



Conceptual Modeling of Web Service Conversations

Boualem Benatallah¹, Fabio Casasti, Farouk Toumani²,
Rachid Hamadi¹

Intelligent Enterprise Technologies Laboratory
HP Laboratories Palo Alto

HPL-2003-60

March 27th, 2003*

E-mail: {boualem, [rhamadi](mailto:rhamadi@cse.unsw.edu.au)}@cse.unsw.edu.au, casati@hpl.hp.com, ftoumani@isima.fr

web services

Web services are emerging as a promising technology for the effective automation of inter-organizational interactions. Several standards that aim at providing infrastructure to support Web services description, discovery, and composition have recently emerged including WSDL, UDDI, and BPEL4WS. Indeed, advances in this area promise to take cross-organizational application integration a step further by facilitating the automatic discovery and invocation of relevant services. However, despite the growing interest in Web services, several issues still need to be addressed to provide similar benefits to what traditional middleware brings to intra-organizational application integration (e.g., transaction support). In this paper, we identify a framework for defining extended service models to enable the definition of richer Web service abstractions. We also identify and define specific abstractions based on an analysis of existing e-commerce Web portals. Finally, we show how the model and the abstractions are supported by a conversation manager implemented on top of the SELF-SERV platform.

* Internal Accession Date Only

Approved for External Publication

. To be published in CAiSE '03, 16-20 June 2003, Klagenfurt, Austria

¹ School of Computer Science and Engineering, The University of New South Wales, Sydney NSW 2052, Australia

² Laboratoire LIMOS, ISIMA, Campus des Cezeaux, BP 125, 63173 Aubiere Cedex, France

© Copyright Springer-Verlag

Conceptual Modeling of Web Service Conversations

Boualem Benatallah¹, Fabio Casati², Farouk Toumani³, and Rachid Hamadi¹

¹ School of Computer Science and Engineering
The University of New South Wales, Sydney NSW 2052, Australia
{boualem,rhamadi}@cse.unsw.edu.au

² Hewlett-Packard Laboratories
Palo Alto, CA, 94304 USA
casati@hpl.hp.com

³ Laboratoire LIMOS, ISIMA, Campus des Cezeaux
BP 125, 63173 Aubiere Cedex, France
ftoumani@isima.fr

Abstract. Web services are emerging as a promising technology for the effective automation of inter-organizational interactions. Several standards that aim at providing infrastructure to support Web services description, discovery, and composition have recently emerged including WSDL, UDDI, and BPEL4WS. Indeed, advances in this area promise to take cross-organizational application integration a step further by facilitating the automatic discovery and invocation of relevant services. However, despite the growing interest in Web services, several issues still need to be addressed to provide similar benefits to what traditional middleware brings to intra-organizational application integration (e.g., transaction support). In this paper, we identify a framework for defining extended service models to enable the definition of richer Web service abstractions. We also identify and define specific abstractions based on an analysis of existing e-commerce Web portals. Finally, we show how the model and the abstractions are supported by a conversation manager implemented on top of the SELF-SERV platform.

1 Introduction

Web services are emerging as the technology of choice for application integration across wide area networks and across companies [1–4]. The essential benefit they bring to application integration is standardization. In fact, Web service technology builds on top of Web standards and extends them with additional languages and protocols (e.g., WSDL, UDDI, and SOAP [5]) that enable service description, discovery, and interaction. Standardization is important in Enterprise Application Integration (EAI) [6, 3], but is critical in the Web where the interaction occurs without a central coordinator and without a central authority.

However, despite the growing interest in Web services, several issues still need to be addressed to provide similar benefits to what traditional middleware

brings to intra-organizational application integration. Indeed, EAI middleware provides much more than basic features such as service description, discovery, and invocation. For example, it supports *transactions*, a very useful abstraction when it comes to developing reliable distributed applications. Having a shared understanding of this abstraction makes it easier to describe the properties of a system. In addition, automated tools can support, manage, and enforce transactions, without the developer having to worry about writing ad-hoc code for this purpose. Without the help of transactional middleware, the development of reliable applications would be quite hard.

Abstractions provided by EAI middleware can be beneficial for modeling and implementing Web services as well, although their notion needs to be reinterpreted in this new context. For example, a notion of transactions for Web service operations may define whether from a users' perspective, an operation can be aborted at any time without effect. A more complex transaction model may also allow for operation or conversation description languages that specify that an operation can only be rolled back within a specified time or with the payment of a fee. Just like for EAI middleware, this abstraction has the potential of simplifying the interpretation of service behaviors and allowing the development of tools that support the definition of transactional services and enforce transactional properties, without the application developers having to worry about it. Indeed, it is not surprising that proposals in this direction are emerging [7, 8].

Endowing Web services with abstractions analogous to those of traditional application integration is not, however, sufficient to support many of the requirements posed by application integration over the Web, since the problem is more complex in this domain. In fact, in EAI many of the properties and semantics of the services are assumed to be known a priori. Very often, clients and servers are deployed by the same project team. This is why the service interface (for example specified in terms of CORBA-IDL) is often all that is available in terms of service description. Properties and semantics are discussed face to face, and are either documented informally or not at all.

In Web services this is not the case (or, at least, this scenario is not the one Web services are targeting). The idea here is to enable developers to discover (at development time) service descriptions on the Web and, by reading these descriptions, be able to code client applications that can (at run time) bind to and interact with services of a specific type (i.e., compliant to a certain interface and protocol). As such, richer service descriptions and richer description models are needed, so that users can better understand the service execution semantics and how to interact with the service. In addition, richer service description model also allows the development of tools that better support Web service deployment, execution, monitoring, and management, as the transaction example demonstrate. Finally, it enables a more sophisticated dynamic binding, as clients can be more selective on the properties of the services they bind to when they search for a service. Referring again to the transactional example, clients can for instance require that the service supports transactions. The ultimate goal is that a service description includes all that is needed for developers to understand how

to write clients that interact with the service and for automated tools to dynamically bind to a service, based on the specified characteristics. This is essential especially as the number of services to be integrated grows and the environment becomes more dynamic. This is the scenario that Web services eventually aim at addressing, despite the many hard challenges it presents.

In this paper we identify a framework for defining extended service models, to enable the definition and description of richer abstractions and achieve the above-mentioned benefits. The main goal that guided us in the design of the framework is that of enabling the definition of service properties in a way that can support: (i) humans in understanding the service properties, (ii) clients in searching services based on these properties, and (iii) applications in automating the enforcement of the properties, much like transactional middleware supports transactional abstractions.

In addition to the framework, we identify and specify a set of abstractions that we have found useful and commonly needed in many practical situations. We observe that defining framework and properties that cover many different aspects of Web services is relatively easy. Indeed, there are tons of service description models around, developed in many different fields of computer science, and trying to extend service descriptions with functional and non-functional properties (we discuss some of them in Sect. 5). The problem, often overlooked, is that adding abstractions to models and primitives to languages is a delicate issue. In fact, while in general new abstractions may provide the benefits described above, they also make the service model more complex. Complexity severely compromises the usability (and therefore the adoption) of models and languages. Indeed, simple things are always the ones that work best, because they are both easier for users to understand and for developers to implement. Therefore, the hard part lies in striking a balance between expressive power and simplicity. As a consequence, another goal that guided our work is exactly that of striking this balance and “right-sizing” the model, while providing room for it to evolve as the need arises. In order to achieve this goal, we tried to understand what is the minimal set of features and abstractions that are useful and needed in practice to describe a Web service. This required an analysis of existing e-commerce applications and of their behaviors so that we could:

- determine a set of abstractions that could adequately model most or even all of them, and
- avoid the artificial introduction of complex abstractions that we could have thought useful, and that may even be needed in some occasion, but that are rarely used in practice.

In the following, we describe in detail how we approached the problem (Sect. 2), what are the resulting frameworks and models that we have developed (Sect. 3), and how they are supported by extending the SELF-SERV service development platform [9] (Sect. 4). Finally, in Sect. 5, we review some related work and give concluding remarks.

2 Web Portal Interaction Analysis: In Search of Real Needs

This section describes the rationale that guided the analysis of existing Web applications to identify the abstractions needed in real scenarios.

2.1 Web Portals versus Web Services

When starting the research described in this paper, our original intent was that of analyzing existing Web services to understand their characteristics and requirements in terms of description languages. However, we quickly recognized that a better approach was to analyze e-commerce Web portals rather than Web services. There are two main reasons for this choice. The first is that the Web services area is still rather immature. Only few Web services are available on the Internet, and they typically provide very simple functionalities (such as conversions from postscript to PDF), without any commitment required on either the client or the service side (e.g., no guarantees and no payments). There are indeed a few contexts in which Web services are available and are used for e-commerce transactions, but this mostly happens within a closed community of business partners, so that these services (and, most of all, the description of their interface and conversation) are not publicly available.

On the other hand, Web-based commerce is now a mature area. There is a huge number of Web portals that enable B2B, B2C, and C2C business transactions. In particular, e-commerce Web portals often include “terms and conditions” documents that describe the semantics of many operations (in particular those that involve some form of commitment on the client’s or provider’s side). However, Web portals are oriented to humans, while Web services are oriented to applications. Nonetheless, we believe that by analyzing a Web portal it is possible to extrapolate what would be the behavior of an “equivalent” Web service. For example, by analyzing a Web site (such as Travelocity.com) and by understanding the operations it makes available via a browser as well as the semantics of such operations, we can extrapolate what would be the behavior of its dual “Web service”.

2.2 An Embryonic Conversation Model

To perform this kind of reverse engineering analysis we needed a Web service description model, so that we could use it to abstract the Web service characteristics of a Web portal. However, we intentionally wanted a model that was very simple, so that it could help us start from a minimal base and progressively extend it as needed, as opposed to start from a rich model that included many possibly unnecessary features, thereby defeating our purpose of determining a “right-sized” Web service description model. A key ingredient of any service model is the interface definition language. For this, we simply use WSDL as a base, as it is now an accepted standard. Besides, WSDL is quite simple, and as

such it suits our purposes. Another important aspect of a service is the *conversation* it supports, i.e., the set of acceptable message exchanges and the order in which they should occur. For this, we defined a very simple conversation model, with the idea of progressively extending it according to the requirements derived from the analysis of real applications.

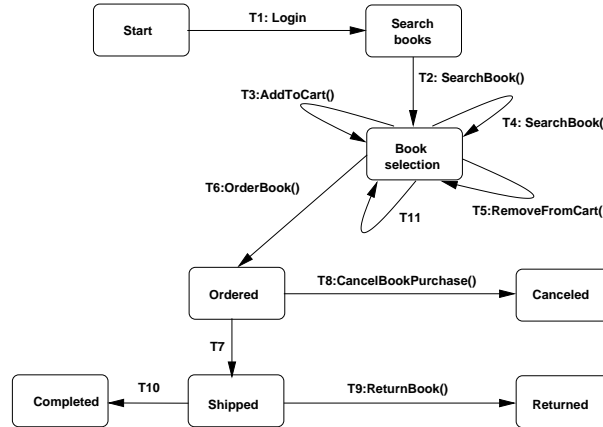


Fig. 1. The behaviour of Amazon.com as a Web service

The proposed embryonic conversation description model is based on the traditional state machine formalism, since it is simple, well known, and suited to describe reactive behaviors, which is the most relevant one as far as conversations are concerned (although, as we will see, other aspects of conversation behaviors need to be modeled as well). We assume that a conversation a Web service supports is modeled by a set of states and state transitions. States are labeled with a logical name, such as `logged_in`, `seat_reserved`, or `conversation_completed`. Transitions are labeled by service operations. When an operation `op` is invoked within a conversation that is in state `src`, then:

- if `src` has an output transition `tr` labeled with operation `op`, then the conversation moves into the destination state of `tr`,
- if `src` has no output transition labeled with operation `op`, then the conversation remains in state `src`.

This model is analogous to that of WSCL [10], although it is expressed through a state machine rather than a simplified version of UML activity diagrams, since we believe that state machines are better suited, for the reasons described above. In the following, we will use the term *conversation schema* to denote the specification of a conversation, while the term *conversation instance* will refer to an individual execution of a conversation between a service and a client. Figure 1 presents an example of a conversation schema supported by Web portal, namely Amazon.com, modeled with the formalism described above. Some transitions are unlabeled as they are not caused by explicit operation invocations. This is one of the indicators that this simple model is not enough, as

anticipated, and that extensions are needed. The figure shows a slightly simplified version of the actual interfaces and conversations supported by the site, but this is solely for ease of presentation.

2.3 Discussions and Observations

In this section, we discuss the main lessons learned during the analysis of about twenty Web portals including `Amazon.com`, `Travelocity.com`, and `Expedia.com`. They will provide the basis for progressively extending the Web service conversation model in a way that it retains the same simplicity (to the possible extent), but that also allows the specification of the features that are really needed in practice.

Implicit and timed transitions. Most transitions between states occur due to explicit operation invocations. However, there are cases in which transitions occur without operation invocations, or at least without an explicit invocation by requesters. An example is the transition between states `Ordered` and `Shipped`. This transition occurs when the ordered books are physically shipped to the customer. An important type of *implicit transitions* are *timed transitions*. A timed transition occurs automatically after a time interval is elapsed since the transition is *enabled* (i.e., the conversation state is the transition’s source state), or as a certain date and time is reached. As an example, flights on `Travelocity.com` can be put on hold for a period of time, or until midnight of the next day.

Compensation. In several Web portals we found operations (called *compensating* operations in the following) whose purpose is to semantically cancel the effects of other operations (called *forward* operations). For example, there are operations to cancel the purchase of a book or the reservation of a flight. Note that, unlike rollback in database transactions, “cancel the effects” here is meant from the perspective of the client. On the provider’s side, executing forward operations and subsequently compensating them may indeed be very expensive in terms of money, time, and other resources. The execution of a compensation operation is typically allowed only within a certain time period. For example, this is how most online booksellers handle returns. Compensation may also have an associated cost (e.g., cancellation fee). Cost and time constraints are sometimes combined, so that the fee is charged only after a certain date and time, or after some time has elapsed from the execution of the forward (i.e., compensated) operation.

Resource locking. The execution of some operations seems to acquire (or, using database terminology, lock) some resources for the client. For example, flight reservation Web portals allow customers to hold seats on a plane. Analogously to compensations, resource reservation may be associated to a cost, and may have a validity limited in time, after which the resource is released.

Conditions and instance-specific properties. In some conversations, transitions may require that certain conditions be verified in order to be enabled. For example, some operations may only be available to “premium” customers. As another example, `Amazon.com` has the concept of “gold box”, where special

one-day discounts are offered to certain customers. Other aspects of a conversation may also be instance-specific. For example, the cancellation fee and the time constraint may vary depending on the client or on the kind of goods being purchased.

Multi-state enabled operations. Many operations can be executed in more than one state. In particular, most conversations have a subset of operations that can be executed at any time. For example, it is always possible to search for books or flights, without “loosing” previously done work (e.g., the content of the shopping cart). Another related observation is that, in many cases, the execution of such operations does not cause a state change.

3 Modeling Multiple Aspects of Web Service Conversations

The previous section has outlined many aspects to be included when specifying Web services, thereby paving the road for extending the embryonic model according to what appear to be the needs of service developers. We focus specifically on conversations, both because conversation modeling is one of the most interesting (and innovative) aspects of service descriptions, and because the discussion in the previous chapter has emphasized that many service properties are defined in the context of a conversation. In particular, we intend to provide two different contributions in this section: the first is to define an extensible conversation meta-model that enables the definition of conversation properties. The second is to identify specific properties, based on the analysis of the previous section. We observe that this analysis, being driven by and extrapolated from Web portal analysis, may be polarized towards certain classes of Web services and may not fully account for the requirements of applications that are more cross-organizational and loosely coupled in nature. A more comprehensive analysis will be possible only when the requirements of such Web services applications become better known.

3.1 Requirements for a Conversation Model

We start the description of the conversation model by characterizing the requirements for such a model, based on the findings discussed in the previous section as well as on the intended usage of the model.

Genericity: The model should provide a set of *horizontal* properties that can be used to expose the semantics of a service, so that the concepts and the supporting tools are generally applicable. As such, the semantics of the defined properties should go across domains. Transactionality is one such example. We are not interested in defining vertical (domain-specific) properties: this is the job of standardization consortia, and besides we do not have the required knowledge and expertise.

Automated support: While a main goal of the model is to provide human readable description of a service, our aim is also that of enabling the development of tools that provide automated conversation support. This also means

that, although at the conceptual level the semantics of properties should be independent of any particular representation language, an agreed upon notation for representing properties is also essential. We will propose one such notation, which is also the one supported by the conversation management tool presented in Sect. 4.

Extensibility: While we have tried to identify the main properties in our Web portal analysis, there are other properties that we failed to recognize, or that may become relevant in the future. In addition, each vertical domain may recognize the need for adding more properties. Therefore, the model should be extensible in that it should allow the use of any identified property to describe a specific abstraction.

Relevance: As mentioned before, adding abstractions to models and primitives to languages is a delicate issue. In fact, while in general new abstractions may provide the benefits described above, they also make the model more complex. Therefore, the model should not contain artificial and complex abstractions that could have been thought as useful, and that may even be needed in some very rare occasion, but that are not actually used in practice. Indeed, we argue that the identification of properties should be based on the analysis of existing Web and business to business applications. The identified properties should provide the basis for an effective reasoning paradigm for understanding the behavior of service conversations (e.g., a composite service can reason about the transactional effect of one of its components on the overall service). Their use should benefit several automation activities of the service life cycle including services selection, composition, monitoring and management.

Compliance with Web services standards: The model should build upon the building blocks of XML and Web service concepts and standards (e.g., XML Schema, SOAP, WSDL, and UDDI).

3.2 Conversation Meta-Model

The “skeleton” on which we base the proposed conversation model is essentially a state machine, just like in the embryonic model presented above. We have motivated earlier why state machines are an appropriate paradigm for defining the set of conversation supported by a service (although other analogous approaches are possible). State and transitions have the same meaning as those described earlier. However, we generalize the approach by enabling the association of several *descriptive properties* with transitions, to characterize when the transition should occur and what are its implications (e.g., transactional semantics). In addition to transition properties, we also characterize properties of the conversations as a whole. In particular, we have identified the following characterizations as being useful for both conceptually describing a conversation and for automatically supporting its execution:

Conversation objects. Service objects refer to service main information such as product sold by an e-commerce portal. In our model, we consider service requests (i.e., operation names and input parameters) and responses (i.e., output

parameters) as conversation objects. This is compatible with the Web services model where users submit requests, whose structure (e.g., SOAP message) is represented according to service interfaces. As illustrated below, among other usages, service objects can be referenced in the pre-conditions of operation invocations (i.e., state transitions). In addition to service requests and responses, a service object may consist of an internal variable. Internal variables can represent service data items such as service-log (i.e., a variable that keeps a reference to the logged service invocations). They can also represent conversation-specific data items such as conversation instance identifier or the maximum number of operation invocations in a conversation instance.

Requester profiles. Requester profiles characterise users invoking operations. A requester profile consists of a set of attributes such as identity of user, purchase history, membership of a user to group(s) (e.g., `premier_member`), etc. Similarly to conversation objects, requester profiles may be used in the description of conversations (e.g., in pre-conditions of operation invocations).

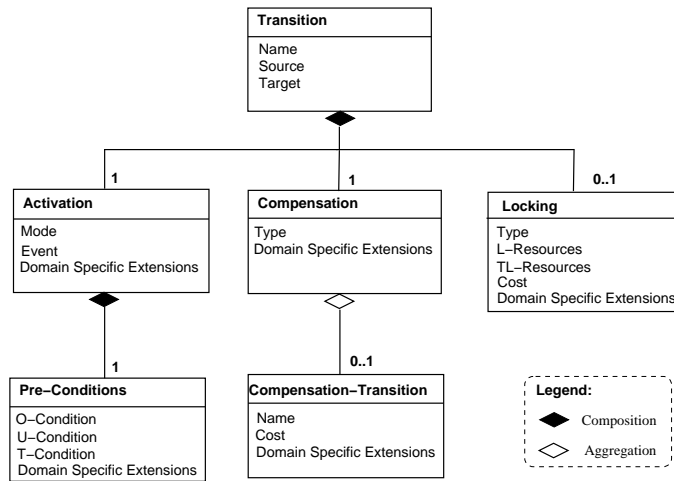


Fig. 2. UML Conceptual Model for Transition Properties

3.3 Transitions with Multiple Properties

In this section, we will describe transitions using our conversation description model. We discuss a list of properties that can be used to capture abstractions identified in Sect. 2.3. These properties consist of an initial set of abstractions that we have found to be useful and commonly needed in many practical situations, namely *description*, *activation*, *compensation*, and *locking* properties. The model is extensible in the sense that other properties may be defined and used. The description property provides human-understandable description (e.g., textual or HTML document) about the Web service conversation. It also contains any information that is not easily formalized and can not be interpreted by an automated tool such as a conversation controller.

The conceptual model shown in Fig. 2 represents a UML static model for the different components that constitute the properties of a transition. Each property is described using a set of attributes. The model is also open to the extension of definitions of properties by adding new domain-specific attributes⁴. The remainder of this section gives details about the identified transition properties, namely *activation*, *compensation*, and *locking*.

Activation Property. This property allows to describe the triggering features of a transition. Besides the fact that a transition is activated by invoking an operation, an activation property specifies an *activation mode*, the *activation event* and *pre-conditions*. The activation mode indicates whether the triggering of the transition is *explicit* (`mode="user"`) or *implicit* (`mode="provider"`). When the activation mode is explicit, the transition is activated by explicitly invoking a service operation. In this case, the value of the attribute `event` is the name of the corresponding operation. When the activation mode is implicit, the value of the attribute `event` is a specification of a temporal event (i.e., the transition will occur automatically after the occurrence of this event). A pre-condition is a triple (`O-condition`, `U-condition`, and `T-condition`), where:

- An `O-condition` specifies conditions on service objects.
- A `U-condition` specifies conditions on user profiles. It is used to specify the fact that an operation can be invoked by certain users (e.g., an operation is only available to "premium" customers).
- A `T-condition` specifies *temporal constraints* to allow the description of *timed* transitions (e.g., a transition can occur only within a certain time period).

A part of a pre-condition may be missing, in which case, the associated condition is `True`. An `O-condition` (respectively, `U-condition`) is a predicate or query over service objects (respectively, user profiles). It is satisfied if the predicate is `True` or the query result is not empty. We adopt XPath [11] as a language to express queries as service objects and user profiles are represented using XML.

The definitions of temporal constraints use XPath time functions (e.g., current time) and some pre-defined time functions in our model. A complete description of the temporal model, we use to specify temporal constraints, is outside the scope of this paper. In this section, we describe the constructs that are used to model timed transitions.

To keep track of beginning dates, termination dates, and number of invocations of transitions, we introduce the following functions: `beginT` and `endT`. `beginT(T)` (resp., `endT(T)`) denotes the beginning (resp., termination) date of the last invocation of the transition `T` within the same conversation instance. The conversation model features also the following temporal predicates:

- `M-Invoke` prescribes when an implicit transition must be automatically fired.
- `C-Invoke` prescribes a deadline or a time window within which a transition can be fired.

⁴ Attributes named in Fig. 2 are cross-domain attributes.

- **L-Invoke** prescribes the maximum number of invocations of a given transition within a time window.

M-Invoke is used to specify temporal events. **C-Invoke** and **L-Invoke** are used to specify temporal pre-conditions of a transition. For the sake of simplicity, we assume that all temporal values (time instants, durations, and intervals) are expressed at the same level of granularity. Formally, a temporal constraint is specified as either $\text{Pred}(\text{op}, d)$, where $\text{Pred} \in \{\text{M-Invoke}, \text{C-Invoke}\}$, or $\text{L-Invoke}(\text{op}, n, d1, d2)$. op is a comparison operator (e.g., $=$, \leq , and **between**) and d (resp., $d1$, and $d2$) is either an absolute time or a relative time (e.g., **beginT(T)**). The constraint **M-Invoke**(op, d) is only authorized for implicit transitions and means that the transition is automatically fired when the condition **current-date op d** is evaluated to **True**. Here, **current-date** denotes the system time. The constraint **C-Invoke**(op, d) means that the transition can be triggered only if the condition **current-date op d** is evaluated to **True**. The constraint **L-Invoke**($\text{op}, n, d1, d2$) means that the transition can be invoked n times within the time interval $[d1, d2]$. In all the previous predicates, a time value can be expressed as a function on service object or user profiles attributes. This allows to cater for requirements such as the time constraint may vary depending on the client or on the kind of goods being purchased.

Let us consider the descriptions of the activation properties of the transitions T9 and T10 of Amazon.com example (see Fig. 1). The following XML code represents the descriptions of the respective activation properties.

```
<transition name="T9" source="Shipped" target="Returned">
  <activation mode="user" event="ReturnBook">
    <pre-conditions O-condition="True"
      U-condition = "boolean(/user [@membership='gold'])"
      T-condition="C-Invoke(<,end(T7) + 30 days)"/>
  </activation>
</transition>
<transition name="T10" source="Shipped" target="Completed">
  <activation mode="provider" event="M-Invoke(>=, end(T7) + 30 days)">
    <pre-conditions O-condition="True" U-condition = "True"
      T-condition="True"/>
  </activation>
</transition>
```

The transition T9 can be explicitly performed (**mode="user"**) by invoking the operation **ReturnBook** within 30 days after the completion of the transition T7 by only “gold” customers. After this period of time, the transition T9 cannot be performed (constraint **C-Invoke**). However, the transition T10 is implicit (**mode="provider"**). This transition is automatically performed 30 days after the completion of the transition T7 (constraint **M-Invoke**).

Compensation Property. This property specifies the effect of a transition on the client state. With regard to this property, we distinguish the following types of transitions:

- *Effect-less* to denote a transition which has no effect on the client state. Cancelling this kind of transition does not require the execution of any particular

operation. For example, the transition T2, carried out during the execution of the operation `SearchBook()`, do not have any effect on the client state. The cancellation of these transitions is implicit and does not require the execution of any particular operation.

- *Credential-disclosure* to denote a transition which has no effect on the client state from transactional point of view (e.g., the client is not going to make a payment), but client may be required to reveal certain credentials (e.g., postal address, credit card number) to the provider. In general, the release of these credentials is governed by privacy policies. We do not consider further this aspect as it is a challenging topic by its own. We assume here that cancellation of these kind of transitions is implicit and does not require the execution of any particular operation.
- *Definite* to denote a transition whose transactional effects are permanent (i.e., are not compensatable). For example, after the delivery of the purchased items, the `Amazon.com` service remains in the state `shipping` during 30 days, corresponding to the period of time where the user can, under certain conditions, return the purchased items. After this period of time, the transition cannot be undone. This abstraction is conveyed by labeling the transition T10, for instance, as `definite` transition.
- *Compensatable* to denote a transition which has some effect on the client state but this effect can be undone by explicitly invoking a compensation operation. A compensatable transition is characterized by giving the name of the corresponding compensation transition and its cancellation cost. Similarly to time values in `T-condition`, values of `cost` attribute can be expressed as functions on service objects and user profiles. Consider, for instance, the transition T7. The effect of this transition consists of transferring money from the client bank account to the provider account. However, the effect of this transition can be (partially) undone (i.e., the client can be refunded) if the client decides to return the purchased items (operation `ReturnBook()`). The transition `T8:CancelBookPurchase()` can be used to compensate the transition `T6:OrderBook()`.

The examples below illustrate the transactional properties of the transitions T6, T7, and T10 of `Amazon.com` example (see Fig. 1).

```
<transition name="T6" source="BookSelection" target="Ordered">
  <transaction type="Compensatable">
    <compensation-transition name="T8" cost="0"/>
  </transaction>
</transition>
<transition name="T7" source="Ordered" target="Shipped">
  <transaction type="Compensatable">
    <compensation-transition name="T9" cost="/books/book/price * 0.1"/>
  </transaction>
</transition>
<transition name="T10" source="Shipped" target="Completed">
  <transaction type="Definite"/>
</transition>
```

The transitions T6 and T7 are **compensatable** and their effects can be respectively undone using the compensation transitions T8 and T9. The effects of the transition T10 cannot be undone (i.e., it is a **definite** transition).

Resource Locking Property. This property specifies temporary reservation of provider resources for a client when invoking a transition. We distinguish the following types of resource locking:

- *Lock* to denote that certain resources are locked for the client. A resource is a service object. The list of locked objects is specified by means of an XPath query.
- *Tentative-Lock* to denote that there is a tentative non-blocking reservation on certain resources. In fact, this kind of transitions is similar to the lightweight reservations in the *Tentative Hold Protocol*⁵. It allows several clients to place locks on the same item, and then once one client completes the purchase of the item the other clients receive notifications that their locks are no longer valid. An example of *Tentative-Lock* can be found in the travel arrangements domain. Some airlines companies allow travel agents to make flight reservations without effectively locking the seats until the tickets are paid. When the number of available seats in a given flight decreases, a warning is sent to the agents that have reservations in that flight. Whenever one flight becomes fully booked, the agents are notified that their reservations are no longer valid.

Information about the status of the resources is conveyed by the attribute **type** of the element **locking**. The locking type indicates whether some resources are locked (**type="L"**), some resources are on tentative lock (**type="TL"**), or both (**type="mixed"**), i.e., some resources are locked and others are on tentative lock. The attribute **L-resources** (respectively, **TL-resources**) is a query that specifies the resources to be locked (respectively, on tentative lock). Finally, the attribute **cost** indicates the cost of resources locking.

Assume that, in the example of Fig. 1, the transition T7 locks the items that are in the shopping cart (i.e., purchased books) for which the price is above \$100 at a cost of \$10. In this case, the locking property is specified as follows:

```
<transition name="T7" source="Ordered" target="Shipped">
  <locking type="L" L-resources="/books/book/price > 100"
          TL-resources="" cost="10"/>
</transition>
```

4 Automated Support for Conversation Management

In this section, we present the design and implementation of a tool called *conversation manager*. This tool is used to facilitate the creation, monitoring, and control of conversation life cycle operations. It is implemented as an extension of the SELF-SERV service development platform. A description of SELF-SERV prototype can be found in [9]. The prototype architecture (see Fig. 3) features

⁵ <http://www.w3.org/TR/tenthhold-1>.

a *service manager* and a *conversation manager*. These modules have been implemented using Java and the IBM Web Services Toolkit (WSTK) [12]. WSTK provides several components and tools for Web service development (e.g., UDDI, WSDL, and SOAP). Services communicate via (SOAP) messages. The implementation of the conversation manager is an ongoing effort. Here we describe an initial design and implementation of the conversation manager. The conversation manager consists of two modules, namely, *conversation builder* and *conversation controller*. Section 4.1 overviews the design of the conversation controller. Section 4.2 describes the creation and management of conversations using the conversation builder and conversation controller.

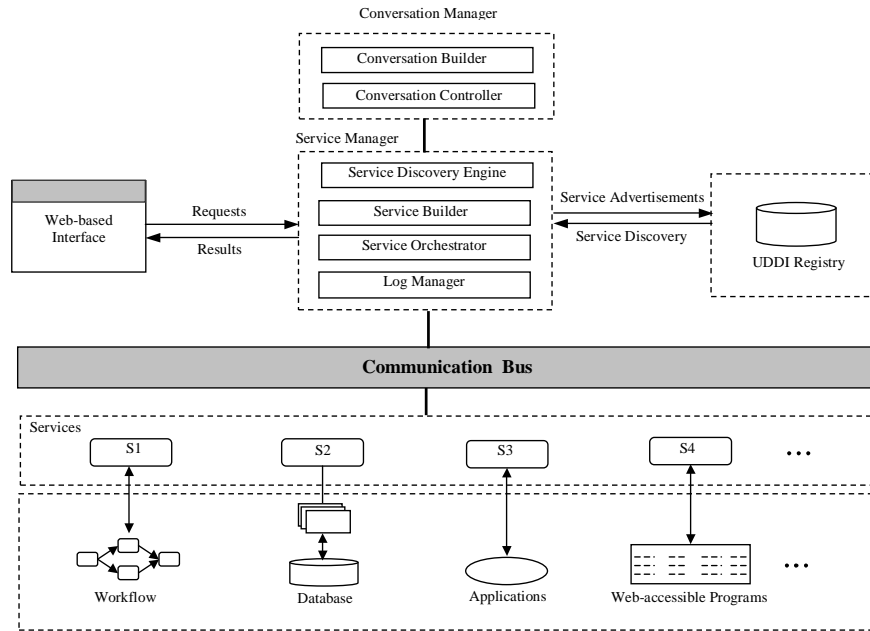


Fig. 3. Prototype Architecture

4.1 Conversation Controller: Overview

The conversation controller is essentially an extensible object attached to a service. It contains operational knowledge (e.g., conversation states). It also provides operations for monitoring conversations (e.g., triggering a transition). At run-time, a conversation controller is responsible for:

- Receiving service requests (i.e., SOAP request messages), determining if new conversation instances should be created, and removing conversation instances when they are no longer valid.
- Checking whether messages received and sent are in accordance with conversation definitions.
- Triggering transitions whenever all their pre-conditions are met.
- Tracing service executions.

The information required by a conversation controller to conduct the above tasks is extracted from the conversation definition and represented in the form of *control tables*. Control tables are associated to states. A control table of a state **S** is a set of rules of the form **E**[**C**]/**A** such that:

- **E** is an event of the form : (i) **explicit**(**op**), meaning that an invocation of an operation **op** has been received, or (ii) **implicit**(**t**), meaning that a temporal event **t** has occurred.
- **C** is a conjunction of conditions appearing in the pre-conditions clause of a transition (i.e., **O-condition**, **U-condition**, and **T-condition**).
- **A** is an action of the form **trigger**(**T**), meaning that the transition **T** of the state **S** needs to be triggered. The triggering of a transition causes: (i) a state transition, meaning that the conversation controller marks the conversation instance as moved from the source state into the target state of the transition, and (ii) the invocation of the corresponding Web service operation⁶.

Briefly, the basic semantics of a control table, is as follows: While at state **S** (i.e., state **S** is active), when one of the elements (i.e., one of the event) of the table is triggered, if its condition evaluates to true, the corresponding transition is activated. The control tables of a conversation are statically derived by analysing the activation properties of transitions. The detailed description of the generation algorithm is outside the scope of this paper due to space reasons.

4.2 Creating and Managing Conversations

The conversation builder assists providers to create conversation definitions and generate control tables. A conversation definition is edited through a visual interface and translated into an XML document for subsequent processing. The visual interface offers an editor for describing a state machine diagram of a conversation. It also provides means to describe the properties of transitions. The conversation builder generates the control tables using the XML representation of a conversation. The control tables are formatted in XML as well.

The functionalities of the conversation controller are realised by a pre-built class called **ConversationController**. This class extends a service with pre-built capabilities to participate in conversations which are defined using the model proposed in this paper by providing access to operational knowledge such as control tables. It provides methods for receiving service requests, managing conversation instances (i.e., creating and deleting instances), detecting transition activation events (i.e., explicit operation invocations, temporal events), triggering transitions, tracing service executions, and communicating with service requesters in accordance with conversation definition (e.g., sending a notification informing the requester that deadline for cancelling an operation is passed).

More precisely, the class **ConversationController** implements a software module made up of a *container* and a *pool of objects*. There is one container per conversation schema. The container is a process that runs continuously, listening

⁶ In the case of an implicit transition, the corresponding operation is an internal service operation.

to service request messages. When the container receives a request message, it proceeds as follows⁷:

- If the message does not carry an existing conversation instance identifier (i.e., the conversation instance is unknown to the container because it is a new or expired conversation), a new controller object is created, and this object is given access to the conversation control tables. The task of handling the request message is delegated to this newly created object by invoking a method called `process_request` on it. The newly created object is temporarily added to the pool of objects so that it can handle subsequent messages related to the same conversation as they arrive.
- If on the other hand the container has previous knowledge about the conversation instance to which the request message relates, the message is forwarded to the controller object that was created when the first message related to that instance was received. This object is retrieved and the method `process_request` is invoked on it.

A service may participate in several conversations simultaneously. Each controller object in the pool is dedicated to a particular conversation instance, and processes all the incoming and outgoing messages related to that instance. By keeping track of the status of these requests and by having access to the relevant control tables, the controller object is able to check whether messages received and sent are in accordance with conversation definition and detect when should a given transition of the conversation be activated. When a controller object detects that a given transition needs to be activated, it sends an invocation message to the related service. Once the corresponding completion message is received, the object sends the response to the requester. The lifespan of a controller object is bound to the life span of the associated conversation. This means that, once the conversation expires (e.g., because of timeout), the associated controller object is no longer needed. The container removes the object from the pool and destroys it. It is worth-noting that the conversation controller relies on the log manager to trace conversation executions and interactions.

It should be noted that controller objects are different from service instances. Controller objects are responsible for managing conversation instances. Service instances are responsible for processing invocations to the service, initiating the service, collecting the outputs, and returning them back to controller objects that initiated the invocations. The messages exchanged between a controller object and a service instance are SOAP request/response messages. The `controller class` is provided by our system. A service provider only needs to download and install the class `ConversationController` in order to support the functionalities of the conversation controller (i.e., message correlation, conversation/messages conformance checking, etc). In fact, a service creator needs to provide only the business logic of the service. The class `ConversationController` shields the service creator from the implementation details of the functionalities of the conversation controller.

⁷ Conversations are tracked in a similar manner as in WS-Coordination, i.e., we assume that SOAP Messages carry identifier of conversations in their headers.

5 Related Work and Conclusions

In this paper, we argue that abstracting Web services in terms of generic properties that describe conversation behaviors (e.g., transactional semantics and temporal constraints), will benefit several automation activities in cross-organizational application integration.

Several ongoing efforts recognize the need to extend the current technological infrastructure for Web services in order to effectively support cross-organization application integration [8]. Emerging standards such as BPEL4WS [13], WSCI [14], WS-Coordination [15], and WS-Transaction [7], layer up functionality related to composition and transactions on top of the basic Web service standards such as SOAP, WSDL, and UDDI [5]. BPEL4WS is particularly related to the work presented in this paper since, in addition to proposing a model for composing Web services, it also presents a way for defining the conversations that a Web service supports. However, the conversation functionality provided by BPEL4WS is essentially driven from its composition nature: in other words, BPEL4WS has been primarily designed as a composition language, in which the same formalism used for composition (a process) can also be used for defining conversations. As such, many of the properties needed for defining conversations (such as activation and compensation properties) are missing from BPEL4WS. WS-Coordination and WS-Transaction are also related to the work presented in this paper since they deal with conversations, and in particular with transactional conversations. However, their goal is that of providing a framework through which conversation properties can be enforced, rather than providing a conversation model. Other efforts which aim at addressing issues related to reliable coordination and transactional execution of integrated services include the OASIS Business Transaction Protocol (BTP)[16]. In general, the focus in this area is on extending traditional transaction techniques to provide reliable and dependable execution of integrated services. In the context of Web service conversations, WSCL [10] builds upon WSDL to describe valid interactions that a service can support, but focuses only on choreography aspects (i.e., acceptable message exchanges and the order in which they should occur). Within the research community, a Web service middleware which is based on the notion of *transactional attitudes* was proposed in [17]. The middleware monitors and controls client transactions according to transactional capabilities of provider services. Finally, we mention that other complementary proposals such as [18, 19] focus on enhancing the descriptions of services with non-functional properties (e.g., quality of service).

Our work makes complementary contributions to the efforts mentioned above. Our approach builds upon the building blocks of Web standards and provide a framework for defining extensible conversation meta-model, to enable the description of generic abstractions such as temporal constraints and implications of service conversations. We presented a conceptual model for the identified abstractions. We derived most of the concepts of abstractions from the analysis of real e-commerce Web portals. We presented a conversation management tool that supports the identified abstractions.

References

1. Aissi, S., Malu, P., Srinivasan, K.: E-Business Process Modeling: The Next Big Step. *IEEE Computer* **35** (2002) 55–62
2. Benatallah, B., Casati, F., eds.: Special Issue on Web Services. Volume 12 of Distributed and Parallel Databases., Kluwer Academic Publishers (2002)
3. Bussler, C.: B2B Protocol Standards and their Role in Semantic B2B Integration Engines. *IEEE Data Engineering Bulletin* **24** (2001) 3–11
4. Weikum, G., ed.: Special Issue on Infrastructure for Advanced E-Services. Volume 24 of *IEEE Data Engineering Bulletin*, IEEE Computer Society (2001)
5. Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S.: Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing* **6** (2002) 86–93
6. Cobb, E.E.: The Evolution of Distributed Component Architectures. In: Proceedings of the 9th International Conference on Cooperative Information Systems (CoopIS'01), Trento, Italy (2001)
7. Cabrera, F., Copeland, G., Cox, B., Freund, T., Klein, J., Storey, T., Thatte, S.: Web Services Transaction (WS-Transaction). <http://dev2dev.bea.com/techtrack/ws-transaction.jsp> (2002)
8. Papazoglou, M.P.: The World of e-Business: Web Services, Workflows, and Business Transactions. In: Proceedings of the CAiSE'02 International Workshop on Web Services, e-Business, and the Semantic Web (WES'02), Toronto, Canada (2002)
9. Sheng, Q.Z., Benatallah, B., Dumas, M., Mak, E.: SELF-SERV: A Platform for Rapid Composition of Web Services in a Peer-to-Peer Environment. In: Proceedings of the 28th Very Large Data Base Conference (VLDB'02), Hong Kong, China (2002)
10. Banerji, A., Bartolini, C., Beringer, D., Chopella, V., Govindarajan, K., Karp, A., Kuno, H., Lemon, M., Pogossiants, G., Sharma, S., Williams, S.: Web Services Conversation Language (WSCL). Note, W3C (2002) <http://www.w3.org/TR/wscl10>.
11. Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0. <http://www.w3.org/TR/xpath> (1999)
12. IBM WSTK Toolkit. (<http://alphaworks.ibm.com/tech/webservicestoolkit>)
13. Curbera, F., Golland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., Weerawarana, S.: Business Process Execution Language for Web Services (BPEL4WS). <http://dev2dev.bea.com/techtrack/BPEL4WS.jsp> (2002)
14. Arkin, A., Askary, S., Fordin, S., Jekeli, W., Kawaguchi, K., Orchard, D., Pogliani, S., Riemer, K., Struble, S., Takacsi-Nagy, P., Trickovic, I., Zimek, S.: Web Service Choreography Interface (WSCI). Note, W3C (2002) <http://www.w3.org/TR/wsci>.
15. Cabrera, F., Copeland, G., Freund, T., Klein, J., Langworthy, D., Orchard, D., Shewchuk, J., Storey, T.: Web Services Coordination (WS-Coordination). <http://dev2dev.bea.com/techtrack/ws-coordination.jsp> (2002)
16. OASIS Committee Specification: Business Transaction Protocol, version 1.0 (2002)
17. Mikalsen, T., Tai, S., Rouvellou, I.: Transactional Attitudes: Reliable Composition of Autonomous Web Services. In: Workshop on Dependable Middleware-based Systems (WDMS'02), Washington DC (2002)
18. Maximilien, E.M., Singh, M.P.: Conceptual Model of Web Service Reputation. *SIGMOD Record* **31** (2002)
19. O'Sullivan, J., Edmond, D., ter Hofstede, A.: What's in a Service? Distributed and Parallel Databases **12** (2002) 117–133