DBCF provides an extensible dependency model based on the ACTA framework. Transactional concerns can be expressed with dependencies already defined in previous works, like [CR91] and [CR92], or new kind of dependencies can be defined if needed, easing in this way the use of new transactional models.
- The model proposed in DBCF does not require the tangled code that is produced in BPEL when applying synchronization concerns. In BPEL, a simple synchronization concern needs the scattering of different constructs over the main business concern of the application. In DBCF, only one ACTA dependency has to be defined for expressing the same concern.

- DBCF reduces the number of constructs that are needed for expressing transaction management concerns. The concepts of ACTA dependencies with implicit constraints and secondary transactions are well suited for expressing straightforwardly the concerns of *synchronization of activities*, *functional replication* and *compensation management*.

- BPEL in the other hand, needs three different constructs for expressing *synchronization of activities*, another construct for *compensation management*, and the *functional replication* concerns are tangled with other concerns related with exception handling.

- Therefore, while DBCF proposes a simple common model for managing the transactional concerns in services composition, BPEL requires dedicated constructs for each different transactional concern.

6.4.2 Advantages of BPEL over DBCF

In this section we will mention some advantages of BPEL over DBCF. Although BPEL is a well proven composition language, and DBCF is experimental, we have decided to mention here only the transactional related features that can contribute a significative improvement to the current transactional proposal of DBCF, ignoring the features related to infrastructural services.

- The exception management is more powerful than the currently implemented in DBCF. Exceptions in BPEL are propagated following the hierarchical composition of activities, and can be handled by any of the activities that receive the exception. In DBCF, exception management is limited to the Java exception handling mechanism, and if an exception is not caught for an activity, it will stop the life cycle of the process instead of being thrown to an upper activity.

- Although most of the timing management features of BPEL have been implemented in DBCF, in BPEL it is possible to express certain sophisticated ideas that in DBCF cannot be expressed. Like that an activity can be scheduled to be executed on a particular time, but only if the associated scope where the activity is defined still has not finished its execution.

- In addition, BPEL has already defined a mechanism for expressing temporal predicates, while in DBCF it has not yet been done.

- The synchronization model of BPEL can use boolean logic expressions in terms of the outcomes of transactions, for expressing conditions in which a specific activity can begin its execution. Though the only constraints that can be expressed are sophisticated *begin on commit* or *begin on abort* dependencies (instead of the broader

kind of dependencies that can be specified with DBCF), the possibility of including logic predicates for expressing dependencies should be taken into account as a future direction of research.

6.4.3 Problems related to transaction management of the two proposals

- Both in BPEL and DBCF, the transaction management concerns crosscut and are tangled with the main concern of the application. Although as stated before DBCF has less tangled code than BPEL.

- There is no mechanism for the business process be distributed or span multiple platforms. The BPEL specification suggests the use of the WS-Transaction specification for accomplishing it. However, as discussed in chapter 2, this specification only defines a protocol for signaling and distributing transactional messages, it is not a complete solution since it needs to be complemented with a great amount of similar specifications [CCF+05c] [CCF+05b] [CCF+05a] [BCC+04] [BBC+06b] [BBC+06a], it is not clear if it will be an open standard, and similar competitive specifications are present in industry [BCH+03a] [BCH+03b] [BCH+03c] [BTP], obstructing in this way the emerging of a universally accepted standard.

6.5 Conclusion

In this chapter we have compared extensively DBCF with BPEL, the state of the art in industry for Web Services composition. We have showed first a brief overview of the differences of these two proposals, and after we have presented a detailed comparison of them. Once this work concluded we have presented a summary of the differences, and an analysis section has been developed. This determines in the context of transaction management, what are the advantages and disadvantages that these proposals present when compared with each other, and the common disadvantages that they have for dealing with advanced transaction management concerns.

In the next chapter, we will demonstrate how the concepts discussed in our model, when applied to BPEL, will bring a significant improvement to the language, and therefore to the current state of the art on Web Services composition.

Chapter 7

An extension to BPEL4WS that follows our model

The transactional concerns present in BPEL have already been discussed in chapter 4 (e.g. activities synchronization, activities compensation and functional replication), and in chapter 6 we claimed that the use of the BPEL constructs for expressing transaction management issues is limited and not extensible, involves a great amount of code and it is not the best simple way for expressing these concerns.

We will continue discussing this claim here, and afterwards we will present the features of a BPEL extension that overcomes these limitations, following the conceptual model employed in DBCF for managing advanced transactional concerns in composition languages like BPEL. We do not provide an implementation of a BPEL engine with support for our proposed extension, but such implementation is part of our future work.

7.1 Why an extension

We believe that the explanation of the necessity of an extension for BPEL can only be completed with a case study example comparing a traditional BPEL implementation and one implementation using our proposed constructs. For that reason, although here we will present theoretically the problems of the BPEL constructs and a description of how our extension manages the same problems, we have decided to dedicate the entire chapter 8 to the presentation of a practical code-based demonstration.

The following table shows a classification of the constructs used for transaction manage-

Transactional concern	**Constructs**
Activities synchronization	Control links declaration, source links elements, target links elements
Activities compensation	Compensation Handlers
Functional replication	Fault handlers

Table 7.1: BPEL constructs for dealing with transaction management concerns

ment in BPEL[1]:

All these constructs have already been discussed in chapter 2, therefore we will not give a semantic description of them again. However, some features of this model require review:

- The proliferation of different constructs for managing these few concerns: There are five constructs in total. Three for expressing each synchronization concern, one for compensation handling and one for functional replication handling.
- The scattering of different constructs when expressing synchronization dependencies between activities: The three constructs needed for declaring a synchronization dependency must be located in different places of the code. Therefore, for expressing something as simple that in a flow element activity A cannot begin until activity B has finished, it is necessary: 1) to declare a control link at the beginning of the flow element, 2) to insert a source element referencing this link in activity B and 3) to insert a target element referencing this link in activity A.
- The lack of extensibility of the model: All the possible transaction management issues have to be expressed combining these fixed constructs, there are no defined extension points in this model. Therefore, dealing with new transactional models can be a cumbersome or even impossible task.
- The lack of a common basis for expressing transactional concerns: In this model, each transactional concept has its own unrelated set of constructs. This causes a proliferation of constructs that can eventually complicate the development.

We believe that these arguments are enough for demonstrating the necessity of an improved alternative solution in BPEL. In the next section we will offer such solution.

[1] Although fault handlers have been presented as a construct for transaction management in BPEL, they are not always necessarily related to transaction management. Only when used for expressing functional replication concerns this is the case.

7.2 Extending BPEL

In this section we will recommend constructs that propose an alternative to the way BPEL implements transactional concerns. We will provide here semantic explanations of the constructs and short examples, since full code-based examples for demonstrations will be given in our case study presented in chapter 8.

In general, we have preferred to describe our proposed BPEL elements using the same simple notation used in the BPEL specification, instead of providing lengthy XSD or DTD formal descriptions.

The objectives of the BPEL extension we are going to describe are:

- Presenting a common model with a simple unified set of constructs for dealing with the transactional concerns currently managed in BPEL: Dealing with different and no related set of constructs for each transactional concern bring complication to the programming process, since more concepts need to be memorized and the code grows in size. Conversely, the adoption of just one strongly expressive model with fewer constructs would accelerate the learning process and reduce the size of our programs.
- Providing well defined extension points in our model: There are a great amount of transactional models, and each business can have different and very particular requirements related to transaction management. No providing extension points implies that we are limited to the expression of the transactional concerns that can be written with our language elements. Additional transactional issues would be very difficult to be expressed, or impossible at all.
- Reducing the scattering of code when applying synchronization of activities: The scattering of code creates software evolution problems, since the code pertaining to one particular concern is not present in one single place, but distributed along code related to other concerns. This problem is present in BPEL each time an activity synchronization concern is expressed.

We will follow as a strategy the selection of constructs that allows us to integrate the proposals of the conceptual model we presented in chapter 5 into BPEL. This is because this strategy fulfills all the requirements exposed above.

Therefore, the main ideas that our extension should be able to abstract are:

- The notion of associating state with activities, where possible states can be: *non-initiated*, *began*, *committed* or *aborted* (as specified before, this is an open list of possible states, since more sophisticated transactional models can have additional states).

- The possibility of establishing dependencies between different activities using the well known extensible dependency model of the ACTA framework.
- The concept of activities that are not part of the business case that the application is trying to solve, but that are related to other non-functional concerns like transaction management.

We will discuss in the rest of this section the constructs that should be added to BPEL in order to implement these ideas.

7.2.1 Suppressing failures

For being compatible with the idea of associating state with BPEL activities, the only additional concept that should be added to the language is the notion of "failed activities". A failed or aborted activity is an atomic or structural activity in BPEL that has already been executed, and that such execution has not been successful. This lack of success can be due to various circumstances, in the case of an invocation activity for example, it can be the presence of a fault in the output message of a web service invocation (see chapter 2 for a description of faults in BPEL). In other more complex cases, such as structural activities (activities that enclose other activities), a failure can be due to the throwing of a fault in one of the internal activities.

The standard behaviour in BPEL when this kind of problems arise, is to manage the fault if a fault handler exists in the scope where such fault happened. If no appropriate fault handler is available, the fault will be thrown to the enclosing scope. This behaviour is not compatible with our model, where sometimes a fault occurrence only means the assignation of a new state to the activity (aborted) where this fault occurred, without necessarily implying the throwing of the fault notifying a problem to the enclosing scope.

We propose a simple syntactic construct that expresses this idea, and that could be added to the common *standard-elements* of BPEL. As we saw in chapter 2, the BPEL specification proposes a set of standard elements that can be potentially included in all the activities. This set of elements is currently limited to *source* and *target* links. The additional element proposed for us follows this structure:

```
<suppressFailure> boolean <suppressFailure/>
```

The following example shows an invocation activity with a nested *suppresFailure* element:

```
<invoke name="requestShipping" ... >
```

```
  <supressFailure>true</supressFailure>
</invoke>
```

The inclusion of a *suppressFailure* element with a true value inside a BPEL activity would be semantically equivalent to the inclusion of an implicit scope immediately containing the activity, with a dummy fault handler. The following example illustrates this:

```
<scope>
  <invoke name="requestShipping" ... />
  <catchAll />
<scope/>
```

7.2.2 Dependencies between activities

Once we have adopted the notion of activities with state in BPEL, the next step proposed in our model is establishing relationships between activities present in a particular BPEL scope using ACTA dependencies. Since the ACTA formal model and the concept of ACTA dependencies have been already explained in chapter 3, we will only explain here our proposed extensions for including these concepts in BPEL.

Identification issues

The ACTA framework proposes a way of declaring dependencies between pairs of activities (transactions). Therefore, for defining ACTA dependencies in a programing language like BPEL, we need an identification mechanism for representing the activities that are declared in the dependencies. Since in BPEL the *name* attribute is part of the collection of attributes defined as *standard-attributes* at the specification (*standard-attributes* are defined as all such attributes that can be present in any BPEL activity), we have preferred to use this identifier for defining ACTA dependencies between two activities, instead of defining a new dedicated attribute.

Defining ACTA dependencies

With the identification problem solved, we have created a construct for defining dependencies between activities, using activity names for source and destination of the dependency, and a string representing the type of the ACTA dependency that is being established between the source and destination activities. The following code shows how the structure of a dependency is written:

```
<dependency type="ncname" source="ncname"? destination="ncname" />
```

In this example [2], the *source* attribute represents the source activity, and the *destination* attribute represents the destination activity. The question mark near the *source* attribute, denotes that it is optional, since when omitted, it will represent by default the activity that declares the dependency.

The possible values of the type attribute can be some of the traditional ACTA dependencies that have been discussed in other works (see [CR92] for example), or any additional dependency that could be defined between BPEL activities, according to the characteristics of a specific transactional model.

The following is an example of the first case, where the "BAD" (*Begin on Abort Dependency*) is defined in [CR92] as a dependency for specifying that the source activity (A) will not begin until the destination activity (B) have failed or aborted:

```
<invoke name="A" ... >
  <dependency type="BAD" destination="B" />
</invoke>
```

In the other hand, the following example shows a relationship based on the success of a particular activity. The "BCD" (*Begin on Commit Dependency*) is defined in [CR92] as a dependency for specifying that the source activity (C) will not begin until the destination activity (D) have successfully finished or committed:

```
<invoke name="C" ... >
  <dependency type="BCD" destination="D"/>
</invoke>
```

Extending the ACTA dependency model

Now we could define a new kind of ACTA dependency for BPEL, no mentioned in previous works about ACTA like the example discussed above. Lets say that we want to express in a single construct the concept of compensation. As mentioned in previous chapters, subtransactions in a long lived transaction cannot block resources until they finish, instead they release the resources after they have finished to use them. Therefore, the problem is what happen with the committed subtransactions if the overall long lived transaction

[2] *ncname* stands for "non-qualified name". It is a term used in XML for denoting a name that can be expressed without specifying a particular namespace.

eventually fails. It is solved defining compensation transactions for each atomic subtransaction, where the former should be executed only if the latter commits (Begin on Commit Dependency), and the enclosing long lived transaction fails (Begin on Abort Dependency). In addition, the compensating transaction should always finalize with a commit, since an abort would produce an inconsistent state. A compensation dependency (*CMD*) has been already defined in [CR92] in the following way:

A transaction *Tj* has a *Compensation Dependency* with a transaction *Ti* (or: "*Tj CMD Ti*") when: if *Ti* aborts, *Tj* must commit.

However, this dependency only expresses one of the concerns related to compensation, since it does not mention the Begin on Commit Dependency with the compensated transaction, and the Begin on Abort Dependency with the long lived transaction. Therefore, if our simplification objective is to provide a single construct, we should define one single expression that represents all the mentioned ideas . Therefore, we have defined a "Complete Compensation Dependency" as a dependency that express all these ideas. For simplification however, we will refer to this dependency in the rest of this work as a Compensation Dependency, but we will identify it with the letters *CPD* instead of the more limited ACTA definition of Compensation Dependency *CMD*. Our new created dependency could be written in the following way in BPEL:

```
<invoke name="cancelShipping" ... >
  <dependency type="CPD" destination="requestShipping"/>
</invoke>
```

In this example, the source activity (*cancelShipping*), can only begin if the destination activity (*requestShipping*) has first committed and if the enclosing scope of this activity has aborted [3].

In the above example, we have created an abstraction for an already existing concept in BPEL: the compensation activities (see chapter 2). Now we would like to create an abstraction for an idea that cannot be straightforwardly expressed in BPEL: Lets consider an hypothetical situation where a particular iteration activity *test2* nested in a sequence activity, cannot begin until another iteration activity *test1* nested in another sequence activity has begun [4]. Our new dependency could be named *begin on begin* dependency and our hypothetical situation could be represented as:

[3] If there is more than one enclosing scope surrounding *requestShipping*, the dependency is more complex, since the compensation activity should be triggered if one of the enclosing scopes fails, no necessarily the immediately enclosing scope. In fact, this is always the case when declaring activities inside a BPEL scope, since there is always at least one additional enclosing scope: the process scope.

[4] This is a variation of the classical ACTA *begin on commit dependency*, where a particular activity cannot begin until another one has successfully finished. This concept is easily expressed with synchronization links in BPEL, but that is not the case with our variation.

```
<flow>
  <sequence>
    ...
    <while name="test1" ... />
    ...
  </sequence>

  <sequence>
    ...
    <while name="test2" ... >
      <dependency type="BBD" destination="test1"/>
    </while>
    ...
  </sequence>
</flow>
```

Where "BBD" represents the *begin on begin* dependency described above. In this example, both sequence activities are executed concurrently since they are nested in a flow activity. However, the second sequence will only execute their activities until the activity before test2, and at this point, the sequence will suspend its execution until the moment that test1 begins.

We could indefinitely continue extending the possible dependencies of our model, rising the level of programming abstraction as needed in particular problems and situations, improving in this way our capacity for dealing with new transactional models.

Simplifying BPEL process with dependencies

We have mention that our model based in ACTA dependencies is powerful enough both for expressing the transactional concerns already present in BPEL and for new kind of transactional models that can be easily represented using the extension points of the constructs. In addition to that, our solution is simpler. For demonstrating that, lets consider the extract of code showed in listing 7.1 (a fragment of our case study showed in chapter 8). Here we have three concurrent sequence activity blocks (they are concurrent since they are nested in a flow activity). We can see one activity in each one of the sequences: a switch activity in the first sequence, and invocation activities in the second and third sequence. In this problem we want to express the fact that the invocation activities of both the second and third sequences, have a synchronization dependency in relationship with the switch activity of the first dependency. It can be expressed in BPEL using synchronization links, therefore for representing the first synchronization dependency, we have to follow these steps:

- Declaring the link *ship-to-invoice* in the flow activity.
- Nesting a source link element referencing the *ship-to-invoice* link, inside the switch activity of the first sequence.
- Nesting a target link element referencing the *ship-to-invoice* link, inside the invoke activity of the second sequence. activity.

Similarly, for representing the second synchronization dependency, we have to follow these steps:

- Declaring the link *ship-to-scheduling* in the flow activity.
- Nesting a source link element referencing the *ship-to-scheduling* link, inside the switch activity of the first sequence.
- Nesting a target link element referencing the *ship-to-scheduling* link, inside the invoke activity of the third sequence. activity.

At this point we want to make emphasis in a fact: Although what we have are two activities (one invocation activity at the second sequence and another at the third sequence) with begin dependencies to the same activity (the switch inside the first sequence), we do not have in BPEL simple constructs for expressing straightforwardly this idea. And since the same links cannot be reused for expressing more than one synchronization dependencies[5], for each new synchronization dependency we will need always a new link declaration, a new source link construct and a new target link construct.

We could express that arithmetically as:

```
SC = 3 * N
```

Where:

SC represents the number of Synchronization Constructs.

N represents the activities synchronization concerns of the application.

Clearly, this relationship will create an excessive number of constructs related only to activities synchronization in the same business process, overwhelming the development and maintainability of the application. In our particular example, the number of synchronization concerns (N) is two, therefore we need six constructs only for expressing these

[5]Textually from the BPEL specification: "Every link declared within a flow activity MUST have exactly one activity within the flow as its source and exactly one activity within the flow as its target."

concerns: two links declarations (lines 5 and 6), two source link references (lines 13 and 14), and two target link references (lines 21 and 29). As one additional complication, all these constructs are scattered in the process code (at the beginning of a flow activity, inside the source activities and inside the target activities).

In the other hand, listing 7.2 shows the same example using our extension. Since we need only one ACTA dependency for expressing each transactional concern, the number of constructs is the same that the number of synchronization concerns needed. Therefore, if we have two synchronization concerns, we need only two constructs for expressing them (line 14 and 22).

7.2.3 Secondary Activities

We have defined *secondary activities* as activities that are not part of the business concern of the application, but related to other concerns like transaction management. They are associated with a BPEL scope, and they express transactional relationships with the *primary activity* of a scope (or activities nested in it) using ACTA dependencies.

Secondary activities can be represented inside a scope, using the following construct:

```
<scope>
  ...
  <secondaryActivities>
    ...
  </secondaryActivities>
</scope>
```

As discussed in chapter 2, BPEL scopes can have nested only one activity, called *primary activity*. This activity can be however a complex structured activity, allowing in this way the nesting of more than one activity in a single scope. Additionally, a scope is considered finished when its primary activity has finished.

In our extension, we consider that a scope cannot be finished if at the moment of the completion of its primary activity, secondary activities are running. If that is the case, the scope has to wait for the completion of these activities before being considered as finished. If no secondary activities are running at the moment of the completion of the primary activity, the scope should be considered as finished, although secondary activities have not yet begun.

The rationale because a scope should not wait for their secondary activities that have not begun after the completion of the primary activity, is because some of these activities will

```
<process ...>
    ...
    <flow>
      <links>
        <link name="ship-to-invoice"/>
        <link name="ship-to-scheduling"/>
      </links>

      <sequence>
        ...
        <switch>
          ...
          <source linkName="ship-to-invoice"/>
          <source linkName="ship-to-scheduling"/>
        </switch>
      </sequence>

      <sequence>
        ...
        <invoke ...>
          <target linkName="ship-to-invoice"/>
        </invoke>
        ...
      </sequence>

      <sequence>
        ...
        <invoke ...>
          <target linkName="ship-to-scheduling"/>
        </invoke>
      </sequence>
    </flow>
    ...
</process>
```

Listing 7.1: Synchronization concerns using BPEL

```
<process >
   ...
   <flow>
     <sequence>
       ...
       <switch name="requestShipping">
         ...
       </switch>
     </sequence>

     <sequence>
       ...
       <invoke ...>
         <dependency type="BCD" destination="requestShipping"/>
       </invoke>
       ...
     </sequence>

     <sequence>
       ...
       <invoke ...>
         <dependency type="BCD" destination="requestShipping"/>
       </invoke>
     </sequence>
   </flow>
   ...
</process>
```

Listing 7.2: Synchronization concerns using our extension

never be executed at all, and the others could begin in a no determined moment in the future, when their begin dependencies with activities no belonging to the current scope be satisfied. Therefore, if the scope waits for such activities in order to be considered as completed, it would block indefinably the execution of other activities that are waiting for its finalization. We will illustrate all these possible situations with the following examples:

Example 1:
Lets consider a functional replication problem, where more than one service can be invoked in order to supply an equivalent behaviour (from the client point of view). In this example we have defined a *requestShipping* service as our first invocation alternative, but in case of problems, we have the alternative of invoking the *requestShippingAlt* service. We can express this transactional relationship with the ACTA *begin on abort* dependency discussed in chapter 3 and in the previous section.

```
<scope>
  <invoke name="requestShipping" ... >
<supressFailure>true</supressFailure>
<invoke>

  <secondaryActivities>
    <invoke name="requestShippingAlt" ... >
      <dependency type="BAD" destination="requestShipping"/>
    </invoke>
  </secondaryActivities>
</scope>
```

Here, the *requestShipping* service plays the role of primary activity, and *requestShippingAlt* is the only secondary activity declared in the scope.

In this example we will consider that *requestShipping* throws a fault when invoked. Since the activity declares a *supressFailure* element, the fault is not going to be propagated to an outer scope, but the state of this activity is going to be changed to *aborted*. After the scope has finished the execution of its primary activity, it will check the state of its secondary activities before being able of changing its state to *committed*. Since one of its secondary activities: *requestShippingAlt* has already started (because its begin on abort dependency has been satisfied), the scope should wait for the finalization of this activity before completing. If *requestShippingAlt* success, the scope will change its state to *committed*. Conversely, if this activity fails, the scope will change its state to *aborted* and the fault will be thrown to the outer scope.

Example 2:
This example will discuss the same code of the example 1, but now we will assume that

requestShipping is executed successfully and get the state of *committed*. When this happen, the primary activity of the scope has finished, and one of its secondary activities is not able to begin, because its begin on abort dependency has not been satisfied. This is a case of an activity that never will begin. Therefore, waiting for it will cause a block in the execution of all the activities that are waiting for the scope completion.

Example 3:
Finally lets consider a case that involves the transactional concern of activities compensation. The concept of compensation has been already discussed in chapter 3, and the ACTA *compensation* dependency ("CPD") was already discussed in the last section as an example of the extensibility properties of the ACTA framework. Revisiting this dependency, in the following code *requestShipping* is a web service invocation that plays the role of primary activity in a scope, and *cancelShipping* is a secondary activity that has a compensation dependency with the primary activity.

```
<scope>
  <invoke name="requestShipping" ... />

  <secondaryActivities>
    <invoke name="cancelShipping" ... >
      <dependency type="CPD" destination="requestShipping"/>
    </invoke>
  </secondaryActivities>
</scope>
```

Here, *cancelShipping* should be executed only if the primary activity commits and one of the enclosing scopes aborts (as mentioned in the previous section, there is at least one additional enclosing scope: the *process scope*). Therefore, after a successful execution of the primary activity, the scope should be considered completed, since no secondary activities have started at this point, and an hypothetical execution of them could begin far a way in the future (in this particular example, it could be that later the entire BPEL process is forced to abort its execution).

Before finishing this section, we would like to define also a simplified method for expressing secondary activities. The *secondaryActivities* construct could be directly declared inside of BPEL activities, with an equivalent semantic meaning of including an implicit scope surrounding such activity and declaring the same secondary activities. In that way, the functional replication example could be simplified as:

```
<invoke name="requestShipping" ... >
  <supressFailure>true</supressFailure>
```

```
  <secondaryActivities>
    <invoke name="requestShippingAlt" ... >
      <dependency type="BAD" destination="requestShipping"/>
    </invoke>
  </secondaryActivities>
<invoke>
```

and the compensation example as:

```
<invoke name="requestShipping" ... >
  <secondaryActivities>
    <invoke name="cancelShipping" ... >
      <dependency type="CPD" destination="requestShipping"/>
    </invoke>
  </secondaryActivities>
</invoke>
```

7.3 Combining PADUS and our BPEL extension for applying transactional concerns

In the previous sections, we proposed new constructs that simplify the programming and reduce the scattering of code related to transaction management in BPEL. However, one problem remains: the modularization of the tangled and crosscutting code related to this concern. Neither the traditional BPEL model nor our extension have satisfactorily managed this problem (our extension reduces the scattering of synchronization related code, but not the tangling and crosscutting code related in general to transaction management).

This section goes one step further in our research for ways of improvement the management of transactional concerns in services composition languages like BPEL. Now we will employ the concepts of aspect oriented programming applied to web services composition that were presented in chapter 2. In that chapter we discussed PADUS [BVJ+06], an aspect language for BPEL. This language will be our tool for modularizing transactional code expressed with our extension constructs.

We have chosen PADUS, since this language is the best suited tool for expressing in a modular way the constructs presented for our extension (e.g. the *suppressFailure*, *dependency* and *secondaryActivities* elements, for representing the concepts of: suppressing failures, ACTA dependencies and secondary activities, respectively). For justifying this statement, we have to make first an observation about a similar characteristic shared for all these

```
1 ...
2 <in joinpoint="Jp" pointcut="shippingInvocation(Jp, activityName, inputVariable)">
3   <advice name="shippingTransaction(activityName, inputVariable)"/>
4 </in>
5 ...
```

Listing 7.3: Using *in* advices in PADUS

constructs: they all are XML elements that must be declared nested *inside* the activities they affect. This observation is important since in general, most aspect oriented languages propose ways of including code before, after, or instead of a particular programming construct, but not a straightforward and direct way of including code nested inside a particular construct [6]. This is not the case of PADUS, since it gives us a lot of flexibility for defining aspect modules, in particular, the possibility of declaring "*in*" advices in addition to the usual constructs *before*, *after* and *around*, gives a strong advantage to PADUS, given our requirements, over other aspect languages for BPEL like AO4BPEL [CM04], that does not have an equivalent construct for it. As we saw in chapter 2, the *in* advices permit us to include any BPEL elements (and therefore transactional properties if the elements are our constructs) nested inside other BPEL elements.

Listing 7.3 shows an example of a PADUS aspect module where an *in* advice is used. In this case, the advice *shippingTransaction* (line 3) will be applied inside all the activities were the related pointcut matches. In other words, after the application of the aspect, the BPEL code defined in the *shippingTransaction* advice will be nested into the activities indicated by the pointcut *shippingInvocation* (line 2).

Additionally, in listing 7.4, we show the definition of the *shippingTransaction* advice (from line 2 to line 24). Since the elements of the PADUS language have already been discussed in chapter 2, we will not describe again the syntax and semantic of the showed PADUS constructs. Instead, we will recall only that an advice in PADUS is composed by BPEL elements that can be located *before*, *after*, *instead*, or *inside* a particular BPEL activity, and we will center our attention, in the fact that this advice definition leverage only on elements of our extension for expressing transactional concerns: *secondaryActivities* elements are present in lines 4 to 23 and lines 9 to 16 (the second block of *secondaryActivities* is nested inside the first one). *dependency* elements are present in lines 8, 13, 21 and a *suppressFailure* element in line 3. Therefore, since all the extension elements we have proposed have to be nested inside BPEL activities, and PADUS provides a straightforward construct for doing such nesting declaratively, we have found a way for applying transactional concerns, using our extension and aspect oriented programming.

[6] *Nested inside* is referred here in the context of BPEL, a XML based language where nesting means the inclusion of XML elements inside of other XML elements. Therefore, the semantic meaning of this inclusion depends on both the nested and the nesting elements. We are not talking about the possibility of including code inside a procedure, like in a procedural oriented languages context.

```
...
<advice name="shippingTransaction(activityName, inputVariable)">
  <supressFailure>true</supressFailure>
  <secondaryActivities>
    <invoke name="$activityName+Alt" partnerLink="shippingAlt"
      portType="lns:shippingPT" operation="requestShipping"
      inputVariable="$inputVariable">
      <dependency type="BAD" destination="$activityName"/>
      <secondaryActivities>
        <invoke name="cancel+$activityName+Alt" partnerLink="shippingAlt"
          portType="lns:shippingPT" operation="cancelShipping"
          inputVariable="$inputVariable">
          <dependency type="CPD" destination="$activityName+Alt"/>
          <!-- compensation dependency -->
        </invoke>
      </secondaryActivities>
    </invoke>
    <invoke name="cancel+$activityName" partnerLink="shipping"
      portType="lns:shippingPT" operation="cancelShipping"
      inputVariable="$inputVariable">
      <dependency type="CPD" destination="$activityName"/>
    </invoke>
  </secondaryActivities>
</advice>
...
```

Listing 7.4: Declaring transactional properties using PADUS

It could be argued that the transactional concerns of BPEL could be declaratively applied anyway using PADUS and traditional BPEL constructs. However, the same reasons we gave for applying our constructs instead of the BPEL ones inside BPEL process, are valid also in a PADUS aspect definition. For example: as mentioned in the previous section, each activity synchronization concern in BPEL needs of three constructs located in different places of the BPEL process (the definition of the synchronization link, the source link and the target link). Modularizing this idea in an aspect therefore involves three insertions of code in different points of the program. It represents the definition of three advices and three pointcuts for each synchronization concern. Conversely, since our extension only need one construct for express this idea, only one advice and an associated pointcut is needed.

We will present a broader discussion of the examples showed above in chapter 8, where a complete case study, taking as starting point a traditional BPEL process, is developed.

7.4 Conclusions

At the beginning of this chapter, we defined the objectives that our extension should be able to accomplish in order to make a significant improvement to BPEL. We have achieved them since:

- We have presented three basic related constructs (*supressFailure*, *dependency* and *secondaryTransactions*) for implementing the concepts of activities with state, ACTA dependencies between activities, and the definition of activities no pertaining to the business functional concern of the process, but to transaction management. As showed along this chapter, this constructs are complementary, and used together permit the definition of all the transactional concerns currently managed by BPEL. As discussed before, this more simpler set of constructs will ease the maintainability of BPEL programs and facilitate the learning of the language.
- We have incorporated the extensibility of the ACTA dependency model to our constructs, in such a way that new kind of dependencies can always be added to our model, supporting in this way new transactional models and very particular kind of transactional problems.
- The scattering of code related to the synchronization of activities has been eliminated, since the BPEL constructs of link declarations, link sources and link targets that were supposed to be located in different parts of the code for each synchronization concern, were replaced by a simple equivalent ACTA dependency.

In addition to these achievements, we give a further step, proposing a way of modularizing crosscutting and tangled code related to transaction management using aspect oriented

programming and our extension to BPEL. We chose PADUS as our aspect language, since it permits the definition of advices for including code inside BPEL activities (the *in* advices) and this is a fundamental requirement of our approach.

All the constructs introduced for us, follow the same conceptual ideas presented in chapter 5 for expressing transactional issues in services compositional languages like BPEL. A full practice demonstration of all the theoretical discussions accomplished in this chapter will be presented in chapter 8.

Chapter 8

Case study: A purchase order processing system

This chapter presents a service composition problem with the intention of validating the main ideas discussed in this thesis. We first develop a state of the art solution using BPEL and examine some limitations of this proposal. We then talk about the reasons why our model brings an improvement. Consequently, the next step is a presentation of a solution to the same problem using our extension to BPEL.This shows in practice our ideas of how transaction issues should be managed in services composition languages. A further improvement is proposed later. This is the inclusion of an implementation for the same problem that goes one step further, modularizing the transactional concerns that crosscut and are tangled in our example. This improvement uses aspect oriented technologies for web service composition.

Our example has been adapted from the Purchase Order example provided in the BPEL specification. We have however introduced some additional complexity related to transaction management in order to have additional elements for the discussion.

8.1 Description of the example

Figure 8.1 shows the components that are part of our hypothetical application. This chart employs the same informal graphical notation used in the BPEL specification, so that it is easily understood by everyone familiarized with the specification. Single boxes represent activities. Boxes containing other boxes represent structural BPEL activities. Dotted lines represent sequencing and free grouping of sequences represents concurrent sequences. Solid arrows represent synchronization dependencies across concurrent activities.

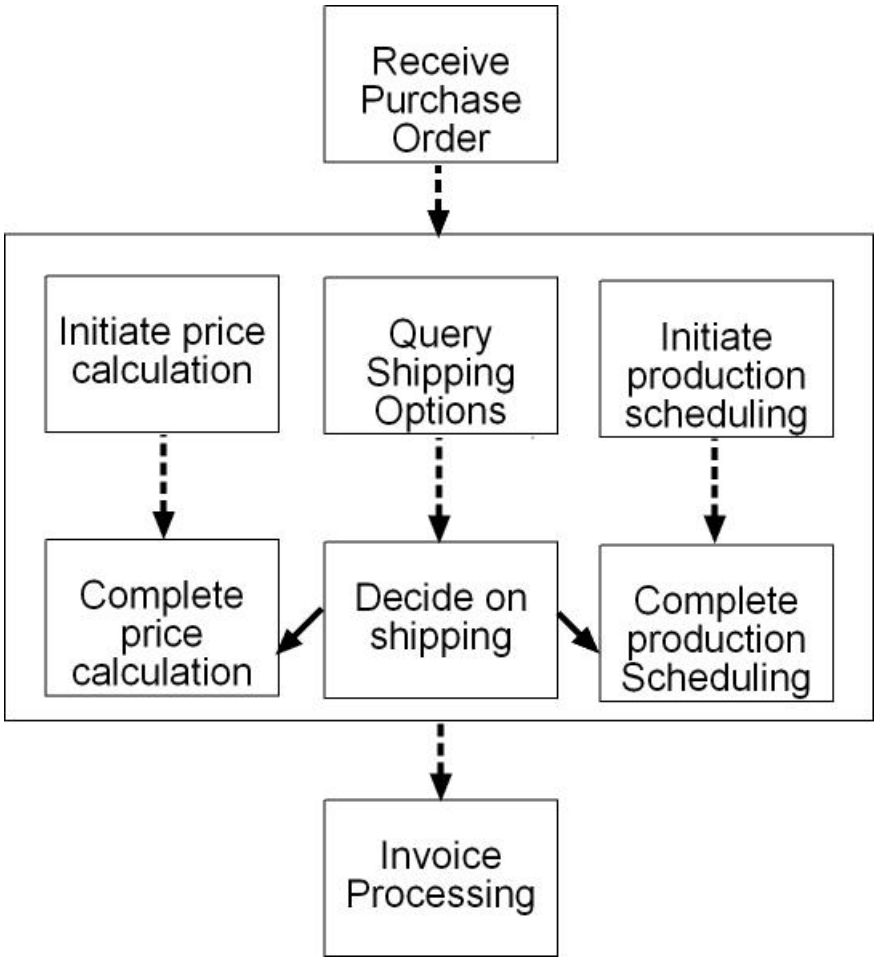

Figure 8.1: A graphical representation of our purchase order processing system

Although the process described in the chart has some changes in comparison to the one provided in the BPEL specification (mainly related to the shipping process), the general overall description of the problem has not changed, therefore we will cite this description from there:

"On receiving the purchase order from a customer, the process initiates three tasks concurrently: calculating the final price for the order, selecting a shipper, and scheduling the production and shipment for the order. While some of the processing can proceed concurrently, there are control and data dependencies between the three tasks. In particular, the shipping price is required to finalize the price calculation, and the shipping date is required for the complete fulfillment schedule. When the three tasks are completed, invoice processing can proceed and the invoice is sent to the customer."[ACD+03]

We will add to this description a more detailed explanation of the shipping process: The purchase order message that arrives first includes two possible delivery addresses: a main address of the client and an alternative one. The deliver can be sent to one of these addresses, basing the decision on which shipping time is shortest. Therefore the shipping process is responsible for querying first the shipping information related to these addresses, making an evaluation of the shipping time and according to that deciding for one of the options. After the decision has been taken, a shipping confirmation should be sent, and the information related to shipping price and schedule is obtained in order to be used for the process of pricing and scheduling.

8.2 An implementation using BPEL

This section presents the implementation of the proposed example using only BPEL. Here we discuss the most relevant fragments of the code, for a complete presentation of all the implementation code please see appendix A. Appendix B also provides a WSDL document for the BPEL example developed here if additional clarity of the message structure and operations used in the services is required. Similarly to the implementation showed here, this WSDL document is also an adaptation of the one presented in the BPEL specification.

The XML code in listing 8.1 represents the general structure of our example. Lines 16 and 17 show the receiving of an initial Web Service request, porting a purchase order as an input message. As we will see soon, the flow activity (line 18 to 24) contains the three concurrent activities that represents the shipping, pricing and scheduling process. Finally lines 25 and 26 show the reply to the client that began the entire process. Control links (lines 19 to 22) are required for establishing synchronization dependencies between activities.

The listing 8.2 shows one of the concurrent process inside the flow activity: the shipping.

```
<process name="purchaseOrderProcess" ...>
  ...
  <variables>
    <variable name="PO" messageType="lns:POMessage"/>
    <variable name="Invoice" messageType="lns:InvMessage"/>
    <variable name="POFault" messageType="lns:orderFaultType"/>
    <variable name="shippingRequestAddress1" messageType="xsd:string"/>
    <variable name="shippingRequestAddress2" messageType="xsd:string"/>
    <variable name="shippingInfoOption1" messageType="lns:shippingInfoMessage"/>
    <variable name="shippingInfoOption2" messageType="lns:shippingInfoMessage"/>
    <variable name="shippingDetails" messageType="lns:shippingDetailsMessage"/>
    <variable name="shippingSchedule" messageType="lns:scheduleMessage"/>
  </variables>
  ...
  <sequence>
    <receive partnerLink="purchasing" portType="lns:purchaseOrderPT"
      operation="sendPurchaseOrder" variable="PO" />
    <flow>
      <links>
        <link name="ship-to-invoice"/>
        <link name="ship-to-scheduling"/>
      </links>
      ...
    </flow>
    <reply partnerLink="purchasing" portType="lns:purchaseOrderPT"
      operation="sendPurchaseOrder" variable="Invoice"/>
  </sequence>
</process>
```

Listing 8.1: General structure of the Purchase Order processing example

This activity first extracts a main address (*shippingRequestAddress1*) and an alternative address (*shippingRequestAddress2*) from the purchase order message sent by the client (lines 5 to 14). Lines 15 to 17 show an invocation activity that queries shipping information using the main address. The obtained message (*shippingInfoOption1*) will provide the partial information related to pricing and scheduling related to this shipping option. The invocation activity showed in lines 18 to 20 will obtain an equivalent shipping information related to the alternative address. The *switch* activity (lines 21 to 47) will select the most convenient shipping option (lines 25 to 26 for the first case and 41 to 42 for the second) according to a particular shipment parameter (in this sample, the number of days for shipping). The shipping information related to pricing (*shippingDetails*) and to scheduling (*shippingSchedule*) are extracted from the message representing the shipping option selected. This data will be used later in the processes of pricing and scheduling respectively.

Finally lines 45 to 46 show source links. They indicate that other activities will have synchronization dependencies with the switch activity. As we will see soon, in this particular case one activity from the pricing process and other from the scheduling process will be unable to initiate until the switch activity has finished (according to the BPEL specification, a switch activity will finish only when the selected activity from the case branches have finished).

Listing 8.3 shows the pricing process. This process is composed of a request for initiating the calculus of the price for the order (lines 5 to 6), a sending of the information related to the price of the shipping (lines 7 to 10), and the receiving of an invoice message representing the total price of the purchase order (lines 11 to 12). The calculus of the price can begin with the information provided by the purchase order message at the initial request received by the process, but in order to continue, it needs also that the selection of a shipping alternative has been done. This is expressed here with a synchronization dependency: the target link showed in line 9, indicates that this activity cannot begin until the activity with the same link as a source have finished (in this case, the switch activity of the shipping process).

Finally, the listing 8.4 shows the scheduling process. This activity is composed of an initial requirement for schedule the production of the order (lines 5 to 6) and an invocation sending the schedule of the shipping (lines 7 to 10). Similarly to the pricing process, this second activity express a synchronization dependency using links (line 9) with the the switch activity of the shipping process.

At this point we have two activities (one at the pricing process and another at the scheduling process) with synchronization dependencies to the same activity (the switch inside the shipping process). As discussed in chapter 7, we do not have in BPEL simple constructs for expressing straightforwardly this idea. Therefore, for each synchronization dependency we need to add to the process a new link declaration (listing 8.1: lines 20 and 21), a new

```
...
<flow>
  ...
  <sequence>
    <assign>
      <copy>
        <from variable="PO" part="address1"/>
        <to variable="shippingRequestAddress1"/>
      </copy>
      <copy>
        <from variable="PO" part="address2"/>
        <to variable="shippingRequestAddress2"/>
      </copy>
    </assign>
    <invoke partnerLink="shipping" portType="lns:shippingPT"
      operation="requestShippingInfo" inputVariable="shippingRequestAddress1"
      outputVariable="shippingInfoOption1"/>
    <invoke partnerLink="shipping" portType="lns:shippingPT"
      operation="requestShippingInfo" inputVariable="shippingRequestAddress2"
      outputVariable="shippingInfoOption2"/>
    <switch>
      <case condition="bpws:getVariableData(
        'shippingInfoOption1', 'numberOfDays', ...)
        <= bpws:getVariableData('shippingInfoOption2', 'numberOfDays', ...)">
        <invoke partnerLink="shipping" portType="lns:shippingPT"
          operation="requestShipping" inputVariable="shippingInfoOption1" />
        <assign>
          <copy>
            <from variable="shippingInfoOption1" part="shippingDetails"/>
            <to variable="shippingDetails"/>
          </copy>
        </assign>
        <assign>
          <copy>
            <from variable="shippingInfoOption1" part="shippingSchedule"/>
            <to variable="shippingSchedule"/>
          </copy>
        </assign>
      </case>
      <otherwise>
        <invoke partnerLink="shipping" portType="lns:shippingPT"
          operation="requestShipping" inputVariable="shippingInfoOption2" />
        ...
      </otherwise>
      <source linkName="ship-to-invoice"/>
      <source linkName="ship-to-scheduling"/>
    </switch>
  </sequence>
  ...
</flow>
...
```

Listing 8.2: The shipping process

```
...
<flow>
  ...
  <sequence>
    <invoke partnerLink="invoicing" portType="lns:computePricePT"
      operation="initiatePriceCalculation" inputVariable="PO"/>
    <invoke partnerLink="invoicing" portType="lns:computePricePT"
      operation="sendShippingPrice" inputVariable="shippingDetails">
      <target linkName="ship-to-invoice"/>
    </invoke>
    <receive partnerLink="invoicing" portType="lns:invoiceCallbackPT"
      operation="sendInvoice" variable="Invoice"/>
  </sequence>
  ...
</flow>
...
```

Listing 8.3: The pricing process

```
...
<flow>
  ...
  <sequence>
    <invoke partnerLink="scheduling" portType="lns:schedulingPT"
      operation="requestProductionScheduling" inputVariable="PO"/>
    <invoke partnerLink="scheduling" portType="lns:schedulingPT"
      operation="sendShippingSchedule" inputVariable="shippingSchedule">
      <target linkName="ship-to-scheduling"/>
    </invoke>
  </sequence>
</flow>
...
```

Listing 8.4: The scheduling process

```
1  <invoke partnerLink="shipping" portType="lns:shippingPT"
2    operation="requestShipping" inputVariable="shippingInfoOption1" >
3    <catchAll>
4      <invoke partnerLink="shippingAlt" portType="lns:shippingPT"
5        operation="requestShipping" inputVariable="shippingInfoOption1" >
6        <compensationHandler>
7          <invoke partnerLink="shippingAlt" portType="lns:shippingPT"
8          operation="cancelShipping" inputVariable="shippingInfoOption1" />
9        </compensationHandler>
10     </invoke>
11   </catchAll>
12   <compensationHandler>
13     <invoke partnerLink="shipping" portType="lns:shippingPT"
14     operation="cancelShipping" inputVariable="shippingInfoOption1" />
15   </compensationHandler>
16 </invoke>
```

Listing 8.5: The shipping request activity with compensation and functional replication

source link construct (listing 8.2: lines 45 and 46) and a new target link construct (listing 8.3: line 9, and listing 8.4: line 9).

Activities synchronization is however only a part of the transactional concerns that can be expressed when composing Web Services. We will add now some complexity to our BPEL implementation including activities for functional replication and compensation. In order to do that, first the shipping request activity presented in the listing 8.2 (lines 25 and 26) will be modified. This time we include an activity associated with a fault handler (listing 8.5: lines 3 to 11), therefore, if the main activity fails, this activity will be automatically invoked. We also add a compensation handler, both for the main activity (lines 12 to 15) and for the activity that replicates the functionality in case that the main invocation fails (lines 6 to 9).

We have to add even more new constructs to our example for managing transactional concerns: the *compensationHandler* and the general fault handler *catchAll*, even when the concerns we are dealing with are very similar to the activities synchronization concerns presented before, in the sense that all of them could be expressed only with transactional dependencies[1] with fewer and more powerful set of constructs.

In addition to the quantity of code we have to write, this model is not extensible. Although it is possible to express certain transactional concerns, other hypothetical transactional relationships between activities (such as the *Begin on Begin Dependency* discussed in chapter 7) are not always possible, and if they are will require a large amount of code.

[1] As discussed in chapter 7: a compensation activity can be expressed with a *compensation dependency* and a functional replication activity can be represented with a *begin on abort dependency.*

In this section we have demonstrated that currently BPEL needs a great quantity of constructs for expressing transactional concerns and that even the expression of simple ideas as synchronization dependencies demand a big amount of coding work. We have also talked about the lack of extensibility of the transactional model used in BPEL.

The following section shows how we can write the same application with the model proposed in our extension for BPEL, presenting a substantial simplification and an improvement of the expressiveness power.

8.3 An implementation using our proposed extension for BPEL

Now we will present for comparison an alternative implementation of the same case study using our proposed BPEL extension. Similarly to the last section, we discuss here only relevant fragments of the sample code, the complete code is showed in appendix C.

Listing 8.6 shows a fragment of the shipping process. We can observe that all references to BPEL control links have been removed (control links declarations, source and target elements). The same has happened with the compensation handling elements and the exception handling related to the implementation of functional replication of activities.

On the other hand, name attributes for identification of some activities have been added (lines 7 and 26), and *suppressFailure* (line 8) and *secondaryActivities* (line 9 to 21) elements have also been added.

As explained in chapter 7, the *suppressFailure* element indicates that in case of failure of a particular activity, a fault should not be thrown, instead the state of the activity will be marked as "aborted". The advantages of using this construct will bring simplification to the code, since the traditional BPEL alternative of adding a surrounding scope with an appropriate fault handler could not be the best solution for all the cases. This is because it always creates an extra burden for the programmer.

The *secondaryActivities* element contains all the activities that are not part of the business case of the application, but of the transactional concerns. We can see here two invocation activities: the *requestShipping1Alt* activity (lines 10 to 17) has a begin on abort dependency (line 11) with the *requestShipping1* activity. It means that this activity should begin only if the *requestShipping1* activity throws a fault. In addition, *requestShipping1Alt* also includes a *secondaryActivities* element (lines 12 to 16), where it is declared an activity with a compensation dependency toward *requestShipping1Alt* (lines 13 to 15). In a similar way, lines 18 to 20 show a compensation activity for *requestShipping1*. The use of a construct for representing general secondary activities is better solution than the use of a specialized one

for englobing compensation activities, since the concept of secondary activities can have more general applications and compensation is a particular case of transaction management that can be expressed with secondary activities and dependencies.

Listings 8.7 and 8.8 show the application of dependencies related to activities synchronization. These dependencies can be expressed in our extension only using our dependency construct, in both listings the code use a begin on commit dependency (8.7: line 8 and 8.8: line 9) with respect to the *requestShipping* activity in order to express a synchronization dependency. This reduces the amount of code to one dependency construct per synchronization concern, and since we have already demonstrated in chapter 7 that the same implementation using BPEL requires three links constructs per synchronization concern, it is a significant simplification improvement.

This section has taken all the transactional concerns present in our case study and has offered an alternative implementation using our extension to BPEL. We have shown that the notion of activities with state, use of ACTA dependencies for establishing transactional dependencies between activities and the concept of secondary activities is a simpler solution than the use of the transactional constructs currently available in BPEL (three different links constructs for activities synchronization, a dedicated construct for defining compensation of activities, and the use of fault handlers for expressing concepts related to functional replication).

We have lowered the number of constructs needed, and we have presented an unified a simpler strategy for dealing with transactional concerns, since all of these concerns can be expressed straightforwardly using the same general constructs. Also, we have brought the extensibility of the ACTA framework to BPEL (where the availability of new kind of dependencies is virtually unbounded), facilitating in this way the inclusion of new dependency models to BPEL.

This conclude the analysis of our constructs, next section will solve one problem still pending, the modularization of crosscutting and tangled code related with transactional management.

8.4 Modularization of crosscutting concerns using PADUS

In this section we go one step further in the improvement of the way transactional issues are managed in BPEL. We previously brought simplification and extensibility to the BPEL proposal, but still one problem is present related to transactional management: The transactional code is completely tangled and crosscuts the business concern of the application.

We had discussed in chapter 2 the necessity of modularizing these crosscutting and tangled concerns and we have given a short introduction to PADUS, an aspect language for BPEL.

```
...
<flow>
  <sequence>
    ...
    <switch name="requestShipping">
      <case condition="...">
        <invoke name="requestShipping1" ...>
          <supressFailure>true</supressFailure>
          <secondaryActivities>
            <invoke name="requestShipping1Alt" ...>
              <dependency type="BAD" destination="requestShipping1"/>
              <secondaryActivities>
                <invoke name="cancelShipping1Alt" ...>
                  <dependency type="CPD" destination="requestShipping1Alt"/>
                </invoke>
              </secondaryActivities>
            </invoke>
            <invoke name="cancelShipping1" ... >
              <dependency type="CPD" destination="requestShipping1"/>
            </invoke>
          </secondaryActivities>
        </invoke>
        ...
      </case>
      <otherwise>
        <invoke name="requestShipping2" ... >
          <supressFailure>true</supressFailure>
          <secondaryActivities>
            <invoke name="requestShipping2Alt" ... >
              <dependency type="BAD" destination="requestShipping2"/>
              <secondaryActivities>
                <invoke name="cancelShipping2Alt" ... >
                  <dependency type="CPD" destination="requestShipping2Alt"/>
                </invoke>
              </secondaryActivities>
            </invoke>
            <invoke name="cancelShipping2" ... >
              <dependency type="CPD" destination="requestShipping2"/>
            </invoke>
          </secondaryActivities>
        </invoke>
        ...
      </otherwise>
    </switch>
  </sequence>
  ...
</flow>
...
```

Listing 8.6: Shipping process implemented using our extension

```
...
<flow>
  <sequence>
    <invoke partnerLink="invoicing" portType="lns:computePricePT"
      operation="initiatePriceCalculation" inputVariable="PO"/>
    <invoke partnerLink="invoicing" portType="lns:computePricePT"
      operation="sendShippingPrice" inputVariable="shippingDetails">
      <dependency type="BCD" destination="requestShipping"/>
    </invoke>
    <receive partnerLink="invoicing" portType="lns:invoiceCallbackPT"
      operation="sendInvoice" variable="Invoice"/>
  </sequence>
  ...
</flow>
...
```

Listing 8.7: Pricing process implemented using our extension

```
...
<flow>
  ...
  <sequence>
    <invoke partnerLink="scheduling" portType="lns:schedulingPT"
      operation="requestProductionScheduling" inputVariable="PO"/>
    <invoke partnerLink="scheduling" portType="lns:schedulingPT"
      operation="sendShippingSchedule" inputVariable="shippingSchedule">
      <dependency type="BCD" destination="requestShipping"/>
    </invoke>
  </sequence>
</flow>
...
```

Listing 8.8: Scheduling process implemented using our extension

```
1  <aspect name="requestShippingTransaction">
2   <using>
3     ...
4     <partnerLink name="shipping" ... />
5     <partnerLink name="shippingAlt" ... />
6     ...
7   </using>
8   <pointcut ... />
9   <advice ... >
10    ...
11  </advice>
12  <in ... >
13    ...
14  </in>
15 </aspect>
```

Listing 8.9: Structure of the PADUS aspect

In this section we will show how these concepts can be applied to our particular example.

If we revisit listing 8.6, we see that the activities representing shipping requests (lines 7 to 22 and 26 to 41) present both code of the business case (the shipping request itself) and code related to transaction issues (the definition of compensation activities and the management of functional replication). In addition, since the shipping confirmation request code is present in more than one single location, a fairly similar transactional code also has to be repeated more than one time, obstructing maintainability and adding complexity to the original problem.

This section will show relevant fragments of an aspect written in PADUS with the intention of modularizing these transactional concerns. For a complete review of the PADUS aspect code please refer to appendix D. First we will show a general structure of the PADUS aspect, and after the main elements that comprise it (pointcut definition, advice and aspect module).

In listing 8.9, lines 2 to 7 shows the BPEL elements (e.g. variables and namespaces) that will be required inside the aspect, line 8 shows the declaration of the pointcut, lines 9 to 11 the declaration of an advice and lines 12 to 14 the aspect module.

Listing 8.10 shows a detail of the pointcut. This pointcut intercepts the joinpoints associated with the invocation of the activity representing the shipping confirmation request. As showed in line 2, it externalizes the values of the activity name and the variable used as an input parameter.

The listing 8.11 shows the advice that "injects" the code related with transaction management inside the shipping request activity. Line 2 shows that this advice is parametrized with the activity name of the shipping confirmation request, and the input variable used

```
1 ...
2 <pointcut name="shippingInvocation(Jp, activityName, inputVariable)"
3   pointcut="invoking(Jp, activityName, 'shipping', 'shippingPT', 'requestShipping',
4   inputVariable, _)"/>
5 ...
```

Listing 8.10: A PADUS pointcut

in the original invocation. The code is basically the same discussed when showed listing 8.6 (lines 8 to 21 and 27 to 40) therefore it will not be discussed intensively here, the main difference with the example without aspects and this one, is that in the first both the names of the activities and the names of the input variables were hardcoded, whereas in this example both the activity name and its input variable name are parametrized. In Particular, names of the secondary activities are functions of the activity name passed as a parameter to the advice (lines 5, 10 and 17). This is necessary since if we used fixed names instead of derived names, we will have name collisions if the advice is executed more than once (as occurs in our example). We have used the plus symbol (+) as an hypothetical concatenation function. In the line 5 for example, the intention is to concatenate the parameter representing the activity name ($activityName) with the string "Alt". This kind of scripting is not possible in PADUS, therefore we have considered this issue as a proposal for future work in PADUS.

Finally listing 8.12 shows the aspect module. This is responsible of realizing the mapping between a specific pointcut (line 2) and an associated advice (line 3). As stated in chapter 2, four possible modifiers can be specified in a PADUS aspect module: *before*, *after*, *around* and *in*. The *in* advice permits a straightforward injection of the code related to transactional concerns, since it will insert related code inside the activity selected for the associated pointcut. In this way, the concerns related with functional replication and activity compensation are fully modularized in this aspect, and the programmer of the business case can use all his efforts in thinking only about business case, instead of also transaction management issues.

If we try to solve the same problem using only BPEL constructs, we had to write and maintain transaction management code tangled with the business problem that the BPEL process is trying to solve. This tangling appears for example, each time it is needed to define a compensation handler in an activity, or to define a fault handler for specifying an alternative service invocation if a particular invocation throws a fault.

We have not chosen however, to modularize all the transactional concerns present in our application, and the reason is because some of them are a primary part of the main concern of the application. In our example, the synchronization dependencies present in our example are a direct consequence of associated data dependencies (one activity will get some data that is needed for other activities). In this case, we believe that this transaction

```
...
<advice name="shippingTransaction(activityName, inputVariable)">
  <supressFailure>true</supressFailure>
  <secondaryActivities>
    <invoke name="$activityName+Alt" partnerLink="shippingAlt"
      portType="lns:shippingPT" operation="requestShipping"
      inputVariable="$inputVariable">
      <dependency type="BAD" destination="$activityName"/>
      <secondaryActivities>
        <invoke name="cancel+$activityName+Alt" partnerLink="shippingAlt"
          portType="lns:shippingPT" operation="cancelShipping"
          inputVariable="$inputVariable">
          <dependency type="CPD" destination="$activityName+Alt"/>
          <!-- compensation dependency -->
        </invoke>
      </secondaryActivities>
    </invoke>
    <invoke name="cancel+$activityName" partnerLink="shipping"
      portType="lns:shippingPT" operation="cancelShipping"
      inputVariable="$inputVariable">
      <dependency type="CPD" destination="$activityName"/>
    </invoke>
  </secondaryActivities>
</advice>
...
```

Listing 8.11: A PADUS advice

```
...
<in joinpoint="Jp" pointcut="shippingInvocation(Jp, activityName, inputVariable)">
  <advice name="shippingTransaction(activityName, inputVariable)"/>
</in>
...
```

Listing 8.12: An aspect module in PADUS

issue should not be taken away of the process definition, since if we do so, our program becomes inconsistent.

This section has presented one topic that had not been yet touched for our extension: The modularization of the transactional concerns that have been of our interest in BPEL. We have showed how PADUS, the aspect language we discussed in chapter 2, can be employed for this goal. The writing of aspects related to transactional issues is also eased for our extension, since it uses less amount of constructs that are also less scattered along the process definition. Finally, we have mentioned the fact that not all the transactional concerns should be aspectized, since some of them are part of the base aspect of the program.

8.5 Conclusions

In this chapter we have demonstrated in practice the inconvenience of working only with the BPEL constructs for expressing transaction management. We have mentioned as problems the abundance of constructs, the lack of relationship between them although they all are defined for managing transactional concerns, and the lack of extensibility of this model. We also reviewed how these problems are solved using our proposed extension, overcoming all the limitations mentioned before.

Finally, we also have mentioned that both the traditional BPEL strategy and our extension shares a problem, and it is that most of the time they produce tangled and crosscutting transaction management code. We have shown an aspect oriented strategy that solves this problem. We have also made the observation that not all the transactional related code should be always modularized in aspects, taking as an example synchronization of activities that are direct consequence of data dependencies.

This chapter finishes the discussion of our work, the next and last chapter will present the overall conclusions of our thesis and will discuss future work.

Chapter 9

Conclusions and Further Research

9.1 Summary of this dissertation

In chapter 2 we presented an introduction to Web Services. We described this technology as the state of the art approach for solving interaction problems between components in loosely-coupled distributed systems. We described how Web Services facilitate the interaction between such components, defining a protocol based in previously well accepted standards in industry, such as XML. We also mentioned that in the Web Services community, there is a lack of common agreements for defining new specifications, and that this is particularly true when talking about advanced transaction management. Afterwards we discussed BPEL, a standard related to Web Services composition that includes transaction management issues, and that has the advantage of being a clear leader amongst other proposals for Web Services composition. Finally, we discussed the use of aspect oriented programming techniques, as an advanced technology for separation of concerns in BPEL programs.

Chapter 3 presented a review of some proposals for advanced transaction management. We discussed the transactional concerns that different proposals have solved, and we reviewed how the solutions were accomplished. The ConTract model showed a strategy for composing atomic transactions into long lived ones, putting a special emphasis in the services that a middleware must provide in order to fulfill high requirements for consistency and fault tolerance. The model proposed by Interbase made an emphasis on three critical concerns in distributed systems and give us important clues about how to solve them. These concerns were: management of functional replication, mixed transactions and timing issues. The ACTA formal model showed us the power of a general mechanism for expressing ATMS, and emphasized the importance of an extensible model as a tool, since possible future problems can depend on new non-predictable business situations. Finally, KALA

proposed a way of implementing the concepts of ACTA in an aspect language, contributing both with the expressiveness of the ACTA model and the advantages of the separation of concerns.

Chapter 4 presented the infrastructural services and the transactional requirements that a framework for transaction management in Web Services composition should fulfill. The main infrastructural services were defined as: management of workflow instances life cycle, management of events, management of correlation issues and management of instances state. The transaction management requirements were classified in the following concerns: synchronization concerns, compensation concerns, functional replication concerns, timing concerns and exception handling concerns. After, we developed a detailed analysis of each transaction management requirement, showing different perspectives for solving them.

In chapter 5 we presented a model for transaction management in Web Services composition that surpasses the one used in BPEL, the state of the art for Web Services composition. We demonstrated the flexibility and extensibility of the ACTA formal model for expressing transactional concerns in BPEL like languages. We noted that additional extensions to the ACTA framework could improve the utility of this model in our domain. Among these extensions, we distinguished the necessity of establishing dependencies implicitly and between more than pairs of activities, since it can bring more abstraction to the programmer, and reduce the probabilities of error when maintaining the application. Also, we have emphasized the necessity of a mechanism for specifying transactions that are not part of the main concern of the application, and we used the concept of secondary transactions, as a strategy for expressing this idea. Afterwards we presented DBCF, a framework for services composition that implements the directions of our model, and we discussed both its infrastructural and transaction management features.

In chapter 6, we have compared extensively DBCF with BPEL, the state of the art in industry for Web Services composition. We have showed first a brief overview of the differences of these two proposals, and after we have presented a detailed comparison of them. Once concluded this work we have presented a summary of the differences, and an analysis section has been developed in order to determine, in the context of transaction management, what are the advantages and disadvantages that these proposals present when compared with each other, and the common disadvantages that they have for dealing with advanced transaction management concerns.

Chapter 7 presents an extension to BPEL that makes use of the ideas developed in chapter 5, with the objective to bring additional simplicity and extensibility to the BPEL language. We presented new constructs for expressing the concepts of activities with state, ACTA dependencies between activities, and the definition of activities no pertaining to the business functional concern of the process, but to transaction management. These constructs are complementary, and used together permit the definition of all the transactional concerns currently managed by BPEL, with the advantages of:

- Easing the maintainability of BPEL programs and facilitating the learning of the language, since the language was simplified.

- The incorporation of the extensibility properties of the the ACTA dependency model to BPEL. Therefore, new kind of transactional dependencies can be always defined, supporting in this way new transactional models.

- Eliminating the scattering of code related to the synchronization of activities, since the BPEL constructs of link declarations, link sources and link targets that were supposed to be located in different parts of the code for each synchronization concern, were replaced by a simple equivalent ACTA dependency.

In addition to these achievements, we give a further step, proposing a way of modularizing crosscutting and tangled code related to transaction management using aspect oriented programming and our extension to BPEL.

Finally, chapter 8 has introduced a case study where is demonstrated in practice, the inconvenience of working only with the BPEL constructs for expressing transaction management concerns. We mentioned as problems the abundance of constructs, the lack of relationship between them although they all are defined for managing transactional concerns, and the lack of extensibility of this model. We also reviewed how these problems are solved using our proposed extension, overcoming all the limitations mentioned before. We also mentioned that both the traditional BPEL strategy and our extension shares a problem, and it is that most of the time they produce tangled and crosscutting transaction management code. As a solution, we showed an aspect oriented strategy that solves this problem. We also made the observation that not all the transactional related code should be always modularized in aspects, taking as an example synchronization of activities that are direct consequence of data dependencies.

9.2 Conclusions

The objectives of our work were to improve the current mechanisms for transaction management in Web Services composition languages, based on relevant elements of advanced transactional models. We have accomplished these objectives since:

- We have demonstrated the flexibility of the ACTA framework, as the base for the development of a model based on transactional dependencies, that can be used for dealing with advanced transaction management concerns in Web Services composition languages.

- We adapted and extended the dependency model of ACTA, in such a way that it can fit more appropriately in our domain. We added the concept of implicit dependencies between activities and we used the notion of secondary transactions presented in [Fab05].

- The correctness of our model has been demonstrated with the implementation of the DBCF framework, a prototype that follows the main directions of the above model.

- Our framework, in addition to proving our concepts and serving as a tool for implementing Web Services composition applications, is well suited for implementing BPEL like composition languages.

- This is possible since DBCF, in addition to implementing the transactional requirements of our work, implements the infrastructural requirements that a service composition engine should provide, such as a Workflow Manager and an Event Manager implementations.

- DBCF provides high level abstractions mechanisms based on Java annotations. When using these annotations for providing semantic information associated with elements of our language, a lot of concerns can be managed transparently to the programmer (like state sharing amongst activities, or the handling of Web Services invocations).

- Additional abstractions mechanisms are provided with aspect oriented programming. In particular, Web Services interface definitions implemented with the JAX-WS specification, will be automatically linked with our workflow engine, since the binding code will be automatically injected in these classes using AOP techniques.

- All our work on DBCF, has been mapped on an extension of BPEL, the current state of the art for Web Services composition. For doing this, this extension surpasses the BPEL mechanisms for managing the transactional concerns of activity synchronization, functional replication and activity compensation.

- Our solution, in addition to be simpler and powerful that the one used in BPEL, is extensible, since it inherits the extension mechanisms of the ACTA formal model. In this way, it can be adapted in the future to deal with new transactional issues.

- Aspect Oriented programming has been employed as a further final step, bringing the advantages of the separation of concerns principle, to our model for advanced transaction management in BPEL. Since in BPEL all the transactional concerns are tangled with other no related concerns, the necessity of an alternative for modularizing such transactional concerns arises. We have chosen PADUS as a tool for accomplishing it, since all the transactional concerns in our model can be described as nested properties of BPEL activities, and PADUS provides the *in* advices, a special kind of advice that permits the declarative nesting of code inside any BPEL

activity matched by a pointcut. Therefore, we can use the *in* advices of PADUS for declaring transactional properties in BPEL.

9.3 Future work

The future work that we will accomplish is:

- Inclusion of temporal constructs in our dependency model based on ACTA. In this way we will add temporal logic to our dependencies. For example, we could define that a transaction can commit only until one hour after another transaction has committed. This could be done adapting the transactional processing monitor used in our work, for accepting such kind of dependency predicates.

- Inclusion of dependencies that involve more than two activities, relating them with boolean logic predicates. In this way, we could specify predicates such as: a transaction A can begin only if transaction B OR C have committed. This goal also implies the modification of the transactional processing monitor used in our work.

- Definition of dependencies between activities executed in different hosts, in this way remote Web Services composition systems could establish transactional constraints between them. For doing this, a central transactional monitor could be developed. Different workflow instances should register themselves and their transactions in this central monitor, and they would be notified when new dependencies were added at runtime.

- The development of a transformation engine that can transform BPEL processes written in XML, to modules written in DBCF, in order that such processes can be executed with the services of our framework. For doing this, the XML document that defines a BPEL process should be parsed, and each BPEL activity should be mapped to a corresponding DBCF activity and a composition script should be generated.

- The implementation of a BPEL engine with support for our proposed BPEL extension. This engine could be implemented as an extension of the transformation engine described above, in order that the engine recognize also the elements of our BPEL extension.

- Although PADUS itself is not part of our work, the additional scripting capabilities mentioned in chapter 7, could be an interesting future work for the PADUS development team.

Appendix A. An implementation of the Purchase Order example using BPEL

```
<process name="purchaseOrderProcess"
    targetNamespace="http://acme.com/ws-bp/purchase"
    xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    xmlns:lns="http://manufacturing.org/wsdl/purchase">
  <partnerLinks>
    <partnerLink name="purchasing" partnerLinkType="lns:purchasingLT"
        myRole="purchaseService"/>
    <partnerLink name="shipping" partnerLinkType="lns:shippingLT"
        myRole="shippingRequester" partnerRole="shippingService"/>
    <partnerLink name="invoicing" partnerLinkType="lns:invoicingLT"
        myRole="invoiceRequester" partnerRole="invoiceService"/>
    <partnerLink name="scheduling" partnerLinkType="lns:schedulingLT"
        partnerRole="schedulingService"/>
  </partnerLinks>
  <variables>
    <variable name="PO" messageType="lns:POMessage"/>
    <variable name="Invoice" messageType="lns:InvMessage"/>
    <variable name="POFault" messageType="lns:orderFaultType"/>
    <variable name="shippingRequestAddress1" messageType="xsd:string"/>
    <variable name="shippingRequestAddress2" messageType="xsd:string"/>
    <variable name="shippingInfoOption1" messageType="lns:shippingInfoMessage"/>
    <variable name="shippingInfoOption2" messageType="lns:shippingInfoMessage"/>
    <variable name="shippingDetails" messageType="lns:shippingDetailsMessage"/>
    <variable name="shippingSchedule" messageType="lns:scheduleMessage"/>
  </variables>
  <faultHandlers>
```

```
  <catch faultName="lns:cannotCompleteOrder" faultVariable="POFault">
    <reply partnerLink="purchasing" portType="lns:purchaseOrderPT"
        operation="sendPurchaseOrder" variable="POFault"
        faultName="cannotCompleteOrder"/>
  </catch>
</faultHandlers>
<sequence>
  <receive partnerLink="purchasing" portType="lns:purchaseOrderPT"
      operation="sendPurchaseOrder" variable="PO" />
  <flow>
    <links>
      <link name="ship-to-invoice"/>
      <link name="ship-to-scheduling"/>
    </links>
    <sequence>
      <assign>
        <copy>
          <from variable="PO" part="address1"/>
          <to variable="shippingRequestAddress1"/>
        </copy>
        <copy>
          <from variable="PO" part="address2"/>
          <to variable="shippingRequestAddress2"/>
        </copy>
      </assign>
      <invoke partnerLink="shipping" portType="lns:shippingPT"
          operation="requestShippingInfo"
          inputVariable="shippingRequestAddress1"
          outputVariable="shippingInfoOption1"/>
      <invoke partnerLink="shipping" portType="lns:shippingPT"
          operation="requestShippingInfo"
          inputVariable="shippingRequestAddress2"
          outputVariable="shippingInfoOption2"/>
      <switch>
        <case condition="bpws:getVariableData('shippingInfoOption1',
            'numberOfDays', 'shippingSchedule/numberOfDays') &lt;=
            bpws:getVariableData('shippingInfoOption2', 'numberOfDays',
            'shippingSchedule/numberOfDays')">
          <invoke partnerLink="shipping" portType="lns:shippingPT"
              operation="requestShipping"
              inputVariable="shippingInfoOption1" />
          <assign>
```

```
        <copy>
          <from variable="shippingInfoOption1" part="shippingDetails"/>
          <to variable="shippingDetails"/>
        </copy>
      </assign>
      <assign>
        <copy>
          <from variable="shippingInfoOption1" part="shippingSchedule"/>
          <to variable="shippingSchedule"/>
        </copy>
      </assign>
    </case>
    <otherwise>
      <invoke partnerLink="shipping" portType="lns:shippingPT"
          operation="requestShipping"
          inputVariable="shippingInfoOption2" />
      <assign>
        <copy>
          <from variable="shippingInfoOption2" part="shippingDetails"/>
          <to variable="shippingDetails"/>
        </copy>
      </assign>
      <assign>
        <copy>
          <from variable="shippingInfoOption2" part="shippingSchedule"/>
          <to variable="shippingSchedule"/>
        </copy>
      </assign>
    </otherwise>
    <source linkName="ship-to-invoice"/>
    <source linkName="ship-to-scheduling"/>
  </switch>
</sequence>
<sequence>
  <invoke partnerLink="invoicing" portType="lns:computePricePT"
      operation="initiatePriceCalculation" inputVariable="PO"/>
  <invoke partnerLink="invoicing" portType="lns:computePricePT"
      operation="sendShippingPrice" inputVariable="shippingDetails">
    <target linkName="ship-to-invoice"/>
  </invoke>
  <receive partnerLink="invoicing" portType="lns:invoiceCallbackPT"
      operation="sendInvoice" variable="Invoice"/>
```

```
      </sequence>
      <sequence>
        <invoke partnerLink="scheduling" portType="lns:schedulingPT"
            operation="requestProductionScheduling" inputVariable="PO"/>
        <invoke partnerLink="scheduling" portType="lns:schedulingPT"
            operation="sendShippingSchedule" inputVariable="shippingSchedule">
          <target linkName="ship-to-scheduling"/>
        </invoke>
      </sequence>
    </flow>
    <reply partnerLink="purchasing" portType="lns:purchaseOrderPT"
        operation="sendPurchaseOrder" variable="Invoice"/>
  </sequence>
</process>
```

Appendix B. A WSDL document for the Purchase Order example

```
<definitions targetNamespace="http://manufacturing.org/wsdl/purchase"
    xmlns:sns="http://manufacturing.org/xsd/purchase"
    xmlns:pos="http://manufacturing.org/wsdl/purchase"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
  <import namespace="http://manufacturing.org/xsd/purchase"
      location="http://manufacturing.org/xsd/purchase.xsd"/>
  <message name="POMessage">
    <part name="customerInfo" type="sns:customerInfo"/>
    <part name="purchaseOrder" type="sns:purchaseOrder"/>
    <part name="address1" type="xsd:string"/>
    <part name="address2" type="xsd:string"/>
  </message>
  <message name="InvMessage">
    <part name="IVC" type="sns:Invoice"/>
  </message>
  <message name="orderFaultType">
    <part name="problemInfo" type="xsd:string"/>
  </message>
  <message name="shippingInfoMessage">
    <part name="shippingDetails" type="sns:shippingDetailsInfo"/>
    <part name="shippingSchedule" type="sns:scheduleInfo"/>
  </message>
  <message name="shippingDetailsMessage">
    <part name="shippingDetails" type="sns:shippingDetailsInfo"/>
  </message>
  <message name="scheduleMessage">
```

```
  <part name="schedule" type="sns:scheduleInfo"/>
</message>
<!-- portTypes supported by the purchase order process -->
<portType name="purchaseOrderPT">
  <operation name="sendPurchaseOrder">
    <input message="pos:POMessage"/>
    <output message="pos:InvMessage"/>
    <fault name="cannotCompleteOrder" message="pos:orderFaultType"/>
  </operation>
</portType>
<portType name="invoiceCallbackPT">
  <operation name="sendInvoice">
    <input message="pos:InvMessage"/>
  </operation>
</portType>
<!-- portType supported by the shipping service -->
<portType name="shippingPT">
  <operation name="requestShippingInfo">
    <input message="xsd:string"/>
    <output message="pos:shippingInfoMessage"/>
  </operation>
  <operation name="requestShipping">
    <input message="pos:shippingInfoMessage"/>
    <fault name="cannotCompleteOrder" message="pos:orderFaultType"/>
  </operation>
  <operation name="cancelShipping">
    <input message="pos:shippingInfoMessage"/>
  </operation>
</portType>
<!-- portType supported by the invoice services -->
<portType name="computePricePT">
  <operation name="initiatePriceCalculation">
    <input message="pos:POMessage"/>
  </operation>
  <operation name="sendShippingPrice">
    <input message="pos:shippingDetailsMessage"/>
  </operation>
</portType>
<!-- portType supported by the production scheduling process -->
<portType name="schedulingPT">
  <operation name="requestProductionScheduling">
    <input message="pos:POMessage"/>
```

```
    </operation>
    <operation name="sendShipingSchedule">
      <input message="pos:scheduleMessage"/>
    </operation>
  </portType>
  <plnk:partnerLinkType name="purchasingLT">
    <plnk:role name="purchaseService">
      <plnk:portType name="pos:purchaseOrderPT"/>
    </plnk:role>
  </plnk:partnerLinkType>
  <plnk:partnerLinkType name="invoicingLT">
    <plnk:role name="invoiceService">
      <plnk:portType name="pos:computePricePT"/>
    </plnk:role>
    <plnk:role name="invoiceRequester">
      <plnk:portType name="pos:invoiceCallbackPT"/>
    </plnk:role>
  </plnk:partnerLinkType>
  <plnk:partnerLinkType name="shippingLT">
    <plnk:role name="shippingService">
      <plnk:portType name="pos:shippingPT"/>
    </plnk:role>
  </plnk:partnerLinkType>
  <plnk:partnerLinkType name="schedulingLT">
    <plnk:role name="schedulingService">
      <plnk:portType name="pos:schedulingPT"/>
    </plnk:role>
  </plnk:partnerLinkType>
</definitions>
```

Appendix C. An implementation of the Purchase Order example using the propossed extension for BPEL

```
<process name="purchaseOrderProcess"
    targetNamespace="http://acme.com/ws-bp/purchase"
    xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    xmlns:lns="http://manufacturing.org/wsdl/purchase">
  <partnerLinks>
    <partnerLink name="purchasing" partnerLinkType="lns:purchasingLT"
        myRole="purchaseService"/>
    <partnerLink name="shipping" partnerLinkType="lns:shippingLT"
        myRole="shippingRequester" partnerRole="shippingService"/>
    <partnerLink name="shippingAlt" partnerLinkType="lns:shippingLT"
        myRole="shippingRequester" partnerRole="shippingService"/>
    <partnerLink name="invoicing" partnerLinkType="lns:invoicingLT"
        myRole="invoiceRequester" partnerRole="invoiceService"/>
    <partnerLink name="scheduling" partnerLinkType="lns:schedulingLT"
        partnerRole="schedulingService"/>
  </partnerLinks>
  <variables>
    <variable name="PO" messageType="lns:POMessage"/>
    <variable name="Invoice" messageType="lns:InvMessage"/>
    <variable name="POFault" messageType="lns:orderFaultType"/>
    <variable name="shippingRequestAddress1" messageType="xsd:string"/>
    <variable name="shippingRequestAddress2" messageType="xsd:string"/>
    <variable name="shippingInfoOption1"
        messageType="lns:shippingInfoMessage"/>
    <variable name="shippingInfoOption2"
        messageType="lns:shippingInfoMessage"/>
```

```
    <variable name="shippingDetails"
        messageType="lns:shippingDetailsMessage"/>
    <variable name="shippingSchedule" messageType="lns:scheduleMessage"/>
  </variables>
  <faultHandlers>
    <catch faultName="lns:cannotCompleteOrder" faultVariable="POFault">
      <reply partnerLink="purchasing" portType="lns:purchaseOrderPT"
          operation="sendPurchaseOrder" variable="POFault"
          faultName="cannotCompleteOrder"/>
    </catch>
  </faultHandlers>
  <sequence>
    <receive partnerLink="purchasing" portType="lns:purchaseOrderPT"
        operation="sendPurchaseOrder" variable="PO">
</receive>
    <flow>
      <sequence>
        <assign>
          <copy>
            <from variable="PO" part="address1"/>
            <to variable="shippingRequestAddress1"/>
          </copy>
          <copy>
            <from variable="PO" part="address2"/>
            <to variable="shippingRequestAddress2"/>
          </copy>
        </assign>
        <invoke partnerLink="shipping" portType="lns:shippingPT"
            operation="requestShippingInfo"
            inputVariable="shippingRequestAddress1"
            outputVariable="shippingInfoOption1"/>
        <invoke partnerLink="shipping" portType="lns:shippingPT"
            operation="requestShippingInfo"
            inputVariable="shippingRequestAddress2"
            outputVariable="shippingInfoOption2"/>
        <switch name="requestShipping">
          <case condition="bpws:getVariableData('shippingInfoOption1',
              'numberOfDays', 'shippingSchedule/numberOfDays') &lt;=
              bpws:getVariableData('shippingInfoOption2', 'numberOfDays',
              'shippingSchedule/numberOfDays')">
            <invoke name="requestShipping1" partnerLink="shipping"
                portType="lns:shippingPT" operation="requestShipping"
```

```
      inputVariable="shippingInfoOption1">
    <supressFailure>true</supressFailure>
    <secondaryActivities>
      <invoke name="requestShipping1Alt" partnerLink="shippingAlt"
          portType="lns:shippingPT" operation="requestShipping"
          inputVariable="shippingInfoOption1">
        <dependency type="BAD" destination="requestShipping1"/>
        <secondaryActivities>
          <invoke name="cancelShipping1Alt"
              partnerLink="shippingAlt" portType="lns:shippingPT"
              operation="cancelShipping"
              inputVariable="shippingInfoOption1">
            <dependency type="CPD"
                destination="requestShipping1Alt"/>
            <!-- compensation dependency -->
          </invoke>
        </secondaryActivities>
      </invoke>
      <invoke name="cancelShipping1" partnerLink="shipping"
          portType="lns:shippingPT" operation="cancelShipping"
          inputVariable="shippingInfoOption1">
        <dependency type="CPD" destination="requestShipping1"/>
        <!-- compensation dependency -->
      </invoke>
    </secondaryActivities>
  </invoke>
  <assign>
    <copy>
      <from variable="shippingInfoOption1" part="shippingDetails"/>
      <to variable="shippingDetails"/>
    </copy>
  </assign>
  <assign>
    <copy>
      <from variable="shippingInfoOption1" part="shippingSchedule"/>
      <to variable="shippingSchedule"/>
    </copy>
  </assign>
</case>
<otherwise>
  <invoke name="requestShipping2" partnerLink="shipping"
      portType="lns:shippingPT" operation="requestShipping"
```

```
          inputVariable="shippingInfoOption2">
        <supressFailure>true</supressFailure>
        <secondaryActivities>
          <invoke name="requestShipping2Alt" partnerLink="shippingAlt"
              portType="lns:shippingPT" operation="requestShipping"
              inputVariable="shippingInfoOption2">
            <dependency type="BAD" destination="requestShipping2"/>
            <secondaryActivities>
              <invoke name="cancelShipping2Alt" partnerLink="shippingAlt"
                  portType="lns:shippingPT" operation="cancelShipping"
                  inputVariable="shippingInfoOption2">
                <dependency type="CPD" destination="requestShipping2Alt"/>
                <!-- compensation dependency -->
              </invoke>
            </secondaryActivities>
          </invoke>
          <invoke name="cancelShipping2" partnerLink="shipping"
              portType="lns:shippingPT" operation="cancelShipping"
              inputVariable="shippingInfoOption2">
            <dependency type="CPD" destination="requestShipping2"/>
            <!-- compensation dependency -->
          </invoke>
        </secondaryActivities>
      </invoke>
      <assign>
        <copy>
          <from variable="shippingInfoOption2" part="shippingDetails"/>
          <to variable="shippingDetails"/>
        </copy>
      </assign>
      <assign>
        <copy>
          <from variable="shippingInfoOption2" part="shippingSchedule"/>
          <to variable="shippingSchedule"/>
        </copy>
      </assign>
    </otherwise>
  </switch>
</sequence>
<sequence>
  <invoke partnerLink="invoicing" portType="lns:computePricePT"
      operation="initiatePriceCalculation" inputVariable="PO"/>
```

```
        <invoke partnerLink="invoicing" portType="lns:computePricePT"
            operation="sendShippingPrice" inputVariable="shippingDetails">
          <dependency type="BCD" destination="requestShipping"/>
        </invoke>
        <receive partnerLink="invoicing" portType="lns:invoiceCallbackPT"
            operation="sendInvoice" variable="Invoice"/>
      </sequence>
      <sequence>
        <invoke partnerLink="scheduling" portType="lns:schedulingPT"
            operation="requestProductionScheduling" inputVariable="PO"/>
        <invoke partnerLink="scheduling" portType="lns:schedulingPT"
            operation="sendShippingSchedule" inputVariable="shippingSchedule">
          <dependency type="BCD" destination="requestShipping"/>
        </invoke>
      </sequence>
    </flow>
    <reply partnerLink="purchasing" portType="lns:purchaseOrderPT"
        operation="sendPurchaseOrder" variable="Invoice"/>
  </sequence>
</process>
```

Appendix D. Applying PADUS to our example

```
<aspect name="requestShippingTransaction">
  <using>
    <namespace name="xmlns:lns" uri="http://manufacturing.org/wsdl/purchase"/>
    <partnerLinks>
      <partnerLink name="shipping" partnerLinkType="lns:shippingLT"
          myRole="shippingRequester" partnerRole="shippingService"/>
      <partnerLink name="shipping2" partnerLinkType="lns:shippingLT"
          myRole="shippingRequester" partnerRole="shippingService"/>
    </partnerLinks>
  </using>
  <pointcut name="shippingInvocation(Jp, activityName, inputVariable)"
      pointcut="invoking(Jp, activityName, 'shipping', 'shippingPT',
      'requestShipping', inputVariable, _)"/>
  <advice name="shippingTransaction(activityName, inputVariable)">
    <supressFailure>true</supressFailure>
    <secondaryActivities>
      <invoke name="$activityName+Alt" partnerLink="shipping2"
          portType="lns:shippingPT" operation="requestShipping"
          inputVariable="$inputVariable">
        <dependency type="BAD" destination="$activityName"/>
        <secondaryActivities>
          <invoke name="cancel+$activityName+Alt" partnerLink="shipping2"
              portType="lns:shippingPT" operation="cancelShipping"
              inputVariable="$inputVariable">
            <dependency type="CPD" destination="$activityName+Alt"/>
            <!-- compensation dependency -->
          </invoke>
        </secondaryActivities>
      </invoke>
```

```
      <invoke name="cancel+$activityName" partnerLink="shipping"
          portType="lns:shippingPT" operation="cancelShipping"
          inputVariable="$inputVariable">
        <dependency type="CPD" destination="\$activityName"/>
        <!-- compensation dependency -->
      </invoke>
    </secondaryActivities>
  </advice>
  <in joinpoint="Jp" pointcut="shippingInvocation(Jp, activityName,
      inputVariable)">
    <advice name="shippingTransaction(activityName, inputVariable)"/>
  </in>
</aspect>
```

Bibliography

[ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, and Johannes Klein. Business process execution language for web services (version 1.1). May 2003.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. Web services. concepts, architectures and applications. Springer, 2004.

[Ark02] A. Arkin. Business process modeling language- bpml 1.0. In *Technical Report. BPMI Consortium*, June 2002.

[Asp] The aspectj project. http://www.eclipse.org/aspectj/.

[BBC+06a] S. Bajaj, D. Box, D. Chappell, F. Curbera, G. Daniels, P. Hallam, and M. Hondo. Web services policy attachment (version 1.2). March 2006.

[BBC+06b] S. Bajaj, D. Box, D. Chappell, F. Curbera, G. Daniels, P. Hallam, and M. Hondo. Web services policy framework (version 1.2). March 2006.

[BCC+04] D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, M. Hadley, and C. Kaler. Web services addressing. August 2004.

[BCE+06] J. Ball, D. Carson, I. Evans, K. Haase, and E. Jendrock. The java ee tutorial. February 2006.

[BCH+03a] D. Bunting, M. Chapman, O. Hurley, M. Little, J. Mischkinsky, E. Newcomer, J. Webber, and K. Swenson. Web services context service specification(version 1.0). July 2003.

[BCH+03b] D. Bunting, M. Chapman, O. Hurley, M. Little, J. Mischkinsky, E. Newcomer, J. Webber, and K. Swenson. Web services coordination framework specification (version 1.0). July 2003.

[BCH+03c] D. Bunting, M. Chapman, O. Hurley, M. Little, J. Mischkinsky, E. Newcomer, J. Webber, and K. Swenson. Web services transaction management specification (version 1.0). July 2003.

[BPM] Business process modeling language. http://www.ebpml.org/bpml.htm.

[BTP] Business transaction protocol. http://www.oasis-open.org/committees/tchome.php?wgabbrev=business-transaction.

[BVJ+06] Mathieu Braem, Kris Verlaenen, Niels Joncheere, Wim Vanderperren, Ragnhild Van Der Straeten, Eddy Truyen, Wouter Joosen, and Viviane Jonckers. Isolating process-level concerns using padus. In *Accepted for the fourth international conference on Business Process Management*, September 2006.

[CCF+05a] Luis Cabrera, George Copeland, Max Feingold, Robert Freund, and Tom Freund. The ws-atomictransaction specification. 2005.

[CCF+05b] Luis Cabrera, George Copeland, Max Feingold, Robert Freund, and Tom Freund. The ws-businessactivity specification. 2005.

[CCF+05c] Luis Cabrera, George Copeland, Max Feingold, Robert Freund, and Tom Freund. The ws-coordination specification. 2005.

[cef] United nations centre for trade facilitation and electronic business. http://www.unece.org/cefact/.

[CM04] Anis Charfi and Mira Mezini. Aspect-oriented web service composition with ao4bpel. In *Proc. of the European Conference on Web Services ECOWS 2004*, September 2004.

[CR91] Panos K. Chrysanthis and Krithi Ramamritham. A formalism for extended transactional models. In *Proceedings of the 17th International Conference on Very Large Data Bases*, 1991.

[CR92] Panos K. Chrysanthis and Krithi Ramamritham. Acta: The saga continues. In *Database Transaction Models for Advanced Applications. Editeed by Ahmed K. Elmargarmid, Morgan Kaufmann Publishers*, 1992.

[DB05] Rémi Douence and Didier Le Botlan. Towards a taxonomy of aop semantics. In *AOSD-Europe-INRIA-1*, July 2005.

[ebX] ebxml specification. http://www.ebxml.org/specs/index.htmtechnicalspecifications.

[ELLR90] A. Elmagarmid, Y. Leu, W. Litwin, and Marek Rusinkiewicz. A multidatabase transaction model for interbase. In *Sixteenth international conference on Very large databases, pages 507518, San Francisco, CA, USA*, 1990.

[Fab05] Johan Fabry. Modularizing advanced transaction management. In *PhD thesis, Vrije Universiteit Brussel*, 2005.

[Fis02] M. Fisher. Introduction to web services. In *The Java Web Services tutorial*, AUGUST 2002.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *ACM Conference on Management of Data, pages 249-259*, May 1987.

[GR93] J. Gray and A. Reuter. Transaction processing, concepts and techniques. Morgan Kaufmann, 1993.

[Gra81] J. Gray. The transactions concepts: Virtues and limitations. In *International Conference on Very Large Databases, pages 144-154*, 1981.

[HL95] Walter L. Hursh and Cristina Videira Lopes. Separation of concerns. In *Technical report, College of Computer Science, Northeastern University*, 1995.

[HTT] Http - hypertext transfer protocol. http://www.w3.org/Protocols/.

[JAX] The jax-ws project. https://jax-ws.dev.java.net/.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 1997.

[LER89] Y. Leu, A. Elmagarmid, and M. Rusinkiewicz. An extended transaction model for multidatabase systems. In *Technical Report, Department of Computer Science, Purdue University*, 1989.

[Ley01] F. Leymann. Web services flow language (version 1.0). May 2001.

[MA01] D. Menasce and V. Almeida. Capacity planning for web services. In *The Java Web Services tutorial*. Prentice Hall, 2001.

[oas] Organization for the advancement of structured information standards. http://www.oasis-open.org/home/index.php.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. In *Communications of the ACM*, 1972.

[Sch] Xml scheme. http://www.w3.org/XML/Schema.

[SOA] Xml protocol working group. http://www.w3.org/2000/xp/Group/.

[Sun] The java language. http://java.sun.com/.

[Tom] Apache tomcat. http://tomcat.apache.org/.

[UDD] Uddi.org. http://www.uddi.org/.

[w3c] World wide web consortium. http://www.w3.org/.

[wmc] The workflow management coalition. http://www.wfmc.org/.

[WR92] Helmut Wachter and Andreas Reuter. The contract model. In *Advanced Transaction Models and Architectures*, 1992.

[ws002] Web services architecture requirements, OCTOBER 2002.

[WSD] Web services description language (wsdl) version 2.0 part 1: Core language. http://www.w3.org/TR/wsdl20/.

[WWC92] G. Wiederhold, P. Wegner, and S. Ceri. Towards megaprogramming: A paradigm for component-based programming. In *Communications of the ACM*, 1992.

[XML] Extensible markup language (xml). http://www.w3.org/XML/.

[XPA] Xml path language (xpath). http://www.w3.org/TR/xpath.