A Framework for the Server-Side Management of Conversations with Web Services

Liliana Ardissono, Davide Cardinio, Giovanna Petrone and Marino Segnan
Dipartimento di Informatica, Università di Torino
Corso Svizzera 185
10149 Torino, Italy
{liliana, giovanna, marino}@di.unito.it

ABSTRACT

The emerging standards for the publication of Web Services are focused on the specification of the static interfaces of the operations to be invoked, or on the service composition. Few efforts have been made to specify the interaction between a Web Service and the individual consumer, although this aspect is essential to the successful service execution. In fact, while "one-shot" services may be invoked in a straightforward way, the invocation of services requiring complex interactions, where multiple messages are needed to complete the service, depends on the fact that the consumer respects the business logic of the Web Service.

In this paper, we propose a framework for the server-side management of the interaction between a Web Service and its consumers. In our approach, the Web Service is in charge of assisting the consumer during the service invocation, by managing the interaction context and instructing the consumer about the operations that can be invoked and their actual parameters, at each step of the conversation. Our framework is based on the exchange of SOAP messages specifying the invocation of Java-based operations. Moreover, in order to support the interoperability with other software environments, the conversation flow specification is exported to a WSDL format that enables heterogeneous consumers to invoke the Web Service in a seamless way.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.2.12 [**Software Engineering**]: Interoperability

General Terms

Languages, Standardization

Keywords

Service Oriented Architectures, Tools and Technologies for Web Services Development

1. INTRODUCTION

Web Services are subject to several limitations that reduce their applicability to realistic domains. For instance, the emerging service publication standards, such as WSDL [32], support the specification of the static interfaces of elementary services. However, these standards do not enable the service provider to specify the flow of the operations to be invoked by the consumer during the service execution. Therefore, the management of the interaction between the consumer and the service provider is difficult, unless very simple tasks are requested. Moreover, these standards support the invocation of operations characterized by very specific signatures that fail to enable the peers to dynamically specify the actual parameters of the operations to be performed. Although this is not a problem when the requirements for the service can be specified by the consumer in a pre-determined way, it challenges the provision of highly interactive services, such as those supporting the customization of complex services and products. In fact, in that case, the Web Service has to identify the product features to be configured at run time, depending on the product structure and on the consumers' requirements.

The current work on workflow management is focused on the service composition in the Web but most of the proposed approaches assume a rather simple type of interaction with the suppliers whose services have to be invoked. For instance, BPEL4WS [13] supports the specification of complex service compositions and offers advanced solutions to the management of transactions [10] and failure recovery during the service execution [14]. However, the service compositions are defined as aggregations of standard WSDL operations with fixed parameters. Therefore, the service choreography has to be specified at the same level of detail. Different from BPEL4WS, Maamar etal [22] specify a service composition framework that also handles conversation aspects, but they are concerned with the establishment of the interaction with the service provider, more than with the management of the interaction during the service execution.

In order to make the service execution possible even when the involved suppliers require complex interactions, the invocation has to be modeled as a conversation where the peers may exchange several messages before the service is completed; e.g., requirements acquisition, negotiation and so forth. For instance, during the interaction with a Web Service supporting the configuration of medium complexity products, the specification of the item features can require more than one step; moreover, failures may occur and have to be repaired before the solution for the consumer is generated. Finally, in some cases, the Web Service may require to suspend the interaction, e.g., waiting for a sub-supplier or a human operator to contribute to the generation of the solution.

In this paper, we present a framework for the management of the communication between Web Service consumers and providers. The conversation model we propose supports complex interactions where several messages have to be exchanged before the service is completed, and the conversation may evolve in different ways, depending on the state and the needs of the two participants. We have defined our model by taking the speech-act theoretical model of

Copyright is held by the author/owner(s).

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-912-8/04/0005.

dialog management as a starting point [26, 12], but we have simplified it in order to meet the scalability and applicability requirements of Web Services. Our proposal has the following peculiarities:

- The conversation model clearly separates communicative behavior from the domain-level actions that the conversation participants perform during the service execution.
- In the conversation flow specification, the signatures of the operations admit generic arguments that can be instantiated with the actual parameters during the service execution.
- Our conversation model introduces explicit methods that enable the Web Service provider to guide the consumer during the service invocation, e.g., by informing it about the objects to which the generic parameters of the operations have to be bound.

Our contribution concerns the service execution, leaving the service discovery apart, as this represents a separate, complex activity, to be carried out by exploiting UDDI registries [30] and mediation agents [21].

The remainder of this paper is organized as follows: Section 2 provides some background about the dialog models based on Speech Act Theory and discusses some issues that limit their applicability to Web Services. Section 3 describes the approach to conversation flow specification we propose and Section 4 presents our conversation management framework at the conceptual level. Section 5 sketches the infrastructure for the development of conversational Web Service providers and consumers we are developing. Section 6 addresses the exploitation of our framework from the viewpoint of the Web Service consumer. Section 7 compares our proposal to the related work and Section 8 concludes the paper.

2. BACKGROUND

Starting from the approach proposed in the Speech Act Theory [7, 26], the communication between agents has often been modeled by exploiting communicative actions, called speech acts, that define the roles (speaker, hearer) played by the participants and separate the illocutionary force of the agents' messages from the object-level actions underlying the execution of the conversation turns.

As the speech acts represent individual communicative actions, various approaches have been proposed to model inter-agent communication at the conceptual level. For instance, in [28] Finite State Automata have been applied to specify the possible sequences of conversation turns to be performed. Moreover, plan-based approaches have been introduced to model goal-oriented behavior and to separate the management of the conversational behavior from the domain-dependent activities carried out by the agents [12, 11, 25].

Two of the most important contributions provided by the Speech Act Theory and the related dialog management models are the introduction of explicit conversation roles played by the interacting agents and the clear separation of communicative and domain-level behavior. Specifically, the dialog management models assume that the agents interact with each other to coordinate the domain-level activity they are cooperating at. These contributions have been recognized as basic elements for the management of the communication between distributed processes and, especially in the research about Multi-Agent systems, speech acts have been widely applied to manage the inter-agent communication. For instance, see the KQML [16] and the FIPA ACL [17] agent communication languages.

Although the speech-act based languages have been successfully applied to support the communication between large, but closed communities of interacting agents, we believe that they are hardly applicable to Web Services, given the scalability and applicability requirements of the open Internet environment. The point is that, in order to exploit Web Services in an effective way, the consumers should be able to start several e-business interactions with heterogeneous providers, also outside well established B2B relationships. This means that the management of the communication between Web Service consumers and providers should be loosely coupled and that the conversation management should be lightweight, at least at the consumer side. Unfortunately, the traditional speech-act based approaches fail to meet these requirements in several aspects. In particular:

- From a pragmatic point of view, the imposition of complex speech acts on Web Services is not realistic because current services are published by means of very simple languages such as RPC invocations or WSDL operations.
- At the conceptual level, these approaches propose a mixed-initiative model of dialog management, opposed to the client-server interaction adopted in the interaction with Web Services. In order to correctly interact with each other, the agents have to agree on a shared interaction protocol, to separately handle the interaction context and to autonomously regulate the turn management activity. Each agent is expected to maintain an internal representation of the information exchanged during the conversation and to exploit this information to track the evolution of the conversation with respect to the agreed protocol. In the world of Web Services, this requirement raises serious interoperability issues. In fact, in order to track the evolution of the conversation and to select at each step the next speech act to be performed, the consumer has to locally execute a copy of the possibly complex conversation management tool (e.g., a finite state automaton) employed by the provider.

In the rest of this paper, we describe a conversation model that addresses these requirements while preserving most of the concepts introduced in the speech-act based dialog management models. We define a flexible, but simple representation formalism supporting the specification of the correct sequence of turns to be exchanged between Web Service provider and consumer without the overhead of the pure speech-act model. Moreover, being based on the emerging standards for the publication of Web Services, our model can be applied to current Web Service providers and consumers in a reasonably straightforward way.

3. CONVERSATION FLOW SPECIFICATION

One way to satisfy the scalability and interoperability requirements described in the previous section is to place the overhead of the communication management on the service provider, which should assist the consumers during the service invocation, e.g., by controlling and guiding the invocation of the operations at each conversation step. Given the public specification of the operations offered by the service provider, the provider should maintain a local interaction context for each active conversation. Moreover, at each conversation step, the provider should enrich the outbound messages for the consumer with contextual and turn management information, in order to make the consumer aware about the eligible turns it may perform. In this way, the consumer can participate to the conversation by interpreting the instructions it receives and selecting the operation to invoke next.

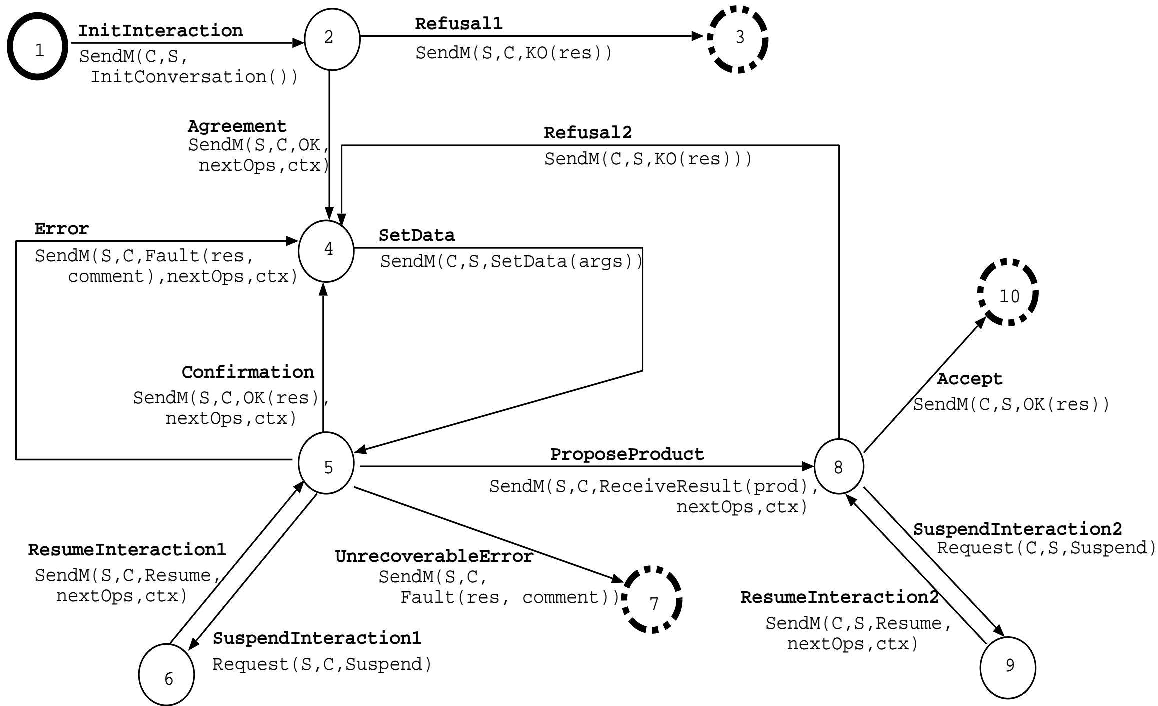

Figure 1: Conversation flow specification of a product customization Web Service.

This approach explicitly puts the service provider in control of the interaction, with the following advantages in the management of the overall service:

- At each conversation step, the consumer is guaranteed to invoke an operation that is eligible in the provider's business logic, without managing the interaction context. This aspect reduces the probability of a failure in the overall service.
- In the dynamic specification of the operations that can be invoked, a highly interactive provider may also specify the actual parameters of these operations. For instance, a configuration Web Service may guide the consumer in the choice of a large number of features by specifying, at each step, the features to be considered and the admissible values that the consumer may select.

With respect to the original representation of Speech Acts, we propose to simplify the specification of the interaction flow by modeling the interaction turns as generic conversational activities where the performed speech act (e.g., *Request, Inform*, etc.) is not specified. Moreover, in order to assist the consumer in the invocation of the operations offered by the service provider, the interaction turns have to be enriched with the necessary turn-management information. Each conversational action is thus represented as a simplified speech act (*SendM* message) where the sender asks the recipient to perform the operation specified as an argument. The conversational actions have the following structure:

SendM(sender, recipient, operation, nextOps, ctx)

where the last two arguments are optional. The arguments of the actions have the meaning reported below.

- *Sender*: message sender, which may be either the consumer *C*, or the service provider *S*.
- *Recipient*: receiver of the message (similar).
- *Operation*: operation that the sender invokes on the recipient. It should be noted that the actor of the requested operation may be omitted because it coincides with the recipient of the message. In other words, each message is aimed at requesting that the recipient performs the operation reported as the argument of the message.
- *NextOps*: list of the possible continuations of the conversation. As the service provider has the control of the interaction, this argument is only present in the messages to be received by the consumer. The argument includes the set of alternative operations offered by the provider which the consumer may invoke in the next conversation step. For each operation, the actual arguments are reported, together with their domains, i.e., the sets of values that the arguments can take.[1]
- *Ctx*: context argument, storing information about the state of the interaction. Similar to the *nextOps* argument, the *ctx* one is only present in the messages directed to the consumer. The structure of the context argument depends on the application domain. For instance, the context object can be empty in simple and deterministic interactions, where the consumer has at most one conversation turn to choose from, i.e., the next operation argument of the *SendM* message includes at most one element. We will not consider the context argument in the rest of this presentation.

The conversation flow of a Web Service can be specified by defin-

[1]The domain of a variable may change during the interaction with the Web Service and thus has to be specified each time the variable occurs as an argument of an operation. For instance, during the configuration of a product, the set of admissible feature values is progressively narrowed by constraint propagation (domain reduction).

ing the sequence of (simplified) speech acts that each conversation partner has to perform. At the conceptual level, this can be done by applying Finite State Automata (FSA), which ar a simple, well-known and easily understandable formalism, and have been applied in several proposals for the management of Web Service choreography; e.g., see [8, 9].

We have selected the customization of bicycles as a sample application domain to present our conversation model. Figure 1 shows the FSA representing the conversation flow specification of the service. The states of the automaton represent the dialog states: the plain circles denote the conversation states and the thick one (state 1) is the initial state. The thick dotted states (3, 7, 10) are final states of the dialog. The conversation turns are represented as send message (*SendM*) activities specifying the related arguments: message sender, recipient, invoked operation, and so forth. The turns to be performed by each of the peers are specified as labels of the arcs. The states having more than one output arc represent situations where alternative turns may be performed by the peer. Notice that we have also labeled the arcs with a boldface identifier to simplify the identification of the conversation turns in the rest of this presentation.

The interaction starts with the consumer *C* (e.g., the personal assistant of an end-customer) asking the service provider *S* to start the interaction (*InitInteraction: SendM(C, S, InitConversation)*). The provider may accept to perform the request (*Agreement: SendM(S, C, OK, nextOps, ctx)*), or reject it (*Refusal1: SendM(S, C, KO(res))*).

If the provider accepts the interaction, a data acquisition phase starts during which the consumer specifies (*SetData* arc) the end-customer's data and her/his requirements on the product to be customized. The consumer informs the provider about these types of information by invoking the *SetData(args)* operation on the list of data/features to be set. For instance, the consumer might set the number of gears of the bicycle equal to 4. Notice that the consumer may retrieve this list from the *nextOps* argument of the last message it received. *SetData* is an example of the generic kind of operation we have introduced in our conversation model. Specifically, the *args* parameter does not refer to any particular end-customer data or product feature. When the consumer invokes the operation, it binds the parameter to the actual list of features to be set.[2]

When the consumer specifies a set of data, e.g., some features of the bicycle, the service provider may react in different ways. Specifically, it may:

- Confirm the successful data acquisition (*Confirmation* arc) and enable another invocation of the *SetData* operation.
- Notify the consumer that there was a failure in the product customization process (*Failure*) and enable the consumer to select other values for the conflicting features.
- Notify the consumer that the interaction is suspended (*SuspendInteraction1*) and resume it later on (*ResumeInteraction1*). The possibility to suspend and restart the interaction enables the service provider to synchronize with its own sub-suppliers, if necessary.

Finally, the data acquisition phase can end negatively or positively. The first case happens when an unrecoverable error occurs; for instance, the end-customers' requirements are incompatible with one another and thus they cannot be concurrently satisfied. If, however, the customization process succeeds, the provider continues the interaction by proposing the product (*ProposeProduct*). The consumer may react in three different ways. If it accepts the proposal (*Accept*), the conversation terminates; note that a payment phase should start, but we omit it for simplicity. If the consumer rejects the proposal (*Refusal2*), the conversation goes back to the data acquisition phase to start the configuration of another product. Otherwise, the consumer might suspend the interaction (*SuspendInteraction2*) and resume it later on (*ResumeInteraction2*). This alternative behavior enables the consumer to submit the proposal to the end-customer and get the authorization to accept/reject it.
Looking at the FSA, we can see that two kinds of object-level operations may be the arguments of a conversation turn:

- *Domain-dependent operations*, such as *SetData*, offered by the service provider to execute its service.
- *Domain-independent operations*, such as *InitConversation*, *OK*, *Fault*, *Suspend*, *Resume* and *ReceiveResult*, focused on the coordination of the conversation between the partners.

Except for special cases, such as *InitConversation*, the domain-independent operations have to be offered by the provider as well as by the consumer. For instance, the consumer must offer the *ReceiveResult* operation, which corresponds to the WSDL *output messages* sent by the service provider to acknowledge the service execution and communicate results.

```
SendM message:
  operation = Fault(nrGear,
                "invalid parameter value");
  nextOps = {SetFeatures(nrGear);
             PostponeSet(nrGear);
             Suspend(id)};
  ctx = null;
```

Figure 2: Sample *SendM* message.

Figure 2 shows a sample *SendM* message sent by the service provider to the consumer during the configuration of a bicycle. In the message, the provider sends a failure message: *Fault(nrGear, "invalid parameter value")*. This message notifies the unsuccessful execution of a previously invoked operation: the consumer previously set the *nrGear* parameter to an invalid value. The provider also specifies that the consumer may invoke *SetData* to set the number of gears again, *PostponeSet* to postpone the setting to a later stage of the interaction, or it may suspend the conversation.

4. INTERACTION MANAGEMENT

In order to manage the conversation at both sides, the two peers should run, respectively, a *Conversation Manager* and a *Conversation Client* modules. The former is employed by the provider to manage the interaction with the consumers, which would only rely on the lightweight Conversation Client to parse the incoming messages and return the responses. The situation is shown in Figure 3, that sketches the architecture of the proposed framework.

Notice that each turn (*SendM* message) is an asynchronous message that one of the participants should send to continue the conversation. If the message does not reach the recipient. the interaction has to be suspended, by the conversation module, with a time out.

[2]The selection of the feature values would not be possible without a suitable binding phase, where the consumer analyzes the product structure specified in the Web Service ontology. Sharing the product ontology is thus necessary to let the consumer understand the individual product features to be set. See Section 6.

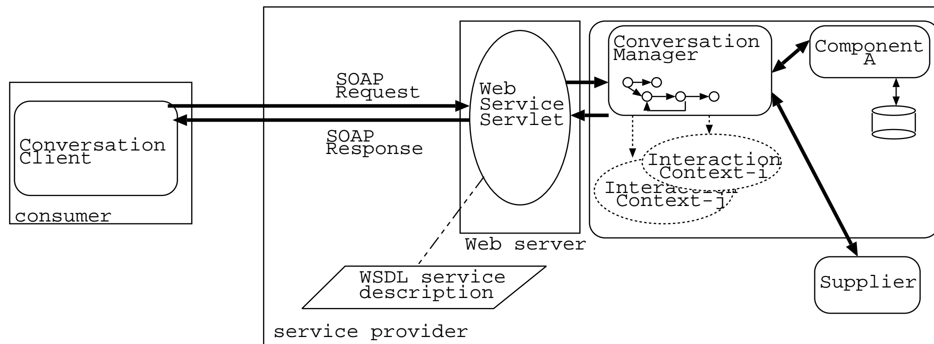

Figure 3: Interaction with a web service provider.

4.1 Conversation Manager

The Conversation Manager should exploit a conversation automaton, such as the one depicted in Figure 1, to control the service provider's behavior. For each interaction session with a consumer, the Conversation Manager of the provider should maintain the active state of the interaction as a description of the contextual information concerning the specific conversation, moving the state from the initial state of the automaton to the other ones, depending on the messages it sends or receives. The Conversation Manager would also be in charge of computing the next operations available to the consumer and to include them in the outbound *SendM* messages.

The operations that the consumer may perform in the next conversation turn can be identified by performing a look-ahead in the conversation automaton, starting from the active state. In fact, the arcs of the automaton are associated to the signatures of the operations to be invoked. However, if the next operations have generic parameters, such as the *SetData* operation of the product configuration service, their signature is underspecified and the Conversation Manager has to retrieve the actual parameters to include in the next operations specification. A highly interactive service provider has thus the responsibility of exposing the list of data that may be specified, at each step of the interaction.

We are not concerned about how the service provider addresses this issue because it strictly depends on the provider's implementation and has thus to be solved case by case. We want however to point out that the inference engines typically exploited to carry out problem solving activities are designed to stop and question the calling modules when they need to set some variables in order to carry the problem solving activity out. For instance, the configuration systems based on constraint satisfaction, such as the ILOG JConfigurator [20], explores the solution space by generating partial solutions, which can be further specified, until all the variables of the problem are instantiated, in an interactive way. Similarly, BDI agent development tools, such as JACK [2], support the development of agents that can interleave planning activities with sensing actions, aimed at acquiring the values of their unbound variables.

4.2 Conversation Client

The Conversation Client is a lightweight conversation component and has to be downloaded by the consumer before invoking the Web Service. This component enriches the consumer's conversation capabilities with the following functionalities:

1. Establishing an endpoint to catch the incoming messages. The Conversation Client provides the domain-independent operations, such as *Suspend*, *Resume*, *ReceiveResult*, etc., that have to be offered by the consumer in order to manage the interaction with the service provider.

2. Interpreting the inbound messages that the consumer receives from the service provider. The Conversation Client extracts the eligible continuations of the interaction by parsing the *nextOps* argument of the inbound messages. For each operation, this list reports the parameters and their domains. In [24, 5], we provide an XML schema specifying the format adopted in the definition of the operations. Notice that, given the list of alternatives, the consumer is responsible for choosing the most convenient option and deciding the details of the invocation, depending on its own business logic.

3. Guiding the consumer in the instantiation of the selected operation. When the consumer sets the values of the parameters of the invoked operation, the Conversation Client performs type checks to ensure that the selected values belong to the associated domains. The Client does not finalize the instantiation of the operation until all the parameters are correctly set by the consumer.

4. Generating the outbound *SendM* messages and sending them to the service provider.

Item 3 deserves further discussion because it plays a critical role in the enforcement of the correct invocation of the provider, therefore reducing the occurrences of failures in the service execution. In fact, the consumer might make a mistake when binding the arguments of the object-level operation to be invoked. For instance, the *SetData* operation could be invoked by specifying values that violate the parameter domains, with a consequent failure in the execution of the operation at the provider side and the need to compensate the failure. Of course, although the local checks support the repair to several problems within the consumer, they do not prevent the failure of the overall product customization. Problems might occur, for instance, if the overall set of features specified during the customization of the product is inconsistent. This general type of failure is detected at the service provider side and is handled as specified in the *UnrecoverableError* transition of the automaton depicted in Figure 1.

5. A FRAMEWORK FOR THE SERVER-SIDE MANAGEMENT OF COMPLEX INTERACTIONS

We are developing a set of Java libraries aimed at facilitating the development of a Conversation Manager and a Conversation Client modules supporting the communication between Web Service providers and consumers. For the implementation, we are exploiting the Sun Microsystem Web Service Developer Pack [29] and in particular the JAXP-RPC package of the Pack.

We have developed a Java-based Conversation Manager module that enables the Web Service provider to control the communication with the interacting consumer applications. The proposed architecture of the Web Service Provider is shown in Figure 3. A Servlet supports the (SOAP) HTTP-based communication with the consumer, by catching the incoming requests and forwarding them to the Conversation Manager for their management.

The Conversation Manager is the core of the Servlet listening to the incoming requests, invoking the appropriate components to execute the services and sending the SOAP response messages to the consumer applications. Depending on the binding to the Web Service provider's business logic, this module can invoke different types of components to provide the service. For instance, a service could rely on an internal component, such as Component A in Figure 3, or by an external Supplier.

The Conversation Manager may execute a conversation flow automaton for the management of the interaction sessions with the consumers in order to compute the possible continuations of each interaction. Although a general infrastructure, supporting the definition of general purpose conversation automata, is not yet available, we have developed a prototype Conversation Manager that implements the FSA shown in Figure 1 and that can be easily customized to satisfy the interaction requirements of different application domains.

As previously mentioned, for interoperability purposes we rely on an XML representation of the signatures of the operations and of the parameter domains. In our infrastructure, the generation of Java classes and objects starting from the XML representation is carried out by exploiting the JAXB tool offered by the Pack.

We have also developed a Java-based Conversation Client that may be downloaded by the consumer and run in order to handle the interaction with a Web Service.

The Conversation Client catches the messages that the Web Service sends to the consumer and extracts the operations that may be invoked on the Web Service next (*nextOps*). Moreover, the Client supports the generation of the *SendM* messages to be sent in response to the Web Service; for the generation of messages, a subset of the libraries available to the Conversation Manager is exploited.

Similar to the Conversation Manager, the execution of the Client has the prerequisite that the consumer binds the invocation of the operations to its own business logic. This issue is discussed in detail in Section 6. Here, we only mention the fact that the Conversation Client offers suitable APIs for the invocation of its functionalities; e.g., extraction of next operations with parameters and domains, instantiation of operations and message generation. In particular, the instantiation of the operations enables the consumer to set the values of the parameters. During the execution of the instantiation procedure, the Conversation Client checks whether the selected values belong to the domains of the parameters, returning a failure in case of type violation.

It should be noted that the XML representation of the *SendM* messages supports the interoperability between service consumers and providers, but the Conversation Client we implemented can only be run in a Java-based environment. The idea is therefore that other versions of the Client should be implemented to provide advanced conversation capabilities in different environments such as, for instance, .Net.

5.1 WSDL Specification of Operations

It should be noticed that JWSDP allows the development of Java code that is automatically translated to SOAP messages following the WSDL specification. In our work, we have exploited this feature to support the interoperability of a Java-based Web Service with consumers developed in heterogeneous environments.

More specifically, we enable the service provider running our Conversation Manager to handle inbound and outbound messages by applying the SOAP communication protocol to exchange Java-based messages. However, we also provide a declarative representation of the format of the *SendM* messages, including the WSDL specification of the object-level operations requested by means of the *SendM* messages. Following the WSDL specification, the schema defining datatypes and object-level operations, such as *SetData*, are generated, as well as the ports and bindings needed to interpret WSDL messages.[3] Thus, a generic consumer, which does not exploit our Conversation Client, may invoke the Conversation Manager by conforming to the WSDL specification of its services. Moreover, the consumer may be guided by the provider in the correct invocation of services if it correctly parses the *SendM* messages it receives, by suitably extracting the *nextOperations* information stored in the messages.

From our perspective, the *SendM* operation is a service offered by both conversation participants. In fact, the consumer needs to offer the *SendM* operation in order to receive messages from the service provider. Therefore this operation must be added to the WSDL specification of both types of peers. However, the publication of the *SendM* operation does not impose any development overhead because it is automatically generated by our framework. Figure 4 shows a portion of the WSDL declarations generated to support the management of *SendM* messages and includes the specification of the *SetData* object-level operation specific for the customization of products. For readability purposes we have removed some information from the WSDL specification. See [3] for the complete representation of this portion of the product customization service.

6. DISCUSSION

In this section, we discuss the binding requirements imposed by our approach on the administrator of the service consumer. This aspect is very important, as binding the consumer's business logic to the invocation of operations on the Web Service is a prerequisite to the service execution.

The first proposals for the management of Web Services assumed a synchronous type of interaction between service provider and consumer. Although this requirement imposed that the consumer waits for the result message each time it invokes an operation on the provider, the synchronization facilitated the set up of the communication. The consumer could in fact associate the invocation of operations to its own business logic by inspecting the WSDL ports published by the provider and by connecting the invocation of operations to its own internal processes. The results of the invoked operations could then be collected by the consumer by exploiting an implicit return channel.

In order to overcome the limitations of synchronous communi-

[3] The infrastructure enforces the WS-I standards on complex object type to ensure interoperability between Web Services providers and consumers developed in heterogeneous environments.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MyConversationService" targetNamespace="urn:Foo"
     xmlns:tns="urn:Foo" xmlns="http://schemas.xmlsoap.org/wsdl/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <types>
    <schema targetNamespace="http://java.sun.com/jax-rpc-ri/internal"
          xmlns:tns="http://java.sun.com/jax-rpc-ri/internal" ..... ">
      <import namespace=..."/>
      <complexType name="vector"> ...
      <complexType name="collection">
        <complexContent>
          <restriction base="soap11-enc:Array">
            <attribute ref="soap11-enc:arrayType" wsdl:arrayType="anyType[]"/>
          </restriction>
        </complexContent>
      </complexType>
        ...
      <complexType name="SendMArgs">
        <sequence>
          <element name="context" type="anyType"/>
          <element name="convId" type="string"/>
          <element name="currentOperation" type="string"/>
          <element name="nextOperations" type="ns2:vector"/>
        </sequence>
      </complexType>
    </schema>
  </types>

  <message name="Conversation_sendM">
    <part name="SendMArgs_1" type="tns:SendMArgs"/>
  </message> ... DEFINITION OF OBJECT-LEVEL OPERATIONS, i.e., SetData, FOLLOWS ...

  <portType name="Conversation">
    <operation name="sendM" parameterOrder="SendMArgs_1">
      <input message="tns:Conversation_sendM"/>
      <output message="tns:Conversation_sendMResponse"/>
    </operation>
  </portType>

  <binding name="ConversationBinding" type="tns:Conversation">
    <operation name="sendM">
      <input>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" ... />
      </input>
      <output>
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" ... />
      </output>
      <soap:operation soapAction=""/>
    </operation>
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
  </binding>

  <service name="MyConversationService">
    <port name="ConversationPort" binding="tns:ConversationBinding">
      <soap:address location="REPLACE_WITH_ACTUAL_URL"/>
    </port>
  </service>
</definitions>
```

Figure 4: Portion of the WSDL specification of a product customization service.

cation, the emerging Web Services standards have recently evolved to support asynchronous communication based on *call back* messages; see [1]. This change has made the binding phase more complex because now the Web Service has to publish different port types, one for each conversation participant, and the consumer becomes itself a Web Service that offers the ports required by the role it wants to play in the interaction. For instance, in our configuration example, the consumer would play a *purchaser* role that specifies all the operations to be offered in order to participate to a purchase business interaction. Roughly speaking, the role-based port types represent the participants' viewpoints on the shared interaction protocol.

In our approach, binding the consumer's business logic is not dramatically different from the previous case. Similar to BPEL4WS1.1, our conversation model relies on the definition of conversational roles for the specification of the services that have to be offered by each participant; e.g., consider the domain-dependent and domain-independent operations offered by service providers and consumers. Moreover, our model assumes that the service provider publishes the conversation flow specification, so that the consumer can inspect the specification and bind the invocation of services to its own internal processes. At the same time, the conversation management features offered by our framework are partially handled by the Conversation Client libraries we offer. Specifically:

- The public conversation flow specification published by the service provider describes all the possible sequences of turns between the interactants and the next operations represent a subset of those sequences. Therefore, the next operations are not discovered by the consumer starting from zero knowledge. The consumer can analyze the conversation flow at business logic binding time; it then acquires at run time the subset of operations that can be invoked in the context of the individual interaction session.

- The presence of operations having generic arguments adds complexity to the specification formalism, hiding the individual parameters of the operations from the conversation flow specification. However, the type of information needed to enable the consumer to bind the invocation of operations (with instantiated parameters) to its own business logic is the same as that of the standard fixed parameters approach. The only difference is that, instead of directly analyzing the parameters specified in the signatures of the operations, the consumer has to analyze a separate representation structure, published by the service provider, where the possible parameters are described. For instance, in our sample domain, the Service must publish the specification of the list of product features that can be arguments of the *SetData* operation. The publication of this information enables the consumer to bind the possible invocations of the operations to its own business logic. For example, the operation could have as actual parameters the product features defined in the public specification of the product structure; e.g., gears, frame, and so forth.

- Our Conversation Client implements and publishes the domain independent operations that the consumer has to offer in order to receive the provider's messages, interpret them and send the responses to the provider. Therefore, no overhead is imposed on the consumer to set up the conversation framework; it only has to download the Client and associate internal processes to object-level actions. Moreover, at run time, the Conversation Client supports the consumer by extracting from the next operations information the actual parameters to be applied to the invoked operations.

As already mentioned, our conversation model handles operations having generic arguments in order to overcome the limitations of the emerging Web Service publication standards. Indeed, the exploitation of operations characterized by fixed parameters would not support a concise and efficient specification of the conversation flow of highly interactive service providers. Moreover, it would not take priorities in the assignment of parameters into account. The sample domain that we selected provides an example of this problem. For instance, in the configuration of complex products, the operations to be performed can be clearly defined, e.g., specifying the needed components and their features. However, the features whose values have to be set depend on the optionals required by the consumer and the expressed requirements may change the strategy adopted by the configuration Web Server in the specification of the other product features. Therefore, assuming that the consumer has bound the setting of all the possible features to its own business logic, the Web Service has to dynamically determine the correct invocation of operations during the exploration of the search space, by taking contextual information about the interaction into account.

7. RELATED WORK

The Semantic Web community [27] is defining standards for the publication of Web Services that support the specification of the domain ontology underlying the service, the meaning of the operations to be invoked and the service choreography. The main advantage with respect to the pure XML approach is that the service offered by a provider can be unambiguously understood by the (UDDI) registries, therefore enhancing their capability to redirect consumers to the most suitable providers. In order to facilitate the interoperability between service consumers and providers, some recent work also proposes prototype infrastructures for the run-time conversation management. These infrastructures rely on the semantic representation of the services to guide the interaction, e.g., by instructing the consumer during the service invocation [23].

Although the semantic Web approach is the most promising solution to the interoperability in the internet, the current proposals are still too complex to be applied in real-world examples. For instance, these approaches rely on sophisticated representations of the services to be invoked; although translation tools assist the binding to the consumers' business logics, this remains a rather complex process. Moreover, the selection of the operations to be invoked, depending on the service choreography, is based on the exploitation of inference engines, such as rule-based ones, imposing relevant overhead on the management of the interaction. In our work we are deeply concerned with the scalability and applicability constraints imposed by the Web. For this reason, we try to reduce the complexity added by semantic information as much as possible, and to handle the interaction between service consumer and provider in a lightweight fashion, at least at the consumer side.

Other XML-based standards for the specification of the conversation flow in e-business interactions with Web Services have been recently proposed and are currently submitted as W3C standards. For instance, WSCL (Web Services Conversation Language [31]) and WSCI (Web Services Choreography Interface [6]) introduce an explicit representation of Web Services interaction processes, aimed at defining the admissible sequences of messages to be exchanged by the peers. To this purpose, WSCL exploits a sequence diagram model that the peers should interpret to handle the conversation, while WSCI introduces the notion of interaction process, with the specification of timing constraints on the service invoca-

tion. Moreover, cpXML (IBM's Conversation Support [18]) introduces an explicit notion of Conversational Policy as a machine readable specification of a pattern of message exchange in a conversation, which can be used to make the interaction with complex Web Services easier from the consumer application viewpoint.

Our interaction model differs from the previously mentioned ones because they assume that the participants share the interaction protocol and autonomously plan the conversational behavior, according to that protocol. In those approaches each participant separately maintains its own internal record of the conversation state, which is necessary to understand how to continue the interaction. In contrast, we propose that only the service provider maintains the state of the conversation and that the consumer application is assisted in the service invocation. Another difference with respect to WSCL and WSCI is the fact they conform to WSDL in the specification of *request-response* and *solicit-response* operations; thus, they cannot support a fine-grained specification of the conversation turns.

Among the mentioned approaches, cpXML deserves a more thorough comparison with our work, because it aims at developing an implementation framework supporting the communication, while the other approaches only contribute with a specification of a communication protocol. In cpXML the consumer and provider roles require the same effort from the conversational point of view. In contrast, we try to simplify the implementation of the consumer; moreover, the consumer is supplied with the minimum amount of information needed to interact with the provider. This is positive, because it supports the implementation of lightweight consumers. At the same time, our approach does not impose extra overhead on the service provider, for two reasons. First, the provider has to maintain the context for each interaction session, otherwise, it would not be able to correctly execute the service requests. Second, the provider knows the details of the execution of its own services. Therefore, it may suitably guide the consumers during the management of complex interactions.

Both cpXML and our work aim at decoupling the peers' business logics by mediating their interaction by means of the conversational activity. However, our model has the potential to separate the consumer from the provider in a clearer way because it does not impose the execution of a specific conversation policy.

8. CONCLUSIONS

We have presented a conversational model aimed at filling the conversational gap between consumers and service providers in the internet. Our approach is based on the idea that the accessibility of Web Services and the establishment of short-term interactions with consumers are facilitated by the following factors:

- A flexible but simple conversation model supporting the exchange of asynchronous messages during the interaction.
- The provider-side control of the interaction, aimed at enforcing the correct invocation of services and at minimizing the communication overhead posed on the consumer. Our conversation model supports the participation of the consumer in the interaction with the service provider by dynamically specifying the eligible operations that can be invoked at each step.
- The possibility to perform local type and consistency checks at the consumer side, before invoking the operations on the provider.

As discussed in [15], the invocation of Web Services should be as seamless as possible for the consumers that should not be forced to implement specific procedures for the service invocation. Although the current Web Service specification frameworks address this constraint by providing stubs that can be invoked by consumers, we aim at providing interactivity between the peers and we take into account the fact that the consumer has to be additionally guided by the provider whenever the sequence of operations to be invoked and their actual parameters have to be specified during the service fruition. To this extent, we propose a framework that enables the consumer to correctly invoke the service provider and to locally perform simple type checks aimed at enforcing the correct invocation of operations and at minimizing the occurrence of failures, with the consequent need to exchange repair messages.

Our framework supports the consumer by offering a Conversation Client that may be downloaded and run to handle the interaction with a Web Service after having bound the invocation of operations to the consumer's business logic. Moreover, the framework offers the libraries needed by the service provider to generate the messages guiding the consumer in the correct invocation of services and the libraries for the specification and implementation of the conversational flow automaton held by the Web Service.

At the current stage, the automaton specifying the service provider's conversation flow is represented in an XML language that we defined by taking inspiration from the WSFL flow specification language; see [24, 4, 5]. Moreover, our infrastructure offers the Conversation Manager module that can be exploited by the service provider to control the interaction with the customer, according to the conversation automaton. As discussed in Section 5.1, the representation of the conversation turns is compatible with the WSDL format: the only difference is that the arguments of the *SendM* operations may be object-level operations, or turn-management information. The conversation flow could therefore be represented in a standard process language, such as, for instance, BPEL4WS1.1. In our future work, we want to investigate this issue and see whether a different version of our Conversation Manager can be developed and plugged into a suitable flow engine, e.g., BPWS4J [19], in order to integrate workflow management and conversation management into a single process.

9. REFERENCES

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. T. I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services version 1.1j. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, 2003.

[2] AOS. JACK Intelligent Agents [tm]. http://www.agent-software.com/shared/products/index.html, 2002.

[3] Appendix. Automatic generation of a WSDL interface for the invocation of a service provider. http://www.di.unito.it/l̃iliana/appendix.txt, 2003.

[4] L. Ardissono, A. Goy, and G. Petrone. Enabling conversations with Web Services. In *Proc. 2nd Int. Joint. Conf. on Autonomous Agents and MultiAgent Systems*, pages 819–826, Melbourne, Australia, 2003.

[5] L. Ardissono, G. Petrone, and M. Segnan. Enabling flexible interaction with web services. Forthcoming.

[6] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimek. Web Service Choreography Interface 1.0. http://ifr.sap.com/wsci/specification/wsci-specp10.html, 2002.

[7] J. Austin. *How to Do Things with Words*. Harvard University Press, Cambridge, Mass, 1962.

[8] B. Benatallah, F. Casati, F. Toumani, and R. Hamadi. Conceptual modeling of Web Service conversations. In *Proc. Advanced Information Systems Engineering, 15th Int. Conf., CAiSE 2003*, Klagenfurt, Austria, 2003.

[9] D. Berardi, F. D. Rosa, L. D. Santis, and M. Mecella. Finite state automata as a conceptual model of *e*-services. In *Proc. Integrated Design and Process Technology (IDPT 2003)*, Austin, Texas, 2003.

[10] F. Cabrera, G. Copeland, T. Freund, J. Klein, D. Langworthy, D. Orchard, J. Shewchuk, and T. Storey. Web Services Coordination (WS-Coordination). http://www-106.ibm.com/developerworks/library/ws-coor/, 2002.

[11] J. Chu-Carroll and S. Carberry. Collaborative response generation in planning dialogues. *Computational Linguistics*, 24(3):355–400, 1998.

[12] P. Cohen and H. Levesque. Rational interaction as the basis for communication. In P. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in communication*, pages 221–255. MIT Press, 1990.

[13] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services, version 1.0. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, 2002.

[14] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. The next step in Web Services. *Communications of the ACM, Special Issue on Service-Oriented Computing*, 46(10), 2003.

[15] H. Deo. The need for a dynamic invocation framework. *WebServices.org*, http://www.webservices.org/index.php/article/articleview/469/1/24, 2002.

[16] T. Finin, Y. Labrou, and J. Mayfield. KQML as an agent communication language. In J. Bradshaw, editor, *Software Agents*. MIT Press, Cambridge, 1995.

[17] FIPA. Foundation for Physical Intelligent Agents. http://www.fipa.org/, 2000.

[18] J. Hanson, P. Nandi, and D. Levine. Conversation-enabled Web Services for agents and e-Business. In *Proc. of the Int. Conf. on Internet Computing (IC-02)*, pages 791–796, Las Vegas, Nevada, 2002.

[19] IBM AlphaWorks. BPWS4J. http://www.alphaworks.ibm.com/tech/bpws4j, 2003.

[20] ILOG. ILOG JConfigurator. http://www.ilog.com/products/jconfigurator/, 2002.

[21] M. Klusch and K. Sycara. Brokering and matchmaking for coordination of agent societies: A survey. In A. Omicini, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, chapter 8, pages 197–224. Springer-Verlag, 2001.

[22] Z. Maamar, B. Benatallah, and W. Mansoor. Service chart diagrams - description & application. In *Proc. of WWW'2003*, Budapest, 2003.

[23] M. Paolucci, K. Sycara, T. Nishimura, and N. Srinivasan. Toward a semantic web e-commerce. In *Proc. of 6th Int. Conf. on Business Information Systems (BIS'2003)*, Colorado Springs, Colorado, 2003.

[24] G. Petrone. Managing flexible interaction with Web Services. In *AAMAS-03 workshop on Web-services and agent-based engineering*, pages 41–48, Melbourne, Australia, 2003.

[25] C. Rich, D. McDonald, N. Lesh, and C. Sidner. COLLAGEN: Java middleware for collaborative agents services with multiple suppliers. http://www.merl.com/projects/collagen, 2002.

[26] J. Searle. Indirect speech acts. In P. Cole and J. Morgan, editors, *Syntax and Semantics: Speech Acts*, volume 3, pages 59–82. Academic Press, New York, 1975.

[27] Web Services Coalition. DAML-S: Web Service description for the Semantic Web. In *Proc. Int. Semantic Web Conference*, pages 348–363, Chia Laguna, Italy, 2002.

[28] A. Stein and E. Maier. Structuring collaborative information-seeking dialogues. *Knowledge-Based Systems*, 8(2-3):82–93, 1994.

[29] Sun Microsystems, Inc. Java Web Services Development Pack 1.3. http://java.sun.com/webservices/webservicespack.html/, 2003.

[30] UDDI Org. Universal Description, Discovery and Integration of Business for the Web. http://www.uddi.org/.

[31] W3C. Web Services Conversation Language (WSCL). http://www.w3.org/TR/wscl10, 2002.

[32] W3C. Web Services Definition Language. http://www.w3.org/TR/wsdl, 2002.