In *Proceedings of the 22^{nd} IEEE Symposium on Reliable Distributed Systems*, Florence, Italy, October 2003.

Coordinated Forward Error Recovery for Composite Web Services

F. Tartanoglu, V. Issarny
INRIA Rocquencourt
78153, Le Chesnay, France
Galip-Ferda.Tartanoglu@inria.fr
Valerie.Issarny@inria.fr

A. Romanovsky
University of Newcastle
upon Tyne
NE1 7RU, UK
Alexander.Romanovsky@newcastle.ac.uk

N. Levy
Université de Versailles
Saint-Quentin
78035 Versailles, France
Nicole.Levy@prism.uvsq.fr

Abstract

This paper proposes a solution based on forward error recovery, oriented towards providing dependability of composite Web services. While exploiting their possible support for fault tolerance (e.g., transactional support at the level of each service), the proposed solution has no impact on the autonomy of the individual Web services, Our solution lies in system structuring in terms of co-operative atomic actions that have a well-defined behaviour, both in the absence and in the presence of service failures. More specifically, we define the notion of Web Service Composition Action (WSCA), based on the Coordinated Atomic Action concept, which allows structuring composite Web services in terms of dependable actions. Fault tolerance can then be obtained as an emergent property of the aggregation of several potentially non-dependable services. We further introduce a framework enabling the development of composite Web services based on WSCAs, consisting of an XML-based language for the specification of WSCAs.

1 Introduction

Systems that build upon the Web services architecture are expected to become a major class of wide-area open distributed systems in the near future. The Web services architecture targets the development of applications based on XML-based standards, hence easing the development of distributed systems through the dynamic integration of applications distributed over the Internet, irrespective of their underlying platforms. However, the provision of effective support for the dependable integration of Web services is still an open issue, which has led to tremendous research effort over the last few years, in both industry and academia (e.g., [1, 3, 9, 12, 13, 23]).

1.1 Composition of Web services

Although the definition of the overall Web services architecture is still incomplete, the base standards have already emerged from the W3C[1], which define a core middleware for Web services, partly building upon results from object-based and component-based middleware technologies. These standards relate to the specification of Web services and a supporting interaction protocol. SOAP (Simple Object Access Protocol) defines a lightweight protocol for information exchange that sets the rules of how to encode data in XML as well as the SOAP mapping to an Internet transport protocol (e.g., HTTP) [18]. Specification of Web service interfaces relies on the WSDL (Web Services Description Language) [19] declarative language that is used to specify: (i) the service's abstract interface that describes the messages exchanged with the service, and (ii) the concrete binding information that contains specific protocol-dependent details including the network end-point address of the service. Complementary to the above core middleware for the integration of Web services is UDDI (Universal Description, Discovery and Integration); this specifies a registry for dynamically locating and advertising Web services [14].

Composing Web services relates to dealing with the assembly of existing services so as to deliver a new service out of them, given the corresponding published interfaces (see Figure 1). Integration of Web services is then realized according to the specification of the overall process composing the Web services. The process specifying the composition must actually not solely define the functional behaviour of the process in terms of interactions with the composed services, but also the process's non-functional properties, possibly exploiting middleware-related services. Various non-functional properties (e.g., availability, extendibility, reliability, openness, performance, security, scalability) should be accounted for in the context of Web services.

[1] World Wide Web Consortium, http://www.w3.org

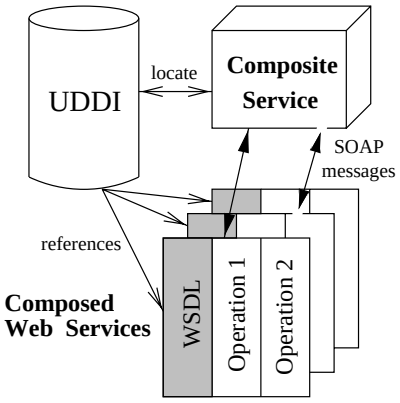

Figure 1. Web services and their composition

However, enforcing dependability of composite Web services is one of the most challenging issues due to the concern for supporting business processes, combined with the fact that the composition process deals with the assembly of loosely-coupled autonomous components.

1.2 Dependability of composite Web services

In general, Web services run on heterogeneous platforms, and are autonomous components with different characteristics (e.g., transactional supports, concurrency policies, access rights). Moreover, Web services can use different transport protocols (e.g., HTTP, SMTP) and interacting with them requires dealing with limitations of the Internet, which is not a reliable media [7]. In addition, availability is a major issue in addressing dependability in Web service applications. Web services may be unavailable for an unknown reason and for an unknown amount of time. Moreover, the overall network status and server loads may cause extensively long delays on responses from Web service servers.

Composite Web services have high dependability requirements that call for dedicated fault tolerance mechanisms due to both the specifics of the Web services architecture and limitations of the Internet. The autonomy of component Web services raises challenging issues in specifying composition processes and in particular behaviour of composite services in the presence of faults. These faults include but are not limited to (i) faults occurring at the level of the Web services, which may be notified by error messages, (ii) faults at the underlying platform (e.g., hardware faults, timeouts), and (iii) faults due to online upgrades of component services and/or of their interfaces. These specifics of Web services require special care in the design of supporting fault tolerance mechanisms, which is the focus of our paper.

1.3 Contributions

This paper introduces a solution based on forward error recovery, for making composite Web services fault tolerant. The proposed solution has no impact on the autonomy of the individual Web Services, while exploiting their possible support for fault tolerance (e.g., transaction support at the level of each service). We define the notion of Web Service Composition Action (WSCA), which allows structuring composite Web services in terms of coordinated atomic actions that have a well-defined behaviour, both in the absence and in the presence of service failures. We then introduce an XML-based language for the specification of WSCAs. The language is used to define fault tolerant composite Web services and provides the application logic.

Section 2 presents proposed fault tolerance mechanisms for composite Web services, identifying limitations of mechanisms based on backward error recovery. This leads us to introduce Web Services Composition Actions (WSCA) for structuring the composition of Web services into fault tolerant functional units in Section 3. The associated development process and deployment of composite Web services is described in Section 4. Section 4.2 then defines the XML-based WSCA Language (WSCAL) to be used for specifying Web services composition based on WSCA, which may further be exploited for the automatic generation of the implementation of composite services. Finally, Section 5 summarises our contribution, and discusses our current and future work.

2 Fault tolerance mechanisms for Web services

The choice of fault tolerance techniques to be exploited for the development of dependable systems depends very much on the fault assumptions and on the system's characteristics and requirements (Section 2.1). Developing fault tolerant mechanisms for composite Web services has been an active area of research over the last couple of years (see [16] for a survey). Existing proposals mainly exploit backward error recovery, and more specifically, transactions (Section 2.2). However, the autonomy of Web services and the Web latency have led to exploit more flexible transactional models and forward error recovery techniques (Section 2.3).

2.1 Backward versus forward error recovery

There are two main classes of error recovery [8]: backward (based on rolling system components back to the previous correct state) and forward error recovery (which involves transforming the system components into any correct state). The former uses either diversely-implemented

software or simple retry; the latter is usually application-specific and relies on an exception handling mechanism [4].

It is a widely-accepted fact that the most beneficial way of applying fault tolerance is by associating its measures with system structuring units as this decreases system complexity and makes it easier for developers to apply fault tolerance [15]. Structuring units applied for both building distributed systems and providing their fault tolerance are well-known: they are *distributed transactions* and *atomic actions* (also referred to as conversations). Distributed transactions [5] use backward error recovery as the main fault tolerance measure in order to satisfy completely or partially the ACID (atomicity, consistency, isolation, durability) properties. Atomic actions [2] allow programmers to apply both backward and forward error recovery. The latter relies on coordinated handling of action exceptions that involves all action participants. Backward error recovery has a limited applicability, and in spite of all its advantages, modern systems are increasingly relying on forward error recovery, which uses appropriate exception handling techniques as a means [4]. Examples of such applications are complex systems involving human beings, COTS components, external devices, several organizations, movement of goods, operations on the environment, real-time systems that do not have time to go back. Integrated Web services clearly falls into this category.

2.2 Backward error recovery for the Web

Transactions have been proven successful in enforcing fault tolerance in closed distributed systems and are extensively exploited for the implementation of primitive (non-composite) Web services. However, transactions are not suited for making the composition of Web services fault tolerant in general, for at least two reasons. First, the management of transactions that are distributed over Web services requires cooperation among the transactional supports of individual Web services, which may not be compliant with each other and may not be willing to do so given their intrinsic autonomy and the fact that they span different administrative domains. Second, locking resources until the termination of the embedding transaction is in general not appropriate for Web services, still due to their autonomy, and also to the fact that they potentially have a large number of concurrent clients that will not stand extensive delays.

Enhanced transactional models have been considered to alleviate the latter shortcoming. In particular, the split model (also referred to as open-nested transactions) where transactions may split into a number of concurrent sub-transactions that can commit independently allows reducing the latency due to locking. Typically, sub-transactions are matched to the transactions already supported by Web services (e.g., transactional booking offered by a service). Hence, transactions over composite services do not increase the access latency as offered by the individual services. Enforcing the atomicity property over a transaction that has been split into a number of sub-transactions then requires using compensation over committed sub-transactions in the case of transaction abortion. However, to support this, Web services should provide compensating operations for all the operations they offer. Such an issue is in particular addressed by the BPEL4WS [9] and WSCI [20] languages for specifying services, which allow defining compensating operations associated with the services' operations. It is worth noting that using compensation for aborting distributed transactions must extend to all the participating Web services (i.e., cascading compensation by analogy with cascading abort). Such a concern is addressed in [12]. This paper introduces a middleware whose API may be exploited by clients of a composite service for specifying and executing a (open-nested) transaction over a set of Web services whose termination is dictated by the outcomes of the transactional operations invoked on the individual services.

In addition to client-side solutions to the coordination of distributed open-nested transactions, work is undertaken in the area of distributed transaction protocols supporting the deployment of transactions over the Web, while not imposing long-lived locks over Web resources. These include BTP (Business Transaction Protocol) [13] and WS-Transaction [11]. The BTP protocol introduces the notion of cohesion, which allows non-ACID transactions to be defined by not requiring successful termination of all the transaction's participants for committing.

WS-Transaction defines a specialization of WS-Coordination [10], which is an extensible framework for specifying distributed protocols that coordinate the execution of Web services, and that can be used in conjunction with BPEL4WS. The *business activity* protocol defined in WS-Transaction specifically serves coordinating the execution of open-nested transactions over a set of business processes.

2.3 Forward error recovery for the Web

In addition to backward error recovery, forward error recovery, using exception handling mechanisms, is extensively exploited in the specification of composite Web services in order to handle error occurrences. For instance, in BPEL4WS [9], exception handlers (referred to as fault handlers) can be associated to a (possibly nested) activity so that when an error occurs inside an activity, its execution terminates, and the corresponding exception handler is executed. However, when an activity is defined as a concurrent process and at least one embedded activity signals an exception, all the embedded activities are terminated as soon as one signaled exception is caught, and only the handler for

this specific exception is executed. Hence, error recovery actually accounts for a single exception and thus cannot ensure recovery of a correct state. The only case where correct state recovery may be ensured is when the effect of all the aborted activities are rolled back to a previous state, which may not be supported in general in the context of Web services, as discussed in the previous section. The shortcoming of BPEL4WS actually applies to all XML-based languages for Web services composition that integrate support for specifying concurrent activities and exception handling.

3 Web service composition actions

Our solution to the above shortcoming of forward error recovery approaches for composite Web services is based on the notion of *coordinated atomic actions* that provides a sound basis for dealing with concurrent exceptions (Section 3.1). However, the notion requires to be adapted to cope with the fault model (Section 3.2) and the specifics of Web services, leading to introduce WSCA (Section 3.3). Illustration of using WSCA is further illustrated using the classical *travel agency* case study (Section 3.4).

3.1 Coordinated atomic actions

The Coordinated Atomic Action (or CA action) concept [21] is a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting components. It provides a conceptual framework for dealing with different kinds of concurrency and achieving fault tolerance by extending and integrating two complementary concepts – atomic actions and ACID transactions. Atomic actions are used to control cooperative concurrency and to implement coordinated error recovery whilst ACID transactions are used to maintain the consistency of shared resources. A CA action is designed as a set of *participants* cooperating inside it and a set of resources accessed by them (see Figure 2).

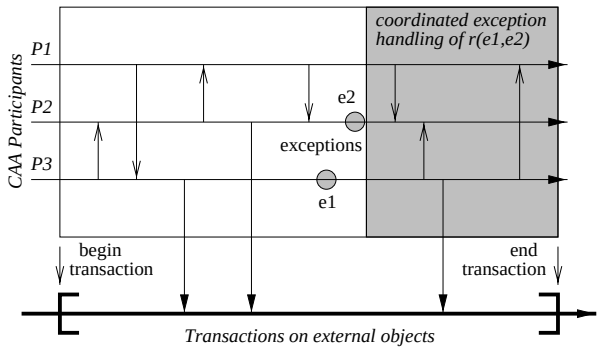

Figure 2. Coordinated atomic actions

In the course of the action, *participants* can access external resources that have ACID properties. Action *participants* either reach the end of the action and produce a normal outcome and commit transactions on external resources or, if one or more exceptions are raised, all the *participants* are involved in their coordinated handling. If several exceptions have been raised concurrently, they are resolved using a resolution tree imposing a partial order on all action exceptions, and all the *participants* handle the resolved exception [2]. If this handling is successful, the action completes normally, but if handling is not possible then an exception is propagated and responsibility for recovery is passed to the caller of the action, while transactions on all external resources are aborted. A formal specification of CA actions can be found in [17].

3.2 Fault model of Web services

The CA action concept for tolerating faults is based on exception handling. It requires the detection of faults, and the mapping of these faults to anticipated exceptions for which handlers are to be defined. Several kinds of errors can occur during the execution of a composite Web service and can be detected at the composition level. These faults relate to: (i) system faults like crashes, timeouts or network errors that can be detected by specialized monitors that generate exceptions at the application level, and (ii) application-level faults. The latter includes a broad range of errors like system exceptions, programmed exceptions, exceptions propagated from other participants, fault messages that are received from SOAP calls and transport level faults (e.g., HTTP errors). Note that the semantics of exceptions is application-dependant. For instance, a timeout does not necessarily mean that a remote system has crashed, and an application-specific handler must be provided to handle this kind of error.

3.3 Adapting CA actions for the Web

CA actions provide a base structuring mechanism for developing fault tolerant composite Web services: a CA action specifies the collaborative realization of a given function by composed services, and Web services correspond to external resources. However, as raised previously, ACID properties over external resources are not suited in the case of Web services. We therefore introduce the notion of Web Service Composition Action (WSCA) that adapts CA actions to the specifics of Web services. WSCAs mainly differ from CA actions in relaxing the transactional requirements over external resources, which are not suitable for wide-area open systems.

WSCA participants specify interactions with composed Web services, stating the role of each Web service in the

composition. Each Web service is viewed as an external resource. However, unlike the base CA action model, interactions do not have to be transactional. The interactions adhere to the semantics of the Web service operations that are invoked. An interaction may then be transactional if the given operation that is called is. However, transactions do not span multiple interactions.

Every WSCA participant further specifies actions to be undertaken when the Web services with which it interacts signal an exception, which may be either handled locally to the participant (system-level exceptions or programmed exceptions signaled by Web services operations that do not need to be cooperatively handled at the WSCA level) or be propagated to the level of the embedding WSCA. The latter then leads to co-operative exception handling according to the exceptional specification of the WSCA. When Web services offer compensating operations, the resulting forward recovery may realize a relaxed form of atomicity. If operations are compensated, the action itself will be atomic in the sense that it will restore its initial state, assuming that the compensating operations succeed. However, the external actions that accessed the service in an intermediate state might have invalidated values.

Compared to solutions that primarily rely on transactional supports for composite Web services, ours mainly differs in that it exploits forward error recovery at the composition level, while enabling exploitation of transactional supports offered by individual Web services, –if any. For many applications, the strict requirement for transactionality can indeed be substitued by the optional mechanisms offered by the underlying services. These concern services offering compensating operations. Hence, the underlying protocol for interaction among Web services remains the one of the Web services architecture (i.e., SOAP) and does not need to be complemented with a distributed transaction protocol. Similarly to our solution, the one of [12] does not require any new protocol to support distributed open-nested transactions. An open-nested transaction is declared on the client side by grouping transactions of the individual Web services, through call to a dedicated function of the middleware running on the client. The transaction then gets aborted by the middleware using compensation operations offered by the individual Web services, according to conditions set by the client over the outcomes of the grouped transactions. Our solution is then more general since we allow forward error recovery involving several Web services to be specified at the composition level, enabling in particular to integrate non-transactional Web services while still supporting dependability of the composite service. Fault tolerance can then be obtained as an emergent property of the aggregation of some potentially non-dependable services, based on the detection of faults and their mapping to application-dependant exceptions.

3.4 WSCA example: the travel agency

For illustration of WSCA, we consider joint booking of accommodation and flights using separate hotel and flight Web services. Then, the composite Web service's operation is specified using WSCAs as follows. The top-level *TravelAgent* WSCA comprises the *User* and the *Travel* participants; the former interacts with the user while the latter achieves joint booking according to the user's request through call to the WSCA that composes the *Flight* and the *Hotel* participants. A diagrammatic specification of the aforementioned WSCAs is shown in Figure 3.

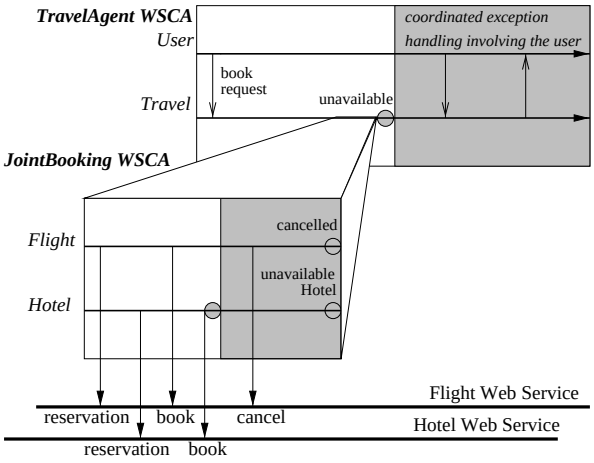

Figure 3. WSCA for composing Web services

In the *TravelAgent* WSCA, the *User* participant requests the *Travel* participant to book a flight ticket and a hotel room for the duration of the given stay. This leads the *Travel* participant to invoke the *JointBooking* WSCA that composes the *Hotel* Web service and the *Flight* Web service. The participants of the *JointBooking* WSCA respectively requests for a hotel room and a flight ticket, given the travel itinerary and dates provided by the user. Each request is subdivided into reservation for the given period and subsequent booking if the reservation succeeds. If both the *Hotel* and the *Flight* participants raise the *unavailable* exception, then concurrent exception resolution applies, leading the *JointBooking* WSCA to signal an exception to the *Travel* participant of the *TravelAgent* WSCA, where the exception gets handled in a cooperation with the *User* denoted by the greyed box in the figure (e.g., the user is requested to give alternative dates). In the case where either the reservation or the booking of one of the participants fails, the participant raises an exception (*unavailableHotel* in the example) that is cooperatively handled at the level of the *JointBooking* WSCA. If one participant has already confirmed a reservation, the booking that has succeeded is cancelled and an *un-*

available exception is signaled to the calling *TravelAgent* WSCA for recovery with user intervention (see Figure 3). Note that if the handling fails and the cancellation is not possible, the exception that is signaled is a failure exception sent to the calling WSCA with sufficient information about failed operations for later recovery.

4 Building WSCA-based composite services

Given the notion of WSCA, composite Web services are built by specifying their operations as WSCAs. The Web services that are composed are further declared in an abstract way to enable more flexible integration of different Web service instances through dynamic binding.

4.1 Composition process

The development process of a WSCA-based composite service comprises the definitions of: (i) the required abstract interfaces of the composed Web services that are to be integrated, and (ii) the composition process that gives the binding information as well as the WSCAs' standard and exceptional behaviours.

Required interfaces of components services to be integrated are declared. The abstract interface of any such service is given in terms of WSDL (limited to the abstract part), which defines the messages exchanged with the Web service. The service interface is further enriched with the characterization of the service's transactional behaviour if offered, in a way similar to existing solutions in the area (e.g., [12],[20]).

The composition process of WSCA-based composite services is expressed as an independent XML document, which specifies the service instances to be integrated together with provided WSCAs, as detailed in Section 4.2. The Web service instance associated to a given composed service may be either statically set or dynamically retrieved according to the service's abstract specification. The only requirement that is demanded from the Web service instances is the conformance of their interfaces with the declared required interfaces. In the most general case, the conformance is verified by syntactically checking WSDL documents of Web services. To provide availability, we allow a participant to be bound to a set of Web service instances implementing the service's specification. WSCAs define the operations provided by the composite Web service. The definition of a WSCA specifies the behaviour of the WSCA's participants including the exception handlers that are executed when an exception occurs.

The WSCA-based composite services are deployed as Web services on top of a base Web service middleware (see Figure 4). Services that provide some WSCA composite services, are referenced in a UDDI Web services registry for client use. Given base middleware for Web services, supporting the deployment of WSCA-based composite Web services requires in addition, a local runtime support for running WSCAs. The runtime support should include mechanisms for WSCA creation, synchronization of participants, exception detection and propagation, and dynamic binding of Web services.

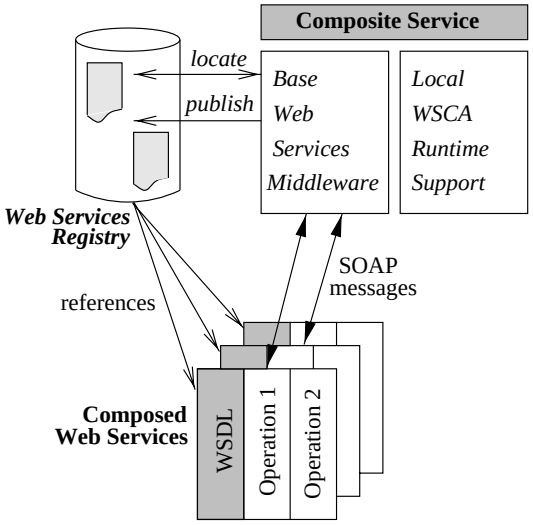

Figure 4. WSCA deployment

4.2 Specifying composite services

This section introduces an XML-based language for the specification of WSCA-based composite Web services, which defines the graph of interactions among Web services. The actual implementation of the service composition may then be decoupled from the specification of the composition process, leaving under the responsibility of the developer to check that his/her implementation conforms to the service's specification. Alternatively, the implementation may be generated from the specification.

The specification of a composite Web service is given by the WSCAL (Web Service Composition Action Language) XML-based language which includes declarations of the service instances and the behaviour of the supported operations:

```
<WSCAL name=nmtoken>
  <services>
    Detailed below
  </services>
  <WSCA operation=nmtoken exceptionTree=anyURI>*
    Detailed below
  </WSCA>
</WSCAL>
```

4.2.1 Service instances

The service instances are given through the *services* element that refers to the corresponding abstract interfaces of composed Web services:

```
<services>
  <service name=qname>*
    <definition hrefSchema=anyURI />
    <staticService instance=anyURI />* |
    <dynamicService onCall=boolean multiple=boolean />?
  </service>
</services>
```

Each service may be statically bound to a specific service instance (defined by the *staticService* element) and/or dynamically bound to an instance matching the abstract definition of the service interface that is given by the corresponding *definition* attribute (defined by the *dynamicService* element). In the former case, concrete binding information is provided through the WSDL document associated with the service's instance. In the latter case, a matching service instance is located at runtime using a location service such as a UDDI service. Dynamic binding of participants with associated Web services may take place either upon first invocation of the service's WSCAs or upon instantiation of the composite Web service, according to the value of the *onCall* Boolean attribute of the given participant. Finally, we allow each service to be bound to a set of instances matching the specification of the associated service rather than a single instance for the sake of availability; this is specified using the *multiple* Boolean attribute in the *DynamicService* element and by stating as many instances as required with the *staticService* element. Then, a unique Web service that is available is chosen for the whole WSCA at the first invocation of the corresponding service. Note that if the Web service becomes unavailable, an exception is raised and another Web service can be chosen if it is explicitly specifed in the corresponding handler.

A sample of the *services* element for the *TravelAgent* composite service is given below, which directly follows from the informal presentation of Section 3.4. The service in particular offers the *JointBooking* WSCA that coordinates booking over the *Hotel* and *Flight* Web services, for which instances are dynamically retrieved upon invocation of the WSCA.

```
<services>
  <service name=''UserBrowser''>
    <definition hrefSchema=''http://ta.com/TAUser.req''/>
    <staticService instance=''http://ta.com/Client.wsdl''/>
  </service>
  <service name=''FlightService''>
    <definition hrefSchema=''http://ta.com/Flight.req''/>
    <dynamicService onCall=true multiple=true />
  </service>
  <service name=''HotelService''>
    <definition hrefSchema=''http://ta.com/Hotel.req''/>
    <dynamicService onCall=true multiple=true />
  </service>
</services>
```

4.2.2 WSCA behaviour

The behaviour of the operations offered by the composite service is defined by the *WSCA* element:

```
<WSCA operation=nmtoken exceptionTree=anyURI>*
  <participant name=nmtoken>*
    <bind service=qname/>*
    <input name=nmtoken ?  message=qname/>
    <output name=nmtoken ?  message=qname/>
    <fault name=nmtoken message=qname/>*
    <state>?
      <xsd:schema .../>*
    </state>
    <behavior>
      <standard>
        Statements ...
      </standard>
      <coordinatedHandler exception=qname>*
        Statements ...
      </coordinatedHandler>
    </behavior>
  </participant>
</WSCA>
```

In the above, the *operation* attribute gives the name of the operation being specified and the *exceptionTree* attribute gives the XML document of the corresponding exception resolution tree. The exception tree is used to resolve the exceptions that are concurrently raised within WSCAs into a single exception, in a way similar to CA actions. Then, a sequence of *participant* elements are declared, each specifying:

- **The services** with which the WSCA participant interacts through the *bind* element that gives the name of the services among those defined by the *service* elements in the embedding *services* element.
- **Parts of the messages** associated with the WSCA that are relevant to the specific participant, which is defined using the *input, output* and *fault* elements. The *fault* elements define the exceptions that may be raised by the participants, which require cooperative exception handling and get composed with the exceptions concurrently raised by peer participants using the exception tree.
- **The local state** of the specific participant through the *state* element that defines the local variables.
- **The behaviour** of the specific participant using the *behavior* element. The participant behaviour subdivides into the participant's standard and exceptional behaviour (as defined by the *standard* and *coordinatedHandler* elements). Each such behaviour is defined as a process using classical statements, including in particular interaction with the Web service instance(s)

associated with the participant, message exchanges with peer participants and exception handling.

WSCAL statements for specifying participants' behaviour are quite similar to the ones introduced by XML-based languages for the specification of composite services. We more specifically base the definition of WSCAL on the CSP language [6] for the base statements, providing a sound basis towards formal reasoning about WSCAL specifications. We thus detail here only some base statements of interests regarding WSCAs. The *call* statement allows specifying (synchronous) operation calls where the invoked operation may be either local to the embedding composite Web service or provided by the Web services to which the participant is bound (which may be a WSCA if the service is itself composite). The *return* statement is the dual statement allowing specifying the message to be returned as partial result of the embedded operations, which is to be merged with the results returned by peer participants. The *send* statement allows specifying the sending of a message to a peer participant whose dual reception may be expressed using either the blocking *wait* or the non-blocking *onInput* statement. Finally, the *raise* statement allows signaling an exception whose handling is specified using the traditional *try* statement for defining exception handling scopes:

```
<try>
  Statements ...
</try>
<localHandler exception=qname>*
  Statements ...
</localHandler>
```

4.3 Coordinated exception handling

The specifics of WSCAL comes from structuring the operations provided by composite Web services as WSCAs that coordinate the execution of composed Web services operations with respect to failure occurrences, in particular introducing the specification of coordinated exception handling.

For each *participant*, the *fault* elements define the exceptions that may be raised and require cooperative exception handling. In addition, a default exception handler is defined for exceptions that are not declared. When a participant raises an exception, it is first handled locally by a local handler, defined using the *localHandler* element, while other participants continue their execution normally without being interrupted. If the local handler fails, or if such a handler is not defined, the participant ultimately raises the exception, which is propagated to all participants, leading to coordinated exception handling, defined using the corresponding *coordinatedHandler* element. Participants synchronize for execution of the coordinated exception handlers. That means in particular that if a participant is engaged in a remote call, all participants wait for the termination of this call. In the case where several participants raise exceptions which are concurrently propagated, then the exceptions are resolved using the document defined in the *exceptionTree* attribute, which imposes a partial order on all exceptions, and all participants handle this unique exception. The exception is then handled cooperatively by participants that can coordinate their execution in the same way as in the standard behavior, i.e., by message passing, and they can invoke external WSCAs if needed. The corresponding exception resolution algorithm can be found in [22].

The WSCA that executes a coordinated exception handler terminates with one of the following outcomes [21]:

- **Exceptional outcome:** All external operations on Web services and nested WSCAs are performed successfully (e.g., all transactions are validated). An exception is signaled to the upper level.
- **Abort outcome:** The WSCA has aborted and all external operations are aborted or compensated according to the transactional behavior of the accessed Web services as specified in the service interface. An abort exception is signaled to the upper level:
- **Failure outcome:** The handling has failed and external operations cannot be aborted or compensated. This can be due to a failure in performing the cancellation operation or if such an operation is not defined. Responsibility for recovery is passed to the caller of the action. A failure exception is signaled to the upper level with information about failed operations:

Adopting the concept of coordinated exception handling of CA actions allows dealing with different types of faults in a disciplined way inside atomic units and partial results of nested actions to be reported to the higher level action, keeping error propagation under control.

4.4 The travel agency example

A sample of the *participant* element specifying the *Hotel* participants behaviour is given below. An example entry of the exception tree document is:

```
<resolve name=''unavailable''>
  <exception name=''unavailableHotel''/>
  <exception name=''unavailableFlight''/>
</resolve>
<resolve name=''unavailable''>
  <exception name=''unavailableHotel''/>
  <exception name=''cancelled''/>
</resolve>
```

Flight participants behavior is not detailed as it is similar to the *Hotel* participant. Coordinated booking is achieved as discussed in Section 3.4, exploiting in particular open-nested transactions of participating Web

services. The two participants of the *JointBooking* WSCA have similar behaviour. The participant first invokes the *reserve* operation of the Web service to which it is bound and then books the proposed selection –if any– through call to *book*. Otherwise, the *unavailable* exception is raised by the reservation operation, leading to retry an alternative reservation, and ultimately propagating the *unavailableHotel* exception for cooperative handling at the level of the WSCA. Finally, cooperative handling of *unavailableHotel* by the WSCA amounts to canceling the performed booking by the peer participant –if any.

```
<WSCA operation=''JointBooking''
    exceptionTree=''http://ta.com/TAExcepTree.xml''/>

  <participant name=''flight''>
    Not detailed, similar to hotel given below
  </participant>

  <participant name=''hotel''>
    <bind service=''HotelService''/>
    <input message=''InputMsg'' />
    <output message=''OutputMsg'' />
    <fault name=''unavailableHotel'' message=''unvHotel''/>
    <state> Not detailed </state>
    <behavior>
      <standard>
        <try>
          <comment text=''Try reserve a room''/>
          <call service=''HotelService'' operation=''reserve''
               input = ''...''  output = ''...''
               fault = ''unavailableHotel''/>
          <comment text=''Book the room that was found''/>
          <call service=''HotelService'' operation=''book''
               input = ''...''  output = ''...''
               fault = ''BookFailed''/>
          <comment text=''Return booking information''/>
          <return element=''...''/>
        </try>

        <localHandler exception=''unavailableHotel''>
          Retry booking, propagates unavailableHotel
          to all participants
          (for coordinated error recovery) otherwise:
          <raise exception=''unavailableHotel''/>
        </localHandler>
      </standard>

      <coordinatedHandler exception=''unavailableHotel''>
        <comment text=''Signal exception to the caller''/>
        <raise exception=''unavailableHotel''/>
      </coordinatedHandler>

      <coordinatedHandler exception=''unavailableFlight''>
        <comment text=''Compensate action''/>
        <try>
          <call service=''HotelService'' operation=''cancel''
            input = ''...''  output = ''...''
            fault = ''CancelFailed''/>
          <raise exception=''cancelled''/>
        </try>
        <localHandler exception=''CancelFailed''>
          <comment text=''Signal exception''/>
          <raise exception=''CancelFailed'' type=''failure''>
            Fault details
          </raise>
        </localHandler>
      </coordinatedHandler>

      <coordinatedHandler exception=''unavailable''>
        <raise exception=''unavailable''/>
      </coordinatedHandler>
    </behavior>
  </participant>
</WSCA>
```

5 Conclusion

Web services are expected to become a major class of systems of systems in the near future. This paper has introduced our work towards supporting the development of dependable systems of systems in the context of the Web service architecture. Our approach primarily lies in the WSCAL XML-based language for the abstract specification of the fault tolerant composition of Web services, which builds upon the CA actions concept for enforcing dependability.

We are currently implementing a base middleware support for WSCAs. The middleware includes the generation of composite Web services from WSCAL specification and a service for locating Web services. We are further working on the formal specification of WSCAL for enabling rigorous reasoning about the behavior of composite Web services regarding both the correctness of the composition and offered dependability properties. The specification of composite Web services using WSCAL allows carrying out a number of analyses with respect to the correctness and the dependable behavior of composite services. Except classical static type checking, the correctness of the composite service may be checked statically with respect to the usage of individual services. In addition, the same specification can be used for implementing executable assertions to check the composite service behaviour online. Reasoning about the dependable behaviour of composite Web services lies in the precise characterization of the dependability properties that hold over the states of the individual Web services after the execution of WSCAs. We are in particular interested in the specification of properties relating to the relaxed form of atomicity that is introduced by the exploitation of open-nested transactions within WSCA.

As discussed in Section 2, there is extensive research work that is ongoing towards supporting the development of fault tolerant composite Web services, relying on the XML-based abstract specification of Web services and of their composition, and on the transactional supports for composite Web services. Our contribution primarily comes from relying on forward error recovery instead of backward error recovery for specifying the behavior of composite Web services in the presence of failures. Forward error recovery is further specified in terms of co-operative actions, building upon the CA actions concept. Our analysis shows that this approach is more effective in dealing with faults at the level of composite Web services.

Acknowledgments

We would like to thank Marie-Claude Gaudel and Brian Randell for their helpful comments and discussions. We also thank Andras Pataricza for useful feedback about the paper. This research is partially supported by the European IST DSoS (Dependable Systems of Systems) project (IST-1999-11585) http://www.newcastle.research.ec.org/dsos/.

References

[1] B. Benatallah, M. D. M.-C. Fauvet, and F. Rabhi. *Patterns and Skeletons for Parallel and Distributed Computing, Eds. F. Rabhi, S. Gorlatch*, chapter Towards Patterns of Web Services Composition, pages 265–296. Springer Verlag (UK), 2002.

[2] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, SE-12(8):811–826, 1986.

[3] F. Casati, M. Sayal, and M.-C. Shan. Developing e-services for composing e-services. In *Proceedings of CAISE'2001, LNCS 2068*, pages 171–186, 2001.

[4] F. Cristian. *Dependability of Resilient Computers*, chapter Exception Handling, pages 68–97. Blackwell Scientific Publications, 1989.

[5] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[6] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[7] M. Kalyanakrishnan, R. Iyer, and J. Patel. Reliability of internet hosts: a case study from the end user's perspective. *Computer Networks*, 31(1-2):47–57, 1999.

[8] P. A. Lee and T. Anderson. *Fault Tolerance Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer - Verlag, 2^{nd} edition, 1990.

[9] Microsoft, BEA and IBM. Business Process Execution Language for Web Services (BPEL4WS) 1.0, 2002. `http://www.ibm.com/developerworks/library/ws-bpel/`.

[10] Microsoft, BEA and IBM. Web Services Coordination (WS-Coordination), 2002. `http://www.ibm.com/developerworks/library/ws-coor/`.

[11] Microsoft, BEA and IBM. Web Services Transaction (WS-Transaction), 2002. `http://www.ibm.com/developerworks/library/ws-transpec/`.

[12] T. Mikalsen, S. Tai, and I. Rouvellou. Transactional attitudes: Reliable composition of autonomous Web services. In *Proceedings of the Workshop on Dependable Middleware-based Systems in DSN 2002*, 2002.

[13] Oasis Committee. Business Transaction Protocol (BTP), Version 1.0, 2002. `http://www.oasis-open.org/committees/business-transactions/`.

[14] Oasis Committee. Universal Description, Discovery and Integration (UDDI), Version 3 Specification, 2002. `http://www.uddi.org`.

[15] B. Randell. Recursive structured distributed computing systems. In *Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems*, 1983.

[16] F. Tartanoglu, V. Issarny, N. Levy, and A. Romanovsky. Dependability in the Web Service Architecture. In *Architecting Dependable Systems, LNCS 2677*, pages 89–108. Springer-Verlag, 2003.

[17] F. Tartanoglu, V. Issarny, N. Levy, and A. Romanovsky. Formalizing Dependability Mechanisms in B: From Specification to Development Support. In *Proceedings of the ICSE Workshop on Architecting Dependable Systems*, Portland, USA, May 2003.

[18] W3C. Simple Object Access Protocol (SOAP) 1.1, W3C Note, 2000. `http://www.w3.org/TR/soap`.

[19] W3C. Web Services Description Language (WSDL) 1.1, W3C Note, 2001. `http://www.w3.org/TR/wsdl`.

[20] W3C. Web Service Choreography Interface (WSCI) 1.0, W3C Note, 2002. `http://www.w3.org/TR/wsci/`.

[21] J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the Twenty-Fifth IEEE International Symposium on Fault-Tolerant Computing*, 1995.

[22] J. Xu, A. Romanovsky, and B. Randell. Concurrent exception handling and resolution in distributed object systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1019–1032, 2000.

[23] J. Yang and M. Papazoglou. Web component: A substrate for Web service reuse and composition. In *Proceedings of CAISE'2002*, 2002.