Policies for Context-Driven Transactional Web Services

Zakaria Maamar[1], Nanjangud C. Narendra[2], Djamal Benslimane[3], and Sattanathan Subramanian[4]

[1] Zayed University, U.A.E
zakaria.maamar@zu.ac.ae
[2] IBM India Research Lab, India
narendra@in.ibm.com
[3] Claude Bernard University, Lyon, France
djamal.benslimane@liris.cnrs.fr
[4] IMRU-FUNDP, University of Namur, Belgium
subramanian.sattanathan@fundp.ac.be

Abstract. This paper presents an approach that uses policies to manage context-driven transactional Web services. Context feeds policies with details on Web services like current status, which permits aligning the behavior of these Web services to the transactional properties they need to satisfy. Context refers here to any information on the interactions a Web service initiates with peers and external environment. Three types of transactional properties are used namely pivot, compensatable, and retriable. Each property satisfaction calls for a set of policies that are specified with a policy language like WSPL. This paper also presents the adaptation strategy that supports developing context-driven transactional Web services. A prototype that implements this strategy is discussed in the paper, too.

Keywords: Adaptation, Context, Policy, Transaction, Web service.

1 Introduction

For the W3C, a Web service *"is a software application identified by a URI, whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts and supports direct interactions with other software applications using XML-based messages via Internet-based applications"*. Though this definition highlights the potential and multiple uses of Web services, it does not stress the obstacles that hinder Web services execution and the way these obstacles could be first, identified prior to execution and second, overcome as part of the exception handling strategy. Guidelines backing the correct execution of a Web service need to be stated and checked prior execution. To this end we suggest mapping these guidelines onto transactional properties to be associated with a Web service. The role of a transactional property is to define the acceptable behavior of a Web service. For example the failure of a Web service could be tolerated in one scenario but not in another one. Different transactional properties are reported in literature and different specifications exist (e.g., Web Services Transaction[1], Web Services Transaction Management[2]). In this paper the focus is on pivot, retriable,

[1] dev2dev.bea.com/pub/a/2004/01/ws-transaction.html
[2] developers.sun.com/techtopics/webservices/wscaf/wstxm.pdf

© Springer-Verlag Berlin Heidelberg 2007

and compensatable transactional properties. A Web service is defined as *retriable* if it can be retried one or more times after failure. A Web service is defined as *compensatable* if it offers mechanisms to undo its effects. Finally, a Web service is defined as *pivot* if once it successfully completes, its effects remain unchanged for ever and cannot be semantically undone. Additionally, a pivot Web service cannot be retried following failure, and thus, will need to be aborted.

Exceptions altering a Web service's behavior need to be monitored so, appropriate corrective actions for satisfying the transactional properties of this Web service are taken. We propose to run the monitoring operation upon a structure, which receives, refines, and stores the necessary information for this operation. We refer to this structure as context. Context "... *is not simply the state of a predefined environment with a fixed set of interaction resources. It is part of a process of interacting with an ever-changing environment composed of reconfigurable, migratory, distributed, and multi-scale resources*" [5]. In this paper, context not only supports the operation of monitoring a Web service execution, but supports also a Web service in making decisions based on the status of the surrounding environment [8]. The environment could be related to users (e.g., stationary user, mobile user), computing resources (e.g., fixed device, hand-held device), time of day (e.g., in the afternoon, in the morning), physical locations (e.g., shopping center, movie theater), etc.

Satisfying the transactional properties of a Web service happens through mechanisms, which we specify and implement as policies. In [7], we used policies to support the behavior flexibility of a Web service, so this latter can align its capabilities to users' requirements and resources' constraints. In this paper we motivate behavior flexibility because of the multiple execution situations a Web service encounters. Indeed a Web service has to consider its internal execution status, has to know how to perform exception handling in case it gets disrupted, etc. In this paper as well, policies not only permit checking the satisfaction of the transactional properties of a Web service, but permit also a clear separation between the functionality of a Web service and the different cases that make up the acceptable behavior of a Web service.

In this paper we discuss our approach for using policies to develop context-driven transactional Web services. Context feeds policies with details required for their execution prior to claiming the satisfaction of the transactional property of a Web service in that specific context. Section 2 presents an illustrative scenario and some related works. Section 3 discusses the approach to develop context-driven transactional Web services using policies. Section 4 presents the adaptation strategy that accommodates these Web services' features and requirements. Prior to concluding and highlighting future work in Section 6, prototype of the approach is presented in Section 5.

2 Background

Illustrative scenario. It is about Amin who travels to Trondheim in Norway to meet his friend Melissa. One day they agree to meet in a coffee shop, not far from Melissa's office. Amin has two options to reach the meeting place: by taxi or by bus. A specification of Amin scenario using state chart diagrams and service chart diagrams [9] is illustrated with Fig. 1. The component Web services of this specification are: trip (`TP`),

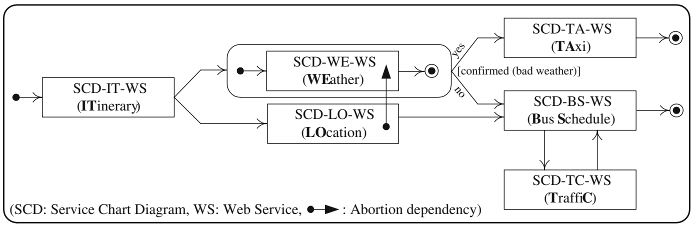


Fig. 1. Specification of Amin scenario

weather (`WE`), location (`LO`), taxi (`TA`), bus schedule (`BS`), and traffic (`TC`). Amin scenario specification could be done with BPEL for example, without any changes in the various policies and strategies that will be defined later.

At his hotel, Amin browses some Web sites about transportation in Trondheim. A site has *Itinerary WS* that proposes routes between two specific places like Amin's hotel and the coffee shop. The proposed routes are subject to weather forecasts: cold weather results in recommending taxis, otherwise public transportation like tramways and buses are recommended. Parallel to checking weather forecasts with *Weather WS*, *Itinerary WS* requests details about the origin and destination places using *Location WS*. Amin appreciates using *Location WS* as he is not familiar with the city.

In case *Weather WS* forecasts bad weather, a taxi booking is made using *Taxi WS* upon Amin's approval. Otherwise, i.e., pleasant day, Amin uses public transportation. The location of both Amin's hotel and coffee shop are submitted to *Bus Schedule WS*, which returns for example the bus numbers Amin has to take. Potential traffic jams force *Bus Schedule WS* to regularly interact with *Traffic WS* that monitors the status of the traffic network. This status is fed into *Bus Schedule WS* so adjustments to bus numbers and correspondences between buses can occur.

From a transactional perspective the designer of Amin scenario needs to pay attention among other things to (i) the Web services that are critical to the successful completeness of this scenario, (ii) the failure details that hinder Web services execution, and (iii) how much these failures impact Web services' and composite Web service's completeness. Hereafter, we list some cases the designer will look into: (i) ensure that either *Taxi WS* or *Bus Schedule WS* completes their execution; (ii) ensure that *Weather WS* successfully completes its execution; and (iii) compensate *Taxi WS* in case the meeting is canceled so the taxi booking is canceled, too.

Related Work. Compared to traditional transactions that comply with the Atomicity, Consistency, Isolation, and Durability (ACID) model, Verma and Deswal discuss the non-suitability of this model for Web services because of the following reasons [14]: transactions may be of a long duration (sometimes lasting hours, days, or more), participants may not allow their resources to be locked for long durations, some of the ACID properties are not mandatory, a transaction may succeed even if only some of the participants choose to confirm and others choose to cancel, transactions that have to be rolled back have the concept of compensation, etc. Interesting to emphasize here the overall success of a transaction despite the failure of some of this transaction's portions.

Bhiri et al. propose a transactional approach to guarantee the failure atomicity of a composite Web service [4]. They use the accepted termination states property as a means for guaranteeing this atomicity. The correctness criterion associated with a composite Web service execution varies from one designer to another. Bhiri et al. claim that this criterion defines the transactional behavior of a composite Web service. This behavior needs to be consistent with the transactional properties that are associated with the component Web services of this composite Web service.

For Younas et al., specifications and protocols developed for Web services transactions such as WS-Transactions, OASIS Business Transaction Protocol (BTP), and Business Transaction Framework are mainly based on the database transaction models such as ACID and extended/advanced transaction models [15]. Although these specifications and protocols have been useful in various domains, they are inappropriate for long running business activities like the ones involving Web services. Younas et al. suggest a new set of transactional properties that are specifically devoted to Web services namely Semantic Atomicity, Consistency, Resiliency, and Durability (SACReD) and are extensively explained in [16]. For instance, semantic atomicity allows the unilateral commit of component service transactions regardless of the commits of their sibling component service transactions.

Pires et al. discuss how to build reliable Web services compositions [10]. Unlike components in traditional business processes, the building task of these compositions is much more difficult due to Web services heterogeneity and autonomy. To face both obstacles, Pires et al. suggest `WebTransact` framework, which is implemented with a multi-layered architecture associated with an XML-based language named *Web Services Transaction Language* (`WSTL`) and a transaction model. Some components that populate this architecture include composite mediator services and remote services.

An interesting perspective on exception handling during process activity failures is built upon forward recovery strategies. This is reported in [3] where Bassil et al. claim that not all failures can be dealt with using roll-back mechanisms such as undoing or compensating activities. Examples of such failures include an already accomplished surgery or a vehicle transporting containers that breaks down. Bassil et al.'s solution suggests a set of factors that may influence the right choice of a forward recovery solution. Two of these factors include knowing the current data context of a failed activity and knowing how far process execution has progressed.

3 Context and Transactional Web Services

3.1 Design and Operation

Achieving transactional Web services using the information that context provides led us to identify the following four levels: composition, component, instance, and state ([9] explains how these levels get deployed). The composition level shows the composite Web services that are developed according to users' needs. The component level shows the Web services that providers develop and advertise so, users' needs are satisfied. The participation of Web services in composite Web services occurs thanks to the instance level [9]. This level shows the Web service instances that are created upon composition participation acceptance. Finally the state level shows the behavior of a

Web service instance using an UML state chart diagram. Each level is associated with a specific type of context: $\mathcal{C}$-context for $\mathcal{C}$omposite Web service, $\mathcal{W}$-context for $\mathcal{W}$eb service, $\mathcal{I}$-context for Web service $\mathcal{I}$nstance, and $\mathcal{S}$-context for $\mathcal{S}$tate chart diagram of a Web service instance. The $\mathcal{W}$-context of a Web service returns information on the participations of this Web service in different compositions. These participations happen according to the Web services instantiation principle [9]. The $\mathcal{C}$-context of a composite Web service is built upon the $\mathcal{W}$-contexts of its component Web services and permits overseeing the progress of a composition. The $\mathcal{I}$-context of a Web service instance records the progress of the execution of this instance, including the states it takes on during execution. Details on the state information of a Web service instance are later recorded in its $\mathcal{S}$-context. Fig. 2 illustrates our proposed context-driven three-level approach for transactional Web services. Not represented in this figure are the composition level and its respective $\mathcal{C}$-context. Interesting to note the $\mathcal{S}$-context of a state chart diagram. $\mathcal{S}$-context tracks the states that permit claiming the satisfaction of the transactional properties of a Web service instance. We recall that composite Web services are made up of Web service instances and not of Web services. In Fig. 2, active means the state that a Web service instance takes now on. Passive means the opposite.

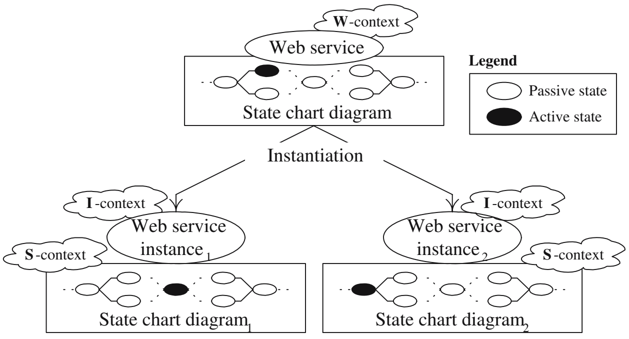


Fig. 2. Context-driven approach for transactional Web services

The operation of this approach concerns Web services of type instance. This operation is about first, context assessment and policy triggering and second, the way context and policy permit meeting the transactional properties of a Web service instance. In this paper context assessment is excluded. Initially the designer associates a Web service, to be deployed later as a Web service instance, with a set of transactional properties like pivot and compensatable. This association per Web service depends on the business logic that underpins the composition scenario. As discussed earlier, the failure of a Web service can be tolerated in one scenario but not in another one.

At run-time, the Web service instance gets triggered according to the specification of the composite Web service. The Web service instance takes on various states like activated, failed, and suspended, which form its state chart diagram. This diagram is context-aware since it has an $\mathcal{S}$-context. As a result tracking the various states that a Web service instance binds to, is now possible. Fig. 3 shows that the monitoring of a

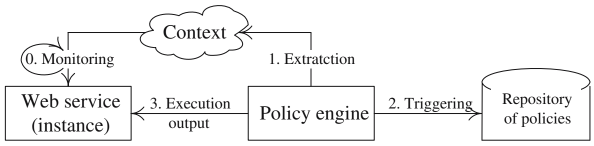


Fig. 3. Operation of the approach

Web service instance is continuous so, relevant details are collected and fed into the context. Additional details are collected as well from the respective contexts of the Web service and composite Web service. All these details are submitted to the policy engine that next, consults the repository of policies. Currently we only assume that one policy executes so conflicts between policies' outcomes are avoided. Execution means making the Web service instance transitions to a new state (active as in Fig. 2), which could allow this Web service instance to satisfy its transactional property. For example if a Web service instance is declared as pivot, then the various policies have to guarantee that this Web service instance only gets aborted in case of execution failure, i.e., no compensation actions are tolerated. We assume in this paper that a Web service does not take on any state that is not included in the acceptable states of its state chart diagram.

We recall that three types of context were defined: $\mathcal{S}$-context, $\mathcal{I}$-context, and $\mathcal{W}$-context. For this paper's requirements the emphasis is on the contexts of state and Web service instance. Each context type has a set of arguments that permit feeding the policy engine with the necessary details for triggering the appropriate policies as depicted in Fig. 3. $\mathcal{S}$**-context**'s arguments include: *StateIdentifier*: Identifier of current state; *StateLabel*: not-activated, activated, suspended, done, compensated, aborted; *PreviousState*: name of previous state from which the Web service instance has transitioned to current state; *NextEffectiveState*: name of next state that the Web service instance has effectively transitioned to; *TransitionIn*: name of transition that permitted transiting the Web service instance to current state; and *TransitionOut*: name of transition that permitted transiting the Web service instance to next effective state; $\mathcal{I}$**-context's arguments** include: *WSIdentifier*: name of Web service instance; *CurrentState*: not-activated, activated, suspended, done, compensated, aborted; *TransactionalProperty*: null, pivot, retriable, compensatable; *MaximumNumberOfRetries*: maximum number of times that the failed execution is authorized to be retried; and *CurrentNumberOfRetries*: current number of times that the Web service instance execution has been retried.

3.2 Transactional Properties and Web Services Modeling

As per Bhiri et al.'s transactional properties namely pivot, compensatable, and retriable (that could be combined as well) [4], we bind to the same properties. We show in the rest of this section how a Web service's behavior is continuously aligned, by using policies, in order to meet the requirements of its associated transactional property. To represent a Web service's behavior we use UML state chart diagram. Since we selected three transactional properties we developed three separate state chart diagrams for clarity reasons. In addition for each state chart diagram we provide a discussion on the role of context

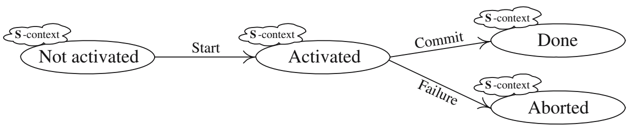


Fig. 4. State chart diagram for a pivot Web service

in feeding the policies with the information that permits achieving the associated transactional property. We recall that the following description applies to Web services of type instance. For illustration purposes we show how a rule is mapped onto WSPL (its syntax is based on the OASIS eXtensible Access Control Markup Language) [2]. The selection of this policy specification language is based on our previous research [7]. In addition, we only detail the pivot transactional property. Fig. 4 shows the acceptable state chart diagram of a pivot Web-service. The key state in this diagram is *activated* from which the Web service could transition to either *done* or *aborted*. We present hereafter the policies that describe the acceptable behavior of a pivot Web service. All the necessary details for policy specification exist in $\mathcal{S}/\mathcal{I}$-contexts.

`WS-Pivot.Policy`$_{done}$ states that a pivot Web service transitions from *activated* state to *done* state if-and-only-if the transactional property is *pivot*, the current state is *activated*, the previous state is *not activated*, and the transition name that was successfully fired is *commit*. This policy is shown below in WSPL.

```
Policy(Aspect="PivotPolicyDone"){
<Rule xmlns="urn:oasis:names:tc:xacml:3.0:generalization:policy:schema:wd:01"
RuleId="PivotPolicyDoneWS">
  <Condition>
    <Apply FunctionId="and">
      <Apply FunctionId="equal" DataType="boolean">
      <SubjectAttributeDesignator AttributeId="TransactionalProperty" DataType="string"/>
      <AttributeValue DataType="string"/> "pivot" </AttributeValue> </Apply>
      <Apply FunctionId="equal" DataType="boolean">
      <SubjectAttributeDesignator AttributeId="CurrentState" DataType="string"/>
      <AttributeValue DataType="string"/> "activated" </AttributeValue> </Apply>
      <Apply FunctionId="equal" DataType="boolean">
      <SubjectAttributeDesignator AttributeId="PreviousState" DataType="string"/>
      <AttributeValue DataType="string"/> "notactivated" </AttributeValue></Apply>
      <Apply FunctionId="equal" DataType="boolean">
      <SubjectAttributeDesignator AttributeId="TransitionOut" DataType="string"/>
      <AttributeValue DataType="string"/> "commit" </AttributeValue></Apply>
    </Apply>
  </Condition>
  <Conclusions> <TrueConclusion PivotPolicyDone = "Permit"/> </Conclusions>
</Rule>}
```

`WS-Pivot.Policy`$_{aborted}$ states that a Web service transitions from *activated* state to *aborted* state if-and-only-if the transactional property is *pivot*, the current state is *activated*, the previous state is *not activated*, and the transition name that was successfully fired is *failure*.

Similar state chart diagrams and their related policies are defined for the retriable and compensatable cases. The retriable case will contain an additional *Suspended* state between *Activated* and *Aborted* states. The compensatable case will extend the retriable case with an additional *Compensated* state between *Done* and *Not-Activated* states.

In addition, the compensatable case will not contain the *Aborted* state; rather, it will contain a transition from *Suspended* to the *Compensated* state.

We discuss hereafter the dependencies among transactional Web services. In particular, we comply with the dependencies suggested by Bhiri et al. in [4], namely activation, abortion, and compensation. We recall that dependencies become effective at the Web service instance level. It is expected that the multiple policies that implement the dependencies will feed also the repository of policies of Fig. 3. Later we will show how these dependencies are used during adaptation (Section 4). In this paper, we expose the transactional properties of the peers of a Web service using two arguments available in $\mathcal{I}$-context: *TransactionalPropertyPerPreviousWebServiceInstance(s)* and *TransactionalPropertyPerNextWebServiceInstance(s)*. For illustration purposes, we present only the compensation dependency. There is a compensation dependency from WS_x to WS_y if the compensation of WS_x fires the compensation of WS_y (or abortion of WS_y, in case WS_y is retriable or pivot). This dependency is reported using $\texttt{WS}_x\texttt{.Policy}_{Compensation(WS_y)}$ and is defined as follows:

```
If      $\text{WS}_x$.I-context.CurrentState(Aborted$\oplus$Compensated) &
        $\text{WS}_y$.I-Context.CurrentState(Done$\oplus$Suspended) &
        $\text{WS}_y$.S-context.TransitionOut(CompensateAfterCommit$\oplus$AbortAfterFailedRetries)
Then    $\text{WS}_y$.S-context.NextEffectiveState=Aborted$\oplus$Compensated &
        $\text{WS}_y$.I-context.CurrentState=$\text{WS}_y$.S-Context.NextEffectiveState &
        $\text{WS}_y$.I-context.CurrentPolicyForNextState=$\text{WS}_x$.Policy$_{Compensation(WS_y)}$
```

Note: $\oplus$ stands for exclusive or.

4 Context-Driven Transactional Web Services Adaptation

In this section, we discuss the adaptation of context-driven transactional Web services during exception handling. Our strategy is to modify the composition specification with minimal disruption to the previously run or already running Web service instances. An exception occurs if a Web service instance execution fails due for example to lack of resources [12]. Exception handling for the Web service instance, called WS.I_{failed}, tightly depends on its transactional property. If it is pivot, then the entire composite Web service will fail, since the effects of the Web service instance cannot be undone. If it is retriable or compensatable, its failure needs to be propagated to the *affected Web service instances* because of the failure's side effects. These affected Web service instances are defined later, but are classified into two types according to their execution order to WS.I_{failed}:

1. Post-affected Web service instances are yet to be performed. This requires a *forward adaptation* strategy.
2. Pre-affected Web service instances are either concurrently executing (perhaps there exists an abortion or compensation dependency from WS.I_{failed} to some of these Web service instance) or have already executed. This requires a *backward adaptation* strategy. This strategy is not discussed in this paper.

4.1 Some Definitions

From now on, we assume that the failed Web service instance WS.I_{failed} is either retriable or compensatable. Before we describe our forward adaptation strategy, some basic definitions are needed. We first, assume that the composition specification like the one in Fig. 1 is mapped onto a graph. The graph's nodes and edges correspond to the Web service instances and the dependencies between these Web service instances, respectively. The dependencies, derived from workflow models [11], are modeled as follows:

- We define the graph of the composition specification as *G = (V,E)*, where *V* is the set of nodes representing Web service instances, and *E* is the set of edges depicting dependencies between the Web service instances. Each edge is a tuple of the form *¡*WS_i*,*WS_j*¿*, where the edge is directed from WS_i to WS_j. The graph also has two unique nodes: *START* (has no predecessors) and *END* (has no successors). The graph is supposed to meet two basic conditions: (a) every node in the graph is directly or indirectly reachable from *START* node, and (b) *END* node is reachable from every node in the graph.
- *Forward edge* ¡$\text{WS}_i \rightarrow \text{WS}_j$¿ depicts the activation dependency between a Web service instance and one of its direct, successor Web service instances in the graph.
- *Backward edge* ¡$\text{WS}_i \leftarrow \text{WS}_j$¿ depicts an edge from a Web service instance to one of its direct, predecessor Web service instances in the graph. This edge depicts repeated execution within a loop, and represents a backward activation dependency.
- Abortion/compensation dependency edges - as described earlier in Section 3.2.

4.2 Forward Adaptation Strategy

In any exception situation, forward adaptation is always preferable, due to its minimal impact on the already executed or currently running Web service instances. The forward adaptation strategy consists of two main steps: (1) determination of the set of the affected Web service instances, and (2) forward adaptation itself.

Determination of the affected Web service instances. Two types of Web service instances are affected by the failed Web service instance WS.I_{failed}: currently executing Web service instances that have an abortion or compensation dependency starting from WS.I_{failed}, and yet to start executing Web service instances that are connected to WS.I_{failed} with forward edges. We extend the former category to include those Web service instances that have abortion/compensation dependencies pointing to the currently executing Web service instances, and so on, in a recursive manner. In other words, the former category of Web service instances includes now all those Web service instances that are directly or indirectly dependent on WS.I_{failed}. The determination algorithm of the set of affected Web service instances is given in Fig. 5 and can be described as follows:

1. Mark all Web service instances that are either currently executing or are yet to execute, in the graph of the composition specification, as "not-visited".
2. For each abortion/compensation dependency pointing from WS.I_{failed}, perform a backward traversal marking all visited Web service instances as "visited", until a Web service instance is reached to which no abortion/compensation dependency edge points.

```
PROC FS(wsif,G)
Input: $wsif$ WebServiceInstanceFailed, $G$ graph
Output: $fs$ SetOfWebServiceInstances
Auxiliary: S, T, K, L
Begin
  ▷ $fs$ represent the Web service instances that are directly or indirectly dependent on $wsif$
  fs$\leftarrow \emptyset$
  WSI$\leftarrow$ WEBSERVINST(G)
  ▷ WebServInst($G$) is a function which returns the whole Web service instances of the graph $G$
  for wsi$\in$WSI do
    if CURRENTSTATE(wsi) $\in \{activated, not - activated\}$ then
      wsi.tag$\leftarrow$ $not\_visited$
    end if
  end for
  ▷ abortion($x$) and completion($x$) are two functions which return the set of Web service instances
  ▷ that have an abortion (respectively a completion) dependency with a Web service instance $x$.
  S$\leftarrow$ ABORTION(wsif) $\bigcup$ COMPLETION(wsif)
  T$\leftarrow \emptyset$
  while S$\neq \emptyset$ do
    for wsx$\in$S do
      wsx.tag$\leftarrow$ $visited$
      T$\leftarrow$T$\bigcup$ ABORTION(wsx) $\bigcup$ COMPLETION(wsx)
      S$\leftarrow$S$-\{$wsx$\}$
      fs$\leftarrow$fs$\bigcup\{$wsx$\}$
    end for
    S$\leftarrow$T
    T$\leftarrow \emptyset$
  end while
  K$\leftarrow$ FORWARD_ALL(wsif)
  ▷ forward_all is a function which returns the set of all Web service instances directly connected to
  ▷ w$sif$ by a forward edge leading out of $wsif$ and its successors.
  L$\leftarrow \emptyset$
  while K$\neq \emptyset$ do
    for wsx$\in$K do
      wsx.tag$\leftarrow$ $visited$
      L$\leftarrow$L$\bigcup$ ABORTION(wsx) $\bigcup$ COMPLETION(wsx)
      K$\leftarrow$K$-\{$wsx$\}$
      fs$\leftarrow$fs$\bigcup\{$wsx$\}$
    end for
    K$\leftarrow$L
    L$\leftarrow \emptyset$
    M$\leftarrow$ IN-LOOP(wsif)
    ▷ if wsif belongs to a loop, this returns the set of instances in the loop. Otherwise, it returns an empty set.
    if M$\neq \emptyset$ then
      wsx$\leftarrow$wsif
      while M$\neq \emptyset and$wsx$\neq$M.first do
        ▷ M.first is the beginning instance in the loop
        wsx$\leftarrow$ PREDECESSOR(wsx)
        wsx.tag$\leftarrow$ $visited$
        fs$\leftarrow$fs$\bigcup\{$wsx$\}$
        M$\leftarrow$M$-\{$wsx$\}$
      end while
      wsx.tag$\leftarrow$ $visited$
    end if
  end while
  return fs
End
```

Fig. 5. Forward sphere calculation

3. Starting at WS.I$_{failed}$, move to each Web service instance along individual forward edges from WS.I$_{failed}$ by marking it as "visited". Continue doing this until the end of the composition specification graph is reached, i.e., *END* node. In case of multiple forward edges leading out of WS.I$_{failed}$, this step should be implemented in parallel for each forward edge.

4. If WS.I$_{failed}$ belongs to a loop, then traverse the composition specification graph backward from WS.I$_{failed}$, until the beginning of the loop is reached, marking all the visited Web service instances as ”visited”. The semantics of the loop dictates that such a case needs to be considered, since control flow in a loop could flow via backward edges also.
5. The collection of Web service instances labeled ”visited” constitute now the *forward sphere* for WS.I$_{failed}$.

```
PROC FA(wsif,G)
Input: $wsif$ WebServiceInstanceFailed, $G$ graph
Output: -
Begin
  for wsx$\in fs$(wsif,G) do
    ▷ suspend all the Web service instances that are in the forward sphere
    if $TransactionalProperty$(wsx) $= retriable$ then
      $executeWS - Retriable.Policy_{suspended}$(wsx)
    else $executeWS - Compensatable.Policy_{suspended}$(wsx)
    end if
  end for
  while $CurrentNumberOfRetries$(wsif) $<$ $MaximumNumberOfRetries$(wsif)$orCurrentState$(wsfi) $\neq done$ do
    $Retryexecutionof$wsif
    $IncrementationofCurrentNumberOfRetriesof$wsif
    ▷ The CurrentState of $wsif$ is automatically updated after each execution
  end while
  if $CurrentState(wsif) = done$ then
    $resumetheexecutionofeach$wsx$fs$(wsif,G)
  else if $TransactionalProperty$(wsif) $= Retriable$ then
      $executeWS - Retriable.Policy_{aborted}$(wsif)
    else if $TransactionalProperty$(wsif) $= Compensatable$ then
        $executeWS - Compensatable.Policy_{compensated}$(wsif)
      end if
      for wsx$\in fs$(wsif,G) do
        if $TransactionalProperty$(wsx) $= retriable$ then
          $executeWS - Retriable.Policy_{aborted}$(wsx)
        end if
        if $TransactionalProperty$(wsx) $= compensatable$ then
          $executeWS - Compensated.Policy_{compensated}$(wsx)
        end if
      end for
    end if
  end if
End
```

Fig. 6. Forward strategy algorithm

Forward adaptation. Once the forward sphere for WS.I$_{failed}$ is calculated, the forward adaptation algorithm given in Fig. 6 is executed. It consists of the following:

1. While WS.I$_{failed}$ is being retried, all currently, running Web service instances in the forward sphere are to be suspended using either WS-Retriable.Policy$_{suspended}$ or WS-Compensatable.Policy$_{suspended}$, as the case maybe.
2. Retry executing WS.I$_{failed}$ until one of the following happens: (i) maximum number of retries is reached without success, or (ii) one of the retries succeeds.
3. If one of the retries of WS.I$_{failed}$ succeeds, then WS.I$_{failed}$ execution will be resumed as well as the execution of the currently running Web service instances in the forward sphere.
4. In case the retry of WS.I$_{failed}$ fails, or the maximum number of retries without success is reached, WS.I$_{failed}$ will be either aborted via WS-Retriable.Policy$_{aborted}$, or compensated via WS-Compensatable.Policy$_{compensated}$ followed by WS-Compensatable.Policy$_{not-activated}$, as per its transactional property.

4.(a) If WS.I$_{failed}$ is either aborted or compensated, the Web service instances in the forward sphere are then aborted (respectively, compensated) if they are retriable (respectively, compensatable) in the reverse order in which they were executed. This is implemented via the pairwise abortion (respectively, compensation) dependency between consecutively aborted (respectively, compensated) Web service instances, as listed in Section 3.2.

4.(b) Execution control now returns to the state before the occurrence of the exception. The composite Web service designer can redesign the rest of the specification composition by taking into account the changed situation after all the aborts and compensations.

4.3 Illustration of the Forward Strategy Using Amin Scenario

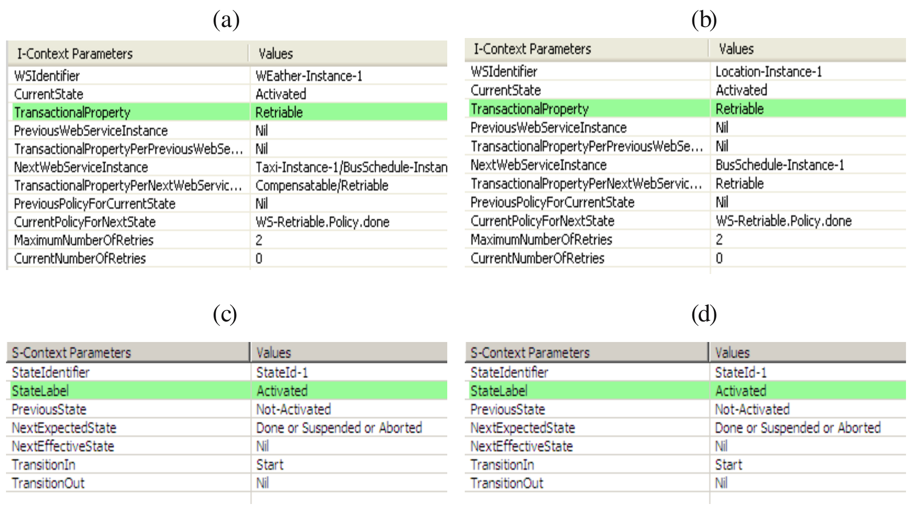
Let us assume that *Location WS* has failed, so its forward sphere consists of {*Bus Schedule WS*, *Traffic WS*, *Taxi WS*, *Weather WS*}.

If *Location WS* can be retried successfully, the execution will then proceed normally. If not, *Location WS* needs to be aborted. While it is being retried, *Weather WS* is kept suspended, until *Location WS* either succeeds or fails. In case *Location WS* fails, *Weather WS* should also be aborted, as per the abortion dependency between *Location WS* and *Weather WS* (Section. 3.2). This will lead to a redesign of the composition specification starting from *Itinerary WS*. Perhaps, during redesign, *Location WS* is associated with another Web service to be offered from within the hotel itself. This extra Web service will be triggered as per an alternate dependency policy.

5 Implementation

Our prototype is developed with the use of *JDK1.4.2* as a high level language, *W3C DOM* for processing XML information, *XACML* for transactional policies, *SWT* for GUI, and *Eclipse 3.2* as a development environment. Fig. 7 shows the initial $\mathcal{I}$-context and $\mathcal{S}$-context values of *Weather-Instance*$_1$ and *Location-Instance*$_1$ when *Itinerary WS* gets requested. In Fig. 7 (a), it can be seen that *Weather-Instance*$_1$ has got *retriable* as transactional property with *2* as *MaximumNumberOfRetries*, and its successor Web services are *Taxi WS* or *Bus Schedule WS* whose transactional properties are *compensatable* and *retriable*, respectively. In Fig. 7-(c) details on the state chart diagram of *Weather-Instance*$_1$ are given. For instance, the current state is *activated* while the transition in is *start*. Finally, Fig. 7-(b,d) show the initial $\mathcal{I}$-context and $\mathcal{S}$-context values of *Location-Instance*$_1$. In Fig. 7-(c,d) it is shown the same $\mathcal{S}$-context for both *Weather-Instance*$_1$ and *Location-Instance*$_1$. This is due to the following reasons: both have got the same transactional property namely retriable; both are getting activated at the same time since they are to be executed in parallel.

For prototyping purposes, we assume that *Location-Instance*$_1$ gets failed in the middle of execution. As a result, $\mathcal{S}/\mathcal{I}$-context's arguments get updated as per WS-Retriable.Policy$_{suspended}$. i.e., $\mathcal{S}$-context.*NextEffectiveState* gets changed to *suspended* and $\mathcal{I}$-context.*CurrentState* gets changed to *suspended*, too. Following the failure of *Location-Instance*$_1$, *Weather-Instance*$_1$ gets also suspended as per the adaptation strategy of Section 4.2. Since *Location-Instance*$_1$ has got *retriable* as transactional


(a)

I-Context Parameters	Values
WSIdentifier	WEather-Instance-1
CurrentState	Activated
TransactionalProperty	Retriable
PreviousWebServiceInstance	Nil
TransactionalPropertyPerPreviousWebSe...	Nil
NextWebServiceInstance	Taxi-Instance-1/BusSchedule-Instan
TransactionalPropertyPerNextWebServic...	Compensatable/Retriable
PreviousPolicyForCurrentState	Nil
CurrentPolicyForNextState	WS-Retriable.Policy.done
MaximumNumberOfRetries	2
CurrentNumberOfRetries	0

(b)

I-Context Parameters	Values
WSIdentifier	Location-Instance-1
CurrentState	Activated
TransactionalProperty	Retriable
PreviousWebServiceInstance	Nil
TransactionalPropertyPerPreviousWebSe...	Nil
NextWebServiceInstance	BusSchedule-Instance-1
TransactionalPropertyPerNextWebServic...	Retriable
PreviousPolicyForCurrentState	Nil
CurrentPolicyForNextState	WS-Retriable.Policy.done
MaximumNumberOfRetries	2
CurrentNumberOfRetries	0

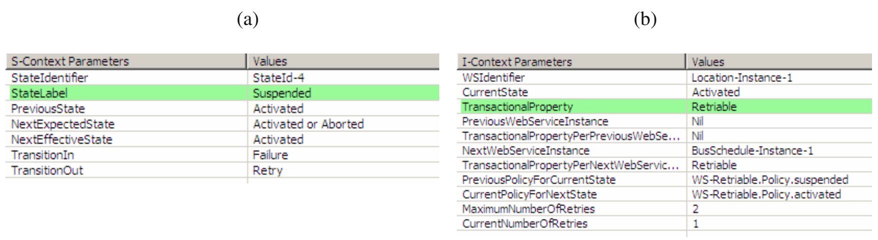
(c)

S-Context Parameters	Values
StateIdentifier	StateId-1
StateLabel	Activated
PreviousState	Not-Activated
NextExpectedState	Done or Suspended or Aborted
NextEffectiveState	Nil
TransitionIn	Start
TransitionOut	Nil

(d)

S-Context Parameters	Values
StateIdentifier	StateId-1
StateLabel	Activated
PreviousState	Not-Activated
NextExpectedState	Done or Suspended or Aborted
NextEffectiveState	Nil
TransitionIn	Start
TransitionOut	Nil

Fig. 7. $\mathcal{I}/\mathcal{S}$-contexts of *Weather-Instance*$_1$ and *Location-Instance*$_1$


(a)

S-Context Parameters	Values
StateIdentifier	StateId-4
StateLabel	Suspended
PreviousState	Activated
NextExpectedState	Activated or Aborted
NextEffectiveState	Activated
TransitionIn	Failure
TransitionOut	Retry

(b)

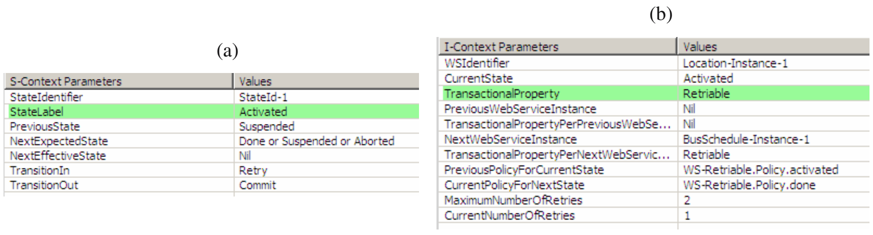
I-Context Parameters	Values
WSIdentifier	Location-Instance-1
CurrentState	Activated
TransactionalProperty	Retriable
PreviousWebServiceInstance	Nil
TransactionalPropertyPerPreviousWebSe...	Nil
NextWebServiceInstance	BusSchedule-Instance-1
TransactionalPropertyPerNextWebServic...	Retriable
PreviousPolicyForCurrentState	WS-Retriable.Policy.suspended
CurrentPolicyForNextState	WS-Retriable.Policy.activated
MaximumNumberOfRetries	2
CurrentNumberOfRetries	1

Fig. 8. Updated $\mathcal{I}/\mathcal{S}$-contexts of *Location-Instance*$_1$ after successful retry

property, it gets retried with *WS-Retriable.Policy*$_{activated}$ transactional policy. Luckily the first attempt of retry itself is successful. Fig. 8 shows *Location-Instance*$_1$'s $\mathcal{I}/\mathcal{S}$-contexts with focus on *NextEffectiveState*, *CurrentState*, and *CurrentNumberOfRetries* arguments.

Next, Fig. 9 shows the further updated $\mathcal{I}/\mathcal{S}$-contexts of *Location-Instance*$_1$. For $\mathcal{S}$-context (Fig. 9-(a)), the state identifier, state label, previous state, next expected state, and transition out values are modified with respect to the current state, i.e., Activated. Same comment is made for $\mathcal{I}$-context's arguments (Fig. 9-(b)).

Once *Location-Instance*$_1$ gets activated, *Weather-Instance*$_1$ is also retried and successfully activated. Its respective $\mathcal{I}/\mathcal{S}$-contexts are updated with respect to its *WS-Retriable.Policy*$_{activated}$ policy. Eventually, both *Weather-Instance*$_1$ and *Location-Instance*$_1$ successfully complete execution. Finally, after the completion of *Weather-Instance*$_1$, *Bus-Schedule-Instance*$_1$ invoked, which in turn invokes *Traffic-Instance*$_1$ for obtaining traffic information. Eventually, both *Bus-Schedule-Instance*$_1$ and *Traffic-Instance*$_1$ successfully complete execution.



(a)

S-Context Parameters	Values
StateIdentifier	StateId-1
StateLabel	Activated
PreviousState	Suspended
NextExpectedState	Done or Suspended or Aborted
NextEffectiveState	Nil
TransitionIn	Retry
TransitionOut	Commit

(b)

I-Context Parameters	Values
WSIdentifier	Location-Instance-1
CurrentState	Activated
TransactionalProperty	Retriable
PreviousWebServiceInstance	Nil
TransactionalPropertyPerPreviousWebSe...	Nil
NextWebServiceInstance	BusSchedule-Instance-1
TransactionalPropertyPerNextWebServic...	Retriable
PreviousPolicyForCurrentState	WS-Retriable.Policy.activated
CurrentPolicyForNextState	WS-Retriable.Policy.done
MaximumNumberOfRetries	2
CurrentNumberOfRetries	1

Fig. 9. Further updated (as per Activated state) $\mathcal{I}/\mathcal{S}$-contexts of *Location-Instance*$_1$

6 Conclusion

In this paper, we presented an approach to develop context-driven transactional Web services. We defined transactional properties on Web services that permit them to be composed together, and their joint execution managed, via policies. We also discussed how this approach helps handle exceptions, via the application of an adaptation strategy.

Our future work concerns a more thorough experimentation and evaluation activity to compare our approach against some other approaches presented in the literature [1,3,4,6,14,15]. In particular, we plan to demonstrate the feasibility of our approach on larger examples. Another interesting future work concerns the definition of a composite Web services framework that can manage transactional properties and ensure substitution mechanisms. This substitution can play an important role, mainly for pivot Web services. Some preliminary results are already reported in [13]. Finally, we plan to study how some fault tolerance concepts of distributed systems can be adapted to the requirements of transactional Web services.

References

1. Alrifai, M., Dolog, P., Nejdl, W.: Transactions Concurrency Control in Web Service Environment. In: Proceedings of The 4th IEEE European Conference on Web Services (ECOWS'2006), Zurich, Switzerland (2006)
2. Anderson, A.H.: An Introduction to The Web Services Policy Language (WSPL). In: Proceedings of The 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'2004), New-York, USA (2004)
3. Bassil, S., Rinderle, S., Keller, R., Kropf, P., Reichert, M.: Preserving the Context of Interrupted Business Process. In: Proceedings of The 7th International Conference on Enterprise Information Systems (ICEIS'2005), Miami, USA (2005)
4. Bhiri, S., Perrin, O., Godart, C.: Ensuring Required Failure Atomicity of Composite Web Services. In: Proceedings of The Fourteenth International World Wide Web Conference (WWW'2005), Chiba, Japan (2005)
5. Coutaz, J., Crowley, J.L., Dobson, S., Garlan, D.: Context is Key. Communications of the ACM, 48(3) (March 2005)
6. Fauvet, M.-C., Duarte, H., Dumas, M., Benatallah, B.: Handling Transactional Properties in Web Service Composition. In: Proceedings of The 6th International Conference on Web Information Systems Engineering, (WISE'2005), New-York, USA (2005)
7. Maamar, Z., Benslimane, D., Anderson, A.: Using Policies to Manage Composite Web Services. IEEE IT Professional, 8(5) (September/October 2006)

8. Maamar, Z., Benslimane, D., Narendra, N.C.: What Can Context do for Web Services? Communications of the ACM, 49(12) (December 2006)
9. Maamar, Z., Mostéfaoui, S.K., Yahyaoui, H.: Towards an Agent-based and Context-oriented Approach for Web Services Composition. IEEE Transactions on Knowledge and Data Engineering, 17(5) (May 2005)
10. Pires, P.F., Benevides, M.R.F., Mattoso, M.: Building Reliable Web Services Compositions. In: Proceedings of The International Workshop on Web Services Research, Standardization, and Deployment (WS-RSD'2002), Erfurt, Germany (2002)
11. Reichert, M., Dadam, P.: ADEPTflex - Supporting Dynamic Changes of Workflows without Losing Control. Journal of Intelligent Information Systems, 10(2) (1998)
12. Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Exception Handling Patterns. In: Process-Aware Information Systems. Technical report, BPM Center Report BPM-06-04, BPMcenter.org. (2006)
13. Taher, Y., Benslimane, D., Fauvet, M.-C., Maamar, Z.: Towards an Approach for Web Services Substitution. In: Proceedings of The 10th International Database Engineering & Applications Symposium (IDEAS'2006), Delhi, India (2006)
14. Verma, M., Deswal, P.: Approaching Web Services Transactions. Technical report, Second Foundation Inc., February 2003. Visited (February 2005) http://www-128.ibm.com/developerworks/webservices/library/ws-tranart
15. Younas, M., Chao, K.M., Lo, C.C., Li, Y.: An Efficient Transaction Commit Protocol for Composite Web Services. In: Proceedings of The IEEE 20th International Conference on Advanced Information Networking and Applications (AINA'2006), Vienna, Austria (2006)
16. Younas, M., Eaglestone, B., Chao, K.M.: A Low Latency Resilient Protocol for E-Business Transactions. International Journal of Web Engineering and Technology, 1(3) (2004)