TWSO – Transactional Web Service Orchestrations

Peter Hrastnik
ec3 – Electronic Commerce Competence Center
Donau-City-Straße 1, A-1220 Vienna, Austria
peter.hrastnik@ec3.at

Werner Winiwarter
Departement of Scientific Computing
University of Vienna
Universitätsstraße 5, A-1010 Vienna, Austria
werner.winiwarter@univie.ac.at

Abstract

There is a need for transactional processing in the Web service world. Software industry responded to this need by publishing a couple of Web service transaction proposals that are quite alike. However, these proposals define basically only communication protocols that indirectly implement advanced transaction models. The proposals lack accurate usage suggestions and the rather obvious question "How can I use transactions in Web service based distributed systems?" is not covered anywhere satisfyingly. The use of arbitrary advanced transaction models is provided only by some of the proposals and likely requires an update of various transaction system components. This paper introduces TWSO (Transactional Web Service Orchestrations), a new approach to integrate transactional processing with Web service orchestrations. It tries to overcome the hassles stated above. TWSO concepts may appear in different manifestations, like an XML vocabulary (TWSOL) or an API for Java (TWSO4J). Constructs of TWSO manifestations are intended to be directly incorporated in Web service orchestration definitions. The usage pattern of TWSO is designed to resemble the programming pattern used when application programmers use transaction–enabled components like databases or application servers. Moreover, arbitrary advanced transaction models can be synthesized by using a basic set of transaction primitives without the demand for system–updates.

1. Introduction

Recently a couple of industrial proposals for Web service transactions have been published. They describe communication–protocols between transaction systems and Web services that take part in a transaction and embed these protocols in a transaction processing architecture that fits well into the Web service world.

Mostly, such protocols adhere to the semantics of advanced transaction models [14]. Advanced transaction models try to relax the rigid demands of ACID transactions [14]. In tightly coupled systems (for example, a client that uses a remote relational database), transactional processing that follows the ACID principles is ubiquitous and works well [17]. However, transactions that follow ACID principles may not be practical in loosely coupled systems, e.g. systems composed of Web services. Potts et al. [17] discuss ACID transactions regarding Web services in detail and even assert that "transaction semantics that work in a tightly coupled single enterprise cannot be successfully used in loosely coupled multi-enterprise networks such as the Internet".

If an application programmer wants to use ACID transactions these days, a ubiquitous pattern is used. Let us assume that we want to use a relational database in some procedural/object–oriented programming language and want to insert a new record into a person and a phone table. We would have to acquire the data (firstname, lastname, birthdate and phonenumber) using the programming language. Then we begin a new transaction, and insert the data into the person and then into the phone table. Users that access the database at the same time of our transaction's runtime will see a semantical correct version of the data, i.e. they will not see the tentative changes made by inserting new records before persisting them. If the insert operations are successful, we would commit the transaction and make the changes to the data persistent and visible. If something goes wrong (e.g. the table space is full when inserting a record into the phone table), we would abort the transaction and the tentatively inserted person record will be undone. In such ACID transactions, we use "begin", "abort" and "commit" transaction primitives. Transaction primitives are basic operations that control transactions.

However, the published industrial proposals for Web service transactions do not provide such a usage pattern. As indicated above, they just provide the communication protocol (under which circumstances which transaction related command can be called) but no clear understanding on how the application programmer can use Web service transactions. Such an understanding would support the basic ideas of transaction–oriented processing, namely "relieving the application programmer from worrying about failure and concurrency interleaving" [12].

Considering Web service orchestrations[1], using transaction concepts may facilitate their design and development process in terms of [12]. Common technologies for Web service orchestrations include, for example, BPEL4WS [2] and XPDL [20]. XPDL does not take transactional processing into account at all. BPEL4WS tries to offer transactional concepts by offering explicit compensation operations. This is a beginning, but far away from a satisfying support of Web service transactions in Web service orchestrations. Moreover, Web service orchestrations can also be implemented on a lower level using common programming

[1] Web service orchestrations tie together a set of existing Web services in order to create a completely new service by employing workflow technologies [15].

languages like Java, C#, etc. To our knowledge, there is no technology that offers transaction support for Web service orchestrations in that area. Thus, there is a significant gap between Web service orchestration technologies and Web service transactions.

Our goal is to fill this gap and facilitate transactional Web service processing in Web service orchestrations. We present an approach — called TWSO (Transactional Web Service Orchestrations) — that allows integrating transactional processing with Web service orchestrations to relieve the Web service orchestration creator from worrying about failure and concurrency interleaving.

2. Related work

2.1. Web service transaction proposals

Industrial organizations have published proposals that deal with transactional processing in the Web service world. Business Transaction Protocol (BTP) [7] was released under the patronage of OASIS. Well–known participating organizations are Oracle, Sun and Bea. WS–Transactions (WS–TX) [6] is a proposal by IBM, Microsoft and BEA, and WS Composite Framework (WS–CAF) [5] stems mainly from Oracle and Sun.

These proposals are very similar and differ only in details while the basic building blocks are elementary the same. They describe an architecture of a Web service transaction system, which includes participating Web services and a hierarchy of central components that offer transaction–related services and communicate transaction–related matters to affected participating components. Based on this framework, different protocols that handle communication between all affected components are defined. These protocols adhere to certain transaction semantics. Thus, if protocol p_x conforms to transaction semantic s_x, and a transaction t_a is executed using p_x, the semantics of t_a conform to s_x. The offered transaction communication protocols are based on the semantics of so–called advanced transaction models (ATMs), as described below. The usage of arbitrary advanced transaction models is provided by WS–TX and WS–CAF. However, to do so it is necessary to introduce a new communication protocol. How this is done is not standardized in the proposals. In addition, this seems to be unwieldy because it is likely that most of the transaction system has to be updated in order to be aware of such new communication protocols.

2.2. Advanced transaction models

Advanced transaction models (ATMs) were presented to overcome the restrictions of ACID style transactions, which are unsuitable for some domains. They offer appropriate transaction semantics for such domains by relaxing the rigid semantics of ACID transactions.

For example, *nested transactions* [11] is a well known advanced transaction model. A nested transaction is a tree of transactions. The sub–trees are either nested transactions or single transactions. The superordinate of a sub-transaction is called parent, the subordinate of a transaction is called child. If a transaction rolls back, all child–transactions have to roll back, too. The commit of a child–transaction does not take effect until the parent–transaction commits. The parent–transaction is the only instance that can see the changes of a child–transaction's commit. Thus, any child-transaction can fully commit only when its parent–transaction commits. However, after a commit, the parent-transaction will see the effects of the child-transaction. Another well–known advanced transaction model is *multilevel transactions* [19]. This model is similar to nested transactions, but allows a complete commit of the results of subtransactions before their parent-transactions commit. The results take effect immediately. However, it is possible to retract the committed results of subtransactions by using compensation transactions. A compensation transaction is a "forward" action that makes some adjustments to reverse the original action. After a compensation transaction, the fact that the original action took place is visible. In contrast, a rollback undoes an action so that it seems like the action never took place. A *saga* is also a common advanced transaction model [13]. It is a chain of transactions. Each transaction in the chain commits when it finishes its work, and provides a compensation transaction. When a transaction t_i fails, t_i aborts, and all previous transactions ($t_1, ..., t_{i-1}$) have to start their compensation transactions.

It should be noted that advanced transaction models may require more transaction primitives than ACID style transactions. For example, in multilevel transactions, besides "begin", "commit", and "rollback", we would need "compensate" and also primitives to set up the tree structure.

2.3. Advanced transaction meta models

Advanced transaction meta models are formal models that can be used to describe advanced transaction models.

ACTA is a framework that can be used to specify, analyze, and synthesize advanced transaction models [8]. It is a very comprehensive meta–model for advanced transactions and it is unlikely that a particular idea for a custom advanced transaction model cannot be represented in ACTA. However, this completeness causes complexity, and ACTA itself is neither easy to understand nor easy to use.

Specialized approaches that use ACTA for defining advanced transaction models have been proposed. For example, ASSET [3] and Bourgogne transactions [18] use the ideas of ACTA but simplify the usage of ACTA significantly. Both approaches are based on a set of general transaction primitives that can be applied to define customized transaction models suitable for specific domains. In ASSET, these transaction primitives are intended to be used in arbitrary programming languages while Bourgogne transactions target the Java Enterprise Edition (J2EE).

Another advanced transaction meta model was proposed in [14] by Jim Gray and Andreas Reuter. It is based on event–state diagrams and is well suited to describe and analyze advanced transaction models in a formal way. In [16], Hrastnik and Winiwarter present an approach for using this advanced transaction meta–model in Web service environments, which is appropriate for a formal description of Web service advanced transaction models.

3. Architecture of a TWSO environment

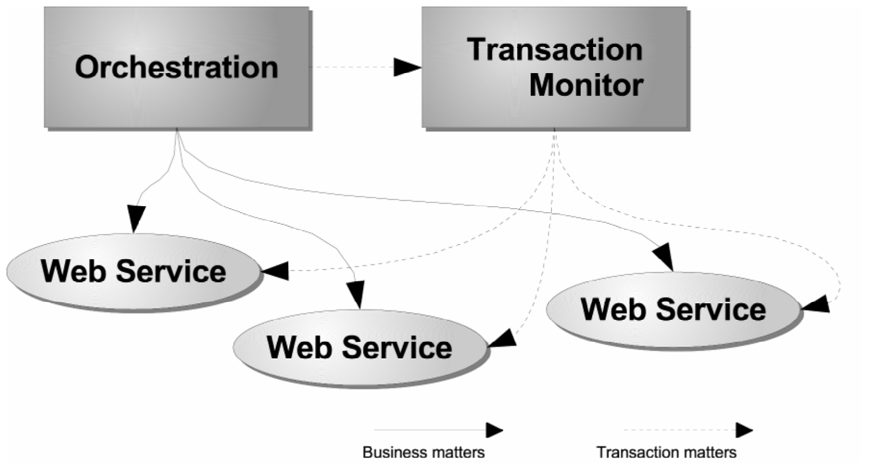

Figure 1. Architecture of a TWSO system

Figure 1 shows the complete architecture of a TWSO environment. There are two major components. The orchestration engine or application combines Web services in terms of workflow, i.e. it calls Web services in some particular order. In addition, it issues transaction specific commands (transaction primitives) to a transaction monitor component. Based on these transaction specific orders, the transaction monitor sends transaction primitives to affected Web services. Transaction primitives are communicated using Web service techniques, e.g. SOAP.

In contrast to the WS–transaction proposals introduced in Sect. 2.1, we do not consider a network of communicating transaction managers, because we think that a single transaction manager is sufficient and the huge complexity of a system of multiple distributed transaction monitors outweighs its advantages, which are organizational/political independence[2] and no single point of failure[3].

We pick up the idea of a transaction context that is forwarded in application calls as suggested in the WS–transaction proposals: When a transaction is instantiated in the transaction monitor it gets a unique identifier. This identifier is sent in each application call and transaction primitive call. Thus, if a Web service is called and the call includes such an identifier, the Web service is aware that transaction specific processing has to be considered. Transaction primitive calls also carry the transaction context, so that the transaction manager knows on which transaction the primitive should be applied. In a SOAP Web service call, using SOAP headers may be a good option to transport the transaction context.

To recapitulate, the orchestration engine or application instantiates one or more transactions and receives a unique identifier for each transaction from the transaction manager. The orchestration engine or application calls Web services in a specific order, and — on another conceptual layer — it calls transaction specific commands. When a Web service receives an application call containing a transaction context, it is aware that this call may involve transaction specific processing. It does such transaction specific processing when it receives transaction primitives from the transaction manager, whereas the transaction manager acts under the control of the orchestration engine or application.

[2] Is trust between organizations that participate in a transaction system high enough so that an organization is willing to obey the orders of another organization's transaction monitor?

[3] Actually, also a single transaction monitor can be designed to be virtually always available, so this motivation may not be accepted.

4. Building blocks of TWSO

TWSO is inspired by the ideas of [3], [18] and [8]. Similar as in ACTA, we distinguish between Web service calls and transaction primitives. Web service calls are operations on the Web service state and may be influenced by executed transaction primitives, e.g. an abort of transaction *t* may undo all changes of the call of Web service *ws*. TWSO constructs are intended to be embedded in arbitrary (as far as possible) orchestration technologies, like XPDL, BPEL4WS or Java. Furthermore, we follow the design of ACTA and Gray and Reuter's transaction meta model [14] and model advanced transaction models as compositions of one or more individual transactions.

TWSO consists of three building blocks. *Transaction primitives* control transactions and are used directly in the orchestration. The *transaction structure* models the interdependencies of the used individual transactions in a TWSO orchestration. The combination of transaction primitives and interdependencies of transactions in an orchestration implements arbitrary advanced transaction models[4], of course, also including the ones introduced in Sect. 2.2. Furthermore, we need *manifestations* of TWSO concepts to be able to use TWSO in as much orchestration techniques as possible.

4.1. Transaction primitives

TWSO is based on the following set of transaction primitives. It should be noted, that solely the Web service is responsible for taking the right steps according to a received transaction primitive. For example, a commit on a Web service that does not manipulate any persistent data may trigger no action at all and is correct.

Begin simply starts a transaction. *Commit* may be issued if the transaction's outcome is considered to be successful and should be finished. A Web service may, for example, persist the transaction's changes and make them visible to all users. *Abort* can be issued if the transaction should be aborted while running. Typically, a classic rollback will be executed by the Web service. In case the changes of a Web service should be undone even after a commit, *compensate* can be used. To assign the responsibility of termination of a transaction to another transaction, the *delegate* primitive can be applied. For example, in nested transactions (see Sect. 2.2), delegate would be issued when a child transaction is ready to commit so that the parent transaction takes command over termination of its child transaction. Because it seems to be adequate, we assume that only termination obligations (i.e. compensation, abort, commit) and only all of them at once can be delegated.

These transaction primitives should be sufficient for many applications. However, if needed, the set can be enhanced (see Sect. 4.3). Of course, all participating components (e.g. transaction managers, Web services, etc.) have to be aware of the transaction primitives.

The transaction primitives cannot be issued in arbitrary order. Thus, we define states of a transaction, valid transaction primitives on this state and state transitions based on the issued transaction primitive. The following states are defined. *Initiated* indicates that a transaction was setup. After a transaction has been begun, it is in the *in–progress* state. Based on the kind of termination, a transaction can be *committed*, *aborted* or *compensated*. If delegation has been applied, the corresponding transaction (i.e. the transaction from which termination obligations have been withdrawn) goes into the *delegated* state. Table 1 shows possible state transitions and effects of issued significant events on the transaction.

Table 1. Significant events and transaction states

state / primitive	initiated	in-progress	committed	aborted	compensated	delegated
begin	⇨$_{\text{in-progress}}$	✗	✗	✗	✗	✗
commit	✗	⇨$_{\text{committed}}$	✓	✗	✗	✗
abort	✗	⇨$_{\text{aborted}}$	✗	✓	✗	✗
compensate	✗	⇨$_{\text{compensated}}$	⇨$_{\text{compensated}}$	✓	✓	✗
delegate	✗	⇨$_{\text{delegated}}$	⇨$_{\text{delegated}}$	⇨$_{\text{delegated}}$	⇨$_{\text{delegated}}$	✗

⇨$_s$ = transition to state *s*, ✗ = exception, ✓ = valid operation but no state transition

4.2. Transaction structure

So far, we have defined some basic transaction primitives. It is possible to synthesize advanced transaction models using these primitives and (massive) explicit control flow logic articulated in the orchestration host–language. For example, to express the parent–child dependencies in nested transactions, one would have to perform something like `if (state(t_parent) == ABORT) { abort(t_child); }` explicitly. However, such an approach is not preferable since it mingles control–flow with transaction logic and obviously violates the *separation of concerns* principle [10]. Thus, it is desirable to remove as much transaction logic as possible from control flow logic by specifying the dependencies between transactions elsewhere and elsewise.

[4] It should be stressed that in contrast to the industrial Web service transaction proposals, there is no need for any software updates when using new transaction models in TWSO.

In TWSO, we define transaction dependencies as follows. A dependency consists of either a single transaction source state or a combination of transaction source states and one or more significant events on transaction(s). As soon as the transaction gets into the source state or the combination of source states of more transactions goes into effect, the significant events are issued to the affected transaction(s). For example, we could define a dependency saying that as soon as t_{s1} gets into state aborted and t_{s2} gets into state compensated, t_{d1} and t_{d2} should do a commit. It is possible to express this in the orchestration language using, for example,

```
if (state(t_s1) == ABORTED && state(t_s2) == COMPENSATED)
       { commit(t_d1); commit(t_d2); }
```

in a suitable[5] place of the workflow. However, if we wanted to change the logic of the dependency, we would have to fiddle the control flow. Using an explicit dependency, we would just change that. Control flow is not touched at all and separation of concerns is honoured.

4.3. Manifestations of TWSO

Until now, we have introduced the concepts of TWSO. However, we have not yet presented techniques that bring the concepts of TWSO to life and thus efficiently incorporate TWSO concepts in different Web service orchestration technologies. As stated before, a claim is that TWSO concepts should be able to be incorporated in as many Web service orchestration host–languages as possible. Moreover, TWSO concepts should interfere with the original orchestration definition (be it an XPDL XML document, Java source code, etc.) as little as possible. Because of flexibility and the separation of concerns principle, it should be possible to remove, add, and modify transaction logic in orchestrations with as little perturbation of the business logic (expressed by the orchestration) as possible.

For XML Web service orchestration technologies like XPDL, we decided to express the concepts of TWSO in XML. The resulting XML language is called TWSOL (Transactional Web Service Orchestration Language) and is introduced in Sect. 5. We intersperse TWSOL elements in Web service orchestration XML documents in order to embed transaction logic into orchestrations. By using XML–Namespaces [4] to discriminate TWSOL elements from the original orchestration elements, we can satisfy the claims above to a high degree. XML–Namespaces also allow extending TWSOL elements in a clean way. For instance, new transaction primitives (besides the ones presented in Sect. 4.1) can be introduced by defining them in new namespaces, and execution environments can decide on the basis of the found namespace whether they are able to execute the namespace's primitive or not.

For programming languages like Java, TWSO APIs suffice. Depending on the capabilities of the programming language, these APIs should be defined in an own namespace and should be defined in a way that separates API implementation and API definition. For example, TWSO4J, a TWSO API for Java that will be implemented, is defined in an own namespace and its implementation is separated from its definition by using Java interfaces.

5. TWSOL – XML language for TWSO

In this section, we present TWSOL XML elements that can be interspersed in orchestration host–languages. We show the syntax of TWSOL using XML–Schema. We do not provide global XML–Schema constructs like namespace or ID/IDREF definitions here. To setup a transaction, we need a unique id by which the transaction is identified. Furthermore, we also need to associate the transaction to one or more orchestration work items (a transaction can control more than one work item) that it is intended to manage. This setup information is kept in the `initiate` element:

```
<xs:element name="initiate">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="tx:activityRef" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="id_tx" type="xs:ID" use="required"/>
   </xs:complexType>
</xs:element>
```

The `<activityRef>` element should include ways to reference activities that occur in the orchestration in order to specify which activities should be managed by the transaction. How this is done depends on the orchestration host–language, therefore the XML–Schema allows any type of content in `<activityRef>`. If the orchestration host–language considers unique identifiers

[5] It should be noted that finding a "suitable place" in the workflow is cumbersome. Besides the actual business logic of the orchestration, one would have to make an additional concurrent execution path that involves a loop that observes the states of the dependencies transactions permanently. This loop would be stopped when the source states of the dependency go into effect (the corresponding significant events are issued) or when the dependency is considered to be not relevant anymore (e.g. when the business logic is finished).

for activities, `<activityRef>` could simply contain the matching identifier. If not, techniques that identify an element unambiguously in an XML document (e.g. XPath [9]) can be used, too:

```
<xs:element name="activityRef"/>
```

After initiating transactions, it should be possible to define their dependencies. The `dependency` element is used for that:

```
<xs:element name="dependency">
   <xs:complexType>
      <xs:sequence>
      <xs:element ref="tx:from"/>
            <xs:element ref="tx:to"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:ID" use="required"/>
   </xs:complexType>
</xs:element>
```

As one can see, `<dependency>` contains two children. The `<from>` child–element specifies the state(s) that have to be in effect in order to trigger the significant event(s) in the `<to>` child–element. The state of a transaction is specified with `<transactionState>` elements. These contain the type of the state (including the namespace it originates from, e.g. `base_states:committed`) and the id of the concerned transaction. The significant event(s) that should be executed is(are) defined via `<significantEvent>` elements in the `<to>` element. The details of `<significantEvent>` elements are covered below:

```
<xs:element name="from">
   <xs:complexType>
      <xs:choice>
         <xs:element ref="tx:transactionState"/>
         <xs:element ref="tx:stateConcatenation"/>
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="transactionState">
   <xs:complexType>
      <xs:attribute name="type" type="xs:NMTOKEN" use="required"/>
      <xs:attribute name="transaction" type="xs:IDREF" use="required"/>
   </xs:complexType>
</xs:element>

<xs:element name="to">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="tx:significantEvent" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

If the `<from>` element contains more than one significant event, the question how they should be combined arises: What combination of significant events has to occur in order to trigger the target significant events? We provide two nestable (a recursion is defined in the XML–Schema of the `<stateConcatenation>` element) different concatenation types: “and” and “or” (the semantic of “and” and “or” is the same as seen in numerous programming languages):

```
<xs:element name="stateConcatenation">
   <xs:complexType>
      <xs:choice minOccurs="1" maxOccurs="unbounded">
         <xs:element ref="tx:transactionState"/>
         <xs:element ref="tx:stateConcatenation"/>
      </xs:choice>
      <xs:attribute name="type" type="tx:concatenationType" use="required"/>
   </xs:complexType>
</xs:element>
```

After we setup the transactions, we have to provide a possibility to issue transaction primitives (i.e. significant events) in the orchestration. We can intersperse `<significantEvent>` elements in suitable places of the orchestration host–language. For example, in XPDL, an XPDL activity element could contain significant event commands. The `<significantEvent>` element has a `type` attribute that specifies the type of the significant event and the namespace it originates from, e.g. `base_events:commit`. To specify the affected transactions, a `to` and a `from` attribute (not to be confused with `<to>` and `<from>` in `<dependency>`) may be used. `To` defines the target transaction and, if necessary, `from` can be used to define a source of a significant event. For example, if one wants to delegate responsibilities using the significant event type `base_events:delegate`, both, the `from` and `to` attribute have to be given: Responsibilities will be transferred from the transaction specified in `from` to the transaction specified in `to`.

```
<xs:element name="significantEvent">
   <xs:complexType>
      <xs:attribute name="type" type="xs:NMTOKEN" use="required"/>
      <xs:attribute name="to" type="xs:IDREF" use="optional"/>
      <xs:attribute name="from" type="xs:IDREF" use="optional"/>
   </xs:complexType>
</xs:element>
```

6. Example

To give a deeper understanding of TWSO and TWSOL, we provide an example here. We will intersperse TWSOL elements into XPDL by using XPDL's built–in extension mechanism, namely "extended attributes".

Let us suppose that we want to synthesize a transaction based on the following scenario for a holiday booking: A flight should be booked, a rental car should be provided at the destination airport, and a hotel room should be prepared. We assume that the local car rental service needs 2 days to provisionally reserve a car and the airline needs just 2 minutes to provisionally reserve a free seat. Because it is probably unacceptable for any airline to hide the seat from other transactions for 2 days as it would be required for a classical ACID transaction (isolation), we have to consider compensation and immediate commitment of successful operations. Moreover, we reckon that if we have a car, it is much easier to find a hotel, and if we have a hotel room, we have a place to stay and can try to rent a car on–site. Thus, we require that the hotel booking and/or the car booking have to be successful to book the itinerary. If both, the car booking and the hotel booking fail, the whole transaction should fail, too. Moreover, if the flight booking fails, the whole transaction should fail because, if we cannot reach the destination, the on–site services would be useless.

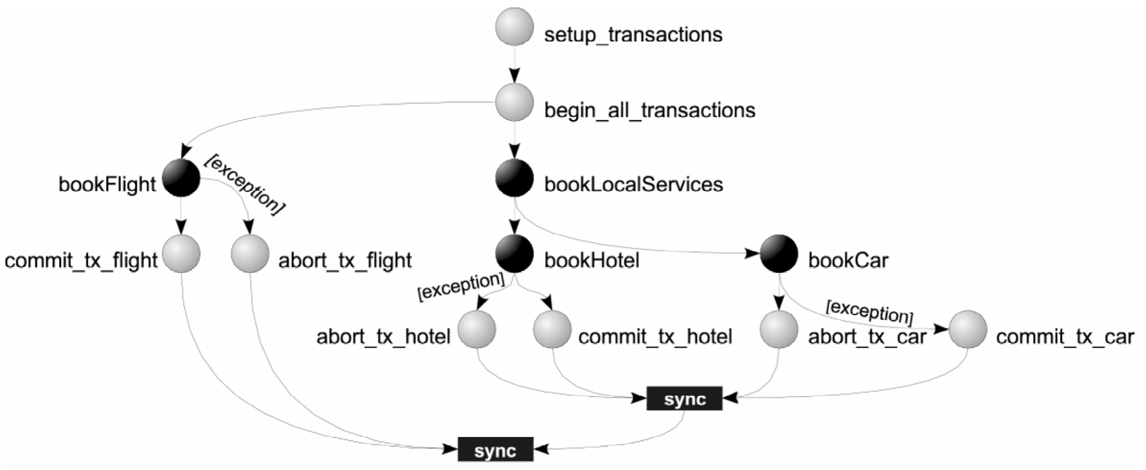

Figure 2. TWSO Web service orchestration using multilevel transaction semantics

This scenario can be modelled using the ideas of multilevel transactions [19]. Significant parts of the corresponding orchestration are illustrated in Figure 2 in an informal way. Circles represent a single unit of work (activity), the lighter ones represent transaction related activities, and the darker ones business related activities like Web service calls. If there are more outgoing transitions and at least one is marked with an expression like "`[condition]`", it means conditional execution. If `[condition]` evaluates to true, this transition is followed. Else, the transition without a condition expression is followed. If there are more outgoing transitions without any conditions, the sibling activities are executed concurrently. The dark bars in Figure 2 represent synchronization activities. Synchronization activities wait until all previous concurrently executed paths are finished.

Most transaction related semantics are specified at the beginning of the workflow in the `setup_transactions` activity. Here, three transactions and their dependencies are specified. Since XPDL requires unique identifiers for activities, we can simply refer to activities by quoting identifiers in the `<activityRef>` elements. The `flight_aborts` dependency causes the compensation of `tx_bookHotel` and `tx_bookCar` at the moment `tx_bookFlight` aborts. The `localServices_abort` dependency causes the compensation of `tx_bookFlight` in case both, `tx_bookCar` and `tx_bookFlight`, abort. This is expressed by the concatenation element of type "`and`" in this dependency:

```
<Activity id="setup_transactions">
   <xpdl:Implementation>
      <xpdl:No/>
   </xpdl:Implementation>
```

```
    <xpdl:ExtendedAttributes> <xpdl:ExtendedAttribute name="tx">

        <tx:initiate id="tx_bookFlight">
            <tx:activityRef>bookFlight</tx:activityRef>
        </tx:initiate>
        <tx:initiate id="tx_bookHotel">
            <tx:activityRef>bookHotel</tx:activityRef>
        </tx:initiate>
        <tx:initiate id="tx_bookCar">
            <tx:activityRef>bookCar</tx:activityRef>
        </tx:initiate>

        <tx:dependency id="flight_aborts">
            <tx:from>
                <tx:transactionState type="tx:abort" transaction="tx_bookFlight"/>
            </tx:from>
            <tx:to>
                <tx:significantEvent type="tx:compensate" to="tx_bookHotel"/>
                <tx:significantEvent type="tx:compensate" to="tx_bookCar"/>
            </tx:to>
        </tx:dependency>

        <tx:dependency id="localServices_abort">
            <tx:from>
                <tx:concatenation type="and">
                    <tx:transactionState type="tx:abort" transaction="tx_bookHotel"/>
                    <tx:transactionState type="tx:abort" transaction="tx_bookCar"/>
                </tx:concatenation>
            </tx:from>
            <tx:to>
                <tx:significantEvent type="tx:compensate" to="tx_bookFlight"/>
            </tx:to>
        </tx:dependency>

    </xpdl:ExtendedAttribute></xpdl:ExtendedAttribute>
</Activity>
```

After specifying the transactions and their dependencies, we start them in activity `begin_all_transactions`. Let us assume that this activity includes a `begin` significant event for each transaction.

The actual work is done concurrently, i.e. the flight booking is done at the same time as the car and hotel booking, and the car booking and hotel booking are done concurrently[6], too. If an exception happens while doing the particular bookings, the corresponding transaction is aborted, otherwise it is committed. To give an example for an activity that causes such a transaction related action, we present the `commit_tx_flight` activity. All other transaction related activities in the orchestration are defined in an analogous way:

```
<xpdl:Activity id="commit_tx_flight">
    <xpdl:Implementation> <xpdl:No/> </xpdl:Implementation>

    <xpdl:ExtendedAttributes>
        <xpdl:ExtendedAttribute name="tx">
            <tx:significantEvent type="tx_base:commit" to="tx_bookFlight"/>
        </xpdl:ExtendedAttribute>
    </xpdl:ExtendedAttribute>

</xpdl:Activity>
```

In case a transaction is aborted, suitable dependencies may be applied. For example, if the flight booking is aborted, the hotel booking and the car booking will be compensated. Execution stops at the synchronization activities until the local services booking path and the flight booking path finish, and, inside the local services booking path, the hotel booking path and the car booking path are finished.

[6] We decided to segment the orchestration into flight booking logic and local services logic to enhance readability. `book_local_services` is a dummy activity that should emphasize this segmentation. Speaking in terms of functionality, concurrent booking of the three services without this segmentation would not differ in any way.

7. Conclusion and next steps

In this paper, we have introduced TWSO, a new approach for transaction–oriented processing in Web service environments. Recent Web service transaction proposals like BTP, WS–TX, and WS–CAF mainly define communication protocols and infrastructure for Web service transactions. In contrast, TWSO focuses on upgrading the ubiquitous usage pattern for ACID transactions (i.e. "begin transaction; do business logic; commit or abort transaction") in order to be sufficient for Web service environments. It is intended to be used and easily integrateable with different existing Web service orchestration technologies. Furthermore, virtually any advanced transaction model can be synthesized using TWSO without any need for software updates.

TWSO is founded on ACTA, a formal model for describing, analyzing, and synthesizing advanced transaction models. In contrast to the ubiquitous ACID transaction usage pattern, TWSO offers an enhanced set of transaction primitives and the possibility to define inter–transaction dependencies to satisfy the demands of Web service transaction systems. We have also presented TWSOL, an XML representation of our approach that should be integrateable with existing Web service orchestration languages without inconveniences.

To enhance comprehension and prove capabilities of the approach, we have presented a real–world example that is based on the multilevel transactions model.

Future work will focus on the implementation of a TWSO system for executing transactional Web service orchestrations. For XML Web orchestration technologies like BPEL4WS, this can involve either the incorporation of native TWSO(L) support into an existing Web service orchestration engine or the translation of orchestrations enriched with TWSOL to pure standard orchestrations, like clean XPDL (according to [1] it may be possible to convert the TWSO logic "down" to pure orchestration constructs). We plan to support Java Web service orchestrations, too. This task includes the definition and implementation of a TWSO API (TWSO4J). To execute such orchestrations, we will implement a TWSO transaction monitor component. In addition, Web services should be able to publish existent TWSO related capabilities (most notably supported transaction primitives) in a machine readable way. Thus, a TWSO WSDL extension is reasonable and scheduled, too.

References

[1] Alonso, G., Agrawal, D., Abbadi, A. E., Kamath, M., Günthär, R., & Mohan, C. (1996). Advanced transaction models in workflow contexts. Proceedings of the 12th International Conference on Data Engineering. 574-581.

[2] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., & Weerawarana, S. (2002). Business process execution language for web services version 1.1. http://www.ibm.com/developerworks/library/specification/ws-bpel/. cited on 2005-09-27.

[3] Biliris, A., Dar, S., Gehani, N.H., Jagadish, H.V., & Ramamritham, K. (1994). ASSET: A system for supporting extended transactions. Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data. 44-54.

[4] Bray, T., Hollander, D., & Layman, A. (1999). Namespaces in XML. http://www.w3.org/TR/REC-xml-names/. cited on 2005-09-27.

[5] Bunting, D., Chapman, M., Hurley, O., Little, M., Mischkinsky, J., Newcomer, E., Webber, J., & Swenson, K. (2003). Web services composite application framework. http://developers.sun.com/techtopics/webservices/wscaf/. cited on 2005-09-27.

[6] Carbrera, L.F., Copeland, G., Feingold, M., Freund, T., Johnson, J., Kahler, C., Klein, J., Langworthy, D., Leymann, F., Nadalin, A., Orchard, D., Robinson, I., Shewchuk, J., Storey, T., & Thatte, S. (2004). Web services coordination, Web services business activity framework, Web services atomic transaction. http://www.ibm.com/developerworks/library/specification/ws-tx/. cited on 2005-09-27

[7] Ceponkus, A., Dalal, S., Fletcher, T., Furniss, P., Green, A., & Pope, B. (2002). Business transaction protocol. http://www.oasis-open.org/committees/download.php/1184/2002-06-03.BTP_cttee_spec_1.0.pdf. cited on 2005-09-27.

[8] Chrysanthis, P.K., & Ramamritham, K. (1994). Synthesis of extended transaction models using ACTA. ACM Transactions on Database Systems, 19(3), 450-491.

[9] Clark, J., & DeRose, S. (1999). Xml path language (XPath) version 1.0. http://www.w3.org/TR/xpath. cited on 2005-09-27.

[10] Dijkstra, E.W. (1997). A Discipline of Programming. Upper Saddle River, NJ, USA: Prentice Hall PTR.

[11] Eliot, J., & Moss, B. (1981). Nested Transactions: An Approach to Reliable Distributed Computing. Unpublished Ph.D Dissertation, Dept. of Electrical Engineering and Computer Science, MIT.

[12] Fekete, A., Greenfield, P., Kuo, D., & Jang, J. (2003). Transactions in loosely coupled distributed systems. Proceedings of the Fourteenth Australasian Database Conference on Database technologies.

[13] Garcia-Molina, H., & Salem, K. (1987). Sagas. Proceedings of the 1987 ACM SIGMOD International Conference on Management of data. 249-259.

[14] Gray, J., & Reuter, A. (2002). Transaction Processing: Concepts and Techniques. San Francisco, California: Morgan Kaufmann Publishers.

[15] Hrastnik, P. (2004). Execution of business processes based on web services. International Journal of Electronic Business. 2(5), 550-556.

[16] Hrastnik, P., & Winiwarter, W. (2004). An advanced transaction meta–model for web services environments. Proceedings of the Sixth International Conference on Information Integration and Web–based Applications & Services (iiWAS2004). 303-312.

[17] Potts, M., Cox, B., & Pope, B. (2002). Business Transaction Protocol Primer. http://www.oasis-open.org/committees/business-transactions/documents/primer/Primerhtml/BTP%20Primer%20D1%2020020602.html. cited on 2005-09-27.

[18] Prochazka, M. (2002). Advanced Transactions in Component-Based Software Architectures. Unpublished Ph.D Dissertation, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague.

[19] Weikum, G., & Schek, H.J. (1991). Multi-level transactions and open nested transactions. Data Engineering, 14, 60-64.

[20] Workflowmanagement Coalition (2002). Workflow process definition interface–xml process definition language. http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf. cited on 2005-09-27.

Biographical Notes

Mag. Peter Hrastnik holds a Scientific Researcher position at the Electronic Commerce Competence Center (EC3) since 2003. He received an MS degree in 1999 from the University of Vienna and is about to finish his PhD studies at the Vienna University of Technology. From 1999 to 2001, he was employed by a mobile telecommunications provider to develop Internet data services. From 2001 to 2003, he worked as a Software Developer at EC3. Peter Hrastnik's main research areas are loosely coupled distributed systems and Web services. He is also interested in technologies for Web application development.

Prof. Dr. Werner Winiwarter holds a tenured position at the Department of Scientific Computing, University of Vienna. He received an MS degree in 1990, an MA degree in 1992, and a PhD in 1995, all from the University of Vienna. The main research interest of Winiwarter is human language technology. In addition, he also works on data mining and machine learning, Semantic Web, information retrieval and filtering, electronic business, digital libraries, and education systems.