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
Institute of Scientific Computing
University of Vienna
Universitätsstraße 5, A–1010 Vienna, Austria
werner.winiwarter@univie.ac.at

Abstract

Software industry responded to the need for transactions in the Web service world by publishing several proposals, that are quite alike. These proposals define basically communication protocols that indirectly implement advanced transaction models. However, the rather obvious question "How can I use transactions in Web service orchestrations?" is not covered anywhere satisfyingly. The use of arbitrary advanced transaction models is provided by some of the proposals, but likely requires an update of various transaction system components. This paper introduces TWSO (Transactional Web Service Orchestrations), an approach to integrate transactional processing with Web service orchestrations. It tries to overcome the hassles stated above and provides an XML vocabulary (TWSOL) that is intended to be incorporated in Web service orchestrations. The usage pattern of TWSO is designed to resemble the programming pattern used when nowadays application developers use transaction–enabled components like databases or application servers. Moreover, arbitrary advanced transaction models can be synthesized by using a set of transaction building blocks without the demand for system–updates.

1 Introduction

Recently a couple of 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 into the Web service world. Such protocols adhere to the semantics of advanced transaction models (ATMs) [8]. ATMs try to relax the rigid demands of ACID transactions. In tightly coupled systems (for example, a client that uses a remote relational database), transactional processing that follows the ACID principles [8] is ubiquitous and works well [11]. However, transactions that follow ACID principles may not be practical in systems composed of Web services. Potts et al. [11] 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".

In general, if an application programmer uses ACID transactions, a ubiquitous pattern is used. First, a new transaction is *begun*. Then (in the context of the new transaction) business logic is executed. If business logic was executed successfully, the transaction is *committed* to make potential business logic's state changes persistent and visible. In case of any error during execution of the business logic, the transaction is *aborted* and application logic's (tentative) state changes are undone without leaving any traces.

The industrial proposals for Web service transactions do not provide such a usage pattern. They just provide the communication protocol (under which circumstances which transaction related command can be called) but no clear understanding on how the application developer 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" [7].

Considering Web service orchestrations[1], using transaction concepts may facilitate their design and development in terms of [7]. Recent technologies for Web service orchestrations either lack (XPDL) or offer only marginal transactional concepts (BPML states that its implementations should support existing transaction proposals but omits how to do this, BPEL4WS offers explicit compensation only). Thus, there is a significant gap between current proposals for Web service orchestrations and Web service transactions. We present an approach called *TWSO* (Transactional Web Service Orchestrations) that tries to fill this gap and allows to integrate transactional processing with Web service orchestrations to relieve the Web service orchestration designer from worrying about failure and concurrency interleaving.

[1]Web service orchestrations tie together a set of existing Web services to create a total new service by employing workflow technologies [9].

2 Related Work

2.1 Advanced Transaction Models, Advanced Transaction Meta 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* is a well known ATM. For Web service environments, the concept of compensation actions that are used by some ATMs (e.g. *multilevel transactions* or *sagas*) can be useful. A compensation action is a “forward” action that makes some adjustments to reverse the original action. After a compensation action, 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.

To describe ATMs, advanced transaction meta models can be used. For instance ACTA is a framework that can be used to specify, analyze, and synthesize ATMs [5]. It is a very comprehensive meta–model for advanced transactions and it is unlikely that a particular idea for a custom ATM cannot be represented in ACTA. However, this completeness causes complexity, and ACTA itself is not easy to use. Specialized approaches that use ACTA for defining ATMs have been proposed. For example, ASSET [1] and Bourgogne transactions [12] use the ideas of ACTA but simplify the usage of ACTA significantly. Both approaches are based on a set of general transaction building blocks that can be applied to define customized transaction models. In ASSET, these transaction building blocks 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 [8] by Jim Gray and Andreas Reuter. It is based on event–state diagrams and is well suited to describe and analyze ATMs in a formal way. In [10], 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 ATMs.

2.2 Web Service Transaction Proposals

Industrial organizations have published proposals that deal with transactional processing in the Web service world, namely Business Transaction Protocol (BTP) [4] (Oracle, Sun, and Bea under patronage of OASIS), WS–Transactions (WS–TX) [3] (IBM, Microsoft and BEA), and WS Composite Framework (WS–CAF) [2] (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 ATMs. The usage of arbitrary ATMs 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.

3 Architecture of a TWSO Environment

Figure 1 shows an architecture of a TWSO environment. The orchestration engine combines Web services in terms of workflow, i.e. it calls Web services in some particular order. In addition, it issues transaction specific matters (transaction primitives and transaction structure, see Sect. 4) to a transaction monitor component. Based on such transaction matters, the transaction monitor may send transaction commands to affected Web services. Transaction matters are communicated using Web service techniques, e.g. SOAP.

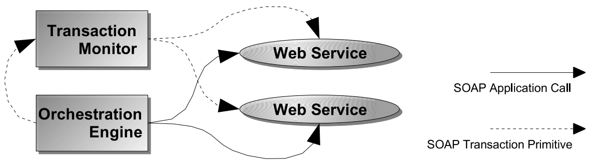

Figure 1. Architecture of a TWSO system

4 Building Blocks of TWSO

TWSO is inspired by the ideas of [1], [12], and [5]. Similar as in ACTA, we distinguish between Web service calls and transaction primitives. Web service calls are operations on the Web service state that 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 host–languages, like XPDL or BPEL4WS. Furthermore, TWSO follows the design of ACTA and Gray and Reuter’s transaction meta model [8] and models ATMs as compositions of one or more individual transactions and (as necessary) their interdependencies.

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 ATMs[2]. Furthermore, we need some technical means to be able to *glue* TWSO to as many orchestration host–languages as possible.

TWSO is based on the following set of *transaction primitives*. *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. E.g. in nested transactions, 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. Only termination obligations (i.e. compensate, abort, commit) and only all of them at once can be delegated. It should be noted, that solely the Web service is responsible for taking the right steps according to a received transaction primitive.

The introduced transaction primitives should be sufficient for many applications. However, if needed, the set could be enhanced easily by using XML-Namespaces, as described below. Of course, all participating components (e.g. transaction managers, Web services, etc.) have to be aware of the new 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 transaction primitives on the transaction.

It is possible to synthesize ATMs 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 [6]. 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.

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 transaction primitives 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 transaction primitives 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 "if then" statements in suitable places 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 and separation of concerns is honored.

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 host–language 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 (by the orchestration expressed) business logic as possible. Since XML is capable to satisfy our demands, we decided to express the concepts of TWSO in XML. Furthermore, virtually all Web service standards are represented in XML. Thus, it is highly reasonable to stick to this de facto standard. 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 to discriminate TWSOL elements from the original orchestration elements, we can satisfy the claims above to a high degree. XML–Namespaces also allow to extend TWSOL elements in a clean way. New transaction primitives can be introduced by defining them in new namespaces. Execution environments may decide on the basis of the found namespace whether they can execute the namespace's primitive.

[2]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.

Table 1. Transaction primitives and transaction states

	initiated	in–progress	committed	aborted	compensated	delegated
begin	$\Rightarrow_{in-progress}$	✗	✗	✗	✗	✗
commit	✗	$\Rightarrow_{committed}$	✓	✗	✗	✗
abort	✗	$\Rightarrow_{aborted}$	✗	✓	✗	✗
compensate	✗	$\Rightarrow_{compensated}$	$\Rightarrow_{compensated}$	✓	✓	✗
delegate	✗	$\Rightarrow_{delegated}$	$\Rightarrow_{delegated}$	$\Rightarrow_{delegated}$	$\Rightarrow_{delegated}$	$\Rightarrow_{delegated}$

$\Rightarrow_s$ = transition to state s, ✗ = exception, ✓ = valid operation but no state transition

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 elements using XML–Schema.

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 includes 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 simply allows any type of content in `<activityRef>`. If the orchestration host–language considers unique identifiers for activities, `<activityRef>` could simply contain the matching identifier. If not, techniques that identify an element unambiguously in an XML document (e.g. XPath) can be used, too.

To define transaction dependencies, the `<dependency>` element is used:

```
<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>
```

`<dependency>` contains two children. The `<from>` child–element specifies the state(s) that have to be in effect in order to trigger the transaction primitive(s) in the `<to>` child–element. The state of a transaction is specified with `<transactionState>` elements. These contain the type of the state and the id of the concerned transaction. The transaction primitive(s) that should be executed is (are) defined via `<txPrimitive>` elements (details of this element are covered below) in the `<to>` element:

```
<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:txPrimitive" minOccurs="1"
                                         maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType></xs:element>
```

If the `<from>` element contains more than one transaction primitive, the question how they should be combined arises: What combination of transaction primitives have to occur in order to trigger the target transaction primitives? 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 in the orchestration. We can intersperse `<txPrimitive>` elements in suitable places of the orchestration host–language. For example, in XPDL, an XPDL activity element could contain transaction primitive commands.

The `<txPrimitive>` element has a `type` attribute that specifies the type of the transaction primitive and the namespace it originates from, e.g. `tx_base: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 transaction primitive. For example, if one wants to delegate responsibilities using the transaction primitive type `tx_base: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="txPrimitive">
    <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.

This scenario can be modeled using the ideas of multilevel transactions [13]. Significant parts of the corresponding orchestration are illustrated in Fig. 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, the sibling activities are executed concurrently. If there are more outgoing transitions and one is marked with an expression like "`[`*condition*`]`", it means conditional execution. If `[`*condition*`]` evaluates to true, the corresponding transition is followed. Else, the transition(s) without a condition expression is (are) followed. The dark bars in Fig. 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

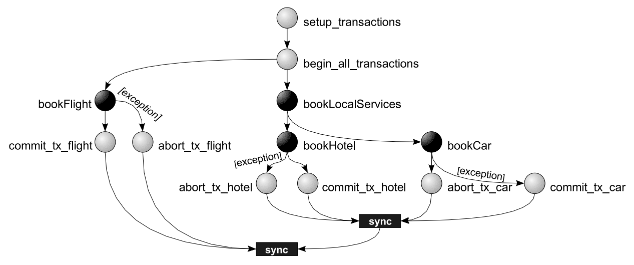

Figure 2. TWSO Web service orchestration using multilevel transaction semantics

`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>

  <!-- <initiate> definitions of tx_bookHotel and tx_bookCar
       are analogous to tx_bookFlight -->

  <tx:dependency id="flight_aborts">
    <tx:from>
      <tx:transactionState type="tx_base:aborted" transaction="tx_bookFlight"/>
    </tx:from>
    <tx:to>
      <tx:txPrimitive type="tx_base:compensate" to="tx_bookHotel"/>
      <tx:txPrimitive type="tx_base:compensate" to="tx_bookCar"/>
    </tx:to>
  </tx:dependency>
  <tx:dependency id="localServices_abort">
    <tx:from>
      <tx:concatenation type="and">
        <tx:transactionState type="tx_base:aborted"
                             transaction="tx_bookHotel"/>
        <tx:transactionState type="tx_base:aborted"
                             transaction="tx_bookCar"/>
      </tx:concatenation>
    </tx:from>
    <tx:to>
      <tx:txPrimitive type="tx_base:compensate" to="tx_bookFlight"/>
    </tx:to>
  </tx:dependency>

  </xpdl:ExtendedAttribute></xpdl:ExtendedAttributes>
</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` primitive 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 is done concurrently[3], 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 activ-

[3]We separated flight booking logic and local services logic to enhance readability. `book_local_services` is a dummy activity that emphasizes this segmentation. Considering functionality, concurrent booking of the three services without this segmentation would not differ in any way.

ity that causes such a transaction related action, we present the `commit_tx_flight` activity. All other transaction related activities 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:txPrimitive type="tx_base:commit" to="tx_bookFlight"/>
    </xpdl:ExtendedAttribute></xpdl:ExtendedAttributes>
</xpdl:Activity>
```

In case a transaction is aborted, suitable dependencies are 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.

7 Conclusion and Next Steps

We have introduced TWSO, an approach for transaction–oriented processing in Web service environments. Recent Web service transaction proposals mainly define communication protocols and infrastructure for Web service transactions. TWSO focuses on upgrading the ubiquitous usage pattern for ACID transactions (i.e. "begin transaction $\rightarrow$ do business logic $\rightarrow$ commit/abort transaction") for Web service environments. It is intended to be used and easily integrateable in Web service orchestrations. Virtually any ATM can be synthesized using TWSO without any need for software updates. TWSO is founded on ACTA, a formal advanced transaction meta model. TWSO offers a comprehensive set of transaction primitives and the possibility to define inter–transaction dependencies to satisfy the demands of Web service transaction systems. We have 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. 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. To execute such an orchestration, we will implement a TWSO transaction monitor component.

References

[1] A. Biliris et al. ASSET: A system for supporting extended transactions. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 44–54, Minneapolis, Minnesota, 1994.

[2] D. Bunting et al. Web services composite application framework, 2003.

[3] L. F. Carbrera et al. Web services coordination, web services business activity framework, web services atomic transaction, 2004.

[4] A. Ceponkus et al. Business transaction protocol, June 2002.

[5] P. K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.

[6] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, October 1997.

[7] A. Fekete et al. Transactions in loosely coupled distributed systems. In *Proceedings of the Fourteenth Australasian Database Conference on Database technologies*, Adelaide, Australia, 2003. Australian Computer Society, Inc.

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, California, 9th edition, 2002.

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

[10] P. Hrastnik and W. Winiwarter. An advanced transaction meta–model for web services environments. In *The Sixth International Conference on Information Integration and Web–based Applications & Services (iiWAS2004)*, pages 303–312, Jakarta, Indonesia, September 2004. Austrian Computer Society.

[11] M. Potts et al. *Business Transaction Protocol Primer*. 2002. cited on 2005-01-28.

[12] M. Prochazka. *Advanced Transactions in Component-Based Software Architectures*. PhD thesis, Charles University, Faculty of Mathematics and Physics, Department of Software Engineering, Malostranske namest i 25, 118 00 Prague 1, Czech Republic, 2002.

[13] G. Weikum and H. J. Schek. Multi-level transactions and open nested transactions. In *Data Engineering*, volume 14, pages 60–64, Los Alamitos, California, March 1991. IEEE Computer Society Press.