Transactional BPEL Processes with AO4BPEL Aspects

Anis Charfi
SAP Research CEC Darmstadt
Darmstadt, Germany

Benjamin Schmeling
UBL Informationssysteme
Neu-Isenburg, Germany

Mira Mezini
Software Technology Group
TU Darmstadt, Germany

Abstract

Recently, OASIS approved two standards respectively for Web Service composition and for Web Service transactions. Nevertheless, it is still unclear how WS-BPEL and the WS-TX family of specifications interoperate, i.e., how to use atomic transactions and business activities in the context of BPEL processes. In this paper, we present several transactional requirements in BPEL processes and argue that BPEL's compensation mechanism provides only limited support for a few of these requirements, e.g., it cannot cope with atomic transactions with the ACID properties. To support transactional BPEL processes, we use the AO4BPEL process container framework. In this framework, the transaction requirements of the process activities are specified declaratively in a deployment descriptor and an aspect-based container is generated automatically to integrate the process execution with the transaction middleware, which is provided as a transaction Web Service based on Apache Kandula.

1 Introduction

The current Web Service stack [24] addresses several advanced issues such as composition, security, reliable messaging, transactions, etc. In this paper, we focus on two of these issues namely composition and transactions.

Recently, OASIS approved WS-BPEL 2.0 [2] as a standard for Web Service composition. In this language, a composite Web Service is implemented by means of a workflow process that consists of activities such as the *messaging* activities *invoke* and *reply*, which are used for interacting with the other Web Services and the *structured* activities *sequence* and *scope*, which act as containers for their nested activities.

BPEL provides some support for transactions through its compensation mechanism, which allows undoing the effects of completed activities. However, the compensation mechanism does not support many important transactional requirements. For instance, it does not support distributed transactions where the process and the partner Web Services have to agree jointly on the outcome of the transaction (as in the 2-Phase-Commit protocol for example). That is, there is no support for external coordination [23] in BPEL. Moreover, the compensation mechanism does not support strict atomic transactions with the traditional ACID properties.

On the other hand, OASIS has also recently approved a Web Services transaction standard that consists of three specifications: WS-Coordination [11], which provides a generic framework for coordinating distributed activities and two other specifications that are built on top of it: WS-AtomicTransaction (WS-AT for short) [20] supports short-lived transactions with the traditional ACID properties and WS-BusinessActivity (WS-BA for short) [21] supports long-running compensation-based transactions.

In this paper, we discuss some transactional requirements in BPEL, which are necessary in many business scenarios. To address these requirements and enable transactional BPEL-based production workflows [19], we integrate BPEL with WS-AT and WS-BA by means of AO4BPEL aspects. More precisely, our approach is based on the AO4BPEL process container framework [7], which uses a declarative deployment descriptor for specifying the non-functional requirements and an aspect-based process container for integrating the process execution with middleware Web Services that provide the capabilities of WS-* specifications. We have already used that framework to support security [5] and reliable messaging [8] in BPEL processes. In this paper, we use it to integrate the process execution by the BPEL engine with a transaction Web Service that is based on Apache Kandula [1], an open source implementation of WS-AT and WS-BA.

The remainder of this paper is organized as follows. Section 2 gives a short overview of the composition and transaction specifications. Section 3 discusses some transactional requirements in BPEL. Section 4 presents the concepts of our approach and Section 5 explains how it is implemented using AO4BPEL aspects and the transaction Web Service. Section 6 reports on related work and Section 7 concludes the paper.

2 Background

This section introduces the WS-* specifications for transactions and composition that are used in this work.

2.1 Web Service Coordination

WS-Coordination [11] provides a generic framework for coordinating distributed activities. WS-Coordination is extensible, i.e., new coordination protocols (e.g., for short-running atomic transactions) can be plugged into it. This specification defines three services: an *activation service*, a *registration service*, and a *coordination service*.

The *activation service* is used to create an activity. When it receives an activity creation request, it creates a *coordination context* that contains a context identifier, a coordination protocol type, and the address of a *registration service*. The *coordination context* can be transported with the application messages to the participants, which will consequently register for a certain coordination protocol (e.g., for the completion protocol of WS-AT) using the *registration service*. The *registration service* responds to the registration request with the address of the *coordination service*, which is responsible for running one or more protocols (referred to as *coordination type*) between the registered parties. WS-AtomicTransaction and WS-BusinessActivity are two coordination types that leverage WS-Coordination.

2.2 Atomic transactions

The atomic transaction coordination type comprises three protocols for supporting atomic distributed transactions with the traditional ACID properties: a *completion* protocol to initiate the commitment of a transaction and two *2-Phase-Commit* protocols (*volatile* and *durable*) to decide whether the transaction should be committed or aborted.

The initiator of the *completion protocol* gets informed about the result of the commitment and can decide whether the outcome of the transaction should be made durable (committed) or must be rolled back. During the commitment the coordinator controls two possible 2PC (Two-Phase Commit) phases. The first phase *Volatile 2PC* is for managing volatile resources like a cache and the second phase *Durable 2PC* is for managing durable resources like a database. The participants of a transaction may choose to register for one of those protocols.

The 2PC protocol is blocking, i.e., after the first phase of the protocol, all participant resources remain blocked until the commit message is generated in the second phase. This is necessary to preserve isolation, i.e., resources are locked to disallow other concurrent transactions from accessing them before the locking transaction completes. Unless strict isolation is required, such an approach is unacceptable in the distributed and heterogeneous Web Service context because it reduces concurrency. Moreover, in cross-organizational distributed contexts, locking resources for a long time is not feasible because the resources are controlled by different parties.

2.3 Business activities

The business activity coordination type supports long-running distributed transactions. Unlike atomic transactions, business activities do not lock resources until the completion of the transaction. These transactions are characterized by relaxed isolation, i.e., intermediary results can be seen by other transactions. If a business activity has to be rolled back, the already completed parts of the transaction must be reversed using appropriate compensation logic.

WS-BusinessActivity defines two coordination types: *AtomicOutcome*, in which the transaction coordinator directs all participants uniformly to close or compensate their work, and *MixedOutcome*, in which the coordinator can direct some participants to close and others to compensate. Moreover, WS-BA defines two coordination protocols. In the *BusinessAgreementWithParticipantCompletion* protocol, a participant informs the coordinator when it completes his/her part of the transaction. Then, the coordinator tells the participant to close or compensate. In *BusinessAgreementWithCoordinationCompletion*, the participant waits until the coordinator tells it to complete.

2.4 Business Process Execution Language

The work described in this paper uses BPEL 1.1 [10], which is similar to a large extent to the WS-BPEL 2.0 standard [2]. The main concepts in BPEL are the *partners*, which represent the parties that the process interacts with, the *variables*, which are containers for the data that is exchanged between the process and its partners, and the *activities*, which are the units of the work in the process.

BPEL distinguishes *atomic* activities such as *invoke*, which is used for calling a partner Web Service, and *structured* activities, which contain other activities and control the order of their execution, e.g. *sequence*. A *scope* is a special structured activity, which provides context for fault handling and compensation handling.

Compensation in BPEL works as follows: the activity that may need to be compensated is nested in a *scope* (named *s* for instance) and a compensation handler, which defines some logic to undo the effects of that activity is attached to that scope. This handler can be called implicitly by the default fault handler of the parent scope of *s* when a fault occurs or explicitly by using the *compensate* activity in a fault handler or in another compensation handler.

3 Transactions in BPEL Processes

In this section, we present some transactional requirements in BPEL processes and discuss the limitations of BPEL's compensation mechanism with respect to them.

3.1 Transactional requirements in BPEL

Web Service based business processes and their partners build distributed applications that have some business state, which changes by means of operation calls (via messaging activities in BPEL). A whole set of operation calls may need to be executed completely and successfully to move the system from one state into another valid state.

```
<process name="BankTransfer">
  <partners>
    <partner name="customer" partnerLinkType="customerSLT"/>
    <partner name="subsidary" partnerLinkType="subSLT"/>
    <partner name="tobank" partnerLinkType="tobankSLT"/>
  </partners>
  <variables>
    ...
  </variables>
  <sequence name="MainSequence">
    <receive name="receiveTransfer" partner ="customer"
      portType="transferServicePT" operation ="transfer "
      variable ="clientrequest " createInstance = "yes"/>
    <assign> ... </assign>
    <scope name="debit-credit-scope">
     <sequence name="debit-credit">
      <invoke name="invokeDebit" partner="subsidary"
        portType="subsidaryBankService" operation ="debit"
        inputVariable ="debitIn" outputVariable ="debitOut"/>
      <invoke name="invokeCredit" partner="tobank"
        portType="toBankService" operation ="credit"
        inputVariable ="creditIn" outputVariable ="creditOut"/>
     </sequence>
    </scope>
    <assign > ... </assign>
    <reply partner ="customer" portType="transferServicePT"
      operation ="transfer" variable ="clientresponse "/>
  </sequence>
</process>
```

Listing 1. The Bank Transfer Process

For example, consider the bank transfer process shown in Listing 1. This process transfers money from one account to another using two *invoke* activities that call the Web Services of two banks: one calls the operation *debit* on the Web Service of the bank that hosts the account to be charged and one calls the operation *credit* on the Web Service of the bank that hosts the account to be credited. If only one of the operations fails, the resulting state would be inconsistent. To avoid such inconsistency, transactions providing all or nothing semantics are needed, i.e., transaction support in BPEL processes is necessary.

In the following, we focus on the relationship of transactions to BPEL constructs such as activities and variables. Thereby, we focus only on the cases where the process is the *initiator* of the transaction, i.e., the party that controls the transaction and the partners are *participants* (i.e., execute some operation in the context of a transaction that is controlled by the process).

3.1.1 T1: Transactional activities

One should be able to define the behavior of each activity in the BPEL process with respect to transactions. As structured activities provide a construct to group a set of child activities, it is quite natural to use them as transaction boundaries. Hence, we derive requirement *T1*: One should be able to define the transactional behavior of structured activities. Moreover, one should also specify for each transactional activity whether the respective transaction is a short-lived atomic transaction or a long-running compensation-based transaction.

Making a structured activity transactional means that all its child activities must participate in the same transaction. For illustration, consider again the *scope* activity *debit-credit-scope* in Listing 1. This scope should be defined as a transaction. As a result, the nested *invoke* activities become transactional and should then use the transaction protocol defined for the parent *scope* activity. For instance, if we declare the scope *debit-credit-scope* as transactional with ACID semantics then the two nested *invoke* activities will have to use the *Durable2PC* protocol defined in WS-AT.

For messaging activities, similar transaction concepts to container-managed transactions in EJB [12] or propagation types in the Spring framework [17] can be used.

3.1.2 T2: Supporting transaction rollback

When a fault occurs during the execution of a transactional activity (e.g., because of a process fault or because a partner is not available), the transaction must be rolled back (requirement *T2*). That is, the BPEL process should immediately send a rollback/compensate message to the transaction coordinator. Related to T2 there are the requirements T3 and T4 below.

3.1.3 T3: Restoring variable values

The data manipulation activity *assign* and the messaging activities *receive*, *reply*, and *invoke* can change the values of certain process variables. When these activities are part of a transaction that should be rolled back the original variable values before the transaction start must be restored (requirement *T3*).

3.1.4 T4: Isolating variable values

In scenarios where isolation is required restoring variables is not enough. One must also hinder activities that are not part of the atomic transaction from accessing variables that are changed during that transaction (requirement *T4*). If isolation is not preserved, activities that execute concurrently to the transactional activity may work with the outcome of the transaction that will not be committed.

Business activities are characterized by relaxed isolation, i.e., the data that is modified by the transaction is made durable without waiting for a commitment phase. Changes are written and made visible to the other parties instantly after executing the respective operation. If the effect of the operation should be canceled, a particular compensation operation has to be called. For example, to compensate the debit operation, one could call the credit operation, which increases the account with the same amount of money.

It is important to note that relaxed isolation is not feasible in all scenarios. For example, if a customer transfers 500 Euros from account A to account B by using a long-running transaction with relaxed isolation for the *debit* operation to A and the *credit* operation to B, then there would not be a 2PC protocol with a prepare phase, i.e., the operations *debit* and *credit* are executed immediately. Consequently, when another client accesses account B and debits 300 Euros, he sees an account balance of 200 Euros. If now the long running transaction fails and all operations are compensated, credit(500) to A and debit(500) to B are called to reverse the actions taken during the transaction. The debit(500) operation will fail due to the overdrawn account that cannot have a negative balance. This results in an inconsistent state, i.e., for the bank transfer scenario strict isolation is required.

Several levels of isolation were defined for SQL databases such as *weak read uncommitted*, *read committed*, *repeatable read*, and *serializable*. The latter means that transactions may only be processed one after another which inhibits concurrency. It could make sense to provide these levels for variable isolation as well as for the partners, but we think that this is too complex for the variables.

Regarding the isolation level of the whole transaction the process partners are required to provide a certain level of isolation, but one cannot assume that these external partners do support the required level. One does not even know if the partner implementation uses a database at all. Consequently, a general-purpose solution that provides a specific isolation level for BPEL transactions does not make sense. This is also the position of WS-AT, which leaves out isolation issues because it considers them as application-specific details. To avoid typical problems in transaction processing such as lost updates, dirty reads, and phantom reads, we propose providing the isolation level *serializable*.

3.2 Limitations of BPEL's compensation mechanism

BPEL's compensation mechanism does not support many of the transaction requirements that were discussed in the previous section. In the following, we discuss three limitations of that mechanism, which are related to the requirements T1 to T4.

First, the compensation mechanism does not support atomic transactions with strict ACID properties because it lacks support for isolation. Consequently, the bank transfer transaction cannot be implemented using BPEL compensation handlers, i.e., additional means are needed.

Second, the process is both the initiator and the coordinator of the transaction in BPEL. It defines the transaction boundaries and is also responsible for managing the transaction (i.e., it listens to faults during the process execution and in the partner calls, it starts the compensation logic if a fault occurs, etc.). In addition, the coordination model of BPEL is local, i.e., the process decides alone on whether compensation is needed. That is, if the process notices some error e.g., during a call to a partner operation it initiates compensation without interacting with the partner to check whether the compensation is really needed. As a result, a fault may occur at the partner without being noticed by the process and consequently no compensation will be done, which leads to an inconsistent state.

Third, in the programming model of BPEL, the process programmer has to write manually a lot of code for handling transactions, e.g., for defining scopes and nesting them correctly, defining compensation handlers, etc. In addition to being a tedious task, such an approach leads to mixing the code that belongs to the process business logic with the transaction code. Moreover, BPEL expects programmers to have knowledge not only on transactions in the processes that they define but also on reversing the operations of partner Web Services. However, it is more logical to suppose that the implementor of the partner Web service knows how to compensate the Web Service operations as assumed in WS-BA.

4 Supporting Transactions in BPEL

This section presents some ideas on how to support the transactional requirements that we presented in the last section. Then, it discusses our solution concept using aspects.

4.1 Addressing the requirements

We integrate the process execution with a middleware that provides the capabilities of WS-AT and WS-BA.

4.1.1 T1: Transactional activities

Figure 1 shows a process with an atomic *scope* activity that contains a *sequence* with two *invoke* activities, interacting with two different partners. To support the execution of that activity as an atomic transaction the BPEL process execution and the transaction middleware are integrated as follows: when the *scope* is started, the process tells the middleware to create a new activity using the activation service of the coordination framework (step 1). The activation service returns an atomic transaction context, which will be sent with the application messages of the nested *invoke* activities (step 2). This transaction context tells the partners (that must support WS-AT) to register at the registration service for the 2PC protocol (step 3). After the *scope* activity completes, the BPEL process registers for the completion protocol at the registration service (step 4), initiates the completion protocol and tells the coordinator to commit (step 5). The latter runs the 2PC protocol with the partners A and B (step 6) and sends the result to the BPEL process using the completion protocol. Depending on that, the process tells the coordinator whether the transaction should be committed or rolled back.

This means that the BPEL process or the BPEL engine must be able to perform the following actions: creating an activity, registering for completion, adding a coordination context to application messages, and supporting the completion protocol. Similar actions are needed for supporting business activities.

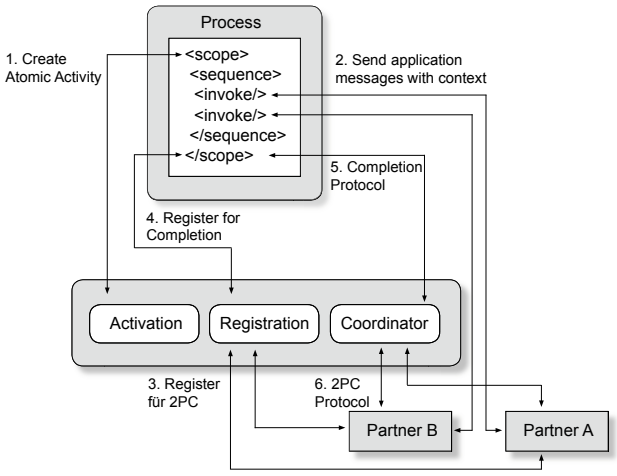

Figure 1. An atomic scope activity.

4.1.2 T2: Transaction rollback

If a transactional activity fails then the transaction coordinator should be told to rollback/compensate because the activity will not complete. To support rollback, we must add a fault handler to the BPEL process that calls the appropriate rollback/compensate operation of the transaction middleware. Adding these fault handlers and the activities for calling the rollback operations leads to mixing the transaction code with the business logic code.

4.1.3 T3: Restoring variable values

In order to restore variables we need to save the variable values before the execution of a transactional activity. Therefore, we could add new process variables to save the original variable values. If the atomic activity fails, the original variable values must be restored using these new variables. This resetting can be done using a compensation handler or a fault handler.

An alternative to this is to call a persistence service to save the original values. The service can also be used to retrieve these values again in the compensation handler or the fault handler. The advantage of using a persistence service is that most of the complexity of storing and restoring variable data will be hidden in the persistence service. That is we do not need to add new variables and *assign* activities to the process, which is quite tedious and leads to very complex process code that is bloated with transaction code.

4.1.4 T4: Isolating variable values

Activities that access variables that are written within an atomic transaction must wait until the transactional activity completes. That is, there is a dependency between the transactional activity and the other activities, which read the variables that are modified within the transaction. This dependency can be enforced by adding a link between both activities, which would result in mixing the transaction code with the code implementing the process business logic. An alternative solution to ensure the required isolation is the use of *serializable scopes* [10], i.e., scopes that provide concurrency control in governing access to shared variables. A serializable scope can be marked as such by setting its attribute *variableAccessSerializable* to *yes*. However, this alternative cannot be used in all cases because serializable scopes must not be nested and they must be leaf scopes.

4.2 Solution concept with aspects

It was clear from the ideas outlined in the paragraph 4.1.1 that the use of an additional technology is necessary for integrating BPEL with WS-AT and WS-BA. At the right points in the process execution, this integration technology should intervene to call the transaction middleware. In our work, AO4BPEL aspects play that integration role. Through their unique support for *cross-layer pointcuts* [3] these aspects are able to integrate process-level specifications such as BPEL with messaging-level specifications

such as WS-AT. That is, AO4BPEL aspects allow to express further transactional aspects that are not possible without using them (i.e., only with using BPEL).

In addition to integrating the process execution with the transaction middleware, aspects allow to separte the transaction code from the process code (i.e., extracting the transaction code from the BPEL processes). In fact, invasive changes are required to support T2, T3, and T4, e.g., to add a fault handler, add variables and activities to restore original variable values, or enforce isolation. These changes lead to mixed and tangled process code. By using AO4BPEL aspects, we ensure that these changes are done in a modular and non-invasive way without modifying the process. Through the reflective capabilities of AO4BPEL it is even possible to modify the attribute *serializable* of a scope using an aspect (to support T4). Adding fault handlers (to support T2) as well as variable and activities (to support T3) can be done using around advice.

5 Implementation

This section shows how our solution is implemented using AO4BPEL aspects and a transaction Web Service.

5.1 The AO4BPEL-based process container framework

AO4BPEL [6, 3] is an aspect-oriented extension to BPEL, which supports the modularization of crosscutting concerns and the dynamic adaptation of BPEL processes. An aspect is an XML document that defines a set *pointcuts* and *advice*. A *pointcut* is an XPath expression for selecting the joint points (i.e., points in the process execution) where the logic of the crosscutting concern should be executed. AO4BPEL supports two types of join points: *activity* join points select activities and *internal* join points select internal points during the execution of activities (e.g., the point where the outgoing message of an *invoke* activity is about to be sent out). A pointcut is associated with an *advice*, which is a BPEL activity that defines some crosscutting logic. The advice type (e.g., before, around, before soapmessageout) defines the order in which the advice is executed w.r.t. the join points that are selected by the pointcut. The advice language of AO4BPEL provides constructs to access the join point context such as the name of the join point activity, its input and/or output variable, the respective SOAP messages, etc. In addition to pointcuts and advice, an AO4BPEL aspect may also declare partners, variables, fault handlers, and compensation handlers. There is one implementation of AO4BPEL, which is based on IBM BPWS4J.

In [7], we presented an aspect-oriented and light-weight process container framework to support non-functional requirements in BPEL processes. This framework introduces a declarative XML-based deployment descriptor to specify the requirements of the activities and a process container, which intercepts the process execution and calls middleware Web Services to enforce these requirements. The process container is implemented by means of AO4BPEL [6] aspects that are generated from the deployment descriptor.

The users of the process container framework need only to know about the deployment descriptor, where they define the transaction requirements of the activities as well as other non-functional requirements such as security [5] and reliable messaging [8]. Then, with appropriate XSLT transformations, a set of AO4BPEL aspects is generated automatically. These aspects call WS-* based middleware Web Services to enforce the non-functional requirements.

The deployment descriptor is an XML document that consists mainly of *selectors* and *requirements*. With a selector, the user defines an XPath expression to choose one or more activities for which a certain requirement will be defined. For instance, with the expression *//invoke[@name='invokeDebit']*, the user can select any *invoke* activity that is called *invokeDebit*. With the *requirement* element, the user can define a non-functional requirement and associate it to the selected activities.

```
<bpel−dd>
  <selectors>
    <selector id="0" name="s0" type="compoundActivity">
     /process //scope[@name="debit−credit−scope"]
    </selector>
  </selectors>
  <services>
    <service name="transaction">
      <requirements>
        <requirement class="atomic" name="req0"
                     selectorid ="0" type="completion"/>
      </requirements>
    </service>
  </services>
</bpel−dd>
```

Listing 2. The deployment descriptor for the bank transfer process

Listing 2 shows the deployment descriptor for making the *scope* activity in the bank transfer process transactional with ACID properties. This requirement will be enforced using the transaction Web Service. The type of this transaction requirement is defined uniquely by combining the values of the attributes *class* (here atomic) and *type* (here completion), which means that for this activity an atomic transaction should be initiated with the completion protocol defined by WS-AT. The selector is associated with the requirement using the attribute *selectorid*. We have also developed a GUI tool that allows to define the requirements graphically and generate the deployment descriptor.

We have implemented XSLT transformations that take the deployment descriptor as input and generate AO4BPEL aspects, which intercept the process execution and call appropriate operations on the transaction Web Service to enforce the transactional requirements of the activities. For the requirement type defined in Listing 2, four aspects will be generated respectively for starting the transaction when the *scope* activity is executed, making the nested messaging activities participate in the transaction, committing the transaction after the completion of the *scope* activity, and rolling back the transaction in case of faults.

Figure 2 shows the interaction of the process container with the transaction Web Service to execute the *debit-credit-scope* as an atomic transaction. Before the *scope* starts, a new transaction is created by a transaction creation aspect that calls the operation *begin* on the transaction Web Service, which returns an identifier for this transaction. This identifier makes it possible to refer to the transaction later. Each nested *invoke* will be intercepted after message creation by the participation aspect, which sends its message to the transaction Web Service to enhance it with the transaction context. The enhanced message overrides the original one and the execution of the *invoke* activity resumes so that the enhanced message is sent to the partner. When the *scope* completes successfully, the commit aspect calls the operation *commit* on the transaction Web Service.

5.2 Examples of transaction aspects

Currently, we have XSLT templates supporting the generation of the four aspect types mentioned in the previous subsection. These aspect types allow supporting T1 and T2. Aspects belonging to the aspect types for creating and committing a transaction are relatively simple as they just contain a call to an operation on the transaction Web Service. In the following, we discuss the other two aspect types.

Listing 3 shows an aspect for participating in a transaction. Like all other transaction aspects, this participation aspect declares the transaction Web Service as partner. It also declares two variables respectively for holding the input and output data for calling the operation *participate*. The pointcut of this aspect selects all *invoke* activities that are nested in the *scope* named *debit-credit-scope*. If one of those activities is executed, the pointcut matches and the advice is executed in the order defined by the advice type.

The type of this advice is set to *before soapmessageout* meaning that the activity selected by the pointcut is intercepted after the SOAP request message is generated but before sending it to the partner. The advice of this aspect defines a *sequence* activity with one nested *invoke* that calls the operation *participate* and an *assign* activity that reads the SOAP message of the selected join point activity and writes it to the input variable of the *invoke* activity. The

```
<aspect name="atomicCompletionParticipate">
 <partnerLinks>
  <partnerLink name="transService" partnerLinkType="txPLT"/>
 </partnerLinks>
 <variables>
  <variable name="inMessage" messageType="rollbackRequest"/>
  <variable name="outMessage" messageType="rollbackResponse"/>
 </variables>
 <pointcut contextCollection ="true" name="s0">
  //scope[@name='debit−credit−scope']//invoke
 </pointcut>
  <advice type="before soapmessageout">
    <sequence>
      <assign>
       <copy>
        <from part="message" variable ="soapmessage"/>
        <to part="soap" variable ="inMessage"/>
       </copy>
       <copy>
        <from part="scopeid" variable ="ThisJPActivity"/>
        <to part="id" variable ="inMessage"/>
       </copy>
      </assign>
      <invoke inputVariable ="inputMessage"
              name=" TransactionService_participate "
              operation ="participate" outputVariable ="outMessage"
              partner =" transService " portType="txPT"/>
      <assign>
       <copy>
        <from part=" participateReturn " variable ="outMessage"/>
        <to part="newmessage" variable ="newsoapmessage"/>
       </copy>
      </assign>
    </sequence>
  </advice>
</aspect>
```

Listing 3. Aspect for participating in an atomic transaction

variable *inputMessage* has a part called *scopeid* that is used to identify the transaction. After calling the transaction Web Service, the operation *participate* returns the enhanced message with a transaction header. This message overrides the original SOAP message of the intercepted join point activity by using the special AO4BPEL variable *newsoapmessage* [6]. When an advice writes a message to this special variable, this message will be injected into the current join point activity.

Listing 4 shows a transaction rollback aspect. The pointcut of this aspect selects the transactional *scope* activity and puts another *scope* activity with a compensation handler around it (using the advice type *around*). The special activity *proceed* is a place holder for the activity that is selected by the pointcut. If a fault occurs during the execution of that activity the compensation handler will be executed and consequently the operation *rollback* will be called on the transaction Web Service.

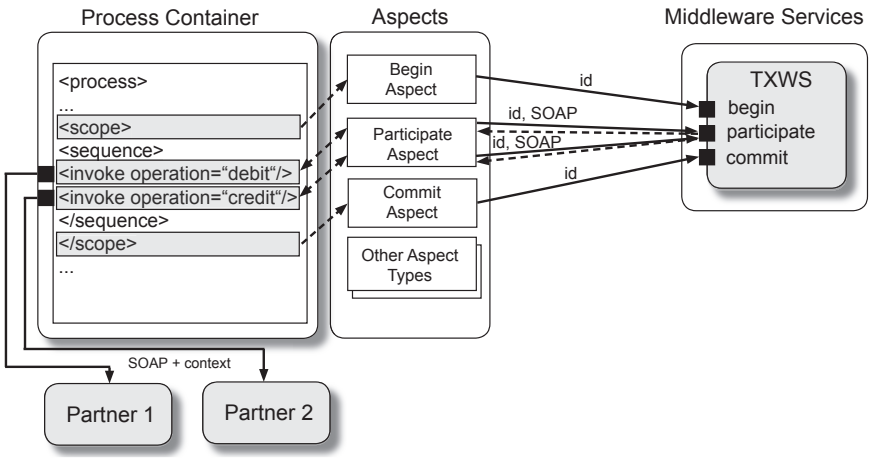

Figure 2. Interactions of the process container and the transaction Web Service

```
1  <aspect name=" transaction_rollback ">
2    ...
3   <pointcutandadvice>
4   <pointcut name=" transactivity ">
5    /process // scope[ @name="debit-credit-scope"]
6   </pointcut>
7   <advice type="around">
8    <scope>
9    <compensationHandler>
10   <sequence>
11    <assign>
12     <copy>
13      <from part="scopeid"  variable =" ThisJPActivity "/>
14      <to part="id"  variable ="inMessage"/>
15     </copy>
16    <assign>
17    <invoke name="trans_rollback"  operation =" rollback "
18          inputVariable ="inMessage"  outputVariable ="outMessage"
19          partner =" transactionService "  portType="txPT"/>
20   </sequence>
21   </compensationHandler>
22   <sequence>
23    <proceed/>
24   </sequence>
25   </scope>
26  </advice>
27  </pointcutandadvice>
28 </aspect>
```

Listing 4. Transaction rollback aspect

5.3 The transaction Web Service

We have implemented a transaction Web Service based on Apache Kandula, which is an open-source implementation of WS-AT and WS-BA.

5.3.1 The atomic transaction port type

For supporting transactions as defined in WS-AT, the transaction service offers the following operations, which are defined in the atomic transaction port type:

- `begin(String id)`
- `participate(String id, String soap)`
- `commit(String id)`
- `rollback(String id)`
- `registerContext(String id, String context)`

The creation of the transaction is done using the operation *begin*. As this operation must be called from the process container before an atomic activity is executed a *before* advice is used. Thereupon, the transaction service creates a new activity.

After executing the method *begin* the transaction service maintains the created context in order to add it to the application messages when the nested messaging activities are executed. In the bank transfer process these are the two *invoke*. Since we have already explained the participation aspect it is now clear how the enhancement of messaging activities with the transaction context is done. By sending the coordination context with the application message, the target Web Services register for the 2PC protocol at the transaction coordinator.

The initiator of the transaction (in this case the transaction Web Service as representative of the BPEL process) uses the completion protocol to control the coordination service. Therefore, it must register for completion at the registration service. To tell the coordinator that it may start

with the 2PC protocol, the transaction Web Service sends a completion commit message to it. This is initiated by a container aspect that calls the operation *commit* on the transaction Web Service after the completion of the atomic activity (as the advice type is *after*).

The explanation above shows that the transaction Web Service supports transactional activities (requirement T1) and transaction rollback (requirement T2). Restoring variables values (requirement T3) can be done by using aspects that interact with a persistence Web Service to save the original values of variables at transaction begin and restore them in case of a fault (in a similar manner to the rollback aspect). Requirement T4 can be supported when the transaction boundary is a scope by using an aspect that sets the attribute *serializable* to the value *yes* using the reflective variable *ThisJPActivity* and its part *variableAccessSerializable*. If this attribute cannot be used e.g., because a *scope* is nested or it is not a leaf scope, appropriate aspects could be deployed to hinder concurrent transactions from reading the data that is modified within another transaction.

5.3.2 The long-running transaction port type

The long running transactions port type of the transaction Web Service provides operations that support the execution of BPEL activities as business activities.

- `boolean begin(String id, boolean atomicOutcome)`
- `void beginNested(String id, String parentContext, boolean atomicOutcome)`
- `String participate(String id, String soap)`
- `boolean commit(String id)`
- `void compensate(String id)`

The operation *begin* is used to create a new business activity. The parameter *atomic outcome* can be set to true if strict atomicity is required or to false if a mixed outcome is required. The same applies to the operation *beginNested*, which takes an additional parameter *parentContext* that is a reference to the parent transaction, in which a new atomic transaction or business activity will be nested.

The operation *participate* is similar to that of the atomic transaction port type, as it enhances the message of a messaging activity with a business activity context that was previously created either by calling *begin* or *beginNested*. The operation *commit* is clear and the operation *compensate* starts the compensation mechanism of WS-BA.

The long running transactions port type is not implemented because Kandula does not yet support WS-BA.

6 Related Work

There are many works on transactional workflows based on advanced transaction models such as [22, 15, 18]. These works incorporate transaction semantics such as atomicity and isolation to insure a reliable workflow execution. For example, Leymann [18] introduced the concept of compensation spheres in the IBM FlowMark workflow system to allow the compensation of actvities. However, we observe that there are a few works on transactions in the context of BPEL such as [14] and [23].

In [14], Choreology proposed dedicated language extensions to BPEL for supporting transactions, e.g., the *businessTransaction* element is used to create or terminate transactions. Moreover, BPEL variables are used to hold coordination contexts and participant identifiers. Messaging activities are extended with two new attributes for the propagation of business transactions. Such an approach increases the complexity as language extensions will be needed for each non-functional concern (e.g., security) and breaks its portability. It is also against the principle of separations of concerns.

In [23], Tai et al. used WS-Policy [16] and WS-PolicyAttachment [9] to specify the transactional requirements of scopes and partner links. Whilst the purpose of the process container and the transaction Web Service is also to support transactions in BPEL processes, it is different from our approach w.r.t requirement specification and requirement enforcement.

With respect to requirement specification, one has to annotate each transactional activity in the BPEL process with an appropriate transactional policy. That is the specification of business logic and transaction code is intertwined. In contrast, our deployment descriptor allows a separate specification of the non-functional properties of the composition. In addition, it uses XPath-based activity selectors, which eliminates the need for attaching policies to BPEL activities in a point-wise fashion. A unified approach combining the benefits of both works (i.e., using WS-Policy with external policy attachment via XPath) is presented in [4].

With respect to requirement enforcement, policies allow to specify what is required but not how it should be enforced. The logic that enforces a certain requirement is hidden inside the policy handlers. Moreover, a special compiler is used in [23] to generate a Java stub that contains the necessary calls to the transaction middleware. As a result, it is not possible to exchange the transaction middleware by another one or to integrate further middleware services without changing the compiler. Such extensions are supported easily in the AO4BPEL container framework by writing appropriate aspects.

There are other works that used AOP to modularize transaction management in object-oriented applications

such as the domain-specific aspect languages proposed in Fabry and Cleenewerk [13] and in Spring AOP [17].

7 Conclusion

To enable transactional workflows in BPEL, we used the AO4BPEL process container framework, which provides several benefits. It is open and light-weight, i.e., it can be easily extended by deploying new aspects. Moreover, further middleware Web Services can be integrated. Another advantage is the separation of concerns as functional and non-functional code are separated. In addition, the use of XPath allows to quantify over several processes. On the other hand our approach has some limitations w.r.t performance (e.g., the overhead for pointcut matching) and works only for our AO4BPEL engine.

As future work, we will provide transaction aspects that allow the BPEL process to be a participant in a transaction. This is needed in many scenarios, e.g., the transfer process may be called as part of another transactional activity such as a rental car booking process that uses the transfer process for payment. Moreover, we will implement appropriate XSLT templates to generate aspects for restoring variable values in case of transaction rollback and for enforcing the variable isolation level *serializable*. In addition, the business activity port type will be implemented as soon as support for WS-BA is available in Sandesha. Another thrust of future work is to decouple the AO4BPEL language and its implementation from SOAP so that it can be used with the other messaging layers that can underly BPEL.

References

[1] Apache. Kandula 0.2. `http://ws.apache.org/kandula/`, May 2006.

[2] A. Arkin, S. Askary, B. Bloch, et al. Web Services Business Process Execution Language 2.0, OASIS Standard, 11 April 2007.

[3] A. Charfi. *Aspect-Oriented Workflow Languages: AO4BPEL and Applications*. PhD thesis, Darmstadt University of Technology, Darmstadt, Germany, 2007. `http://elib.tu-darmstadt.de/diss/000852/`.

[4] A. Charfi, R. Khalaf, and N. Mukhi. QoS-aware Web Service Compositions Using Non-Intrusive Policy Attachment to BPEL. In *Proc. of the 5th International Conference on Service Oriented Computing (ICSOC), Industry track, to appear*. Springer, September 2007.

[5] A. Charfi and M. Mezini. Using Aspects for Security Engineering of Web Service Compositions. In *Proc. of the 3rd IEEE International Conference on Web Services (ICWS)*, pages 59–66. IEEE Computer Society, July 2005.

[6] A. Charfi and M. Mezini. AO4BPEL: An Aspect-Oriented Extension to BPEL. *World Wide Web Journal: Recent Advances on Web Services (special issue)*, March 2007.

[7] A. Charfi, B. Schmeling, A. Heizenreder, and M. Mezini. Reliable, Secure and Transacted Web Service Composition with AO4BPEL. In *Proc. of the 4th IEEE European Conference on Web Services (ECOWS)*, pages 23–34. IEEE Computer Society, December 2006.

[8] A. Charfi, B. Schmeling, and M. Mezini. Reliable Messaging for BPEL Processes. In *Proc. of the 4th IEEE International Conference on Web Services (ICWS)*, pages 293–302. IEEE Computer Society, September 2006.

[9] Chris Sharp (Eds.). Web Services Policy Attachment (WS-PolicyAttachment). `ftp://www6.software.ibm.com/software/developer/library/ws-polat.pdf`, September 2004.

[10] F. Curbera, Y. Goland, J. Klein, et al. Business Process Execution Language for Web Services (BPEL4WS) Version 1.1, May 2003.

[11] David Langworthy (Eds.). Web Services Coordination (WS-Coordination) 1.1, April 2007.

[12] L. G. DeMichiel and M. Keith. Enterprise JavaBeans Specification 3.0, May 2006.

[13] J. Fabry and T. Cleenewerck. Aspect-Oriented Domain Specific Languages for Advanced Transaction Management. In *Proc. of the 7th International Conference on Enterprise Information Systems (ICEIS)*, pages 428–432, May 2005.

[14] T. Flechter, P. Furniss, A. Green, and R. Haugen. BPEL and Business Transaction Management, Choreology submission to OASIS, 2003.

[15] D. Georgakopoulos and M. Hornick. A framework for enforceable specification of extended transaction models and transactional workflow. *Journal of Intelligent and Cooperative Information Systems*, 3(3), 1994.

[16] Jeffry. Schlimmer (Eds.). Web Services Policy Framework (WS-Policy). `ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf`, September 2004.

[17] R. Johnson. Introduction to the Spring Framework. `http://www.theserverside.com/articles/article.tss?l=SpringFramework`, May 2005.

[18] F. Leymann. Supporting business transactions via partial backward recovery in workflow management systems. In *Proc. of BTW*, 1995.

[19] F. Leymann and D. Roller. *Production Workflows*. Prentice-Hall, 2000.

[20] OASIS WS-TX TC. Web Services Atomic Transaction (WS-AtomicTransaction) 1.1, April 16, 2007.

[21] OASIS WS-TX TC. Web Services Business Activity (WS-BusinessActivity) 1.1, April 16, 2007.

[22] A. Sheth and M. Rusinkiewicz. On transaction workflow. *IEEE Data Engineering Bulletin*, 1993.

[23] S. Tai, R. Khalaf, and T. Mikalsen. Composition of Coordinated Web Services. In *Proc. of the 5th International Middleware Conference (Middleware)*, volume 3231 of *LNCS*, pages 294–310. Springer, October 2004.

[24] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Pearson Education, 2005.