Using Advanced Transaction Meta-Models for Creating Transaction-Aware Web Service Environments

Peter Hrastnik
Electronic Commerce Competence Center – EC3, Donau City Straße 1, A-1220 Vienna, Austria
Werner Winiwarter
Institute of Scientific Computing, University of Vienna, Universitätsstraße 5/9, A-1010 Vienna, Austria

Received: December XX 2004; revised: November XX 2004; accepted: April XX 2005

***Abstract*— Recently, the software industry has published several proposals for transactional processing in the Web service world. Even though most proposals support arbitrary transaction models, there exists no standardized way to describe such models. This paper describes potential impacts and use cases of utilizing advanced transaction meta-models in the Web service world and introduces two suitable meta-models for defining arbitrary advanced transaction models. In order to make these meta-models more usable in Web service environments, they had to be enhanced, and XML representations of the enhanced models had to be developed.**

***Index Terms*— Advanced transaction meta models, advanced transaction models, Web services, Web services orchestrations, Web services transactions**

I. Introduction

The publication of Web service transaction proposals [3][4][5] implies that the software industry has recognized that there is a need for transactional processing in the Web service world. In tightly coupled systems, transactional processing that follows the ACID principles [11] is ubiquitous and works well [12].

However, transactions that follow ACID principles may not be practical in loosely coupled systems, e.g. systems composed of Web services. Let us imagine a holiday booking transaction that is modeled according to the ACID properties. A flight should be booked, a rental car should be provided at the destination airport, and a hotel room should be prepared. If 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, then the seat would have to be hidden from other transactions for 2 days because of the required isolation property. This is probably unacceptable for any airline. 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. A traditional ACID transaction cannot support this, because it would have to stick to the atomicity property: If a single service fails, the whole transaction has to fail. Potts et al [12] 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”.

Advanced Transaction Models (ATM) [8][11] offer appropriate transaction semantics for loosely coupled systems. The Web service transaction industry proposals [3][4][5] use the ideas of ATMs and embed them in a transaction processing architecture that fits well into the Web service world. We call an ATM that is supposed to be used in a Web service transaction system *Web Service ATM (WS-ATM)*.

Different (business) domains require different policies for conducting transactional processing. No out-of-the-box set of WS-ATMs can satisfy all requirements of all domains that want to do transactional Web service processing [14]. Besides, a WS-ATM that is used by a domain may have to be adjusted in the course of time and has to be updated from time to time. Such updates should not affect the whole Web service transaction system. Therefore, a Web service transaction system should support arbitrary WS-ATMs, i.e. it should support arbitrary transaction semantics. Generally speaking, the software industry is aware of that requirement because the Web service transaction system proposals [3] and [4] support the idea of incorporating arbitrary WS-ATMs. However, even though formal meta-models for *general* ATMs exist and have been published in [6] and [11], [3] and [4] simply describe a small number of specific WS-ATMs in an informal style.

In this paper we discuss solutions for arbitrary ATMs in the Web service world (i.e. WS-ATMs) that can be described in a standardized way. In Sect. II we present common advanced transaction models whose ideas can build the base of WS-ATMs. Sect. III describes the advantages of standardized WS-

ATMs. In Sect. IV and V, we provide short introductions of two existing advanced transaction meta-models and introduce necessary modifications and appropriate XML representations. Finally, to give a deeper understanding, Sect. VI shows selected aspects of particular WS-ATMs in XML using both advanced transaction meta-models as a base.

II. Advanced Transaction Models

In the past, many advanced transaction models were presented to overcome the restrictions of ACID style transactions, which are unsuitable for some domains. To provide a short introduction and background, some important advanced transaction models are presented here. We refer to [11] for a detailed discussion of advanced transaction models.

A. Transactions with savepoints

Transactions with savepoints [11] allow organizing a transaction into a sequence of actions that can be rolled back individually.

Let us imagine a database transaction where 100,000 records are updated, and isolation of the whole work is important. Each update is a time-consuming task and takes 1 second. The whole transaction would last for a day. If the very last update fails, the whole transaction is aborted and the work of a day is lost.

Savepoints provide a solution for this scenario and relax the atomicity criterion. During a transaction, one can set savepoints and can rollback to an arbitrary savepoint if something fails. It should be noted that setting a savepoint does not commit the modifications that have been done before the savepoint, i.e. another concurrent user cannot see the modifications done so far.

For the previous example, let us assume that a savepoint is set at every 1000th record, then only 15 minutes of work would have to be repeated in case the last transaction fails.

B. Nested transactions

Nested transactions [7] can be seen as a generalization of savepoints. While transactions with savepoints organize a transaction in a sequence, nested transactions form a hierarchy of actions.

A nested transaction is a tree of transactions. The sub-trees are either nested transactions or flat transactions. Leaf level transactions are always flat transactions. The root is called top-level transaction. 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 if its parent-transaction commits. However, after a commit, the parent-transaction will see the effects of the child-transaction.

C. Multilevel transactions

Multilevel transactions [15] are similar to nested transactions, but allow a complete commit of the results of subtransactions before their parent-transactions commit. The results take effect immediately. However, it must be 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.

D. Sagas

Put simply, a Saga [10] is a chain of transactions. Each transaction in the chain commits when it finishes its work, and provides a compensation transaction. If a transaction t_i fails, t_i can do an abort, and all previous transactions (t_1, ..., t_{i-1}) have to start their compensation transactions.

III. Impact of Using Standardized Meta-Model Based WS-ATM Descriptions

A standardized WS-ATM meta-model and its representation in a machine-readable language (used to describe transaction model instances) could facilitate the management of WS-ATMs in Web service transaction systems mostly in the following special areas: *Comprehension* and the process of obtaining comprehension, development of transaction aware *Web services*, and development of transaction aware *Web service orchestrations*. Figure 1 summarizes potential impacts of standardized WS-ATM descriptions.

These areas are related, e.g. comprehension implicitly influences the development of transaction aware Web services and Web service orchestrations. All three areas affect the speed and the extent of the penetration of new or updated WS-ATMs in a particular domain. People usually tend to adopt new techniques like new WS-ATMs faster if they comprehend them thoroughly and if the techniques are easy to use.

Comprehension and the process of obtaining comprehension could be improved by WS-ATM descriptions. If we assume that all details of a WS-ATM are available in a machine-readable format that conforms to a WS-ATM meta-model, then different transformations of the same model could provide appropriate human-readable views for different users. For example, the Web service developer could need a sophisticated HTML document that describes the model in every detail whereas an SVG diagram of basic model properties would serve decision-makers well. In short, WS-ATM meta-models build the foundation of reusable and standardized documentations of WS-ATMs.

WS-ATM meta-models could also improve the development of transaction enabled Web services. Automatic code generators that create abstract base classes, interfaces, or code-skeletons out of WS-ATMs are feasible with machine-readable WS-ATM descriptions. If a Web service that should work in a particular transaction model (e.g. a special kind of multilevel transaction in the tourism domain as presented in Sect. I) has to be created, the developer could download the specification of

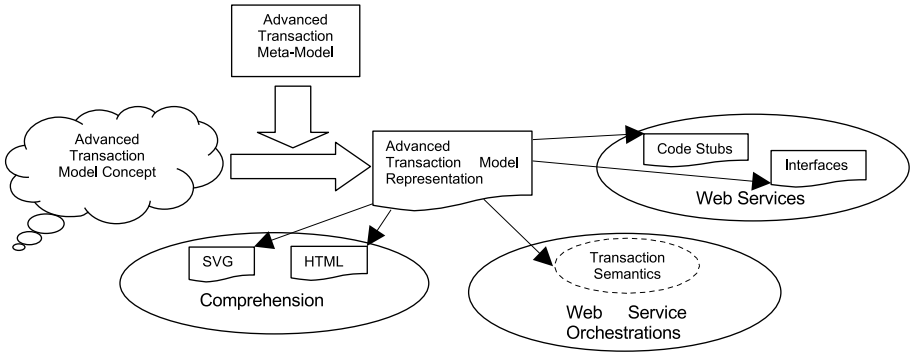

Fig. 1. Impacts of standardized WS-ATM descriptions.

the transaction model and create code fragments out of it. Code fragments clearly show the developer what functionality has to be implemented to support the transaction model. Automatic code generation is a common technique in software development, e.g. Web service clients are generated automatically from WSDL interface descriptions, Corba client stubs from Interface Definition Language descriptions, or GUI code by a graphical GUI builder tool.

The development of transaction-aware Web service orchestrations benefits a lot from advanced transaction meta-models. Conventional ACID transactions are controlled by 3 commands. "begin transaction" sets up a new transaction instance, "commit" will finalize the outcome of all actions that have been executed since the "begin transaction" command, and "abort" will undo ("rollback") the outcome of all actions since "begin transaction". WS-ATMs typically involve several collaborating transactions that have to be set-up and controlled. For example, in nested transactions (see Sect. II-B), the concrete transaction tree structure (e.g. that book-hotel and book-car are subtransactions of book-local-services) has to be defined. Thus, at least in this example, besides "begin", "abort", and "commit", we need methods to specify the inter-transaction relationships. Transaction meta-models can be used to specify the necessary semantics that can be used to set-up and control arbitrary instances of advanced transaction models. If the provided semantics are sufficient, it will be possible to use arbitrary advanced transactions. The actually used (advanced) transaction model is formed by synthesizing it with the provided semantics. Such an approach meets the essence of transaction oriented programming exactly, namely "relieving the application programmer from worrying about failure and concurrency interleaving" [9].

To clarify this aspect, we provide a short example. Let us suppose that we have a set of Web services and we want to use them in a Saga (see Sect. II-D). A fictive advanced transaction meta-model provides the semantics for the inter-transaction dependencies "begin_on_abort" and "begin_on_commit", but does not provide explicit semantics for compensation. Figure 2 depicts the structure and dependencies we have to install. We have regular transactions (tx_n), Web service calls to business logic (ws_{nb}), compensation transactions (ctx_n), and Web service compensation calls (ws_{nc}), for all $n > 0$ and $n < 4$ in this particular example. tx_n manages ws_{nb} and ctx_n manages ws_{nc}. To implement the compensation logic, we have to install the following dependencies for our Saga. For all $n > 1$ and $n < 4$, if tx_n aborts, ctx_{n-1} has to begin (if a transaction aborts, compensation for the preceding transactions compensation transaction has to start). For $n = 2$, if ctx_n commits, ctx_{n-1} has to begin (if a compensation transaction commits, the preceding compensation transaction has to start) and for all $n > 0$ and $n < 3$, if tx_n aborts, tx_n has to begin again (compensation transactions have to be successful, thus, if a compensation transaction aborts, it has to start over again).

It should be noted that it is not necessary for the application programmer to work on this rather low level. Instead, tools can provide a high level interface for frequently used advanced transaction models and translate them to a lower level, where transaction "commands" can be used. For example, the Saga described in Figure 2 could be also defined by declaring: "do a Saga on ws_{1b}, ws_{2b}, and ws_{3b}; respective compensation actions are ws_{1c}, ws_{2c}, and ws_{3c}". This is not covered here but it should be kept in mind when reading Sect. IV and V.

IV. Gray and Reuter's Advanced Transaction Meta-Model

Simple transaction models can be described with finite state-machines. However, most advanced transaction models do not have a fixed number of states. Thus, finite state-machines are not appropriate, and specialized meta-models for advanced transaction models have to be developed. In this section we give a short overview of an advanced transaction model that was introduced in [11] by Jim Gray and Andreas Reuter.

In Gray & Reuter's model, transactions are modeled as compositions of one or more *Atomic Actions (AA)*. AAs have ports that identify possible signals an AA can receive, and final states that indicate the outcome of an action. For example, a simple AA named "T" can have the ports "abort", "commit", and "begin" and the final states "aborted" and "committed".

In a transaction, the included AAs can also be related. Relations can model the invocation hierarchy of the AAs, e.g. if AA_a commits, AA_b has to commit, too. Transactions impose different rules on relations among AAs and the effects they have on related AAs. For each AA in a transaction model, there can be one or more rules. Each rule represents a state transition an AA can perform. Such rules have two parts. The active part of a rule defines conditions

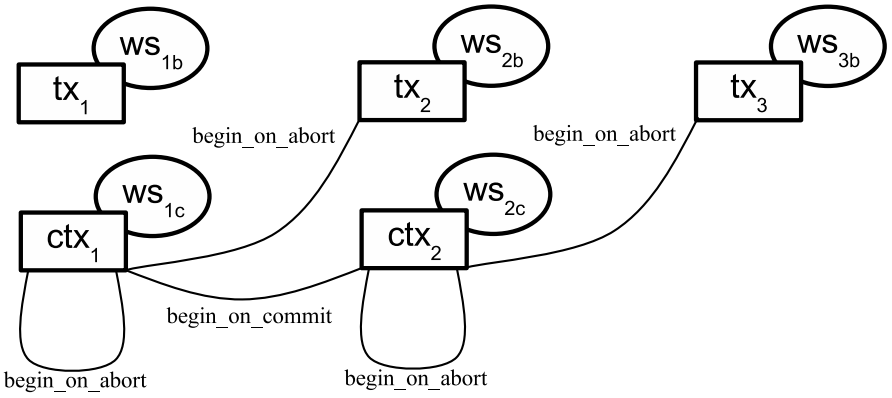

Fig. 2. Structure and dependencies of an example Saga.

that trigger events. For example, "commitment of AA_a triggers commitment of AA_b" is modelled by the active part. These events cause an AA to change its state. The passive part specifies the conditions for performing a state transition. For example, "commitment of AA_a can only happen if AA_x is ready to commit, too" can be defined by the passive part. The structure of a rule can be depicted as follows:

```
<rule identifier>:<preconditions> →
<rule modifier list>, <signal list>, <state
transition>
```

The *rule identifier* indicates the port on a target AA to which a signal should be sent. *Preconditions* are predicates that have to be fulfilled before the corresponding rule is executed.

Rule modifiers capture the dynamic behaviour of a transaction model, i.e. the addition or deletion of rules. The *signal list* contains names of rules that are to be activated in the course of execution of the originating rule. The *rule modifier list* contains one or more rule modifiers. *State transition* is a supplementary element that gives the rule a label. The structure of a rule modifier is the following:

```
<rule modifier> ::=
((+||-) (<rule identifier>|<signal>))
||
(delete (<Atomic Action Identifier>))
```

The first clause of a rule modifier introduces means to dynamically create new rules and dependencies introduced by these new rules. It is also used to delete single rules. The second clause is a shortcut and makes it possible to dynamically delete obsolete rules pertaining to a particular AA.

A transaction model consists of several such rules. Whenever an event occurs, the right side of the rule that identifies the event is executed – of course only if the preconditions of the rule are met. The rule is "marked" to indicate the current state. It remains marked after its execution steps are finished until a new signal comes in. Thus, subsequent emissions of the same signal to the same action are not possible, because the port is "closed" after the first emission. Once an AA reaches a final state, all its rules are deleted. Note that we only write down delete actions if they are essential for the described transaction model.

To illustrate Gray & Reuter's model, we will define a simple transaction model: flat transactions. In flat transactions we have two AAs: The flat transaction action itself and a system action. The "System" action can only be aborted, i.e. the system crashes for some reason. The flat transaction action can be committed and aborted. There is a dependency between the system action and the flat transaction action: If the system action aborts, the flat transaction action has to abort, too. The graphical rendering of the model depicted in Figure 3 describes a particular state of the flat transaction model. The figure would become too complex if we tried to describe the whole model with it. Textual rules are a better way to do that. Each AA has three ports (**A**bort, **B**egin, and **C**ommit) and two states (**A**borted and **C**ommited). Transaction "T" is running, i.e. the begin port has been used.

Shaded ports in the figure cannot be used. Thus, the only ports that can be used in the current state are the abort port of AA "System" and the abort and commit port of action "T". If the system gets into the state "aborted", the abort port of the "T" action is signaled. This emitted signal implies an abort of "T" and, consecutively, a rollback of "T". It should be noted that we expect a transaction that is aborted to perform a rollback. The "textual rules rendering" of the model is as follows:

```
S_A(System): , , System Crash
(rule 1)

S_B(T): +(S_A(system)|S_A(T)), , BEGIN WORK
(rule 2)

S_A(T): (delete(S_B(T)),delete(S_C((T))), , ROLLBACK
WORK
(rule 3)

S_C(T): (delete(S_B(T)),delete(S_A((T))), , COMMIT WORK
(rule 4)
```

The notation $S_X(J)$ means signal X of AA "J". The signals are abbreviated as follows: "A" is short for abort/rollback, "B" means begin, and "C" means commit. The first rule handles

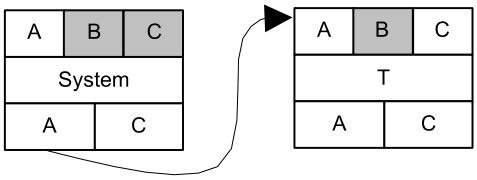

Fig. 3. A single aspect of a flat transaction system.

the case of a system crash: The system action is aborted. Since it does nothing, it is actually redundant. It is only given for the sake of clarity. The second rule installs the structural dependency of the AA "T" and the AA "System", i.e. the arrow in Figure 3. The third rule is executed when an abort signal arrives. All ports are deactivated. The same is true in the case of a commit signal, which is written down in Rule 4.

Nested transactions (see Sect. II-B) can be described with the following rules:

$S_B(T_{kn})$: +($S_A(T_k)$|$S_A(T_{kn})$), ,BEGIN WORK (rule 1)
$S_A(T_{kn})$: , , ROLLBACK WORK (rule 2)
$S_C(T_{kn})$: C(T_k), , COMMIT WORK (rule 3)

Rule 1 introduces a new AA and installs the dependency "if the parent AA aborts, the child AA has to abort, too". Rule 2 establishes the abort signal and Rule 3 manages the commit system: "The child can finally commit only if its parent has committed". The rules use two AA identifiers: T_k and T_{kn}. T_k represents an arbitrary parent AA and T_{kn} an arbitrary child AA of T_k.

Gray & Reuter's model seems to be promising for some of our purposes. As shown in [11], it is able to express the semantics of many well-known ATMs, including those that are also found in Web service transaction framework proposals like nested transactions and multilevel transactions. Thus, chances are good that it can cover a significant majority of all needed ATMs. The model is concise and not hard to understand, which can turn out to be a positive factor regarding acceptance in the Web service world.

A. An XML serialization of Gray & Reuter's advanced transaction meta-model

In order to use Gray & Reuter's model as stated in Sect. III, ATMs have to be described in a machine-readable language that conforms to the meta-model. Besides the claim that the language has to be machine-readable, it would be helpful if humans can read it as well. While the textual rules introduced above would be tolerable regarding these claims, XML is an excellent choice as well. It is verbose enough to provide information for humans, and countless tools and libraries exist that ease the processing of XML. In addition, every standard in the Web service world is using XML, therefore, representing the model in any other way would be unreasonable. Thus, an XML representation of Gray & Reuter's model seems to be the best choice. In the next subsections an XML serialization of a transaction meta-model that is based on Gray & Reuter's ideas is introduced. To be concise, we focus on significant parts of the model. Nevertheless, an XML schema that defines the entire meta-model was developed as well.

A straightforward approach to bring Gray & Reuter's model into the XML world is to map the rules in a one-to-one way. The "begin work" rule of a flat transaction as shown before would look something like this in XML:

```
<rule stateTransition="begin work">
  <identifier>
    <signal type="B">
      <atomicAction type="T"/>
    </signal>
  </identifier>

  <ruleModifiers>
    <add>
      <from>
        <signal type="A">
          <atomicAction type="system"/>
        </signal>
      </from>
      <to>
        <signal type="A">
          <atomicAction type="T"/>
        </signal>
      </to>
    </add>
  </ruleModifiers>
</rule>
```

For such a simple model, this one-to-one mapping seems to be appropriate. However, for more complex transaction models, this kind of mapping has deficits. For example, take a look at the one-to-one mapped commit rule of a nested transaction:

```
<rule stateTransition="COMMIT WORK">
  <identifier>
    <signal type="C">
      <atomicAction type="T" id="kn"/>
    </signal>
  </identifier>

  <precondition>
    <state type="C">
      <atomicAction type="T" id="k"/>
    </state>
  </precondition>
</rule>
```

A human specialist could imagine that the identifier "kn" describes an arbitrary child AA and "k" its parent AA. The precondition for committing the child AA (i.e. the parent transaction has to be in the committed state) is not machine-readable, because the hierarchic relation between "T_k" and "T_{kn}" is not expressed in a machine-readable way. A similar problem arises when mapping flat transactions with savepoints (see Sect. II-A) to XML in a one-to-one way. The abort rule and its one-to-one XML serialization for an arbitrary AA in a flat transaction with savepoints are as follows (let R be the target savepoint, i.e. the transaction should rollback to the AA identified by R):

SA(R) : (R<S_n) → , SA(S_{n-1}), ROLLBACK WORK

```
<rule stateTransition="ROLLBACK WORK">
  <identifier>
    <signal type="A">
      <atomicAction type="S" id="n"/>
    </signal>

    <arguments>
      <arg name="RollbackTargetSavepointAA">
        <constraint> RollbackTargetSavepointAA < Sn
        </constraint>
      </arg>
    </arguments>
  </identifier>

  <signalList>
    <emitSignal>
      <target>
        <signal name="C">
          <atomicAction type="S" id="n-1"/>
        </signal>
      </target>
    </emitSignal>
  </signalList>
</rule>
```

Here we have an argument that defines the identifier more precisely, and a constraint, which describes the allowed values of the argument. The rule and consequently the one-to-one mapping defines this in a language that cannot be understood by a machine without difficulty. Thus, a comprehensive XML mapping should include machine-readable parameter-passing semantics, too. Another problem arises with the signal list. A human can interpret the target of the signal: the linear predecessor AA. Similar to the parent-child relationship problem above, the linear relationship is not expressed explicitly.

Hence we face two obvious key problems when translating Gray & Reuter's rules to XML in a straightforward one-to-one way: We need an explicit definition of relationship types (e.g. previous, parent, etc.) and some kind of parameter passing semantics.

1) Explicit relationship declarations: One has to consider that arbitrary transaction models can have arbitrary relation types between their AAs. While a set of basic relation types can be identified, an extension mechanism is required, too. The basic set of relation types can be separated into *linear* and *hierarchic* types. Linear types are "first", "next", "previous", and "last". The hierarchic types are "parent", "child", and "root". Another special type (see Sect. VI-A) is also needed: "self" for relations to the atomic action itself. Note that the basic set just supports transaction models that follow a linear or hierarchic structure. The names are self-describing and these basic types should be sufficient for quite a few transaction models – at least the set is sufficient for all ATMs presented in [11]. The commit rule of a nested transaction in XML looks like this:

```
<rule stateTransition="COMMIT WORK"
      xmlns:aaRelations="http://wstx.ec3.at/AA_relations">
  <identifier>
    <signal type="C">
      <atomicAction type="T" id="kn"/>
    </signal>
  </identifier>
  <precondition>
    <state type="C">
      <atomicAction type="T" id="k">
        <aaRelations:relationSpecification relatedTo="this"
          as="parent" />
      </atomicAction>
    </state>
  </precondition>
</rule>
```

As can be seen, the embedding of relation specifications is implemented with XML-namespaces. This provides a flexible extension mechanism for AA relation types. The "this" value in the relatedTo attribute represents the current rule. Of course, the semantics of a parent type in the "http://wstx.ec3.at/AA_relations" namespace has to be implemented in the processing software in order to do some "parent relation aware" processing. If this is the case, the processing software is aware that the parent has to commit first. Extension sets reside in other namespaces and are included by declaring their namespace and prefixing the corresponding relation element with the namespace shortcut. Again, the semantics of extension AA relation types has to be implemented in the processing software – at least if it is desired to process these new AA relation types appropriately.

2) Parameter passing: Since the input parameters used in the ATM rules presented in [11] are only AA types, we concentrate on building a parameter passing system that considers just AA types for now. We need to know the allowed type of the AA that can be passed as well as constraints the passed AA instance has to respect.

For the constraints, we use a similar technique as in Sect. IV-A.1: It is sufficient that constraints are expressed in terms of relations to other AAs. Thus, we enhance our relation vocabulary with "linearAncestor" (any previous AA), "linearSuccessor" (any subsequent AA), "treeAncestor" (on a higher tree-level), and "treeSuccessor" (on a lower tree-level). For instance, argument passing in the rollback rule of a transaction with savepoints is specified as follows:

```
<rule stateTransition="ROLLBACK WORK"
      xmlns:aaRelations="http://wstx.ec3.at/AA_relations">
...
  <identifier>
    <signal name="A">
      <atomicAction type="S" id="n"/>
      <arguments>
        <arg type="S">
          <aaRelations:relationSpecification
              relatedTo="this" as="linearAncestor" />
        </arg>
      </arguments>
    </signal>
  </identifier>
...
</rule>
```

3) Supplementary AA type declarations: In [11], there is no explicit definition, which states an atomic action can have and which signals it can accept. This is done implicitly by defining rules accordingly, i.e. if a rule is identified by signal "S" to AA "T", it is assumed that "T" has the port "S". Though it is redundant, an explicit definition of AA types used in the model should be added to enhance readability and to ease processing by software programs. State transitions are also defined implicitly in [11], i.e. if signal "C" arrives at AA "T", "T" gets into the "C" state. This should be stated explicitly in the XML representation as well. A nested transaction AA can be represented in XML as follows:

```
<signalType name="A"/>
<signalType name="B"/>
<signalType name="C"/>
```

```
<stateType name="A"/>
<stateType name="C"/>

<atomicActionType name="T">
  <signals>
    <signal type="A"/>
    <signal type="B"/>
    <signal type="C"/>
  </signals>

  <states>
    <state type="A"/>
    <state type="C"/>
  </states>

  <transitions>
    <transition>
      <fromSignal type="A"/><toState type="A"/>
    </transition>
    <transition>
      <fromSignal type="C"/><toState type="C"/>
    </transition>
  </transitions>
</atomicActionType>
```

V. ACTA

ACTA is a framework that can be used to specify, analyze, and synthesize advanced transaction models [6]. As in Gray and Reuter's model, several transactions are combined to compose advanced transactions models in ACTA. Basically, ACTA distinguishes between *object events* and *significant events*. Object events are calls on operations of objects. Significant events are invocations of transaction management primitives like commit, abort etc. Besides that, ACTA uses the following building blocks to describe an advanced transaction model.

Inter-transaction dependencies are used to specify the relationships between transactions in a transaction model. For example, an abort dependency between transaction t_j and transaction t_i indicates that the abort of t_i causes the abort of t_j. *Views of transactions* allow specifying the state of objects visible to a transaction at a point of time. For example let us assume that under the control of transaction t_a an object event on object o_v has changed the state of o_v and t_a has not committed yet. Transaction views control whether it is possible for object events under a transaction t_b to operate on and/or see the not-yet-committed state of o_v. With *conflict sets* it is possible to define that object events under control of a transaction cannot be called by another transaction while the object events are in-progress (in-progress object events are events that have been started but have not been committed or aborted yet). *Delegations* move the responsibility for object events from one transaction to another and also delegate the responsibility for significant events from one transaction to another. Delegation of object o_d from transaction t_y to transaction t_x means that all method calls to o_d that happened before the delegation (i.e. t_y was de facto in control) are considered to have been happened under t_x and t_x is responsible for doing significant events (e.g. "commit") on o_d.

In [6], example ATMs are described using axioms that are expressed in predicate logic, and this predicate logic uses the building blocks of ACTA. The following axioms describe a simple atomic transaction:

1) $SE_t = \{\text{Begin, Commit, Abort}\}$
2) $IE_t = \{\text{Begin}\}$
3) $TE_t = \{\text{Commit, Abort}\}$
4) t satisfies the fundamental Axioms of Transactions
5) $\mathit{View}_t = H_{ct}$
6) $\mathit{ConflictSet}_t = \{p_{t'}[ob] \mid t' \neq t, \mathcal{I}nprogress(p_{t'}[ob])\}$
7) $\forall ob\ \exists p\ (p_t[ob] \in H) \Rightarrow (ob\ is\ atomic)$
8) $(\text{Commit}_t \in H) \Rightarrow \neg(tC^*t)$
9) $\exists ob\ \exists p\ (\text{Commit}_t[p_t[ob]] \in H) \Rightarrow (\text{Commit}_t \in H)$
10) $(\text{Commit}_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob]) \in H) \Rightarrow (Commit_t[p_t[ob]] \in H))$
11) $\exists ob\ \exists p\ (Abort_t[p_t[ob]] \in H) \Rightarrow (\text{Abort}_t \in H)$
12) $(Abort_t \in H) \Rightarrow \forall ob\ \forall p\ ((p_t[ob] \in H) \Rightarrow (Abort_t[p_t[ob]] \in H))$

Axioms 1–3 define events and their purpose: Significant events (begin, commit, abort), a single initiation event (begin), and two termination events (commit, abort). Axiom 4 refers to basic transaction axioms, i.e. transactions can only be initiated by a single event and can only be terminated by a single event, termination can only occur if a transaction has been initiated before, and only running transactions can invoke operations on objects. Axioms 5–12 define the core semantics of atomic transactions. We will not describe these axioms in detail here. See [6] for a complete explanation of axioms 5–12.

The ACTA framework is a very comprehensive meta-model and it is unlikely that a particular idea for a custom advanced transaction model cannot be represented in ACTA. However, as one can see from the preceding example, this completeness causes complexity, and ACTA itself is neither easy to understand nor easy to use. Moreover, it is questionable that a one-to-one XML representation of such axioms is useful.

Specialized approaches for defining advanced transaction models that use ACTA have been proposed. For example, ASSET [2] and Bourgogne [13] transactions 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. These transaction primitives can be used in arbitrary programming languages. Thus, an advanced transaction model is defined by creating a small program that uses such transaction primitives. Note that such a description of a particular transaction model is not universal. Instead, it is a description of a particular instance of a transaction model. This approach is perfectly qualified for the development of transaction aware Web services or transaction aware Web service orchestrations as described in Sect. III.

A. XML representation

We decided not to develop an XML representation of the ACTA framework because the resulting complexity does not promise wide acceptance. Instead, enhancing Web service orchestrations with arbitrary advanced transaction models in a way that is inspired by ACTA and/or ASSET/Bourgogne transactions seems to be a valuable aim. Web service orchestrations tie together a set of existing Web services in order to create completely new services by employing workflow technologies.

In this section we present such an approach. It should be noted that we do not include views and conflict sets. As also stated in [5], Web service transactions take place in an

inter-organizational network, and the possibility to influence a foreign organization's Web service is not provided. Thus, we have to assume that the Web services that take part in a transaction act correctly regarding concurrent access, i.e. views and conflict sets in terms of ACTA. To give a basic understanding of our approach, we will present significant parts of selected examples.

To represent transaction primitives, we use XML elements. These XML elements reside in a special namespace so that they can be incorporated in various XML Web service orchestration languages like BPEL4WS [1] or XPDL [16]. Here, we will embed the transaction primitive elements in XPDL fragments, i.e. XPDL elements that are irrelevant for the particular issue are omitted. XPDL is a Web service orchestration language that conforms to a graph-oriented workflow paradigm. A graph-oriented workflow has activities and transitions. Activities represent workitems like Web service calls. Transitions link activities and model the execution sequence, i.e. they model the control flow.

An atomic transaction instance can be specified as follows:

```
<Activities>
  <Activity id="start"/>
  <Activity id="setup_tx">
    <tx:initiate id="atom">
      <tx:activityref id="bookFlight"/>
    </tx:initiate>
  </Activity>
  <Activity id="begin_atom">
    <tx:significantEvent type="tx_base:begin" tx="atom"/>
  </Activity>
  <Activity id="bookFlight"/>
  <Activity id="commit_atom">
    <tx:significantEvent type="tx_base:commit" tx="atom"/>
  </Activity>
  <Activity id="abort_atom">
    <tx:significantEvent type="tx_base:abort" tx="atom"/>
  </Activity>
  <Activity id="end"/>
</Activities>

<Transitions>
  <Transition from="start" to="setup_tx"/>
  <Transition from="setup_tx" to="bookFlight"/>
  <Transition from="bookFlight" to="commit_atom"/>
  <Transition from="bookFlight" to="abort_atom">
    <Condition Type="DEFAULTEXCEPTION"/>
  </Transition>
  <Transition from="abort_atom" to="end"/>
  <Transition from="commit_atom" to="end"/>
</Transitions>
```

In this orchestration, we have activities that do transaction related tasks. Before "bookFlight" is executed, the transaction "atom" is initiated, and the activity that should be controlled by "atom" is specified. After the initiation, "atom" is started with the significant event "begin". Then "bookFlight" is executed under control of the transaction "atom". If no exception occurs, "commit_atom" issues a "commit" significant event. Otherwise – when an exception occurs in "bookFlight" – "abort_atom" is executed. "abort_atom" invokes an "abort" significant event. It should be noted that transaction related elements reside in a special namespace that is denoted by the prefix "tx".

The type of a significant event is also defined using namespaces. The basic set of significant events includes "begin", "commit", "abort", and "compensate". Compensation is a "forward" action that makes some adjustments to reverse the original action. After compensation, the fact that the action took place is visible. In contrast, an abort undoes an action so that it seems like the action never took place. We decided to add "compensation" to the basic set of significant events because it will be used frequently in Web service environments. Depending on the state of a transaction at a point in time, only particular significant events make sense. Table I shows possible state transitions and effects of issued significant events on the transaction.

The basic set of significant events should be sufficient for most advanced transaction models in Web service environments. However, if necessary, more significant events from other significant event namespaces can be introduced. Of course, all affected components of the transaction management system have to be able to handle these new significant events.

For advanced transaction models like nested transactions, the delegation concept of ACTA and dependencies between transactions are necessary. Let us assume that we want to synthesize a nested transaction with a root transaction "bookJourney" and two child transactions "bookFlight" and "bookHotel" (also called "booking transactions" here). This can be accomplished as depicted in Figure 4. "bookJourney" is just a routing activity and does actually nothing. In contrast, "bookFlight" and "bookHotel" – which are executed concurrently – call a corresponding Web service. For the sake of clarity, Figure 4 does not take trivial exception handling and resulting abort operations into account.

First, we have to install dependencies between the root transaction and the booking transactions so that, if the root transaction aborts, the booking transactions abort, too. Thus, the time before the booking transactions delegate commit and abort responsibilities to the root transaction is addressed.

```
<Activity id="setup_tx">
  <tx:initiate id="tx_bookJourney">
    <tx:activityref id="bookJourney"/>
  </tx:initiate>

  <tx:initiate id="tx_bookHotel">
    <tx:activityref id="bookHotel"/>
  </tx:initiate>

  <tx:initiate id="tx_bookFlight">
    <tx:activityref id="bookFlight"/>
  </tx:initiate>

  <tx:dependency id="nested_aborts">
    <tx:from>
      <tx:significantEvent type="tx_base:abort"
                           tx="tx_bookJourney"/>
    </tx:from>
    <tx:to>
      <tx:significantEvent type="tx_base:abort"
                           tx="tx_bookHotel"/>
      <tx:significantEvent type="tx_base:abort"
        tx="tx_bookFlight"/>
    </tx:to>
  </tx:dependency>
</Activity>
```

Dependencies between transactions are defined using relationships of significant events. For example, a dependency could be "if significant event 'abort' occurs on transaction t_i or t_j, issue significant event 'compensate' on transaction t_k, t_l, and t_m". The XML representation of a dependency arranges significant events as follows. The "from" group defines the significant events that have to occur in order to trigger the dependency. If there are more significant events in the "from" group, they have to be grouped with "and" or "or" semantics. If they are grouped via "and", the dependency is executed

TABLE I
SIGNIFICANT EVENTS AND TRANSACTION STATES

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

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

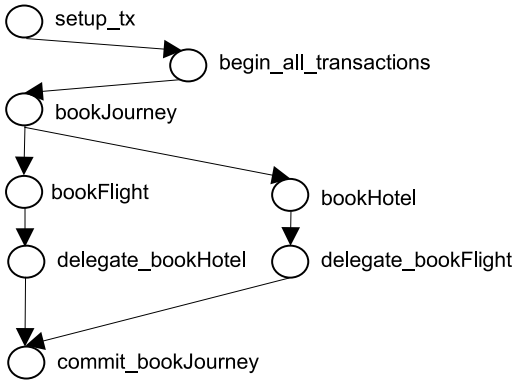

Fig. 4. Nested transaction synthesized using XPDL.

only if all significant events in the group occur. If “or” is used, a single significant event that occurs is sufficient to trigger the dependency. See Sect. VI-B for details of grouping more “from” significant events. The “to” group includes the significant events that should be issued.

The activity “begin_all_transactions” starts all involved transactions, i.e. “tx_bookJourney”, “tx_bookFlight”, and “tx_bookHotel”. After that, “bookJourney” (an activity that has no implementation and is used for routing the concurrency of the booking transactions) is called, and subsequently “bookHotel” and “bookFlight” are executed concurrently.

The final commit of the particular booking transaction has to be issued by the root transaction. Thus, we need to delegate the commit and abort responsibility from the booking transactions to the root transaction at the moment a child transaction is ready to commit:

```
<Activity id="delegate_bookFlight">
  <tx:delegate from="tx_bookFlight" to="tx_bookJourney"/>
</Activity>

<Activity id="delegate_bookHotel">
  <tx:delegate from="tx_bookHotel" to="tx_bookJourney"/>
</Activity>
```

VI. MULTILEVEL TRANSACTIONS IN WS-ATMS

A. Multilevel transactions inspired by Gray & Reuter’s metamodel

To clarify the XML representation in Sect. IV, we present selected aspects of a multilevel transaction model (see Sect. II-C) XML description. We have chosen this particular ATM because its ideas are used in the industrial Web service transaction proposals, i.e. in WS-Transactions [4], WS-CAF [3], and BTP [5]. Besides the “system” transaction (see Sect. IV) and a transaction “T”, which has the usual signal ports (“abort”, “begin”, and “commit”) and states (“aborted” and “committed”), multilevel transactions make use of compensation actions. Hence we have to introduce a corresponding atomic action type:

```
<atomicActionType name="Compensation">
  <signals>
    <signal type="begin"/>
    <signal type="commit"/>
  </signals>

  <states>
    <state type="committed"/>
  </states>

  <transitions>
    <fromSignal type="commit"/>
    <toState type="committed"/>
  </transitions>
</atomicActionType>
```

Compensations are not aborted, so we have no abort port and no aborted state. In contrast to nested transactions, a child-transaction can commit in multilevel transactions before its parent transaction commits. If this happens, a compensation action has to be provided. This is the corresponding rule modifier:

```
<rule stateTransition="COMMIT WORK"
      xmlns:aaRelations="http://wstx.ec3.at/AA_relations">
...
  <ruleModifiers>
    <add>
      <from>
        <signal type="abort">
          <atomicAction type="T">
            <aaRelations:relationSpecification
              relatedTo="this" as="parent"/>
          </atomicAction>
        </signal>
      </from>

      <to>
        <signal type="begin">
          <atomicAction type="compensation"/>
        </signal>
      </to>
    </add>

    <delete>
      <atomicAction type="T">
        <aaRelations:relationSpecification self="true"/>
      </atomicAction>
    </delete>

  </ruleModifiers>
...
</rule>
```

Whenever a child-transaction commits, a new compensation action is installed and all rules pertaining to the current atomic action are removed. The compensation action is connected to the abort state of the parent transaction. If the parent transaction has to abort, the compensation action is started.

B. Multilevel transactions inspired by ACTA

Let us suppose that we want to synthesize a transaction based on the example described in Sect. I. Thus, we want to book a trip that consists of booking a flight (“bookFlight”), a hotel (“bookHotel”), and a rental car (“bookCar”). If “bookFlight” fails, everything else should fail, too. If “bookFlight” is successful and “bookCar” and/or “bookHotel” are successful, the transaction should succeed.

This situation can be modeled using the ideas of multilevel transactions. Significant parts of the corresponding orchestration are illustrated in Figure 5. The black bars in Figure 5 represent synchronization activities. Synchronization activities wait until all previous concurrently executed paths are finished. Thus, in our orchestration the execution stops on 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.

First, we have to setup the transactions we want to use in the orchestration. This is done as follows:

```
<Activity id="setup_transactions">
  <tx:initiate id="tx_bookFlight">
    <tx:activityref id="bookFlight"/>
  </tx:initiate>
  <tx:initiate id="tx_bookHotel">
    <tx:activityref id="bookHotel"/>
  </tx:initiate>
  <tx:initiate id="tx_bookCar">
    <tx:activityref id="bookCar"/>
  </tx:initiate>

  <tx:dependency id="flight_aborts">
    <tx:from>
      <tx:significantEvent type="tx_base:abort"
                           tx="tx_bookFlight"/>
    </tx:from>
    <tx:to>
      <tx:significantEvent type="tx_base:compensate"
                           tx="tx_bookHotel"/>
      <tx:significantEvent type="tx_base:compensate"
                           tx="tx_bookCar"/>
    </tx:to>
  </tx:dependency>
  <tx:dependency id="localServices_abort">
    <tx:from>
      <tx:concatenation type="and">
        <tx:significantEvent type="tx_base:abort"
                             tx="tx_bookHotel"/>
        <tx:significantEvent type="tx_base:abort"
                             tx="tx_bookCar"/>
      </tx:concatenation>
    </tx:from>
    <tx:to>
      <tx:significantEvent type="tx_base:compensate"
        tx="tx_bookFlight"/>
    </tx:to>
  </tx:dependency>
</Activity>
```

Here, three transactions and their dependencies are specified. The “flight_aborts” dependency causes the compensation of “tx_bookHotel” and “tx_bookCar”, if “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.

After specifying the transactions, 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, whereas the car booking and hotel booking is done concurrently, too. If an exception happens while doing the particular bookings, the corresponding transaction is aborted, otherwise it is committed. If a transaction aborts, the defined dependencies may be applied. For example, if the flight booking is aborted, the hotel booking and the car booking will be compensated.

VII. Conclusion and Future Work

In this paper we have introduced two meta-models for advanced transaction models and appropriate XML representations for their model instances. We used Gray & Reuter’s meta-model [11] and ACTA [6].

A one-to-one XML mapping of Gray & Reuter’s meta-model is not favorable, therefore we enhanced this model with a parameter passing system, an extendable mechanism to explicitly express relationships between the components of a transaction, and supplementary transaction component type declarations. We have presented an XML representation of this enhanced meta-model by showing significant sections of well-known ATMs (flat transactions, nested transactions, transactions with savepoints, and multilevel transactions) in XML. Gray & Reuter’s meta-model is appropriate to describe the structure and basic ideas of advanced transaction meta-models. It can be especially useful to comprehend advanced transaction models in Web service systems. However, it is not well suited to facilitate developing transaction aware Web services or transaction aware Web service orchestrations directly.

We refrained from developing an exact XML representation of the ACTA framework in favor of incorporating the ideas of ACTA in Web service orchestrations. We have presented a way to synthesize advanced transaction model instances in the Web service orchestration language XPDL by integrating the ACTA concepts of transaction interdependencies, delegation, and significant events. The ACTA framework seems to be well suited as a basis for the development of transaction aware Web services and transaction aware Web service orchestrations. To gain comprehension of a particular advanced transaction model it can be helpful, but we clearly focused on the facilitation of developing applications that use arbitrary advanced transaction models.

In the next step, we will develop various transformations of XML transaction models based on Gray and Reuter’s model, e.g. to SVG diagrams and HTML documents. SVG seems to be a valuable target because its interactive capabilities provide means for intuitively demonstrating an advanced transaction model’s capabilities.

Furthermore, future work will focus on transaction aware Web service orchestrations. We will implement a system for executing Web service orchestrations that employs transaction primitives to synthesize arbitrary advanced transaction models. This can involve either the incorporation of native transaction primitive support into an existing Web service orchestration engine or the translation of orchestrations enriched with transaction primitives to pure standard orchestration languages, like clean XPDL. To execute such an orchestration, we will

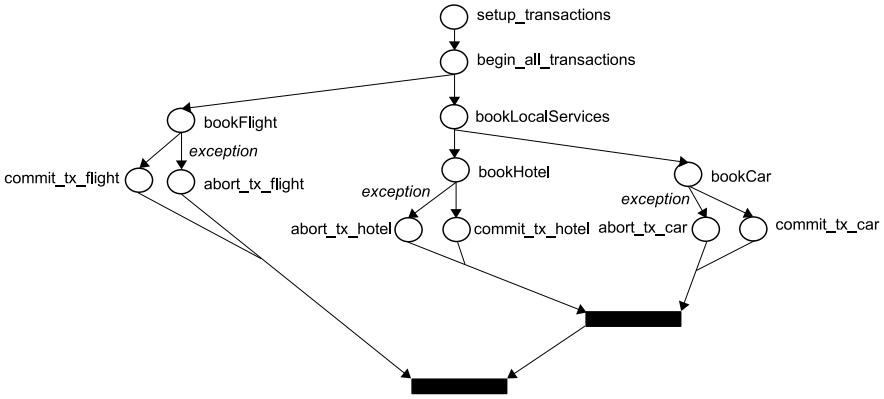

Fig. 5. Multilevel transaction instance.

implement a transaction monitor component, which is able to act on the transaction primitives of an orchestration.

REFERENCES

[1] Andrews T et al. Business Process Execution Language for Web Services Version 1.1. May 2003. Available at: URL http://www.ibm.com/developerworks/library/specification/ws-bpel/. Accessed April 21, 2005.

[2] Biliris A et al. ASSET: A System for Supporting Extended Transactions. In: Proceedings of the 1994 ACM-SIGMOD International Conference on Management of Data. New York, NY, USA: ACM Press; 1994. p. 44-54.

[3] Bunting D et al. Web Services Composite Application Framework. July 2003. Available at: URL http://developers.sun.com/techtopics/webservices/wscaf/primer.pdf. Accessed April 21, 2005.

[4] Cabrera F et al. Web Services Transaction (WS-Transaction). November 2004. URL http://www.ibm.com/developerworks/library/specification/ws-tx/. Accessed April 21, 2005.

[5] Ceponkus A et al. Business Transaction Protocol. June 2002. Available at URL http://www.oasis-open.org/committees/download.php/1184/2002-06-03.BTP_cttee_spec_1.0.pdf. Accessed April 21, 2005.

[6] Chrysanthis P K and Ramamtitham K. Acta: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In: Proceedings of the 1990 ACM-SIGMOD International Conference on Management of Data. New York, NY, USA: ACM Press; 1990. p. 194-203.

[7] Eliot J and Moss B. Nested Transactions: An Approach to Reliable Distributed Computing. Cambridge, MA, USA: MIT Press; 1985.

[8] Elmagarmid A editor. Database Transaction Models for Advanced Applications. 1st ed. San Mateo, California, USA: Morgan Kaufmann Publishers; 1992.

[9] Fekete A et al. Transactions in Loosely Coupled Distributed Systems. In: Proceedings of the Fourteenth Australasian Database Conference. Adelaide, Australia: Australian Computer Society; 2003. p. 7-12.

[10] Garcia-Molina H and Salem K. Sagas. In: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data. New York, NY, USA: ACM Press; 1987. p. 249-259.

[11] Gray J and Reuter A. Transaction Processing: Concepts and Techniques. 9th ed. San Francisco, California, USA: Morgan Kaufmann Publishers; 2002.

[12] Potts M, Cox B, and Pope B. Business Transaction Protocol Primer. June 2002. Available at URL http://www.oasis-open.org/committees/business-transactions/documents/primer/BTP_Primer_v1.0.20020605.pdf. Accessed April 21, 2005.

[13] Prochazka M. Advanced Transactions in Enterprise JavaBeans. Lecture Notes in Computer Science, Revised Papers from the Second International Workshop on Engineering Distributed Objects 2000; 2000:215-230. London, UK: Springer-Verlag.

[14] Roberts J and Srinivasan K. Tentative Hold Protocol Part 1: White Paper. November 2001. Available at URL http://www.w3.org/TR/tenthold-1/. Accessed April 21, 2005.

[15] Weikum G and Schek H-J. Multi-Level Transactions and Open Nested Transactions. Data Engineering 1991 Mar;14(1):60-4. Los Alamitos, CA, USA: IEEE Computer Society Press.

[16] Workflow Process Definition Interface – XML Process Definition Language. October 2002. Available at URL http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf. Accessed April 21, 2005.

Peter Hrastnik Peter Hrastnik studied Business Informatics at the University of Vienna and the Vienna University of Technology. After receiving his MSc., he was employed by an Austrian telecommunications provider to develop Internet services. Since 2001, he has been at the ec3 and has been concentrating on Web, XML, and distributed systems technologies for the creation of virtual enterprises. In addition, Peter Hrastnik is working on his PhD thesis, which aims at the facilitation of transactional processing in the Web service world.

Werner Winiwarter Prof. Dr. Werner Winiwarter holds a tenured position at the Institute 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. His main research interest is human language technology. In addition, Prof. Winiwarter also works on data mining and machine learning, Semantic Web, information retrieval and filtering, electronic business, digital libraries, and education systems.