# *An Advanced Transaction Meta-Model for Web Services Environments*

Peter Hrastnik

E-Commerce Competence Center – EC3
Donau City Straße 1, A-1220 Vienna, Austria
peter.hrastnik@ec3.at

Werner Winiwarter

Faculty of Computer Science, University of Vienna
Liebiggasse 4/3-4, A-1010 Vienna, Austria
werner.winiwarter@univie.ac.at

***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 a suitable meta-model for defining arbitrary advanced transaction models. In order to make this meta-model more usable in Web service environments, it had to be enhanced and an XML representation of the enhanced model had to be developed.*

## 1. Introduction

The publication of Web service transaction proposals [1][2][3] 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 [7] is ubiquitous and works well [8]. However, transactions that follow ACID principles may not be practical in loosely coupled systems, e.g. systems composed of Web services. [8] discusses this in detail and even asserts 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) [6][7] offer appropriate transaction semantics for loosely coupled systems. The Web service transaction industry proposals [1][2][3] 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 [9]. 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 because the Web service transaction system proposals [1] and [2] support the idea of incorporating arbitrary WS-ATMs. However, even though formal meta-models for *general* ATMs exist and were published in [4] and [7], [1] and [2] simply describe a small number of specific WS-ATMs in an informal style.

In this paper we discuss a solution for arbitrary ATMs in the Web service world (WS-ATMs) that can be described in a standardized way. Sect. 2 describes the advantages of standardized WS-ATMs. In Sect. 3, we provide a short introduction of an existing advanced transaction meta-model. Sect. 4 introduces necessary enhancements of this advanced transaction meta-model and an appropriate XML representation. Finally, to give a deeper understanding, Sect. 5 shows selected aspects of a particular WS-ATM in XML.

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

A standardized WS-ATM meta-model and its representation in a machine-readable language (the language is used to describe transaction model instances) could facilitate working with 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 *administration* of a Web service transaction system. Figure 1 summarizes potential impacts of standardized WS-ATM descriptions and gives examples of applications.

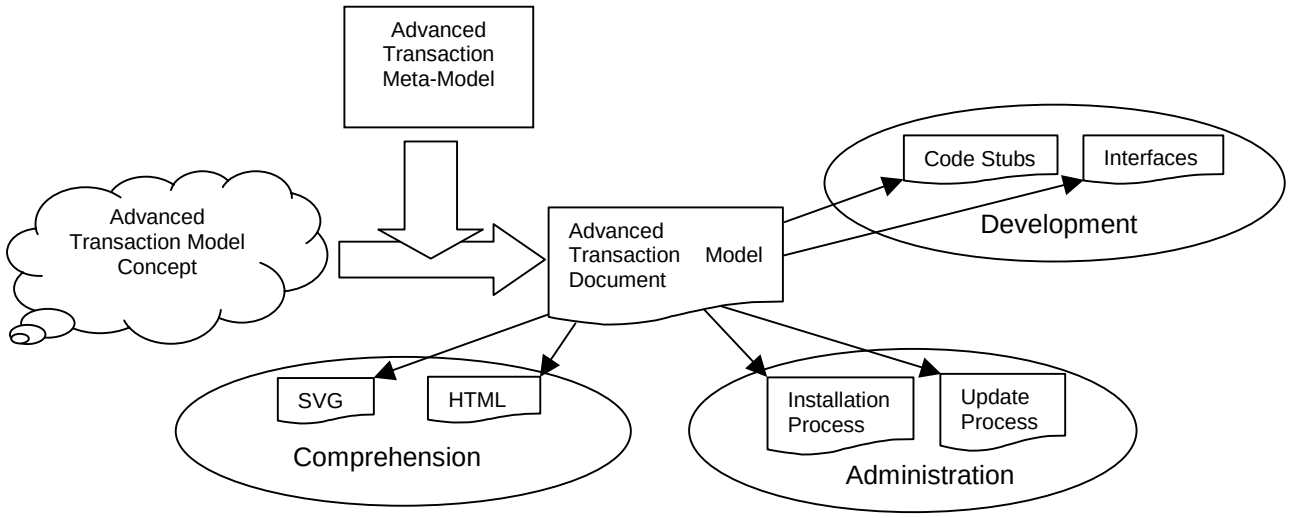


**Figure 1: Impacts of standardized WS-ATM descriptions**

Comprehension, development and administration are related, e.g. comprehension implicitly influences development and administration. 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-ATMs 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 conform to a particular transaction model (e.g. a special kind of multi-level transaction [10] in the tourism domain) has to be created, the developer could download the specification of 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.

Different business domains want to or have to use different WS-ATMs. It is not realistic that a set of "shrink-wrapped" transaction models will be used in all domains or that a WS-ATM will be used forever in a single domain [9]. It is more likely that a WS-ATM changes from time to time to reflect new requirements of the domain. A Web service transaction system has to support WS-ATM variations like the update of existing WS-ATMs or the addition of new WS-ATMs. If the WS-ATMs are described thoroughly in a standardized way, it is imaginable that an existing transaction system can be updated with new WS-ATMs using some kind of easy-to-use installation procedure. At least, a detailed description of a transaction system behaviour that conforms to the new WS-ATM should be possible.

## 3. An Advanced Transaction Meta-Model

Simple transaction models can be described with finite state-machines. However, if it is not possible to identify a fixed number of states of a particular transaction model a priori, finite state-machines are not appropriate. Thus, specialized models for ATMs were developed. In this section, we give a short overview of such a model that was introduced in [7] by Jim Gray and Andreas Reuter.

In Gray & Reuter's model, transactions are modelled 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 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 be aborted only, 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 2 describes a particular state of the flat transaction model. The figure would become burdened 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.

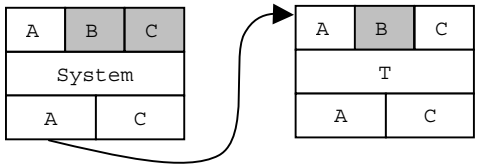


**Figure 2: A single aspect of a flat transaction system**

Shaded ports in the figure cannot be used. Thus, the only ports that can be used in the demonstrated 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 signalled. This emitted signal implies an abort of "T" and consecutively a rollback of "T". Note 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 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 2. The third rule is executed when an abort signal arrives. All ports are deactivated now. The same is true in the case of a commit signal. It is written down in rule 4. Nested transactions (for a detailed discussion of nested transactions see [5]) 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 our purposes. As shown in [7], 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.

## 4. An XML Serialization of Gray & Reuter's Advanced Transaction Meta-Model

In order to use Gray & Reuter's model as stated in Sect. 2, 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 that humans can read it as well. While the textual rules introduced in Sect. 3 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. Representing the model in any other way would just not fit well there. Thus, an XML representation of Gray & Reuter's model seems to be the best choice. In the next sections 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 meta-model solely. Nevertheless, an XML-Schema that defines the complete 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 in Sect. 3 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 computer scientist 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 [7] 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):

$S_A(R) : (R<S_n) \rightarrow , S_A(_{sn-1}), \text{ROLLBACK WORK}$

```
<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.

### 4.1. 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. 5) 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 [7]. The commit rule of a nested transaction in XML looks like this:

```
<rule stateTransition="COMMIT WORK"
        xmlns:aaRelations="http://wsTransactions.ec3.at/2004/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://wsTransactions.ec3.at/2004/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.

### 4.2. Parameter Passing

Since the input parameters used in the ATM rules presented in [7] 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 constraint, we use a similar technique as in Sect. 4.1: 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). E.g., argument passing in the rollback rule of a transaction with savepoints is as follows:

```
<rule stateTransition="ROLLBACK WORK"
      xmlns:aaRelations="http://wsTransactions.ec3.at/2004/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>
```

### 4.3. Supplementary AA Type Declarations

In [7], 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 used AA types in the model should be added to enhance readability and to ease processing by software programs. State transitions are also defined implicitly in [7], 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>
```

## 5. Selected Aspects of an XML Advanced Transaction Model

To clarify the XML representation in Sect. 4, we present selected aspects of a multi-level transaction [10] model XML description. We have chosen this particular ATM because its ideas are used in the industry Web service transaction proposals, i.e. in WS-Transactions [2], WS-CAF [1], and BTP [3].

Besides the “system” transaction (see Sect. 3) and a transaction “T”, which has the usual signal ports (“abort”, “begin”, “commit”) and states (“aborted” and “committed”), multi-level 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="comitted"/>
      </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 multi-level 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://wsTransactions.ec3.at/2004/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.

## 6. Conclusion and Future Work

In this paper we have introduced a meta-model for advanced transaction models and an appropriate XML representation for its model instances. As foundation we used Gray & Reuter's meta-model [7]. A one-to-one XML mapping of this model is not favourable, therefore we enhanced the 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 multi-level transactions) in XML.

In the next step we will analyse another advanced transaction meta-model called ACTA [4]. If it is reasonable, we will create an XML representation of the ACTA model and make a comparison between the two XML ATM meta-models. To prove the capabilities of the model(s) we will create a catalogue of well-known ATMs in XML. Another important task is the development of various transformations of the XML transaction models, e.g. to diagrams and HTML documents (comprehension), to Java code fragments (development), to a requirement report for Web service transaction systems/transaction monitors (administration), etc.

## References


[1] BUNTING, D., CHAPMAN, M., HURLEY, O., LITTLE, M., MISCHKINSKY, J., NEWCOMER, E., WEBBER AND J. AND SWENSON, K., Web Services Composite Application Framework, available from http://developers.sun.com/ techtopics/webservices/wscaf/primer.pdf, cited 2004-04-15.

[2] CABRERA, F., COPELAND, G., COX, B., FREUND, T., KLEIN, J., STOREY, T. AND THATTE, S., Web Services Transaction (WS-Transaction), available from http://www-106.ibm.com/ developerworks/webservices/library/ws-transpec/, cited 2004-04-15.

[3] CEPONKUS, A., DALAL, S., FLETCHER, T., FURNISS, P. AND GREEN, A., Business Transaction Protocol, available from http://www.oasis-open.org/committees/ download.php/1184/2002-06-03.BTP_cttee_spec_1.0.pdf, cited 2004-04-15.

[4] CHRYSANTHIS, P. K. and RAMAMTITHAM, K., Acta: A framework for specifying and reasoning about transaction structure and behaviour, in: Proceedings of ACM-SIGMOD International Conference on Management of Data, pages 194-203, 1990.

[5] ELIOT, J. and MOSS, B., Nested Transactions: An Approach to Reliable Distributed Computing, Cambridge/Massachusetts, MIT Press, 1985.

[6] ELMAGARMID, A. (ed.), Database Transaction Models for Advanced Applications, 1st edn., San Mateo/ California, Morgan Kaufmann Publishers, 1992.

[7] GRAY, J. and REUTER, A., Transaction Processing: Concepts and Techniques, San Francisco/California, Morgan Kaufmann Publishers, 2002.

[8] POTTS, M., COX, B., POPE, B., Business Transaction Protocol Primer, available from http://www.oasis-open.org/committees/business-transactions/documents/primer/BTP_Primer_v1.0.20020605.pdf, cited 2004-07-21.

[9] ROBERTS, J. and SRINIVASAN, K., Tentative Hold Protocol Part 1: White Paper, available from http://www.w3.org/TR/tenthold-1/, cited 2004-04-15.

[10] WEIKUM, G. and SCHEK, H.-J., Multi-level transactions and open nested transactions, in: Data Engineering Archive, Vol. 14, No. 1, pages 60–64, 1991.