Conversations + Interfaces = Business Logic

Harumi Kuno[1], Mike Lemon[1], Alan Karp[1], and Dorothea Beringer[2]

[1] Hewlett-Packard Laboratories
1501 Page Mill Road, MS 1U-14
Palo Alto, CA 94304 USA
[2] Hewlett-Packard Co.
19320 Pruneridge Avenue, MS 49b-26
Cupertino, CA 95014 USA
{harumi_kuno,mike_lemon,alan_karp,dorothea_beringer}@hp.com

Abstract. In the traditional application model, services are tightly coupled with the processes they support. For example, whenever a server's process changes, existing clients using that process must also be updated. However, electronic commerce is moving toward e-service based interactions, where corporate enterprises use e-services to interact with each other dynamically, and a service in one enterprise could spontaneously decide to engage a service fronted by another enterprise. We clarify here the relationship between currently developing standards such as UDDI, WSDL, and WSCL, and propose a conversation controller mechanism that leverages such standards to direct services in their conversations. We can thus treat services as pools of methods, independent of the conversations they support. Even method names can be decided on independently of the conversations. Services can spontaneously discover each other and then engage in complicated interactions without the services themselves having to explicitly support conversational logic. The dynamism and flexibility enabled by this decoupling is the essential difference between applications offered over the web and e-services.

1 Introduction

Electronic commerce is moving towards a model of e-service based interactions, where corporate enterprises use e-services to interact with each other dynamically [1]. For example, a procurement service in one enterprise could spontaneously decide to engage a storefront service fronted by another enterprise. These services can communicate by exchanging messages using some common transport (e.g., HTTP) and message format (e.g., SOAP).

However, suppose that the storefront service expects the message exchanges to follow a specific pattern (conversation), such as the conversation depicted in Figure 1 (shown from the perspective of the storefront service). Service developers must now address several issues. How does the client service know what conversations the storefront service supports? Does the storefront service developer have to code the conversation-controlling logic directly into the service? If

F. Casati, D. Georgakopoulos, M.-C. Shan (Eds.): TES 2001, LNCS 2193, pp. 30–43, 2001.

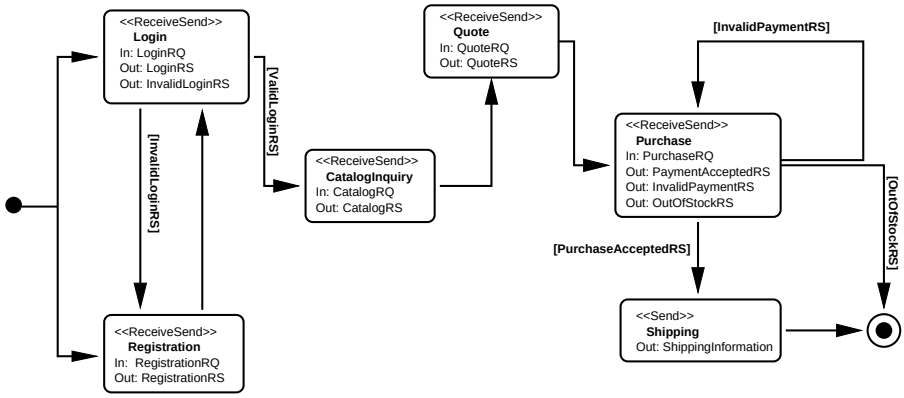

Fig. 1. An Example Conversation Depicted as a UML Activity Diagram. Interactions are represented as action states.

so, do developers have to re-implement the client and storefront services each time a new message exchange is added to the supported conversation?

A model of dynamic service interactions thus imposes the following requirements:

1. Services must be able to describe themselves, and clients must be able to discover them.
2. A service must be able to describe its abstract interfaces and protocol bindings so that clients can figure out how to invoke it.
3. A service must be able to describe the kinds of interactions (conversations) that it supports (e.g., that it expects clients to login before they can request a catalog) so that clients can engage in complex exchanges with the service.

The Universal Description Discovery and Integration (UDDI) [2,3,4] specifications address the first problem by defining a way to publish and discover information about Web services. The Web Services Description Language (WSDL) [5] addresses the second problem, defining a general purpose XML language for describing the interface and protocol bindings of network services. The Web Services Conversation Language (WSCL)[1] [6,7] addresses the last problem, providing a standard way to model the public processes of a service, thus enabling network services to participate in rich interactions. Together, UDDI, WSDL, and WSCL enable developers to implement web services capable of spontaneously engaging in dynamic and complex inter-enterprise interactions.

In this paper, we describe how these standards enable us to separate interface logic from conversation logic. We begin by discussing our perspective of e-services as pools of functional endpoints that can be composed using conversations (Section 2). In Section 3 we provide a brief overview of UDDI, WSDL, and WSCL,

[1] WSCL was originally named the *Conversation Definition Language* (CDL).

and describe how they enable the creation of specifications describing service endpoints and conversations. We have implemented a prototype conversation controller that leverages service interface descriptions and conversation specifications expressed using WSCL and WSDL, which we describe in Section 4. We present related work in Section 5, and summarize our conclusions and discuss future directions in Section 6.

2 Approach

E-services are much more loosely coupled than traditional distributed applications. This difference impacts both the requirements and usage models for e-services. E-services are deployed on the behalf of diverse enterprises, and the programmers who implement them are unlikely to collaborate with each other during development. However, the primary function of e-services is to enable business-to-business interactions. Therefore, e-services must support very flexible, dynamic bindings. E-services should be able to discover new services and interact with them dynamically without requiring programming changes to either service. This distinction is what separates e-services from applications delivered over the web.

In our model, e-services interact by exchanging messages. Each message can be expressed as a structured document (*e.g.*, using XML) that is an instance of some document type (*e.g.*, expressed using XML Schema). A message may be wrapped (nested) in an encompassing document, which can serve as an envelope that adds contextual (delivery or conversation specific) information (*e.g.*, using SOAP).

We define a conversation to be a sequence of message exchanges (interactions) between two or more services. We define a *conversation specification* (also known as a *conversation policy*) to be a formal description of "legal" message type-based conversations that a service supports. Our goal is to enable e-services developed by different enterprises to engage in flexible and autonomous, yet potentially quite complex, business interactions (conversations).

We advocate a service-centric perspective that separates service interfaces from conversation specifications. This approach allows us to treat services as pools of interfaces that can be specified by individual participants and then later composed using separate conversation specifications. We can then create conversation controller services that can use conversation and interface specifications to direct services in their interactions, thus freeing service developers from having to explicitly program conversational logic. Such a single third-party conversation controller could leverage "reflected" XML-based specifications to direct the message exchanges of e-services according to protocols without the service developers having to implement protocol-based flow logic themselves. The conversation controller can assume responsibility for the conversation logic, leaving service developers free to focus on service-specific logic. For example, the controller would handle exceptions due to message type errors, while the service would be responsible for handling exceptions related to message content.

The advantage of this approach is that it enables services to be easily and flexibly composed with a minimum of programming effort. In order to participate in a given conversation type, a service need only to be able to accept and produce messages of the appropriate types. This allows services and clients to discover each other and interact dynamically using published specifications.

That is to say, because the conversation policies are not service-specific, services and clients can interact even if they were not built to use precisely matching conversation policies, as long as both parties are capable of sending and receiving appropriate messages. Furthermore, because the service interfaces and the conversation policies are decoupled, different instances of a service could name their methods independently, e.g., a client could use the same conversation specification to talk to two different book-selling services, despite the fact that one service supports a Login method while the other uses a corresponding Sign-on method.

Finally, this approach gives us a scalable mechanism for handling the versioning (evolution) of conversation policies. Services would not necessarily have to be updated in order to support new or modified conversation policies. For example, suppose that the conversation in Figure 1 were updated to allow the client to send a quote request before it has requested a catalog. We could effect this change by simply updating the conversation specification; we would not have to modify either the storefront or the procurement services' code.

3 Currently Developing Standards for Service Communication Specifications

The prevalent model for e-service communication is that e-services will publish information about the specifications that they support. UDDI facilitates the publication and discovery of e-service information. The current version of WSDL (1.0) is an XML-based format that describes the interfaces and protocol bindings of web service functional endpoints. WSDL also defines the payload that is exchanged using a specific messaging protocol; SOAP is one such possible messaging protocol. However, neither UDDI nor WSDL currently addresses the problem of how a service can specify the sequences of legal message exchanges (interactions) that it supports. (We use the term "conversation" to refer to a legal sequence of message exchanges.)

The Web Services Conversation Language (WSCL) addresses this issue, providing an XML schema for defining legal sequences of documents that e-services can exchange. WSCL and WSDL are highly complimentary – WSDL specifies how to send messages to a service and WSCL specifies the order in which such messages can be sent. The advantage of keeping the two distinct is that doing so allows us to decouple conversational interfaces (represented by WSCL) from service-specific interfaces (represented by WSDL). This means that a single conversation specification can be implemented by any number of services, independent of the protocols supported by the various implementations.

3.1 UDDI Registries

A UDDI business registration is an XML document that describes a business entity and its web services. The UDDI XML schema defines four core types of service information: *business information* (such as business name and contact information), *business service information* (general technical and business descriptions of web services), *binding information* (specific information needed to invoke a service), and *service specification information* (associating a service's binding information with the business service information it implements).

Programmers and programs can use the UDDI Business Registry to locate technical information about services, such as the protocols and specifications that they implement. More importantly, the UDDI Business Registry also serves as a registry for abstract (service-independent) specifications. Services can refer indirectly to the UDDI registrations for specifications they implement, which makes it straightforward to identify the business service information that represents a given service.

The UDDI *tModel* is a meta-data construct that uniquely identifies reusable service-related technical specifications for reference purposes. A service publishes tModelInstanceDetails, which is a list of tModelInfo elements that refer to the tModels that the service supports. A UDDI tModel data structure includes a unique key (*tModelKey* attribute), a *name* element, an optional description, and an overviewDoc element in which we can store a URL for the actual specification document.

For example, suppose we wanted to register a WSCL specification of the "storefront" conversation depicted in Figure 1 in a UDDI registry. We would create a tModel entry within the UDDI registry that referred to the actual WSCL specification document in its *overviewDoc* element. Figure 2 shows a UDDI tModel reference for a WSCL specification for a service conversation.

```
<tModel authorizedName="XXXX" operator="YYYY" tModelKey="ZZZZ">
    <name>storefrontConversation</name>
    <description xml:lang="en">
  WSCL description of a simple storefront conversation
    </description>
    <overviewDoc>
<description xml:lang="eng">WSCL source document.</description>
<overviewURL>http://foo.org/specs/storefrontWSCL.xml</overviewURL>
    </overviewDoc>
</tModel>
```

Fig. 2. A UDDI tModel Referencing a WSCL Specification.

This "storefront conversation" tModel can now be referenced by the tModelInstanceInfo of any service that implements that conversation type (Figure 3.

```
<businessService>
  (. . .)
  <bindingTemplates>
    <bindingTemplate>
      (. . .)
      <accessPoint urlType="http">http://www.foo.com/</accessPoint>
      <tModelInstanceDetails>
     <tModelInstanceInfo tModelKey="ZZZZ">
         (. . .)
         </tModelInstanceInfo>
      </tModelInstanceDetails>
      (. . .)
    <bindingTemplate>
      (. . .)
  <bindingTemplates>
</businessService>
```

Fig. 3. A tModelInstanceInfo Referencing a Conversation tModel.

3.2 Web Service Conversation Language

WSCL addresses the problem of how to enable services (often called web services or e-services in this context) from different enterprises to engage in flexible and autonomous, yet potentially quite complex, business interactions. It adopts an approach from the domain of software agents, modeling protocols for business interaction as *conversation policies,* but extends this approach to exploit the fact that Service messages are XML-based business documents and can thus be mapped to XML document types. Each WSCL specification describes a single type of conversation from the perspective of a single participant. A service can participate in multiple types of conversations. Furthermore, a service can engage in multiple simultaneous instances of a given type of conversation or even conversations of different types.

WSCL specifies the public interface to web-services, but does not specify how the conversation participants will handle and produce the documents received and sent. A conversation definition is thus service independent, and can be used by any number of services with completely different implementations. A conversation developer (e.g. a vertical standards body) can create a WSCL description of some conversation, and publish it in a UDDI directory. A service provider who wanted to create a service that supported that conversation description could create and document service endpoints that support the messages specified by the WSCL document. Any software developer who wants to create an application using the published web-service can download the WSCL files describing the conversations supported, and implement the necessary methods accordingly. Ideally, software developers creating and using web-services will be supported by tools that allow them to map easily and quickly from the interactions outlined in the conversation definition to any existing applications and back-end logic,

while separating cleanly between the public and the private processes. Without any formal definition of the conversations, such tool support will not be possible.

Figure 1 depicts a UML diagram of a simple purchase conversation definition from the perspective of the seller. A service that supports this conversation definition expects a conversation to begin with the receipt of a LoginRQ or a RegistrationRQ document. Once the service has received one of these documents, it answers with a ValidLoginRS, a InvalidLoginRS, or a RegistrationRS, depending on the type and content of the message received. Although this conversation is defined from the perspective of the seller, it can be used to determine the appropriate message types and sequences for both the seller and the buyer. The buyer simply derives his conversation definition by inverting the direction of the messages' halves of a conversation.

There are four elements to a WSCL specification:

- *Document type descriptions* specify the types (schemas) of XML documents that the service can accept and transmit in the course of a conversation. The schemas of the documents exchanged are not specified as part of the WSCL specification document; the actual document schemas are separate XML documents and are referenced by their URL in the interaction elements of the conversation specification.
- *Interactions* model the actions of the conversation as document exchanges between conversation participants. WSCL currently supports four types of interactions: *Send* (the service sends out an outbound document), *Receive* (the service receives an inbound document), *SendReceive* (the service sends out an outbound document, then expects to receive an inbound document in reply), and *ReceiveSend* (the service receives an inbound document and then sends out an outbound document).
- *Transitions* specify the ordering relationships between interactions. A transition specifies a source interaction, a destination interaction, and optionally a document type of the source interaction as additional condition for the transition.
- The *Conversation* element lists all the interactions and transitions that make up the conversation. It also contains additional information about the conversation like its name, and with which interaction the conversation may start and end. A conversation can also be thought of as being one of the interfaces or public processes supported by a service. Yet in contrast to interfaces as defined by CORBA IDE or Java interfaces, conversations also specify the possible ordering of operations, i.e. the possible sequences in which documents may be exchanged.

Although WSCL specifies the valid inbound and outbound document types for an interaction, it does not specify how the conversation participants will handle and produce these documents; it only specifies the abstract interface, the public process. The WSCL specification of a conversation is thus service-independent, and can be used (and reused) by any number of services. We can use the *tModel* structure to register WSCL conversation specifications in UDDI registries (as illustrated above).

Table 1. Comparison of Aspects of WSDL and WSCL.

		WSDL	WSCL
Abstract Interfaces	choreography	*out of scope*	Transition
	messages	Operation	Interaction
Protocol Bindings		Binding	*out of scope*
Concrete Services		Service	*out of scope*

3.3 Web-Service Definition Language (WSDL)

As noted before, WSCL specifications are conversation-specific. WSCL describes the structures (types) of documents a service expects to receive and produce (by either explicitly including or else by referring to the document type definitions), as well as the order in which document interchanges will take place, but does not specify how to dispatch received documents to the service. This is partially addressed by WSDL. WSDL documents describe the abstract interface and protocol bindings of a network service. WSDL specifications that describe abstract protocol interfaces are reusable and thus are registered as UDDI tModels.

A reusable WSDL document consists of four components: document type, message, portType (named set of abstract operations and messages involved with those operations), and binding definitions (define message format and protocol details for a specified portType's operations and messages). For example, the "storefront" conversation shown in Figure 1 requires that a service implementing the "Start" interaction provide some sort of endpoint that can accept a *LoginRQ* or *RegistrationRQ* document and output either a *LoginRS* or a *RegistrationRS* document.

3.4 Mapping between WSDL and WSCL

We identify three main aspects of web services. The *abstract interface* (public process, business model) describes the messages or documents (business payload) a service can exchange, as well as the order in which they are exchanged. The *protocol binding* represents the protocols used for exchanging documents. Finally, the *service* itself consists of a particular location that implements a set of abstract interfaces and protocol bindings.

Table 1 shows how these three different aspects are covered by WSDL and WSCL. We can map the corresponding terminology used by WSDL and WSCL to describe operations and interactions as shown in Table 2. It is evident that the only overlap between WSDL and WSCL exists in the specification of the documents being exchanged.

There are a number of ways that we could extend WSDL or WSCL to make explicit the mapping between WSDL port types/operations and WSCL interactions. For example, we could add protocol bindings in WSDL that refer to WSCL conversation specifications or we could add choreography to WSDL port type descriptions. However, to do so by extending WSDL or WSCL would couple these specifications. Instead, we advocate that services should use other methods

Table 2. Comparison of Terminology of WSDL and WSCL.

WSDL	**WSCL**
Port Type	Conversation
Operations:	**Interactions:**
One-way	Receive
Request-response	ReceiveSend
Solicit-response	SendReceive
Notification	Send
Input	InboundXMLDocument
Output, Fault	OutboundXMLDocument
Names of Operation, Input, Output, Fault	ID of Interaction, InboundXML Document OutboundXMLDocument
Message	URL of XML schema (WSCL delegates the specification of the payload entirely to an external XML schema, whereas WSDL directly uses XML data types)

of mapping between the WSDL and WSCL specifications that they support. For example, one option is that when a service populates its UDDI *businessService* entry, it creates *tModelInstanceInfo* records for the WSDL and WSCL specifications that it supports. The mappings between these specifications can then be deduced by document type (mapping WSDL input message types to WSDL InboundXMLDocument schemas). Alternatively, a separate mapping document could be created to map explicitly between WSCL interactions and WSDL operations and port types.

4 Dynamic Conversation Controller for E-Services

Thus far we have shown how WSDL and WSCL can be used to specify the conversational and functional interfaces of e-services. We have implemented a prototype conversation controller that leverages these specifications to direct services in their conversations. (This prototype is described more fully in [8].) We exploit the fact that e-service messages are XML-based business documents and can thus be mapped to XML document types. Our conversation controller can act as a proxy to an e-service, and track the state of an ongoing conversation based on the types of messages exchanged. Specifically, the Conversation Controller requires two pieces of information: a specification of the structure of the conversations supported by the service (interactions, valid input and output message types of interactions, and transitions between interactions), and a specification of the service's interfaces, mapping document types to appropriate service entry points (for given interactions).

Our Conversation Controller is designed to act as a proxy to a service. Once it has received a message on behalf of an e-service, the Conversation Controller can dispatch the message to the appropriate service entry point, based on the state of the conversation and the document's type.

When forwarding the response from the e-service to the client, the Conversation Controller includes a prompt indicating valid document types that are accepted by the next stage of the conversation. This prompt can optionally be filtered through a transformation appropriate to the client's type. In addition, if the client requests it and provides a specification of its interfaces, the Conversation Controller can also direct the client's side of the conversation. Thus neither the service nor the client developer must explicitly handle conversational logic in their code.

Each time the Conversation Controller receives a message on behalf of the service, it will identify the current stage of the conversation and verify that the message's document type is appropriate; if not, then it will send an exception. If the message type is valid, then the Conversation Controller will invoke the service appropriately. It will then identify the document type of the response from the service, identify the new state and the valid input documents for that state, and format an appropriate response for the client. The Conversation Controller can also pass the response through an appropriate transformation, if requested by the client. (For example, if the client is a web browser and has requested form output, then the Conversation Controller may transform the response into an HTML form prompting for appropriate input.)

Moreover, if the client is another service that can return a specification of its own service entry points, then the Conversation Controller could automatically send the output message to appropriate client entry points; if a valid input document for the new state is returned, the Conversation Controller could then forward it to the service, thus moving the conversation forward dynamically. As a result, the Conversation Controller can help a client and service carry out an entire conversation without either the client or the service developer having to implement any explicit conversation control mechanisms. This means that the client developer does not need complete knowledge of all the possible conversations supported by all the services with which the client might interact in the future. For example, each time the Conversation Controller receives a message on behalf of a service, it could implement the pseudo-code listed below.

1. Look at the message header and determine the current state of the conversation. (Ask the service for specifications, if necessary.)
2. From the conversation specification, get the valid input document types for the current state.
3. Verify whether the current message is of a valid input document type for the current state.
4. If the received message is of a valid type, then look up the inbound document in the dispatch specification and dispatch the message to an appropriate service entry point. If more than one appropriate service entry point exists, then dispatch it to each entry point (in order specified by the service) until the service produces an output document of a valid document type. If no entry point exists or no valid output document is produced, then inform the client, also prompting for valid input document types.

5. From the conversation specification, calculate the conversation's new state, given the document type of the output document returned by the service. Look up the valid input documents for this new state.
6. Format the output document in a form appropriate to the client type, also prompting for the input document types that are valid in the new state.

4.1 Client Automation

An argument can be made that developers implementing e-service clients will not want a conversation controller to direct their part of the conversation, both because they expect to hard-code the client parts of the conversation and also because they will find the idea of using a third-party to control the conversation foreign[2]. However, decoupling conversation logic from business logic on the client side greatly increases the flexibility of a client by allowing it to interact dynamically with services even if their conversation policies do not match exactly. For example, the same client code could be used to interact with two services that support different conversation policies but common interfaces.

In order for a conversation controller to direct the client's part of a conversation, the controller must be able to dispatch messages the client receives from the server in order to generate documents that the server requests. This means that the client must be able to communicate its service interfaces to the Conversation Controller. For example, we can extend the process described in the previous section to allow the Conversation Controller to direct both the server and client sides of the conversation, producing the pseudo-code listed below.

1. Look at the message header and determine the current state of the conversation. (Ask the service for specifications, if necessary.)
2. From the conversation specification, get the valid input document types for the current state.
3. Verify whether the current message is of a valid input document type for the current state.
4. If the received message is of a valid type, then look up the inbound document in the dispatch specification and dispatch the message to the appropriate service entry point; otherwise, inform the client that the message is not a valid type and prompt for the input document types that are valid in the new state.
5. From the conversation specification, calculate the conversation's new state, given the document type of the output document returned by the service. Look up the valid input documents for this new state.
6. If the client wishes to be treated as a browser, then format the output document in an appropriate HTML form, also prompting for the valid input document types for the new state.
7. If the client wishes to be directed by the Conversation Controller and there are valid input documents for the new state, then look up outbound document types in the client's dispatch table, and invoke the appropriate client methods that could produce valid input documents.

[2] Conversation with Kevin Smathers, 1/4/2001.

8. If the client produces a valid input document, then send it to the service, invoking it through the Conversation Controller (recursion takes place here).
9. If the client does not produce any valid input documents, or if there were no valid input documents in the new state, then format and return the output document in an appropriate HTML form, also prompting for the new state.

4.2 Conversation Controller State

The Conversation Controller that we have outlined above does not include any performance management, history, or rollback mechanisms. If one subscribes to the idea that intermediate states of an e-service's conversation are *not* transactional, and one also supposes that Conversation Management functionality (including performance history, status of ongoing conversations, etc.) is distinct from Conversation Control functionality, then the Conversation Controller can operate in a stateless mode.

5 Related Work

In his survey of agent systems for E-Commerce, Griss [9] notes that researchers in the agent community have proposed a number of agent communication systems over the past decade, and indeed agent-based e-commerce systems seem like a natural model for the future of e-services. Griss identifies several kinds of agent systems appropriate for E-Commerce, including personal agents, mobile agents and collaborative/social agents. Griss then lists seven properties that represent dimensions of agent-like behavior: adaptability, autonomy, collaborations, intelligence, mobility, persistence and personality/sociability. We believe that although e-services exhibit some of these properties, e-services are not necessarily adaptable, intelligent or anthropomorphic (they are not required to exhibit personality/sociability). However, since agents dynamically communicate via message exchanges that conform to specified protocols/patterns, agent-based conversations are recognized as an especially appropriate model for e-service interactions.

Several existing agent systems allow agents to communicate following conversational protocols (or patterns). However, to the best of our knowledge, all of these are tightly coupled to specific agent systems, and require that all participating entities must be built upon a common agent platform. For example, the Knowledgeable Agent-oriented System (KaoS) [10] is an open distributed architecture for software agents, but requires agent developers to hard-wire conversation policies into agents in advance. Walker and Wooldridge [11] address the issue of how a group of autonomous agents can reach a global agreement on conversation policy; however, they require the agents themselves to implement strategies and control. Chen, et al. [12] provide a framework in which agents can dynamically load conversation policies from one-another, but their solution is homogeneous and requires that agents be built upon a common infrastructure.

Our Conversation Controller is unique in that we require only that a participating service produce two XML-based documents – 1) a specification of the conversational flows it supports and 2) a specification of the service's functionality (describing how the service can be invoked).

A few E-Commerce systems support conversations between services. However, these all require that the client and service developers implement matching conversation control policies. RosettaNet's *Partner Interface Processes* (PIPs) [13] specify the roles and required interactions between two businesses. *Commerce XML* (cXML) [14] is a proposed standard being developed by more than 50 companies for business-to-business electronic commerce. cXML associates XML DTDs for business documents with their request/response processes. Both RosettaNet and CommerceXML require that participants pre-conform to their standards. Our work is completely compatible with such systems, but is also unique in that we allow a service's clients to share the service's Conversation Controller dynamically – without having to implement the client to the specifications of the service.

Insofar as they reflect the flow of business processes, e-service conversations also resemble workflows. However, as the authors of the Web Services Conversation Language (WSCL) [6] observe, workflows and conversations serve different purposes. Conversations reflect the interactions between services, whereas workflows delineate the work done by a service. A conversation models the externally visible commercial interactions of a service, whereas a workflow implements the service's business functionality. In addition, workflows represent long-running concurrent fully integrated processes, whereas e-service conversations are loosely coupled interactions.

6 Conclusions / Future Work

E-services pose a new set of requirements and usage models for service interactions. E-services must enable business-to-business interactions without requiring intensive collaboration between service developers. Therefore, we advocate a service-centric perspective that separates service interfaces from conversation specifications. Distinguishing between conversation logic and service functionality allows us to treat services as pools of interfaces that can be described using service specifications and composed using conversation specifications.

In this paper, we have sketched how to use WSDL to create specifications describing service interfaces and how to use WSCL to create abstract conversation specifications. We have discussed how these standards relate to each other and how we can use them to compliment each other. We also described how services can refer to the WSDL and WSCL specifications they support in their UDDI registrations. We have built a prototypical conversation controller service that leverages these specifications to direct services in their interactions. This third-party conversation controller uses "reflected" XML-based specifications to direct the message exchanges of e-services according to protocols without the service developers having to implement protocol-based flow logic themselves. The ad-

vantage of this approach is that it treats services as pools of methods that can be easily and flexibly composed with a minimum of programming effort.

In the future, we plan to investigate more sophisticated uses of conversation policies. For example, we would like to provide a model for the explicit support of deciding conversation version compatibility. We would also like to explore how to support both nested conversations and multiparty. Finally, we hope to address how to exploit document type relationships when manipulating message documents. For example, we would like to use subtype polymorphism to establish a relationship between a document type accepted as input by an interface specification and a corresponding document type in a conversation specification.

References

1. Kuno, H.: Surveying the E-Services Technical Landscape. In: International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS). (2000)
2. Boubez, T., Hondo, M., Kurt, C., Rodriguez, J., Rogers, D.: UDDI Data Structure Reference V1.0. Technical report (2000)
3. Boubez, T., Hondo, M., Kurt, C., Rodriguez, J., Rogers, D.: UDDI Programmer's API 1.0. Technical report (2000)
4. Ariba, Inc. and International Business Machines Corporation: UDDI Technical White Paper. Technical report, Ariba, Inc. and International Business Machines Corporation and Microsoft Corporation (2000)
5. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.0. Technical report (2000)
6. 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) 1.0. Technical report, Hewlett-Packard Web Services Organization (2001)
7. Beringer, D., Kuno, H., Lemon, M.: Using WSCL in a UDDI Registry 1.02: UDDI Working Draft Best Practices Document. Technical report, Hewlett-Packard Company (2001)
8. Kuno, H., Lemon, M.: A Lightweight Dynamic Conversation Controller for E-Services. In: International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS). (2001)
9. Griss, M.: My Agent Will Call Your Agent . . . But Will It Respond? Software Development Magazine (2000)
10. Bradshaw, J.M.: KAoS: An Open Agent Architecture Supporting Reuse, Interoperability, and Extensibility. In: Knowledge Acquisition for Knowledge-Based Systems Workshop. (1996)
11. Walker, A., Wooldridge, M.: Understanding the emergence of conventions in multi-agent systems. In: First International Conference on Multi-Agent Systems. (1995)
12. Chen, Q., Dayal, U., Hsu, M., Griss, M.: Dynamic Agents, Workflow and XML for E-Commerce Automation. In: First International Conference on E-Commerce and Web-Technology. (2000)
13. Web page: (`http://rosettanet.org`)
14. Web page: cxml.org. (`http://www.cxml.org`)