Specifying Web Service Recovery Support with Conversations

Ferda Tartanoglu, Valérie Issarny

Specifying Web Service Recovery Support with Conversations

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

Abstract

Web services offer a number of valuable features towards supporting the development of open distributed systems, built out of the composition of autonomous services. Nonetheless, the resulting systems must offer a number of non-functional properties and in particular dependability-related ones, for acceptance by users, including effective exploitation in the e-business domain. However, dependability of composite services can only be achieved according to the recovery property of composed Web services. This calls for the rigorous specification of the standard and exceptional behavior of Web services. This paper introduces the WS-RESC conversation language that addresses this issue. In a way similar to existing conversation languages, WS-RESC includes constructs for defining ordering and choices. However, WS-RESC further includes constructs for specifying concurrency since it is an inherent feature of distributed systems, and for specifying timing constraints and recovery properties of conversation since these are key behavioral properties in the context of dependability.

1. Introduction

Although the modularity and interoperability of the Web services architecture enable complex distributed systems to be easily built by assembling several component services into one composite service, there clearly is a number of research challenges in supporting the thorough development of distributed systems based on Web services. One such challenge relates to the effective usage of Web services in developing business processes, which requires support for composing Web services in a way that guarantees dependability of the resulting composite services. The development of dependable composite services lies in specifying the services' standard and exceptional behavior, as enabled by most composition languages through support for exception handling and possible integration of distributed transaction management [21]. The exceptional specification of the composite service is further closely related to the ones of the Web services that are composed since the latter must ultimately perform some recovery actions (e.g., compensation) in the presence of faults.

The exceptional specification of a Web service is in part given by its WSDL interface [27], which states the exceptions that may be signaled in the form of fault messages. However, the behavior of a Web service is more precisely defined if its specification comes along with the ordering requirements over the service's operations as well as on timing constraints. For example, a Web service may require that a client first authenticate itself using a *Login* operation, and then, that it calls subsequent operations with at most 15 minutes intervals. It is now well recognized that providing machine-readable specifications of such descriptions, called *conversations*, is beneficial for at least three reasons. First, it helps in the discovery process where the service requester may search providers that support a given conversation. Second, the specification may be used to check correctness of the implementation, i.e., that the client side implementation is correct with respect to the server-side specification. Finally, such a specification may be used to automate Web service programming, e.g., by providing tools for the automated generation of correct code skeletons [13, 2]. Moreover, these and further analyses are possible using formal methods provided that an encoding of the conversation to a corresponding formal model is given [13, 17]. A typical use of such a formal model may be to verify refinement relationships over conversations through model checking. For example, one can verify that the required conversation at the client-side matches the provided conversation at the server-side, in a way similar to architectural connector matching [1]. Although several propositions for describing Web service conversations exist (e.g., [2, 7, 9, 24, 25, 26, 29]), they provide no or limited support for describing recovery behavior of Web services.

This paper introduces the WS-RESC conversation language that allows the thorough specification of both the standard and exceptional behavior of autonomous, composable Web services, further assisting the development of de-

0-7695-2268-8/05/$20.00 (C) 2005 IEEE

pendable composite services. The language in particular enables the definition of equivalence relationships over conversations with respect to their recovery behavior, which may be exploited for the design of fault-tolerant composite actions. WS-RESC is an XML-based language to be directly used by Web service developers to describe recovery-related properties of Web services. In addition, we provide a formal specification of the language through translation into the π-calculus [16] that makes available a large number of tools for reasoning about Web service properties. In particular, it allows the automated analysis of the correct composition of Web services with respect to the services' behavior.

Notions that are introduced are illustrated throughout this paper using some base recovery-related properties, i.e., redundancy, retry-ability, compensability and atomicity. The next section presents our approach to the specification of the recovery behavior of Web services, characterizing in particular the above properties, and introduces the notion of equivalence relationships using transition diagrams to specify them at the conversation level. Section 3 then introduces our conversation language, together with its formal semantics using the π-calculus. Section 4 further exemplifies the use of our language, addressing the specification of the aforementioned recovery properties associated with conversations, and its exploitation for developing dependable composite services. Related work is described in Section 5. Finally, Section 6 summarizes our contribution and sketches perspectives for our research.

2. Specifying recovery behavior of Web services

Specifying recovery behavior of Web services is central to the development of dependable composite Web services. Recovery properties of the composite service ultimately depend upon the ones offered by the composed services. For example, a composite service may require all the operations of the composed services to be *atomic* for the duration of all its interactions with them. Alternatively, it may require that a specific operation be *retry-able*. Such requirements may then be matched against the properties of the individual services, so as to allow, e.g., checking the correctness of the composition with respect to dependability properties, or dynamically retrieving instances of Web services that may actually be composed.

Several approaches exist that specify error recovery properties for individual operations, using meta-data. These approaches enforce participant Web services to describe their supported transactional behavior. Then, a client (the service composer), or a middleware service acting on behalf of a client, may exploit those descriptions 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. Such a concern is particularly addressed in the WSTx [15] and WebTransact[18] frameworks. However, these approaches are not sufficient for comprehensively expressing the recovery behavior of a service. The error recovery mechanism that is implemented by the client-side (i.e., the composite service) often involves more than one operation to be invoked on the server-side, and specific orders and conditions under which they should be invoked may be required for delivering the target recovery property. In this section, we address how conversations, in addition to specifying how to use the service in terms of dependencies between operations and time constraints, may be used to specify the recovery properties of Web services. In a first step, we present how recovery properties may be expressed using the notion of state equivalence (§ 2.1). Then, we discuss how state equivalence that is based on the knowledge of the systems' internal states may be used in the modeling of systems that exhibit only their potential behavior, leading us to introduce a specific equivalence relation over conversations (§ 2.2).

2.1. Recovery-related properties

Achieving fault tolerance of composite Web services through some recovery mechanism (e.g., open-nested transaction, replication) implies support of some base properties (e.g., atomicity, compensability) from individual operations or conversations of the composed services. Most of these properties may be characterized in terms of relationships over values of the individual services' states, an in particular state equivalence (also referred to as final state equivalence).

Redundancy, which is a key fault tolerance technique used in both hardware and software systems, implies having multiple systems that behave similarly. Hence, the redundancy property applies to two system activities (e.g., conversation, operation) if the system's state after the execution of either activity is equivalent to the one after the execution of the other, when both are executed with initial system states that are equivalent.

Retry-ability often involves using idempotent activities, i.e., activities that, when executed several times, give the same result. In other words, an activity is retry-able if the states reached after one or more sequential executions of the activity are equivalent.

Atomicity, is another base recovery property, stating that an activity either successfully executes until completion, or aborts by exceptionally terminating in the same state as the one that held before its execution. The atomicity property is then formally expressed in terms of state equivalences over the activity's pre- and post-states [30].

Transactions for long running activities are realized based on the **compensation** of operations that already committed (i.e., externalized), instead of implementing the atomicity property. We can describe compensation-based transactions using the notion of state equivalence, specifying that the successful compensation of an operation, or of a set of operations, brings the system to a state that is equivalent to either the initial one or another consistent state. Indeed, some committed operations may have effects on parts of the system that cannot be recovered, leading to bring the system to a state that is not equivalent to the one that could be restored under the atomicity property. In this case, the state that is reached should still be a consistent state. Applying compensation operations when several operations are already performed on a Web service may be tricky, if there are dependencies between operations, which implies verifying a number of properties for deciding whether the compensation can be applied. For instance, in [12], the authors give a formal definition of compensating transactions based on the equality of histories. The definition makes use of the notion of the *commutativity* of sequence of operations. Then, different types of compensations are defined based on this notion. Commutativity of operations, or sequence of operations, can be expressed with state equivalence relations.

2.2. Recovery properties of conversations

We use labeled transition systems to model conversations, similarly to the UML activity diagram used in WSCL[26]. This approach, contrary to the state-machine modeling as that of [2], has the advantage of relying only on operation names and messages, as they are defined in the service's WSDL document, for the definition of states (or nodes) and labeled transitions. A state then models an operation of the Web service that may potentially be called, and a transitions models the actual call of the operation and gives the next available operations. Transitions are labeled optionally by either the *output* or *fault* return messages of the previously called operation. Starting states are defined using a *Start* state and the end of a conversation is given with an *End* state. In addition, the *Empty* state refers to an operation without messages associated with it and that does nothing. The *Start*, *End* and *Empty* states are the only states that do not refer to any Web service operation. Thus, the transition originating from the *Start* and *Empty* states may not be labeled, and *End* is final.

As an illustration, Figure 1 models a conversation specifying that all the interactions with the specific Web service begin with the *Login* operation. For example, assuming that the *Login* operation is of type request/response, the conversation starts when the client sends the input message of the *Login* operation. Then, if the *Login* operation returns the *LoginOut* output message, the client is allowed to call the *Search* operation, as many times as necessary, followed by a *Buy* operation. The conversation terminates either if the *Buy* operation returns a *Confirmation* output message or if the *Login* operation returns a *LoginError* fault message.

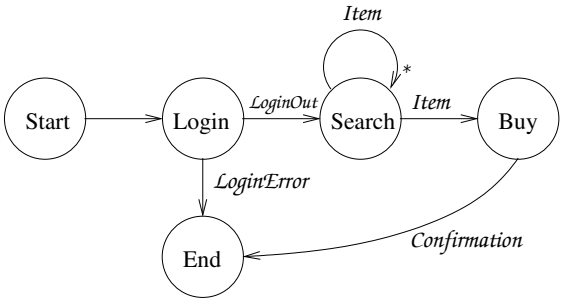

Figure 1. An e-business service conversation

Consider now the conversation depicted in Figure 2. It may be useful for the service's client to know that the internal state of the Web service (which is hidden) is exactly the same after each invocation of the operation modeled by state *A*, i.e., that the operation is *retry-able*. The client (or composite Web service) may use this information either to verify if a particular Web service supports retry-ability in case of a failure, or to implement an application-specific forward error recovery handler based on the retry technique.

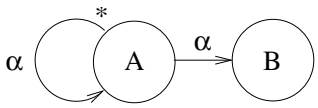

Figure 2. Retry-ability

Consider further the conversation depicted in Figure 3, which specifies that there are two execution paths for reaching the state *D*: $A \xrightarrow{\alpha} B \xrightarrow{\beta} D$, and $A \xrightarrow{\alpha} C \xrightarrow{\gamma} D$. We can deduce that the operations modeled by *B* and *C* are alternative executions, if the system's internal state at node *D* is the same following either execution path, i.e., if the redundancy property holds over the two activities: $A \xrightarrow{\alpha} B \xrightarrow{\beta} End$, and $A \xrightarrow{\alpha} C \xrightarrow{\gamma} End$. Such a property may then be exploited for implementing, e.g., fault tolerance using recovery blocks.

According to [8], there are two different approaches to defining the internal state of a system: (i) the *forward-looking style* with which the internal state of a system at a given instant is a notional attribute of the system that is sufficient to determine the system's potential behavior;

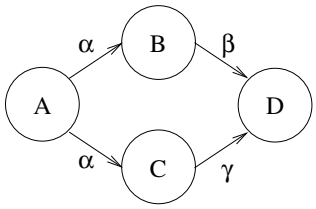

Figure 3. Alternatives

and (ii) the *backward-looking style* with which the internal state of a system is the total information explicitly stored (in state variables) by the system up to the given instant. The state equivalence relation used for specifying recovery-related properties relies on the equivalence of internal states of systems, as given in the second definition. However, what is typically described in conversation-like languages follows the former definition, i.e., the potential behavior of systems. The system is specifically viewed as a process and represented, in general, with labeled transition systems. Conversation languages are further often tightly coupled with formal methods, such as process calculi like CSP [10] or π-calculus [16]. Using these modeling approaches, the notion of equivalence (referred to as observational equivalence) is expressed in terms of the system's external behavior and verified using bisimulations of processes (e.g., see [20] for an exhaustive list of different bisimulations for the π-calculus).

Following the above, we introduce an equivalence relationship over conversations, to express the equivalence of system's internal states after the execution of the specified conversations, without making explicit their values, in a way similar to the work of [5]. Our equivalence relation is a binary relation, noted $\tilde{\sim}$, between two conversations. For two conversations A and B of a Web service W, if $A\tilde{\sim}B$ holds, then the internal state of W after the execution of A is equivalent to that reached after the execution of B, if the initial internal states for both executions are equivalent. Note that the equivalence relation specifies only equivalence over internal states, not behavior. Thus, processes are neither structurally congruent nor observationally equivalent. Our equivalence relation satisfies the following properties:

- Reflexivity: $A\tilde{\sim}A$
- Commutativity: $(A\tilde{\sim}B) \Rightarrow (B\tilde{\sim}A)$
- Transitivity: $((A\tilde{\sim}B) \wedge (B\tilde{\sim}C)) \Rightarrow (A\tilde{\sim}C)$

The specification of Web services conversations together with the equivalence relationships holding over them support the thorough development of dependable composite Web services, defining the standard and exceptional behavior of the Web services to be composed.

3. The WS-RESC language for dependability

Following the specification of Web services conversations discussed in the previous section, we introduce the WS-RESC (Web Service REcovery Support Conversation) XML-based language for specifying conversations and related recovery behavior. Conversations are in particular specified in terms of the Web service's offered operations, which are defined in the related WSDL document. Conversations then set the rules of how to use the specific Web service. In a way similar to existing conversation languages, WS-RESC includes constructs for defining ordering (dependency) with conditions on exchanged messages and choices. However, WS-RESC further includes constructs for specifying concurrency since it is an inherent feature of distributed systems, and for specifying timing constraints and conversation equivalences since these are key behavioral properties in the context of dependability. Finally, WS-RESC supports the composition of conversations.

The definition of WS-RESC comes along with its formal specification through translation in the π-calculus, thus allowing for automated reasoning about behavioral matching of Web services using π-calculus tools. Such a support is crucial in assisting the development of dependable composite Web services, since it enables enforcing the correct usage of Web services in the composition process. We use the following notation to denote π-processes, with Exp denoting boolean expressions:

P, Q ::=		Processes
	$P\|Q$	Parallel
	$P + Q$	Choice
	$!P$	Replication
	$if\ Exp\ then\ P\ else\ Q$	Conditional
	$v(x)$	Input message
	$\bar{v}(x)$	Output message
	$(\nu x)P$	Restriction
		(x is a new name in P)
	$\emptyset$	Null process

We recall that the input process $v(x).P$ is ready to input from channel v, then to run P with the formal parameter x replaced by the actual message, while the output process $\bar{v}(y).P$ is ready to output message y on channel v, then to run P. The $reduction$ relation, noted $\rightarrow$, is further defined over processes, with $P \rightarrow P_1$ expressing that P can evolve to process P_1 as a result of an action within P. For instance, we have: $((\bar{v}(x).P + P')|(v(y).Q + Q')) \rightarrow P|Q\{^x/_y\}$, with $Q\{^x/_y\}$ meaning that x replaces y in Q. In the following, we also use a shorthand notation for input and output messages, denoting the channel and parameter with message names.

We further use the following XML notations within examples: the default namespace is the one of the conver-

sation specification language, namespace *this* refers to the document being specified, and *ws* refers to the WSDL document of the Web service.

3.1. States and transitions

A state of a conversation is declared with the *state* XML element, which has two attributes: (i) *name* names the state and (ii) *operation* relates the state to an operation of the Web service. Note that the Web service operation as well as its associated message exchange pattern, i.e., whether it is of type request-response, one-way or solicit-request, etc. is declared in the related WSDL document. Naming states allows their reuse, and dually introducing several states referring to the same operation, with different transition rules, in different parts of the conversation. As an illustration, the example below introduces two states, *Login* and *Search*, of the conversation depicted in Figure 1:

```
<state name=''Login'' operation=''ws:Login''/>
<state name=''Search'' operation=''ws:Search''/>
```

Transition from a state occurs when the operation that is associated with the state is executed. Destination states then specify the operations that can be subsequently called. Additionally, we may have conditions on transitions, represented as labels on the transitions. Conditions refer to the names of the output and fault messages of the source operation. The transition is only valid if the condition is satisfied, i.e., if the operation returns the given messages. The condition is expressed as an XPath[28] expression and may thus be a boolean expression composed of several messages. A transition is specified using the *transition* XML element, which embeds: the *source* element that gives the source state (or activity), and the *destination* element that gives the target state (or activity). The optional *condition* attribute of the *source* element is defined for transitions that depend on some output or fault messages. In addition, we use the attributes *minOccurs* and *maxOccurs* for the *transition* element to specify how many times the sequence should be repeated. The special keyword *all* for a source state is introduced as a shorthand notation for declaring as many transitions as states. As an illustration, the following defines two transitions of the conversation depicted in Figure 1.

```
<transition name=''t0''/>
  <source state=''this:start'' />
  <destination state=''this:login''/>
</transition>

<transition name=''t2'' >
  <source state=''this:login''
          condition=''ws:LoginOut''/>
  <destination state=''this:search''/>
</transition>
```

A state A and related output transitions directly translate into a π process $A_\pi ::= in.((\overline{out}_1.B^1_\pi) + ... + (\overline{out}_n.B^n_\pi)$, with in being the input message for the operation associated with A, n the number of output transitions, $\overline{out}_i.B^i_\pi$ modeling the transition labeled with message abstracted by out_i, and B^i_π denoting the process associated with the destination state.

3.2. Choice

Using XML, non-deterministic exclusive choices are defined by declaring two or more transitions from the same state and with the same condition. This further directly translates into π processes combined with the choice $(+)$ operator, as above. The following example, taken from the conversation of Figure 1, specifies that after the execution of the *Search* operation, the client is allowed to call either the *Search* or *Buy* operation:

```
<transition name=''t1''>
  <source state=''this:search''
          condition=''ws:Item'' />
  <destination state=''this:search''/>
</transition>

<transition name=''t2''>
  <source state=''this:search''
          condition=''ws:Item'' />
  <destination state=''this:buy''/>
</transition>
```

3.3. Correlation for session management

Since there is not a standard way to manage sessions in Web services, keeping track of Web service instances is usually managed by the application. For example, a Web service implementation may use cookies stored at the client side for handling sessions, or may require a session identifier to be associated with interactions. Such information may be used to identify the client, in the case where some operations should be invoked by the same client, as well as to identify a specific Web service instance on the server-side.

We abstractly represent session information using correlations, which serve as identifiers for states. Therefore, when a state has the same correlation value as another, this means that both states are part of the same session. We define correlations with the element *correlation*. Then, states sharing the same correlation values are identified through the attribute *correlate* referencing this correlation element, as exemplified below.

```
<correlation name=''SessionID'' />
<state name=''Login'' operation=''ws:Login''
       correlate=''this:SessionID''/>
<state name=''Search'' operation=''ws:Search''
       correlate=''this:SessionID''/>
```

Correlations are defined in corresponding π processes using parameters. Specifically, $(\nu i)A_\pi(i)$ is the process A_π with the newly defined correlation value *i*. The correlation value can then be shared with processes within the same session by passing the value in a way similar to input messages.

3.4. Activities and nesting

An activity is a connected graph that defines the conversation supported by a Web service. Activities are named and contains at least one *Start* state and an optional *End* state. Besides the *Start* and *End* states that have no behavior, a state in the graph may define a Web service operation or another activity, which has its own *Start* and *End* states. A nested activity has to terminate on an *End* state to allow the continuation of the main activity. A nested activity is viewed as an isolated execution, thus considered as a single state. Isolation is further enforced by disallowing states to be shared between activities. Activities thus directly translate into π processes, according to rules associated with embedded constructs.

Figure 4. Activities and composition

As an illustration, Figure 4 depicts an activity with a nested activity Y. Activities are declared using the *activity* element and nesting is specified through transition with a destination element of type *activity*, leading to an implicit transition on the *Start* state of the nested activity. The nested activity may then continue until an *End* state is reached. Only then, the containing activity may resume, from a transition that has the nested activity as a source destination. Conditions on this transition may be on the output or fault messages of the state preceding the *End* state in the nested activity. We get the following XML definition for the activity of Figure 4.

```
<activity name=''Main''>
  <transition name=''a''>
    <source state=''A''>
    <destination state=''Activity-Y''>
  </transition>

  <transition name=''b''>
    <source state=''Activity-Y''>
    <destination state=''B''>
  </transition>
  ...
</activity>

<activity name=''Activity-Y''>
  ...
</activity>
```

3.5. Concurrency and join condition

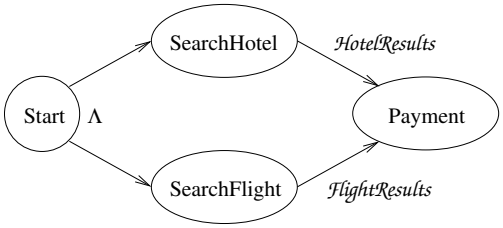

Figure 5. An activity with a join condition

Concurrency refers to activities or processes that are allowed to execute concurrently, which directly translates in the π-calculus using the parallel (|) operator. Figure 5 depicts a conversation where the *concurrent* transition from the *Login* state leads to two concurrent sub-activities: *SearchHotel* and *SearchFlight*. Concurrency is specified using the $\wedge$ symbol on the state-transition diagram. The corresponding XML declaration is given using the *concurrent* construct and by declaring multiple destination states and/or activities. In addition, the *minOccurs* and *maxOccurs* attributes may be specified for the *concurrent* element, to specify that the destination operations may be called several times in parallel:

```
<transition name=''concurrency''>
  <source state=''this:Start''/>
  <concurrent>
    <destination state=''this:SearchHotel''/>
    <destination state=''this:SearchFlight''/>
  </concurrent>
</transition>
```

Synchronization of concurrent activities is further specified using a join condition, which is expressed as a boolean expression on basic communication events that specifies under which conditions the execution is allowed to continue. In the activity depicted in Figure 5, the concurrent activities are both required to terminate before invoking the *Payment* operation. Various conditions can be specified, ranging from the synchronization of all the parallel activities, termination of a subset of the concurrent activities, and no condition at all. In general, the join condition is specified as a boolean expression on the output messages of the last operation of each activity that is joined. Formally, this translates into a conditional π process that is sequentially composed with the concurrent activity, and whose condition is a boolean expression over related output events. The XML representation of the activity depicted in Figure 5 is given below. The corresponding transition specifies the states of all the concurrent activities that are to be joined as source states, and the join condition as an Xpath[28] expression on the destination state:

```
<transition>
  <join>
    <source state=''this:bookHotel''
            condition=''ws:HotelResults''/>
    <source state=''bookFlight''
            condition=''ws:FlightResults''/>
  </join>
  <destination state=''this:PayAll''
               condition=''ws:HotelResults
                    and ws:FlightResults''/>
</transition>
```

3.6. Timing

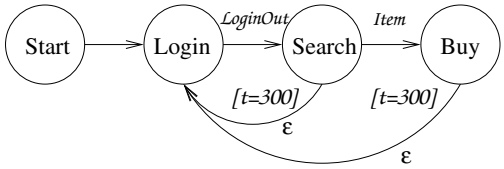

Figure 6. An activity with timers

Web services may have timing constraints on some of their interactions. For example, a Web service that requires authentication before starting subsequent interactions may impose a delay of at most 15 minutes between successive operations, and enforce re-login upon timeout, as illustrated in Figure 6. A timeout is associated with a transition using the *timeout* construct. The embedded *onInput* attribute is set to *true* if the timeout is computed from upon receipt of an input message by the source operation, or set to *false* (default) if the timeout is computed from upon emission of the output message. The *state* attribute further specifies the destination state of the process upon timeout occurrence.

```
<transition>
  <source state=''this:Login''
          condition=''ws:LoginOut''/>
  <timeout duration=''300''
           onInput=false
           state=''this:Login''/>
  <destination state=''this:Search''/>
</transition>
```

Since there is not a standard way to model timers in the π-calculus, we can abstract from time by using a specific process that is run when the timer timeouts and that can be prefixed by an output event relating to a timeout fault message, if any. A timer sets on a *Login* operation that returns a *TimeOut* fault message may then be specified as: $Login.(\overline{OK}.Search_\pi + \overline{TimeOut}.Login_\pi)$. An alternative would be to model the time directly in the calculus, as presented in [3]. In this paper, the authors extend the π-calculus with a timer denoted by $timer^t(\bar{x}(v).P, Q)$, with *t* being a positive integer representing time steps, *Q* the process that is run when the timer timeouts and $\bar{x}(v).P$ is the continuation process. The latter modeling has the advantage of making time explicit and thus allows reasoning about timing properties of processes. However, there does not yet exist any tool assisting such reasoning. We thus undertake the former approach for modeling timeout in the π-calculus, allowing to benefit from existing tools for the analysis of WS-RESC conversations.

3.7. Equivalence

Finally, we introduce the *equivalence* element to specify equivalence between activities, according to our definition introduced in Section 2.2. We get the following XML notation:

```
<equivalence>
  <activity name=''this:activityA''/>
  <activity name=''this:activityB''/>
</equivalence>
```

We recall that the equivalence relationship serves specifying equivalence of internal states and not of behavior. Thus, our definition does not map to any of the equivalence relationships define over π processes. However, two processes can be substituted if they are equivalent according to our definition, when analyzing the behavior of processes from the standpoint of dependability.

Given the process algebra definition of the proposed conversation language, behavioral compatibility of Web service clients (e.g., composite service) with Web services (e.g., composed Web services) may be verified using observational equivalences such as simulation tests between processes. Specifically, the server process must simulate the client process, as discussed in the context of Web services in [17]. In the general case of composite services, i.e., when a client accesses several Web services, additional verifications may be done for checking safety and liveness properties on the composition process. Furthermore, identifying the appropriate Web service that can take part in a composition that requires a specific recovery protocol to be implemented (e.g., WS-Transaction [14]), may be done by: (i) verifying that the Web service may implement the required protocol, i.e., the process defining the protocol simulates the Web service's conversation process, and (ii) checking that required recovery properties hold. Note that the π-calculus formalization of Web services protocols are given in [4] for Web services authentication protocols and in [6] for long-running transactions, which may be conveniently combined with our work.

4. Specification of error recovery properties

The WS-RESC language allows precisely characterizing the recovery behavior of Web services, further easing

the development of dependable composite services. This section illustrates usage of our language through examples of common recovery-related properties, i.e., *retry-ability* (§ 4.1), *compensability*(§ 4.2), and *atomicity* (§ 4.3).

4.1. Retry-ability

A retry-able process is defined as a process that behaves similarly when sequentially executed several times, i.e., the process delivers the same results in terms of output or fault message returned to the caller, and the internal states of the Web service after any number of executions are equivalent, at least as can be perceived by the client. It is quite direct to specify this property in terms of activity equivalence. For instance, consider the example we introduced in Section 2 (Figure 2). The corresponding π process is specified as $A_\pi ::= msg.(\bar{\alpha}.A_\pi + \bar{\alpha}.B_\pi)$, with msg denoting the call message of the operation associated with A, and $\bar{\alpha}$ the output message of the operation associated with A. Consider another conversation that is also supported by the Web service, with the operation associated with A being executed only once (Figure 7), i.e., $A'_\pi ::= msg.\bar{\alpha}.B_\pi$. If $A_\pi \tilde{\sim} A'_\pi$, then this implies that a single execution of the operation has the same effect as several executions and thus that the operation is retry-able.

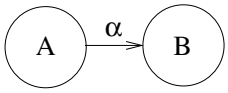

Figure 7. Expressing retry-ability

4.2. Compensation

Consider now the activity depicted in Figure 8 and specified as $C_\pi ::= msg_{Buy}.\bar{\alpha}.Cancel_\pi$ and $Cancel_\pi ::= msg_{Cancel}.\bar{\beta}.End_\pi$. The equivalence relationship: $C_\pi \tilde{\sim} \emptyset$, states that the operation *Cancel* cancels the effects (on the server side) of the operation *Buy*, provided that the operation *Cancel* returns an output message, given by the transition β that confirms the correct execution.

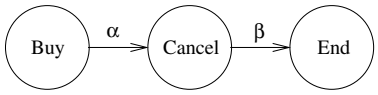

Figure 8. Compensating operations

4.3. Atomicity

Figure 9 illustrates an atomic activity which terminates either successfully by committing performed operations or without completing its task by aborting and rolling back its state. The activity starts with the invocation of the *Begin* operation and then allows calling several times operations *A* or *B* in any order. If any of these operations returns a fault message, the whole activity is aborted on the server side, and the client can no longer continue and may only call the *Abort* operation to confirm the abortion. Otherwise, the client may call, anytime, either the *Commit* operation to validate results and finish or the *Abort* operation to cancel the effects of previous operations. Note that in the first situation, the client calls *Abort* to confirm (i.e., acknowledge) the abortion that is automatically done at the server-side, and that in the second situation the choice of either aborting or validating is left to the user.

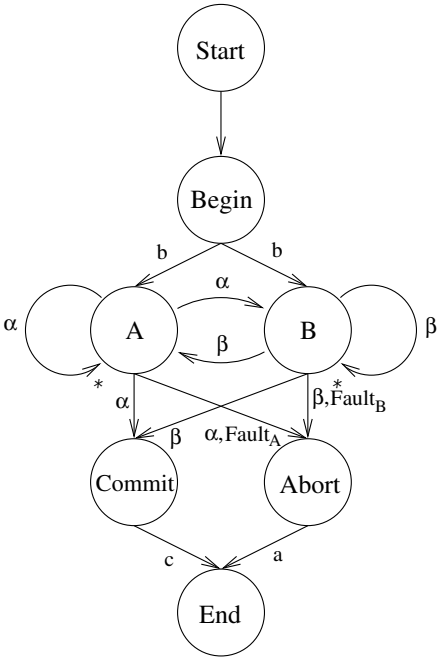

Figure 9. Atomicity

Let *C* be the the activity illustrated in Figure 9 and *C'* the sub-activity that always terminate with an *Abort* and that can be simulated by *C*. *C'* may be expressed with the following process:

$C' ::= msg_{Begin}.(\bar{b}.A + \bar{b}.B)$
$A ::= msg_A.(\bar{\alpha}.A + \bar{\alpha}.B + \bar{\alpha}.Abort + \overline{Fault_A}.Abort)$
$B ::= msg_B.(\bar{\alpha}.B + \bar{\alpha}.A + \bar{\alpha}.Abort + \overline{Fault_B}.Abort)$
$Abort ::= msg_{Abort}.\bar{a}.End$

The equivalence relation $C' \tilde{\sim} (msg_{Begin}.\bar{b}.End)$ states that the internal state of any activity that starts at the *Begin* state and ends (after an undetermined number of transitions) at the *Abort* state, is *equivalent* to the internal state of the starting state *Begin*. That means that all of the executed actions are undone.

5. Related works

Various conversation languages have been introduced in the literature, which may be coupled or not with the specification of Web services composition. The former approach is in particular addressed by choreography languages such as WS-CDL [24]. The latter approach is the most general, as it enables defining reusable autonomous services and does not impose any concrete link to other services nor specific composition processes. Hence, it is more appropriate for defining loosely coupled, reusable services. Relevant conversation languages then include DML [23], WSCL [26, 19], the framework introduced in [2], the *service model* description of OWL-S [22], Abstract BPEL [7], WSCI [25] and the conversation specification presented in [29]. However, most conversation languages lack constructs for specifying recovery properties associated with conversations. They only allow exceptional behavior to be described using transitions on fault messages. Hence, except the framework introduced in [2], specification of recovery properties such as transactional behavior of conversations is not addressed. The language introduced in [2] allows specifying transactional behavior of conversations. However, it uses a list of pre-defined transactional properties in a way similar to approaches that annotates individual operations with transactional properties [15, 18], hence reducing the language's expressiveness. Furthermore, existing conversation languages do not address timing issues, except for CS-WS [9] that introduces an additional timeout attribute associated to operations. Also, it should be possible to specify concurrent activities within conversations, as concurrency allows specifying complex distributed systems involving several participants competing and/or collaborating. However, only workflow-based BPEL [7] and WSCI [25] address the specification of concurrent abstract processes.

6. Conclusion

The Web services architecture offers many attractive features for supporting the development of open distributed systems, spanning various application domains such as e-business processes but also mobile applications [11]. However, the development of distributed systems using Web services still raises numerous challenges, in particular related to the enforcement of non-functional properties. Dependability of Web services is in particular a crucial issue since it greatly conditions acceptance of the Web services technology by both service clients and providers. Such an issue has led to tremendous research effort over the last couple of years, as witnessed by, e.g., the introduction of choreography languages integrating support for the specification of recovery actions in the composition process, and of advanced transaction protocols for distributed composite Web services. However, the recovery behavior of composite services ultimately depend upon the recovery properties of the composed services, which require adequate specification of the individual Web services. Such an issue is partly addressed in the definition of the services' WSDL interfaces through fault messages. However, this does not specify complex recovery properties such as compensation of operations, as exploited by advanced transaction models for Web services. Other attempts define the recovery behavior of Web services operations, using meta-data. However, these approaches are not sufficient for comprehensively expressing the recovery behavior of a service. The error recovery mechanism that is implemented by the client-side (i.e., the composite service) often involves more than one operation to be invoked on the server-side, and specific orders and conditions under which they should be invoked may be required for delivering the target recovery property. This then suggests to specify the recovery properties of Web services at the level of conversation. However, existing conversation languages, do not address such a needs, as they primarily focus on the specification of the services' standard behavior.

This paper has introduced the WS-RESC conversation language, which allows the comprehensive specification of Web services standard and exceptional behavior, further assisting the development of dependable composite services. The language in particular enables the definition of equivalence relationships over conversations with respect to their recovery behavior, which may be exploited for the design of fault-tolerant composite actions. Formal specification of the language through translation into the π-calculus additionally allows the automated analysis of the correct composition of Web services with respect to the services' behavior. In order to make such an automated analysis, formal specification of the composition behaviour should also be provided. In future work, we plan to use such a formally described composition language on a real-life application and show how WS-RESC may be used to correctly integrate component Web services.

The conversation and related recovery properties specified in WS-RESC is to be provided as part of the Web service's interface, extending the WSDL description. However, unlike the WSDL document that gives all provided operations, the WS-RESC description may only expose conversations and related properties considered relevant by the Web service designer. Then, conversations and properties related to the actual composition as it is implemented by a client may be obtained by applying composition rules given by the formal definition on conversations specified by the Web service.

7. References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), 1997, pp. 213-249.

[2] B. Benatallah, F. Casati, and F. Toumani. Web service conversation modeling. *IEEE Internet Computing*, January-February 2004, pp. 46-54.

[3] M. Berger and K. Honda. The two-phase commitment protocol in an extended pi-calculus. In L. Aceto and B. Victor, editors, *Electronic Notes in Theoretical Computer Science, EXPRRESS'00, 7th International Workshop on Expressiveness in Concurrency*, volume 39. Elsevier, 2003.

[4] K. Bhargavan, C. Fournet, and A. Gordon. A semantics for web services authentication. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'01)*, 2004.

[5] A. P. Black, V. Cremet, R. Guerraoui, and M. Odersky. An equational theory for transactions. In *Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science*, Mumbai (Bombay), India, December 2003.

[6] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 2003)*, 2003.

[7] F. Curbero, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.0. Technical report, OASIS, 2002. `http://www.ibm.com/developerworks/library/ws-bpel/`.

[8] M. Gaudel, V. Issarny, C. Jones, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, B. Randell, A. Romanovsky, R. Stroud, and F. Taiani. DSoS Conceptual Model. Technical report, IST Project Dependable Systems of Systems, IST-1999-11585, 2003.

[9] J. E. Hanson, P. Nandi, and D. Levine. Conversation-enabled web services for agents and e-business. In *Proceedings of the International Conference on Internet Computing (IC-02)*, CSREA Press, 2002.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[11] V. Issarny, D. Sacchetti, F. Tartanoglu, F. Sailhan, R. Chibout, N. Levy, and A. Talamona. Developing ambient intelligence systems: A solution based on web services. *Journal of Automated Software Engineering*, 2004. To appear.

[12] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *The VLDB Journal*, 1990, pp. 95-106.

[13] L. Meredith and S. Bjorg. Contracts and types. *Communications of the ACM*, 46(10), October 2003, pp. 41-47.

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

[15] T. Mikalsen, S. Tai, and I. Rouvellou. Transactional attitudes: Reliable composition of autonomous Web services. In *DSN 2002, Workshop on Dependable Middleware-based Systems (WDMS 2002)*, 2002.

[16] R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[17] C. Pahl and M. Casey. Ontology support for Web service processes. In *Proc. of the Joint 9th European Software Engineering Conference (ESEC) and 11th SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2003.

[18] P. Pires, M. Benevides, and M. Mattoso. *Web, Web-Services, and Database Systems 2002*, chapter Building Reliable Web Services Compositions, Springer LNCS 2593, 2003, pp. 59-72.

[19] K. G. S. Frolund. cl: A language for formally defining web services interactions. Technical report, HP Laboratories Palo Alto, October 2003.

[20] D. Sangiorgi and D. Walker. *The pi-calculus: A Theory of Mobile Processes*. Cambridge Universtity Press, 2001.

[21] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Dependability in the Web services architecture. In *Architecting Dependable Systems*. Springer Verlag, LNCS 2677, 2003. Available at http://www-rocq.inria.fr/arles/doc/doc.html.

[22] The OWL Services Coalition at the World Wide Web Consortium. Owl-s: Semantic markup for web service, 2003. `http://www.daml.org/services/owl-s/`.

[23] R. Tolksdorf. A dependency markup language for web services. In *Web Databases and Web Services 2002*, Springer Verlag, LNCS 2593, 2003, pp. 129-140.

[24] Web services choreography description language version 1.0, 2004. W3C Working Draft, `http://www.w3.org/TR/ws-cdl-10/`.

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

[26] Web services conversation language (WSCL), version 1.0, 2002. W3C Note, `http://www.w3.org/TR/wscl10/`.

[27] Web services description language (WSDL), version 1.1, 2001. W3C Note, `http://www.w3.org/TR/wsdl`, Working draft version 2.0 available at `http://www.w3.org/TR/wsdl20/`.

[28] XML Path Language (XPath), Version 1.0, 1999. W3C Recommendation, `http://www.w3.org/TR/xpath`.

[29] X. Yi and K. Kochut. Process composition of Web services with complex conversation protocols: a colored Petri nets based approach. In *Proceedings of Advanced Simulation Technology Conference DASD2004*, Arlington, Virginia, USA, April 2004.

[30] A. Zarras and V. Issarny. A framework for systematic synthesis of transactional middleware. In *Proceedings of Middleware98 – The IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, 1998.