Ensuring Required Failure Atomicity of Composite Web Services

Sami Bhiri
LORIA-INRIA
BP 239, F-54506
Vandoeuvre-les-Nancy Cedex, France
bhiri@loria.fr

Olivier Perrin
LORIA-INRIA
BP 239, F-54506
Vandoeuvre-les-Nancy Cedex, France
operrin@loria.fr

Claude Godart
LORIA-INRIA
BP 239, F-54506
Vandoeuvre-les-Nancy Cedex, France
godart@loria.fr

ABSTRACT

The recent evolution of Internet, driven by the Web services technology, is extending the role of the Web from a support of information interaction to a middleware for B2B interactions.

Indeed, the Web services technology allows enterprises to outsource parts of their business processes using Web services. And it also provides the opportunity to dynamically offer new value-added services through the composition of pre-existing Web services.

In spite of the growing interest in Web services, current technologies are found lacking efficient transactional support for composite Web services (CSs).

In this paper, we propose a transactional approach to ensure the failure atomicity, of a CS, required by partners. We use the Accepted Termination States (ATS) property as a mean to express the required failure atomicity.

Partners specify their CS, mainly its control flow, and the required ATS. Then, we use a set of transactional rules to assist designers to compose a valid CS with regards to the specified ATS.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*; H.2.4 [**Database Management**]: Systems—*Transaction Processing*; K.4.4 [**Computers and Society**]: Electronic Commerce—*Distributed commercial transactions*

General Terms

Design, Reliability

Keywords

Reliable Web services compositions, Failure atomicity, Transactional models

1. INTRODUCTION

Web services are emerging as a promising technology for automating B2B interactions. Nowadays, enterprises are able to outsource their internal business processes as services and make them accessible via the Web. Then they can dynamically combine individual services to provide new value-added services.

A main problem that remains is how to ensure a correct composition and a reliable execution of a composite service (CS) with regards to partners transactional requirements.

Despite growing interest, Web services middleware is still rather primitive in terms of functionality, far from what current EAI middleware can provide for intra-enterprise applications [2].

The current Web services technologies ensure communication interoperability which is a part of the problem when considering the building of reliable Web services compositions [16]. Indeed, unlike activities in traditional workflows, services are defined independently of any computing context. Thereafter, the task of building composite Web services requires mechanisms to deal with the inherent *autonomy*, and *heterogeneity* of Web services.

Although powerful, *Advanced Transaction Models* (ATMs) [6] are found lacking functionality and performance when used for applications that involve dynamic composition of heterogenous services in a peer-to-peer context.

Their limitations come mainly from their inflexibility to incorporate different transactional semantics as well as different interactions patterns into the same structured transaction [8].

In this paper, we propose a transactional approach for reliable Web services compositions by ensuring the failure atomicity required by the designers.

From a transactional point of view, we consider a CS as a structured transaction, Web services as sub transactions and interactions as inter sub transactions dependencies. We use the Accepted Termination States (ATS) property as a correctness criteria to relax atomicity.

To the best of our knowledge, defining a transaction with a particular set of properties, in particulary ATS, and ensuring that every execution will preserve these properties remains a difficult and open problem [18].

The paper is organized as follows. Section 2 introduces a motivating example and gives the main points which has driven our approach. In section 3, we explain the notion of transactional web service and show how we express its transactional properties. Section 4 presents the notion of Transactional Composite (Web) Service (TCS) and explains our transactional point of view. Section 5 presents the notion

WWW2005, May 10–15, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

of Accepted Termination States (ATS) as a mean to express the required failure atomicity. Section 6 illustrates how our approach proceeds (using a set of transactional rules) to assist designers to compose valid TCSs. In section 7, we discuss some related work. Section 8 concludes our paper.

2. MOTIVATING EXAMPLE AND METHODOLOGY

Let us first present a motivating example. We consider an application dedicated to the online purchase of personal computer. This application is carried out by a composite service as illustrated in figure 1. Services involved in this application are: the **Customer Requirements Specification (CRS)** service used to receive the customer order and to review the customer requirements, the **Order Items (OI)** service used to order the computer components if the online store does not have all of it, the **Payment by Credit Card (PCC)** service used to guarantee the payment by credit card, the **Computer Assembly (CA)** service used to ensure the computer assembly once the payment is done and the required components are available, and the **Deliver Computer (DC)** service used to deliver the computer to the customer (provided either by Fedex (DC_{Fed}) or TNT (DC_{TNT})).

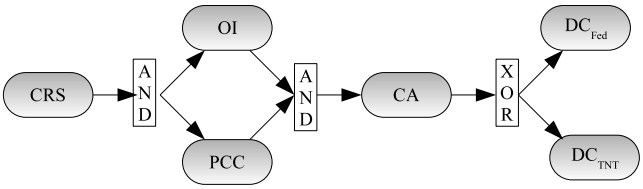

Figure 1: A composite service for online computer purchase.

When a user designs a composite service, he expects the service execution to be reliable. That means he particularly pays attention to failure handling. In our example, the designer may want to be sure: that one of the two delivery services will succeed, that the service CA is sure to complete, and that it is possible for the service OI to undo its effects (for instance when the payment fails). These properties define what we call the *transactional behavior* of the service. This behavior is specified using a set of transactional requirements. Since these requirements may vary from one context to another, the transactional behavior will vary too. For instance, a designer may accept the failure of the DC_{Fed} service in a context, while in another one he may not tolerate such a failure at such an advanced stage. So the mean of a reliable execution is tightly related to transactional requirements and it may vary according to designers.

In the same time, in order to ensure a reliable execution, we have to be sure that a specified transactional behavior is consistent with the set of selected services and the transactional requirements. Back to our example, we can easily notice that since the OI service is not sure to complete, the payment service PCC have to be compensatable (and it must be compensated when the OI service fails).

The following points introduce our approach and its concepts for supporting this kind of scenarios.

First, we believe that we must enhance Web services description for a better characterization of their transactional behavior. This can be done by enhancing WSDL interface with transactional properties.

Second, we have to model services composition and choreography, in particular mechanisms for failure handling and recovery.

Third, we have to provide designers with a mean to express their transactional requirements, in particular their required failure atomicity level. Finally, we have to support composite service validation with regards to designers' requirements.

In the rest of the paper we detail each of these issues and especially our set of transactional management rules for composite service validation.

3. TRANSACTIONAL WEB SERVICE DESCRIPTION

A Web service is a self-contained modular program that can be discovered and invoked across the Internet. Web services are typically built with XML, SOAP, WSDL and UDDI specifications [4] [17]. A transactional Web service is a Web service that emphasizes transactional properties for its characterization and correct usage.

The main transactional properties of a Web service that we are considering are *retriable, compensatable and pivot* [14]. A service s is said to be ***retriable*** if it is sure to complete after several finite activations. s is said to be ***compensatable*** if it offers compensation policies to semantically undo its effects. Then, s is said to be ***pivot*** if once it successfully completes, its effects remains for ever and cannot be semantically undone. Naturally, a service can combine properties, and the set of all possible combinations is $\{r; cp; p; (r, cp); (r, p)\}$.

In order to model the internal behavior of a service, we have adopted a states/transitions model. A service has a minimal set of states (*initial*, *active*, *aborted*, *cancelled*, *failed* and *completed*), and it also includes a set of transitions (*activate()*, *abort()*, *cancel()*, *fail()*, and *complete()*). The figure 2.a shows the internal states/transitions diagram of a pivot service.

When a service is instantiated, the state of the instance is *initial*. Then this instance can be either *aborted* or *activated*. Once it is *active*, the instance can normally continues its execution or it can be *cancelled* during its execution. In the first case, it can achieve its objective and successfully *completes* or it can *fail*.

The requested transactional properties can be expressed by extending the service states and transitions. For instance, for a compensatable service, a new state *compensated* and a new transition *compensate()* are introduced (e.g., service in figure 2.b). Figure 2 illustrates the states/transitions diagram of a retriable service (figure 2.c) and states/transitions diagrams of services combining different transactional properties (figures 2.d and 2.e).

Within a transactional service, we also distinguish between external and internal transitions.

External transitions are fired by external entities. Typically they allow a service to interact with the outside and to specify composite services choreographies (see next section). The external transitions we are considering are *activate(), abort(), cancel(), and compensate()*.

Internal transitions are fired by the service itself (the ser-

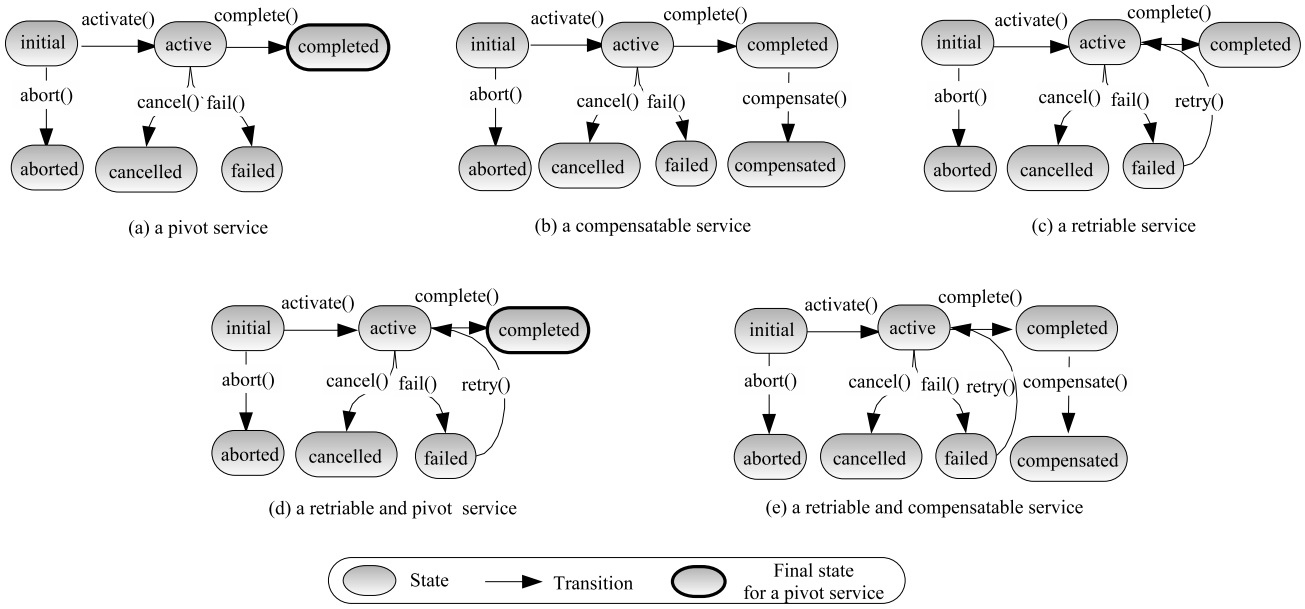

Figure 2: Services states/transitions diagrams according to different transactional properties.

vice agent). Internal transitions we are considering are *complete(), fail(), and retry().*

4. COMPOSITION OF TRANSACTIONAL WEB SERVICES

A composite Web service is a conglomeration of existing Web services working in tandem to offer a new value-added service [13]. It coordinates a set of services as a cohesive unit of work to achieve common goals.

A Transactional Composite (Web) Service (TCS) emphasizes transactional properties for composition and synchronization of component Web services. It takes advantage of services transactional properties to specify mechanisms for failure handling and recovery.

4.1 Dependencies between services

A TCS defines services orchestration by specifying dependencies between services. They specify how services are coupled and how the behavior of certain service(s) influence the behavior of other service(s).

DEFINITION 4.1 (DEPENDENCY FROM s_1 TO s_2). *A dependency from s_1 to s_2 exists if a **transition** of s_1 can fire an external **transition** of s_2.*

A dependency defines for each external transition of a service a precondition to be enforced before this transition can be fired.

In our approach, we consider the following dependencies between services:

Activation dependency from s_1 to s_2: There is an activation dependency from s_1 to s_2 if the completion of s_1 can fire the activation of s_2.

We can tailor activation dependencies between services by specifying the activation condition, $ActCond(s)$, of each service s. $ActCond(s)$ defines the precondition to be enforced before the service s can be activated (only after the completion of other service(s)). There is an activation dependency from s_1 to s_2 iff $s_1.completed \in ActCond(s_2)$. Reciprocally for each service $s_1 \in ActCond(s_2)$, there is an activation dependency from s_1 to s_2 according to $ActCond(s_2)$.

For example, the composite services defined in figure 3 define an activation dependency from OI and PCC, to CA such that CA will be activated after the completion of OI and PCC. That means $ActCond(CA) = OI.completed \bigwedge PCC.completed$.

Alternative dependency from s_1 to s_2: There is an alternative dependency from s_1 to s_2 if the failure of s_1 can fire the activation of s_2.

We can tailor alternative dependencies between services by specifying the alternative condition, $AltCond(s)$, of each service s. $AltCond(s)$ defines the precondition to be enforced before the service s can be activated as an alternative of other service(s). There is an alternative dependency from s_1 to s_2 iff $s_1.failed \in AltCond(s)$. Reciprocally for each service $s_1 \in AltCond(s_2)$, there is an alternative dependency from s_1 to s_2 according to $AltCond(s_2)$

For instance the composite service cs_1 in figure 3.b defines an alternative dependency from DC_{Fed} to DC_{TNT} such that DC_{TNT} will be activated when DC_{Fed} fails. That means $AltCond(DC_{TNT}) = DC_{Fed}.failed$.

Abortion dependency from s_1 to s_2: There is an abortion dependency from s_1 to s_2 if the failure, cancellation or the abortion of s_1 can fire the abortion of s_2.

We can tailor abortion dependencies between services by specifying the abortion condition, $AbtCond(s)$, of each service s. $AbtCond(s)$ defines the precondition to be enforced before the service s can be aborted. There is an abortion dependency from s_1 to s_2 iff $s_1.aborted \in AbtCond(s_2)) \bigvee s_1.failed \in AbtCond(s_2) \bigvee s_1.cancelled \in AbtCond(s_2)$. Reciprocally for each service $s_1 \in AbtCond(s_2)$, there is an abortion dependency from s_1 to s_2 according to $AbtCond(s_2)$.

Compensation dependency from s_1 to s_2: There is

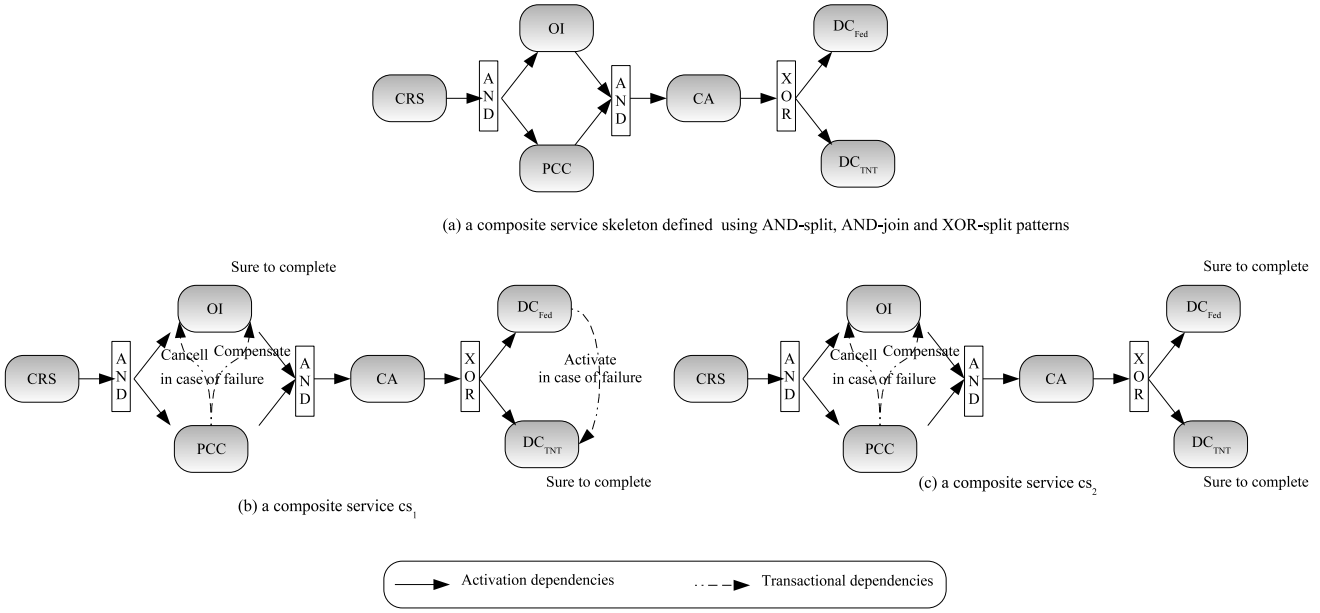

Figure 3: Two composite services defined according to the same skeleton.

a compensation dependency from s_1 to s_2 if the the failure or the compensation of s_1 can fire the compensation of s_2.

We can tailor compensation dependencies between services by specifying the compensation condition, $CpsCond(s)$, of each service s. $CpsCond(s)$ defines the precondition to be enforced before the service s can be compensated. There is a compensation dependency from s_1 to s_2 iff $s_1.failed \in CpsCond(s_2) \bigvee s_1.compensated \in CpsCond(s_2)$. Reciprocally for each service $s_1 \in CpsCond(S_2)$, there is a compensation dependency from s_1 to s_2 according to $CpsCond(s_2)$.

Composite services in figure 3 define a compensation dependency from PCC to OI such that OI will be compensated when PCC fails. That means $CpsCond(OI) = PCC.failed$.

Cancellation dependency from s_1 to s_2: There is a cancellation dependency from s_1 to s_2 if the failure of s_1 can fire the cancellation of s_2.

We can tailor cancellation dependencies between services by specifying the cancellation condition, $CnlCond(s)$, of each service s. $CnlCond(s)$ defines the precondition to be enforced before the service s can be cancelled. There is a cancellation dependency from s_1 to s_2 iff $s_1.failed \in CnlCond(s_2)$.

Reciprocally for each service $s_1 \in CnlCond(s_2)$, there is a cancellation dependency from s_1 to s_2 according to $CnlCond(s_2)$.

Composite services in figure 3 define a cancellation dependency from PCC to OI such that OI will be cancelled when PCC fails. That means $CnlCond(OI) = PCC.failed$.

For clarity reasons, we do not deal with abortion dependencies. We call transactional dependencies the compensation, cancellation and alternative dependencies.

4.2 Relations between dependencies

Dependencies specification must respect some semantic restrictions. Indeed, transactional dependencies depend on activation dependencies according to the following relations:

R_1 : An abortion dependency from s_1 to s_2 can exist only if there is an activation dependency from s_1 to s_2.

R_2 : A compensation dependency from s_1 to s_2 can exist only if there is an activation dependency from s_2 to s_1, or s_1 and s_2 execute in parallel and are synchronized.

R_3 : A cancellation dependency from s_1 to s_2 can exist only if s_1 and s_2 execute in parallel and are synchronized.

R_4 : An alternative dependency from s_1 to s_2 can exist only if s_1 and s_2 are exclusive.

Section 4.4 shows how these relations define potential dependencies induced by given activation dependencies.

4.3 Control and transactional flow of a TCS

Within a transactional composite service, we distinguish between the TCS control flow and the TCS transactional flow.

Control flow: The control flow (or skeleton) of a TCS specifies the partial ordering of component services activations. Activation dependencies between component services define the corresponding TCS control flow.

We use (workflow-like) patterns to define a composite service skeleton. As defined in [7], a pattern "is the abstraction from a concrete form which keeps recurring in specific non arbitrary contexts". A workflow pattern can be seen as an abstract description of a recurrent class of interactions based on (primitive) activation dependency. For example, the *AND-join* pattern [21] (see figure 3.a) describes an abstract services choreography by specifying services interactions as following: *a service is activated after the completion of several other services.*

Example: Figure 3.a illustrates a TCS skeleton defined using an AND-split, an AND-join and an XOR-split patterns.

Transactional flow: The transactional flow of a TCS specifies mechanisms for failures handling and recovery.

Transactional dependencies (like compensation, cancellation and alternative) define the TCS transactional flow.

4.4 Pattern's potential dependencies

Several TCSs can be defined based on a skeleton. Each TCS adopts the activation dependencies defined by the skeleton's patterns and may extend them by specifying additional transactional dependencies.

Example: Figure 3 shows two TCSs, cs_1 and cs_2, defined using the same skeleton. Each of these TCSs adopts this skeleton (figure 3.a) and refines it with an additional transactional flow.

Additional transactional dependencies are a subset of potential transactional dependencies defined by the skeleton's patterns. Indeed, a pattern defines in addition to the default activation dependencies, a set of potential transactional dependencies.

A *potential dependency* is a dependency that is not initially defined by the pattern but that can be added by TCSs using this pattern. Potential dependencies are directly related to the pattern's activation dependencies according to the relations we have introduced in section 4.2.

We have shown above that dependencies between services can be tailored by specifying preconditions on services' external transitions. And potential transactional dependencies are not an exception to this fact. So a TCS skeleton defines for each service the potential conditions corresponding to the potential dependencies. A pattern defines for each service s it is connected with:

- $ptCpsCond(s)$: its potential compensation condition,
- $ptAltCond(s)$: its potential alternative condition,
- $ptCnlCond(s)$: its potential cancellation condition.

We can write each of these conditions in exclusive disjunctive normal form. For instance, we can write the potential compensation condition of a service s as follows: $ptCpsCond(s) = \bigoplus_i ptCpsCond_i(s)$. Then $ptCpsCond_i(s)$ is one potential compensation condition of s.

Example: The TCS skeleton illustrated in figure 3.a uses an AND-join pattern to define activation dependencies between services OI, PCC and CA. According to the relation R_2 given in section 4.2, a TCS based on this skeleton can eventually specifies the following compensation dependencies: from OI to PCC, from PCC to OI, from CA to OI and from CA to PCC. Similarly, according to the relation R_3, this pattern defines the following potential cancellation dependencies: from OI to PCC, and from PCC to OI. That means, among other, that *ptCpsCond(PCC)=OI.failed* $\bigoplus$ *CA.failed* $\bigoplus$ *CA.compensated* and *ptCnlCond(OI)=PCC.failed*.

In the same way, according to the relation R_4, the XOR-split pattern connecting CA, DC_{Fed} and DC_{TNT} defines the following potential alternative dependencies: from DC_{TNT} to DC_{Fed} and from DC_{Fed} to DC_{TNT}.

That means that $ptAltCond(DC_{TNT}) = DC_{Fed}.failed$ and that $DC_{TNT}.failed = ptAltCond(DC_{Fed})$.

Finally, note that both TCSs cs_1 and cs_2 define their transactional flow as a subset of the potential transactional flow presented above. Both define compensation and cancellation dependencies from PCC to OI. cs_1 defines an alternative dependency from DC_{Fed} to DC_{TNT}.

5. FAILURE ATOMICITY REQUIREMENTS OF A TCS

Several executions can be instantiated according to the same TCS. The state of an instance of a TCS composed of n services is the tuple (x_1, x_2, ..., x_n), where x_i is the state of service s_i at a given time. The set of termination states of a TCS cs, $STS(cs)$, is the set of all possible termination states of its instances.

Back to our motivating example limited to the three services CRS, OI and PCC, we can have the following set of termination states: (*CRS.completed, OI.completed, PCC.completed*); (*CRS.completed, OI.failed, PCC.completed*); (*CRS.completed, OI.completed, PCC.failed*); (*CRS.compensated, OI.failed, PCC.initial*); (*CRS.compensated, OI.initial, PCC.failed*); (*CRS.compensated, OI.failed, PCC.failed*).

In order to express the designer's requirements for failure atomicity, we use the notion of *Accepted Termination States* ([18]). In other word, the concept of ATS represents our notion of correction.

Definition 5.1 (Accepted Termination States). *An accepted termination state, ats, of a composite service cs is a state for which designers accept the termination of cs. We define ATS the set of all Accepted Termination States required by designers.*

An execution is correct iff it leads the CS into an accepted termination state. A CS reaches an ats if (i) it completes successfully or (ii) it fails and undoes all undesirable effects of partial execution in accordance with designer failure-atomicity requirements [18].

Back to our example, a designer may choose the following ATS: *ATS(CS)={(CRS.completed, OI.completed, PCC.completed); (CRS.compensated, OI.failed, PCC.failed)}* that means that an execution is correct when all of the services complete, or when CRS is compensated (given the failure of OI and PCC). Obviously, we note that a composite service transactional behavior may vary according to the required ATS.

6. TRANSACTIONAL RULES

To explain the rules and to illustrate how they are working, we go back to our motivating example of personal computer online purchase. We suppose in addition that designers specify the ATS illustrated in the figure 5 to express their required failure atomicity.

Intuitively, the execution of a composite service can generate various termination states. A composite service is not valid if it exists some termination states that do not belong to the ATS specified by the designers.

Definition 6.1 (Validity according to an ATS). *A CS cs is said to be **valid** according to ATS* iff *its set of termination states is included in ATS, written* $STS(cs) \subseteq ATS$.

Example: The composite service cs_1 (illustrated in figure 3.b) is valid because $STS(cs_1) \in ATS$. However regarding the composite service cs_2 (illustrated in figure 3.c), we

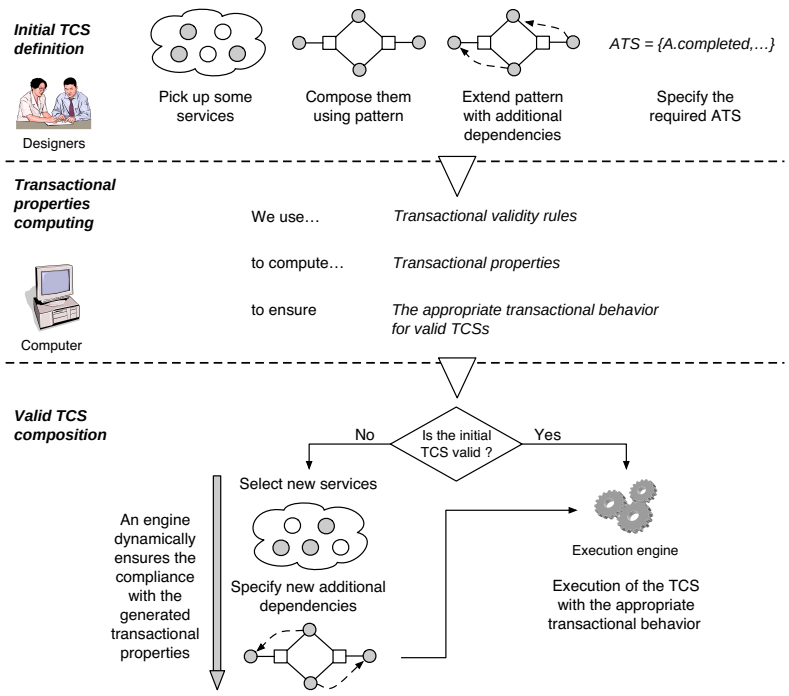

Figure 4: Objective and overview of our approach.

services / ats	CRS	OI	PCC	CA	DC_{Fed}	DC_{TNT}
ats_1	(completed,	completed,	completed,	completed,	initial,	completed)
ats_2	(completed,	completed,	completed,	completed,	completed,	initial)
ats_3	(completed,	compensated,	failed,	aborted,	aborted,	aborted)
ats_4	(completed,	failed,	compensated,	aborted,	aborted,	aborted)
ats_5	(completed,	cancelled,	failed,	aborted,	aborted,	aborted)
ats_6	(completed,	completed,	completed,	completed,	failed,	completed)

Figure 5: ATS used in our example of PC online purchase.

note that $STS(cs_2)$ contains the following termination state, $(CRS.completed, OI.failed, PCC.completed, CA.aborted, DC_{Fed}.aborted, DC_{TNT}.aborted)$, which is not an accepted termination state. Thereafter cs_2 is not valid.

6.1 Objective and overview

As illustrated in figure 4, our approach applies in a top-down manner.

Definition of an initial TCS: First, designers dynamically choose some available services and combine them to offer a new value added service. They compose the new service using a set of interactions patterns (sequence, AND-split, AND-join,...).

They can augment this skeleton by new dependencies selected from the potential dependencies. Then they express their required failure atomicity by specifying the required ATS.

Compute validity transactional properties: We use a set of rules, independent from skeletons and designers' ATS, to compute from the TCS skeleton and the required ATS a set of transactional properties.

These transactional properties tailor the appropriate transactional behavior for valid TCSs. A TCS must satisfy these transactional properties to be valid.

Definition of a valid TCS: If the initial TCS is not valid, designers can (i) select new services (with eventually new transactional properties) and (ii) augment the same skeleton with new dependencies. During this phase an engine assist designers to compose a valid TCS by respecting the generated transactional properties.

Once a valid TCS is reached, it can be deployed and executed.

6.2 Extracting services conditions

Tailoring the appropriate transactional behavior for valid composite services is equivalent to identify the appropriate

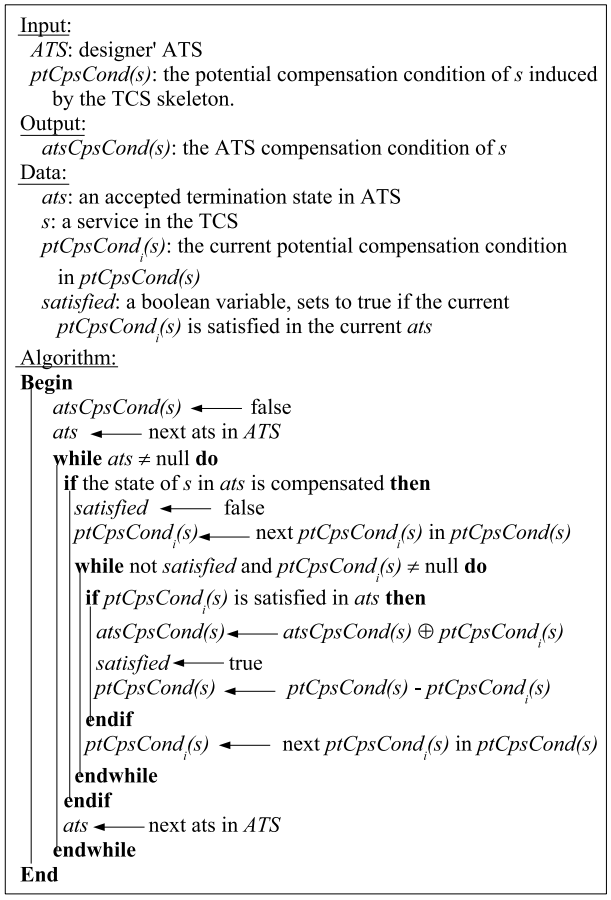

Figure 6: The algorithm for extracting ATS compensation conditions of a service s from the specified ATS and the TCS skeleton.

dependencies between services. We can deduce from the specified ATS and the TCS skeleton the services' conditions corresponding to these dependencies.

For each service s we distinguish (i) $atsCpsCond(s)$, the ATS compensation condition deduced from ATS, (ii) the ATS cancellation condition, $atsCnlCond(s)$, deduced from ATS, and (iii) $atsAltCond(s)$, the ATS alternative condition deduced from ATS. Below, we explain how we can deduce these conditions.

The algorithm given in figure 6 allows to extract the ATS compensation condition for a given service s from the composite service skeleton and the required ATS. The principle is: a potential compensation condition of s becomes an ATS compensation condition if it is satisfied in an $ats \in ATS$ such that the state of s in ats is compensated. We proceed similarly to deduce ATS alternative and cancellation conditions of each service.

For instance in our example, the potential compensation condition of PCC, $OI.failed$, becomes an ATS compensation condition because it is satisfied in $ats4$ (in which the state of PCC is compensated). And since $ats4$ is the only ats in which PCC is compensated then we can deduce that $atsCpsCond(PCC) = OI.failed$. Similarly we can extract the ATS cancellation condition for OI. $ats5$ is the only ats in which OI is cancelled. Furthermore, the potential cancellation condition of OI, $PCC.failed$ is satisfied in $ats5$. Then we can deduce that $atsCnlCond(s) = PCC.failed$. Finally, we can deduce in the same way, from $ats6$ and $ptAltCond(DC_{TNT})$, that $atsAltCond(DC_{TNT}) = DC_{Fed}.failed$.

It is important to note that the ATS specified by the designers must be consistent with the pattern semantics. An ATS is consistent if it satisfies the following two conditions.

First, each of its ats must be well-formed. An $ats \in ATS$ is not well-formed if it exists a service s such that none of its potential (or activation) conditions (corresponding to its state in ats) is satisfied (in ats). We can easily modify the previous algorithm to detect if it exists a not well-formed $ats \in ATS$.

Second, the set of all ats must be consistent. Such inconsistency can be detected after the generation of transactional properties ensuring CSs validity.

For instance given the following termination state (limited to the three services OI, PCC and CA) ts = *(OI.completed, PCC.compensated, CA.aborted)*, we note, among other, that none of the service PCC potential conditions ($OI.failed$, $CA.failed$ and $CA.compensated$) is satisfied in ts. So we can deduce that ts is not well-formed.

To illustrate an ATS inconsistency, let us consider the following ATS=$\{ats_1$ = *(OI.completed, PCC.completed, CA.completed);* ats_2 = *(OI.completed, PCC.failed, CA.aborted);* ats_3 = *(OI.compensated, PCC.failed, CA.aborted)*$\}$. Note that ats_1 and ats_2 are contradictory because the service OI once it is completed (in ats_2) and once it is compensated (in ats_3) for the same condition ($PCC.failed \bigwedge CA.aborted$). Thereafter the given ATS is not consistent although each of its ats is well-formed.

6.3 Transactional validity rules

An ATS also defines the accepted termination states of each component service. We denote $ATS(s)$ the set of accepted termination states of a component service s. Regarding our illustrative example, we can deduce, for instance, that *ATS(PCC)* = {*completed, failed, compensated*} and *ATS(CA)* = {*completed, aborted*}.

We can now introduce validity rules we are using to generate transactional properties that ensure validity (we suppose that $\Diamond$F means that F is eventually true):

$\forall s|$ *s is a component service in TCS*

1. $s.failed \notin ATS(s) \Longrightarrow$ generate the following transactional property TP_s^r : *s must be retriable*
2. $s.compensated \notin ATS(s) \Longrightarrow$ generate the following transactional property TP_s^p : *there is no need for s to be compensatable*
3. $\forall atsCpsCond_i(s) \in atsCpsCond(s)$, generate the following transactional property $TP_s^{cp_i}$: $\Diamond(atsCpsCond_i(s)) \Longrightarrow$
   (a) s must be compensatable and
   (b) $atsCpsCond_i(s) \in CpsCond(s)$.
4. $\forall atsCnlCond_i(s) \in atsCnlCond(s)$, generate the following transactional property $TP_s^{cl_i}$: $\Diamond(atsCnlCond_i(s)) \Longrightarrow$ $atsCnlCond_i(s) \in CnlCond(s)$.

5. $\forall atsAltCond_i(s) \in atsAltCond(s)$, generate the following transactional property $TP_s^{at_i}$: $\Diamond(atsAltCond_i(s)) \Longrightarrow atsAltCond_i(s) \in AltCond(s)$.

The first rule postulates that if the state *failed* does not belong to the ATS of s, then it exists a transactional property saying that s must be *retriable*.

The second rule postulates that if the state *compensated* does not belong to the ATS of s, then it exists a transactional property saying that there is no need for s to be compensatable.

The third rule postulates that for each ATS compensation condition of s, $atsCpsCond_i(s)$, it exists a transactional property saying that: if this condition is eventually true then s must be compensatable and $atsCpsCond(s)$ becomes a compensation condition of s. That means $\forall s' \in atsCpsCond_i(s)$ add a compensation dependency from s' to s according to $atsCpsCond_i(s)$.

The fourth rule postulates that for each ATS cancellation condition of s, $atsCnlCond_i(s)$, it exists a transactional property saying that: if this condition is eventually true then $atsCnlCond(s)$ becomes a cancellation condition of s. That means $\forall s' \in atsCnlCond_i(s)$ add a cancellation dependency from s' to s according to $atsCnlCond_i(s)$.

The fifth rule postulates that for each ATS alternative condition of s, $atsAltCond_i(s)$, it exists a transactional property saying that: if this condition is eventually true then $atsAltCond(s)$ becomes an alternative condition of s. That means $\forall s' \in atsAltCond_i(s)$ add an alternative dependency from s' to s according to $atsAltCond_i(s)$.

Example Back to our example, we can compute the following transactional properties: $\mathcal{TP}_{\mathcal{V}}(ATS, CSskeleton) = \{\boldsymbol{TP}^r_{CRS}, \boldsymbol{TP}^r_{CA}, \boldsymbol{TP}^r_{DC_{TNT}}, \boldsymbol{TP}^p_{CA}, \boldsymbol{TP}^p_{CRS}, \boldsymbol{TP}^p_{DC_{TNT}}, \boldsymbol{TP}^p_{DC_{Fed}}, \boldsymbol{TP}^{cp_1}_{PCC}, \boldsymbol{TP}^{cp_1}_{OI}, \boldsymbol{TP}^{cl_1}_{OI}, \boldsymbol{TP}^{at_1}_{DC_{TNT}}\}$

- By applying the first rule and since the state *failed* does not belong to $ATS(CRS)$ we get the transactional property $\boldsymbol{TP}^r_{CRS}$: *CRS must be retriable*. Similarly, we can compute the following transactional properties: $\boldsymbol{TP}^r_{CA}$: *CA must be retriable* and $\boldsymbol{TP}^r_{DC_{TNT}}$: DC_{TNT} *must be retriable*.

- By applying the second rule and since the state *compensated* does not belong to $ATS(CA)$ we get the transactional property $\boldsymbol{TP}^p_{CA}$: *there is no need for CA to be compensatable*. Similarly we can compute the following transactional properties: $\boldsymbol{TP}^p_{CRS}$, $\boldsymbol{TP}^p_{DC_{TNT}}$, and $\boldsymbol{TP}^p_{DC_{Fed}}$: *there is no need for CRS,* DC_{TNT}, DC_{Fed} *to be compensatable*.

- By applying the third rule and since *atsCpsCond(PCC) = OI.failed* we get the transactional property $\boldsymbol{TP}^{cp_1}_{PCC}$: $\Diamond(OI.failed)$ (means that OI is not retriable) $\Longrightarrow$
  (a) *PCC must be compensatable* and
  (b) *PCC must be compensated when OI fails*.

- By applying the third rule and since *atsCpsCond(OI) = PCC.failed* we get the transactional property $\boldsymbol{TP}^{cp_1}_{OI}$: $\Diamond(PCC.failed)$ (means that PCC is not retriable) $\Longrightarrow$
  (a) *OI must be compensatable* and
  (b) *OI must be compensated when PCC fails*.

- By applying the fourth rule and since *atsCnlCond(OI) = PCC.failed* we get the transactional property $\boldsymbol{TP}^{cl_1}_{OI}$: $\Diamond(PCC.failed)$ (means that PCC is not retriable) $\Longrightarrow$ *OI must be cancelled when PCC fails*.

- By applying the fifth rule and since $DC_{Fed}.failed$ = *atsAltCond(*DC_{TNT}*)* we get the transactional property $\boldsymbol{TP}^{At_1}_{PCC}$: $\Diamond(DC_{Fed}.failed)$ (means that DC_{Fed} is not retriable) $\Longrightarrow$ DC_{TNT} *must be activated when* DC_{FED} *fails*.

The composite service cs_1 verifies all the validity rules and thereafter it is valid. However the composite service cs_2 verifies all the validity rules except the $R^{cp_1}_{PCC}$ rule. This rule postulates that if the compensation condition of PCC (which is the failure of OI) is eventually true (which is the case in cs_2) then PCC must be compensatable and must be compensated when OI fails (which is not the case in cs_2). The composite service cs_1 respects this rule since OI is sure to complete and thereafter never fails.

6.4 Validity rules proof

We use the following lemma (the proof is not shown due to lack of space).

Lemma A TCS termination state ts is not an accepted termination state iff $\exists$ a service s such that

- the termination state of s in $ts \notin ATS(s)$ or
- none of its ATS potential conditions (corresponding to its state in ts) is satisfied (in ts).

Proving that: (cs satisfies all validity rules $\Longleftrightarrow$ cs is valid) is equivalent to proof that: (cs is not valid $\Longleftrightarrow$ $\exists$ a rule such thats cs does not satisfy this rule).

1) $\Longrightarrow$: cs is not valid means that it has a not valid termination state. That means (using the lemma above) either (a) it exists a service s which terminates in a not valid state or (b) it exists a service s which terminates in a valid state but without satisfying one of its ATS conditions corresponding to this termination state. (a) means that cs does not verify the validity rules 1 or 2. (b) means that cs does not verify one of the validity rules 3, 4 or 5.

2) $\Longleftarrow$: (a) If cs does not verify one of the validity rules 1 or 2 then it exists a service s which will terminate in a non valid termination state. (b) if cs does not verify one of the validity rules 3, 4 or 5 then it exists a service s which will terminate in a valid state without satisfying none of its corresponding ATS conditions. (a) and (b) means that (using the lemma above) cs is not valid.

6.5 Implementation

We are currently developing a prototype that supports this work. Our prototype is written in Java.

The first part of the prototype is the transactional engine. It allows the user to select the services (with transactional properties), to define the TCS skeleton using patterns, and to specify the required ATS. The engine uses the transactional rules to compute the appropriate transactional properties for valid TCSs. Then, it assists designers to compose a valid TCS by respecting these transactional properties.

Window 1 of figure 7 shows how designers can choose services from the "Web services" scroll panel. It typically shows the transactional properties of the chosen service.

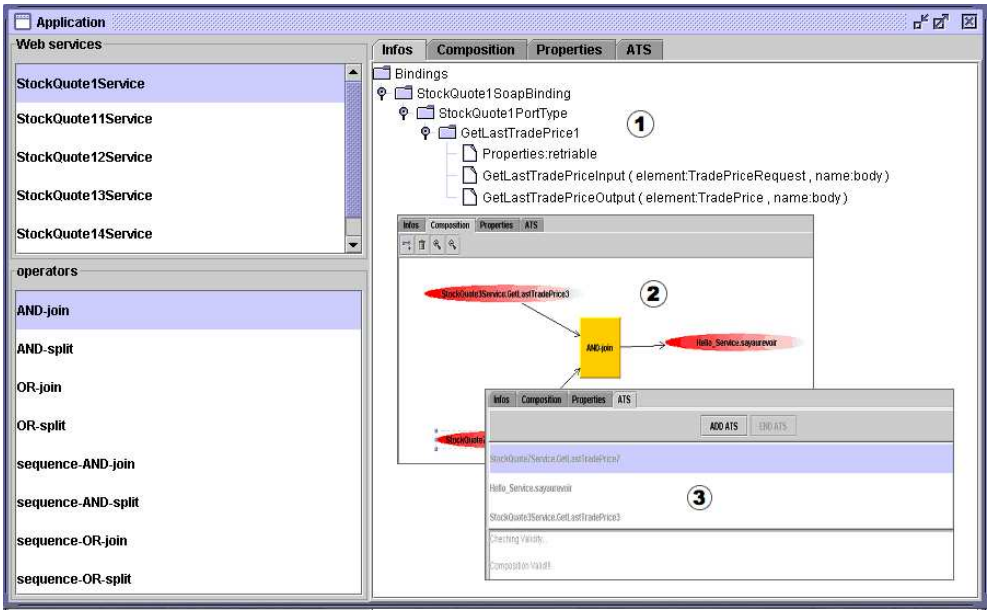

Figure 7: A screen shot illustrating the application of our transactional approach.

Window 2 of figure 7 illustrates how designers can specify the TCS skeleton using patterns from the "operators" panel.

Finally, window 3 of figure 7 illustrates how the transactional engine computes the appropriate transactional properties from the required ATS.

The second part of the prototype is a workflow engine that is able to execute the composite service. Our workflow engine is Bonita, a workflow engine supported by the Object Web consortium ([20]). Bonita is a cooperative workflow system supporting the specification, the execution, the monitoring, and the coordination of the processes. The main features of Bonita are: a third-generation workflow engine that can be **parameterized by an activity model**, a web interface to control workflow processes (accessing workflow methods as J2EE-based web services), an implementation using J2EE Enterprise Java Beans, the possibility to execute code in the server side for different events (e.g., start and cancel activities) by means of hooks (hooks can be for instance Java programs, and may be assigned to process and node events), and the availability of a graphical user interface to design and control workflow processes, based on Java JFC/Swing. Of course, for our concern, the most interesting feature is related to the ability to define a specific model of services, including transactional states.

7. RELATED WORK

Advanced Transaction Models (ATMs) have been proposed to support new database applications by relaxing transaction isolation and atomicity to better match the new requirements. As workflows in the past, services composition requirements either exceed or significantly differ from those of ATMs [6] in terms of modelling, coordination [22] and transactional requirements. Their limitations come mainly from their inflexibility to incorporate different transactional semantics as well as different behavioral patterns into the same structured transaction [8]. To overcome these limitations, [18] proposed a transactional Workflows system supporting multitask, multisystem activities where: (a) different tasks may have different execution behaviors or properties, (b) application or user defined coordination of the different tasks, and (c) application or user defined failure and execution atomicity are supported. In this approach, failure atomicity requirement is defined by specifying a set of ATS. Unfortunately, no transaction management support is provided to ensure this correctness criteria. Accepted termination states as a mean to relax atomicity has been discussed in many previous works [1, 5, 18]. In fact, ATS property has been always implicitly included in most of transactional models. For example, atomicity property implicitly defines ATS for traditional transactions; *all* (success state) and *nothing* (correct failure state). Also, when an advanced transaction model specifies global transaction structure, sub transactions properties, inter sub transaction dependencies, mechanisms of handing-over, success and failure criteria, and so on, it implicitly defines its ATS. In the same way, when [19, 23] define rules to form a well defined flexible transaction, they implicitly define the appropriate ATS for flexible transaction model.

Emerging standards such as BTP [15], WS-transaction (WS-AtomicTransaction and WS-BusinessActivity [11, 12]), and WS-TXM (Acid, BP, LRA)[3] define models to support a two-phase coordination of web services. These proposals are based on a set of extended transactional models to specify coordinations between services. Participants agree to a specific model before starting interactions. Then the corresponding coordination layer technologies support the appropriate messages exchange according to the chosen transactional model. These propositions inherit the extended transactional models rigidity. Besides, there is a potentiel problem of transactional interoperability between services implemented with different approaches. Our approach can complement these efforts and overcome these two gaps. Indeed, our approach allows for reliable, more complex, and

more flexible compositions. In addition, it can coordinate services implemented with different technologies since we use only services transactional features (and not interested in how they are implemented). So, we can use our approach to specify flexible and reliable composite services, while component services can be implemented by one of the above technologies. Once a valid TCS is reached, it can be considered as a coordination protocol and can be plugged in one of the existing coordination technology to be executed.

8. CONCLUSION

In this paper, we have proposed a transactional approach for reliable Web services compositions by ensuring the failure atomicity required by the designers.

Contrary to ATMs, our approach follows the opposite direction by starting from designers requirements to provide correctness rules. Like in [9, 18] (for transactions), designers define the global composite service structure, using patterns, and specifies required ATS as a correctness criteria. Then, we use a set of transactional rules to assist designers to compose a valid CS with regards to the specified ATS.

The main contribution of our approach is that is able to incorporate different interactions patterns into the same structured transaction, and besides it can validate CSs according to designers transactional requirements.

Acknowledgment: We would like to thank Laura Lozano for her implementation efforts.

9. REFERENCES

[1] M. Ansari, L. Ness, M. Rusinkiewicz, A. Sheth. Using Flexible Transactions to Support Multi-System Telecommunication Applications. In *Proc. of the 18th VLDB Conference*, 65-76, August 1992.

[2] B. Benatallah, F. Casasti, F. Toumani. Web Service Conversation Modeling: A Cornerstone for E-Business Automation. In *IEEE Internet Computing*, 8(1), 46-54, January/February 2004.

[3] D. Bunting et al. Web Services Transaction Management (WS-TXM) Version 1.0. Arjuna, Fujitsu, IONA, Oracle, and Sun, July 28, 2003.

[4] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, S. Weerawarana. Unraveling the Web services web: an Introduction to SOAP, WSDL, and UDDI. IEEE Internet Computing, 6(2), 86-93, March/April 2002.

[5] A. Elmagarmid, Y. Leu, W. Litwin, M. Rusinkiewicz. A multidatabase transaction model for Interbase. In *Proc. of the 16th VLDB Conference*, Brisbane, Australia 1990.

[6] A. Elmagarmid, (Ed.). Transaction Models for Advanced Database Applications. Morgan-Kaufmann, 1992.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, Reading, Massachusetts, 1995.

[8] N. Gioldasis, S. Christodoulakis. UTML: Unified Transaction Modeling Language. In *Third International Conference on Web Information Systems Engineering (WISE'02)*, 115-126, IEEE Computer Society, December 2002.

[9] N. Krishnakumar, A. Sheth. Managing Heterogenous Multi-system Tasks to Support Enterprise-wide Operations. Distributed and Parallel Databases, 3(2), 155-186. Kluwer Academic Publishers, 1995.

[10] D. Langworthy et al. Web Services Coordination (WS-Coordination). BEA, IBM, Microsoft, 2003.

[11] D. Langworthy et al. Web Services Atomic Transaction (WS-AtomicTransaction). BEA, IBM, Microsoft, 2003.

[12] D. Langworthy et al. Web Services Business Activity Framework (WS-BusinessActivity). BEA, IBM, Microsoft, 2004.

[13] B. Medjahed, B. Benatallah, A. Bouguettaya, A. H. H. Ngu, A. K. Elmagarmid. Business-to-business interactions: issues and enabling technologies. VLDB Journal: The International Journal on Very Large Data Bases, 12(4), 59-85, April 2003.

[14] S. Mehrotra, R. Rastogi, A. Silberschatz, H. Korth. A transaction model for multidatabase systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS92)* (Yokohama, Japan), IEEE Computer Society Press, 56-63, June 1992.

[15] OASIS Committee Specification. Business Transaction Protocol, Version 1.0 (June 2002).

[16] P. F. Pires. WebTransact: A Framework For Specifying And Coordinating Reliable Web Services Compositions. *Technical report ES-578/02*, Coppe Federal University of Rio De Janeiro, Brazil, April 2002.

[17] P. F. Pires, M. R. Benevides, M. Mattoso. Building Reliable Web Services Compositions. In *Web, Web-Services, and Database Systems*, LNCS 2593, 59-72, Springer, 2003.

[18] M. Rusinkiewicz, A. Sheth. Specification and Execution of Transactional Workflows. In *Modern Database Systems: The Object Model, Interoperability, and Beyond.*, W. Kim Ed., ACM Press and Addison-Wesley, 1995.

[19] H. Schuldt, G. Alonso, C. Beeri, H. J. Schek. Atomicity and isolation for transactional processes. In *ACM Transactions on Database Systems*, 27(1), 63-116, March 2002.

[20] M. Valdès, F. Charoy. Bonita: Workflow Cooperative System. ObjectWeb consortium, http://bonita.objectweb.org/, 2004.

[21] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, A.P. Barros. Advanced Workflow Patterns. In *the 7th International Conference on Cooperative Information Systems (CoopIS 2000)*, LNCS 1901, 18-29. Springer-Verlag, Berlin, 2000.

[22] D. Worah, A. Sheth. Transactions In Transactional Workflows. In *Advanced Transaction Models and Architectures*, S. Jajodia and L. Kerschberg, Eds., chapter 1, pages 3-45. Kluwer Academic Publishers, 1997.

[23] A. Zhang, M. Nodine, B. Bhargava, O. Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. In *Proc. ACM SIGMOD International Conference on Management of Data*, 22(3), 67-78, 1994.