Protocol Engineering for Web Services Conversations

Shamimabi Paurobally

School of Electronics and Computer Science, University of Southampton Southampton SO17 1BJ, UK. sp@ecs.soton.ac.uk

Nicholas R. Jennings

School of Electronics and Computer Science, University of Southampton Southampton SO17 1BJ, UK. nrj@ecs.soton.ac.uk

Abstract

Although web services aim to bring about seamless and effective communication in a wide variety of Internet applications, the interactions between them are currently limited to simple request-response exchanges. However, in the longer term we believe this is unsustainable. In particular, we believe that more complex protocols for web service conversations are necessary if the participants are to tailor their needs and offers to the prevailing context and they are to coordinate multiple services in open and realistic environments. To this end, this paper combines and extends two recent web service languages, *WS-Conversation Language (WSCL)* and *WS-Agreement*, in order to obtain a method for engineering protocols of sufficient expressiveness for the next generation of flexible and autonomous services. Specifically, we propose that the protocols include speech-acts as the individual messages and we show how to model such speech-acts as WS-Agreement schemas, which can, in turn, be imported into the specification of the protocols in WSCL. To demonstrate our approach we express a standard contracting protocol in the extended WSCL/WS-Agreement languages. Furthermore, we use statechart notation as a visual counterpart to help developers write clients that flexibly interact with a service and to help users to better understand how to interact with a service. Finally, we show that the translation between statecharts and WSCL/WS-Agreement protocols is straightforward.

Key words: web services, conversations, interaction protocols, WSCL, WS-Agreement.

1 Introduction

In the last decade, web services [1] have emerged as a new paradigm that supports loosely-coupled distributed systems in service discovery and service execution. In this context, a web service is viewed as an autonomous software component, offering some specified functionality, that can be discovered and invoked across the Internet. Current examples of web services are online travel reservations, purchasing books at `Amazon.com`, map services at `maps.yahoo.com` and currency converters. Generally speaking, the aim of the web services endeavour is to obtain an environment where service customers and service providers can locate one another, connect with each other dynamically, set (negotiate) the terms and conditions of service invocation automatically and then execute the necessary actions according to the prevailing contract. To this end, a web services architecture has been developed that consists of five layers for supporting service description, publishing service descriptions and discovering services. Then standards such as Business Process Execution Language for Web Services (BPEL4WS) [2] are defined over the web services architecture to enable higher level functionality such as service composition, choreography and transactions. The key advantages of these and related standards include interoperability between distributed applications regardless of the underlying platform, implementation language and operating system. Thus, because web services use standard communication protocols (such as HTTP, FTP and SOAP [3]), distributed applications are easily accessible via the Internet, even through firewalls. Another advantage is that web services are specified in the cross-platform modeling language XML (eXtensible Markup Language). This allows heterogeneous distributed applications to be described in a common way, which, in turn, facilitates the adoption of XML-based web services as standards in industry.

Given these advantages, it is clear that web services have much to offer. However there are a number of shortcomings which prevent approaches that may cause this promise to remain unfulfilled. In particular, there are shortcomings with respect to the prevailing views of interaction and negotiation in the next generation of web services where open environments and collaboration between services will be common (see [4] for a general discussion about the importance of flexible interactions in service-oriented systems). There are two proposals that start to capture such flexible interactivity, namely the web services conversation language (WSCL) [5] and the web services agreement specification (WS-Agreement) [6]. The former focuses on synchronisation aspects; what are acceptable message exchanges and the order in which they should occur. The latter specifies the terms of an agreement in the context of the service description. However this work will not fulfill its full potential because it is limited to simple request-response exchanges in which a web service remains a self-contained application without any ability to collaborate with other web

services in order to satisfy a request. In particular, such simple exchanges are unsuitable for coordinating transactions between multiple web services because of the explosion in the amount of communication. Moreover advanced transactional systems where participants continuously tailor their needs and offers are also beyond the scope of request-response messages because of the absence of negotiations. In both cases it can be seen that richer and more flexible interactions such as auctions and contracting protocols are needed. Given this and because such issues have been extensively researched upon for the last decade in the software agents community, this paper argues that work on enabling interaction between web services would benefit from the insights and techniques from the field of multi-agent systems. In more detail, the work we develop in this paper extends the WSCL and WS-Agreement standards to propose a framework for engineering interaction protocols and constructing flexible conversations in the web services domain. Unlike the current simple offer-accept protocols specified in WSCL and WS-Agreement, this paper considers more complex interaction protocols between more than two parties. Moreover, to overcome the un-intuitiveness problem of XML-based specifications, we propose statecharts as a graphical representation of these protocols. We also propose a translation between statecharts and the XML-based protocols in order to ensure there is a clear bridge between the specifications and the implementations.

Against this background, this paper advances the state of the art by developing a framework that enables richer and more flexible interactions between web services to be specified. In particular, the foremost contribution is that we extend the conversational capabilities of web services by supporting non-trivial interactions in which several messages have to be exchanged before the service is completed and/or the conversation may evolve in different ways depending on the state and the needs of the participants. This increase in flexibility and expressiveness is achieved through the use of speech-acts such as *propose*, *call for proposals* and *inform* (rather than just offer-accept as is the case currently). Second, in contrast to WSCL and WS-Agreement which only concretely analyse interaction between two services, we describe a case study involving more than two web services. The third contribution is that these non-trivial forms of interaction allow users to better understand the service execution semantics and how to interact with a service. Fourth, by separately defining speech-acts as WS-Agreement and modeling interaction protocols in statecharts, we contribute to providing a more structured method for engineering protocols and designing web service conversations. Web services are usually not subject to a rigid analysis and design phase and functionality is often published in an ad-hoc way, which can lead to misunderstandings between parties. To overcome this, we propose a design method (different speech-acts that can be imported and modeling protocols using statecharts) to construct protocols which is compatible with the schemas in WSCL and WS-Agreement. Thus, as the number of services to be integrated grows and the environment

becomes more dynamic, our work should help developers to understand how to write clients that flexibly interact with a service and to develop automated tools to dynamically bind to a service based on the specified characteristics. Our fifth contribution is to research in agent interactions, where, to date, much of the work has yet to be put to test in open and dynamic environments. In this vein, our work provides a dynamic environment for studying agent interactions. Last, but not least, we bring together the research on WSCL and WS-Agreement. Now on their own, they respectively lack expressiveness and use of interaction protocols, but when taken together complement each other when it comes to specifying web services conversations.

The paper is structured as follows. Section 2 discusses web services and critically analyses the WSCL and WS-Agreement specifications from the perspective of supporting flexible interactions. Section 3 describes our extensions to the WS-Agreement Schema and our modeling of a number of central speech-acts as WS-Agreement. Section 4 extends the WSCL schema and describes how to construct interaction protocols that are composed of sequences of speech-acts defined in WS-Agreement. Section 5 provides an application of our extensions by using them to specify a standard contracting protocol (the Contract Net protocol [7]). Section 6 discusses the visualisation of these XML-based protocols in statecharts and provides a translation between the two notations. Finally, section 7 presents our conclusions and future work.

2 Web Services

Web services are self-contained software components that expose specific functionality on the Internet, such that other applications can use them by means of established web protocols and data formats such as HTTP and XML [1]. Their appeal over other interoperability standards, such as the Common Object Request Broker Architecture (CORBA) [8] and the Distributed Component Object Model (DCOM) [9], is the simplicity and flexibility of the architecture. Instead of assuming a common set of predefined interfaces, the model relies on a basic set of message and document formats.

In more detail, there are five components to a layered web services architecture (see figure 1). The lowest (first) layer consists of Internet protocols such as HTTP and FTP. The second layer is concerned with message packaging using the Simple Object Access Protocol (SOAP) [3]. The third layer specifies how to describe and connect to networked web services in the Web Services Description Language (WSDL). The fourth layer, Universal Description Discovery Integration (UDDI), aims at creating a framework for publishing and discovering services over the Internet. The fifth, and top layer, the Web Services Flow Language (WSFL), defines a language for composing web services.

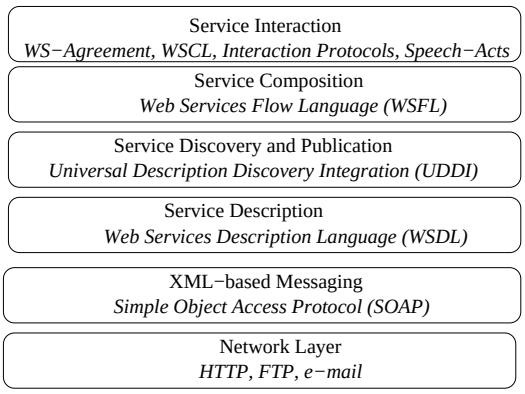

Fig. 1. Web Services Architecture Stack

Standards built over this five layered architecture intend to allow service interactions such as compositions, negotiations and collaborations . However, this leads to the question about how developers will provide and deal with the increasing expressiveness, flexibility, and adaptability that web service systems are likely to require. To this end, in this paper, we take the view that existing work from the fields of multi-agent systems can shed some light on how best to tackle this question.

The relation between web services and agent systems has already been mentioned by Booth et al. in [1] where they state that a web service is viewed as "an abstract notion that must be implemented by an agent. The agent is a concrete entity (a piece of software) that sends and receives messages, while the service is the set of functionality that is provided". Given this, a web service is the representation of a defined functionality of an autonomous software application or component and is located on the Internet. (Thus it shows several of the characteristics of an autonomous agent [4]). We also adopt this view of a web service being published and managed by a software agent. However, despite this relation between web services and multi-agent systems, communication patterns in the web services domain remain at the request-reply level, whereas in the realm of software agents they are much richer. Thus agents can engage in complicated scenarios (such as auctions and negotiations).

To remedy this lack of interactivity between web services, we believe standards should be designed over WSFL so that web services can engage in electronic contracts. Specifically, we propose to achieve this capability with web services by adapting the approach used by agent systems for conversing through agent communication languages and protocols. Thus, we combine and extend existing web services specifications, WSCL and WS-Agreement for describing, respectively, conversations and agreements with web services (sections 4 and 3).

2.1 Web Services Conversation Language (WSCL)

WSCL captures the conversation sequence that a web service is expecting to engage in by describing the order in which its WSDL-described operations should be invoked. In particular, a WSCL schema specifies the documents sent and received and the order of exchange between participants in a conversation. It defines a service behaviour in terms of a list of *interactions* regarding the documents to be exchanged and a list of *transitions* to describe allowable interaction orderings. Thus there are four main elements to a WSCL specification [5]:

(1) *Document type descriptions* specify the types of XML documents that the service can accept (`InboundXMLDocument`) and transmit (`OutboudXMLDocument`) in the course of a conversation.
(2) *Interactions* model the actions of the conversation as document exchanges between two participants. There are five types of interactions: 1) `Send` an outbound document, 2) `Receive` an inbound document, 3) `SendReceive` for sending and then expecting to receive an inbound document as a reply, 4) `ReceiveSend` is the inverse order of `SendReceive`, 5) `Empty` does not contain any exchange and is used at the start and end of a conversation for exchanging documents.
(3) *Transitions* specify the ordering relationships between interactions. A transition specifies the source interaction, a destination interaction, and, optionally, a condition related to the document type of the source interaction.
(4) *Conversation* is the container that lists all the interactions and transitions in a WSCL schema.

An example of a simple conversation is given in figure 2. This can be interpreted as showing a simple purchase conversation from the perspective of a seller when receiving catalogue inquiries. Figure 3 shows the corresponding WSCL representation of the interaction in figure 2. The name of the Conversation in the WSCL specification is given as `StoreFrontServiceConversation`. There are two special interaction types, `Start` and `End` to denote the initial and final transitions. There are two `ReceiveSend Interactions` – `CatalogInquiry` and `Quote`. Through transitions entering `CatalogInquiry`, a seller receives an inquiry for a catalogue in the form of an incoming XML document "CatalogInquiry" located at "http://c123.org/CatRQ" with id="CaRQ". The seller replies with an outbound document "CatRS" located at "http://c123.org/CatRS". The buyer receives a quote from a seller through the transition from `CatalogInquiry` to `Quote` through an inboundXML document "QRQ" located at "http://c123.org/QRQ". After receiving a quote, the buyer may send another inquiry to the seller or end the conversation through a transition to the `End Interaction`.

Fig. 2. Catalogue Inquiry Conversation (taken from [5])

```
<Conversation name="StoreFrontServiceConversation"
   initialInteraction="Start" finalInteraction="End">

<ConversationInteractions>
<Interaction interactionType="ReceiveSend" id="CatalogInquiry">
    <InboundXMLDocument hrefSchema="http://c123.org/CatRQ" id="CatRQ"/>
    <OutboundXMLDocument hrefSchema="http://c123.org/CatRS" id="CatRS"/>
</Interaction>
<Interaction interactionType="ReceiveSend" id="Quote">
    <InboundXMLDocument hrefSchema="http://c123.org/QRQ" id="QRQ"/>
    <OutboundXMLDocument hrefSchema="http://c123.org/QRS" id="QRS"/>
</Interaction>
</ConversationInteractions>

<Conversation Transitions>
<Transition>
  <SourceInteraction> href="Start"/>
  <DestinationInteraction> href="CatalogInquiry"/>
</Transition>

<Transition>
  <SourceInteraction> href="CatalogInquiry"/>
  <DestinationInteraction> href="CatalogInquiry"/>
</Transition>

<Transition>
  <SourceInteraction> href="CatalogInquiry"/>
  <DestinationInteraction> href="Quote"/>
</Transition>

<Transition>
  <SourceInteraction> href="Quote"/>
  <DestinationInteraction> href="CatalogInquiry"/>
</Transition>

  <SourceInteraction> href="Quote"/>
  <DestinationInteraction> href="End"/>
</Transition>

</ConversationInteractions>
</Conversation>
```

Fig. 3. WSCL Specification of the Catalogue Inquiry Conversation

2.2 Weaknesses of WSCL Conversations

The strength of WSCL lies in the specification of the `Interactions` and `Transitions`, which allows some from of sequenced messaging. However, on examining the example in figures 2 and 3, a number of weaknesses in the template for describing conversations appear.

First, WSCL conforms to the WSDL specification of request-response operations between two participants (the service provider and requestor). Such

a request-response protocol is too simple for participants wishing to bargain and dynamically tailor their needs according to their available resources (as argued in section 1).

Second, the participants' identities are not explicit, neither are actions (such as `Receive` or `Send`) explicitly associated with the perpetrator. This lack of expressiveness is not suitable for open environments with many participants. For example, consider the case of a group of m agents, where only agents A and B send a message or expect to receive a message from C. Now since there is no binding of messages to who sent or received what, an agreement from A to C may be interpreted by C as an agreement from B to C, which can cause disputes between parties. In particular, we have interpreted the catalogue inquiry example to be a seller receiving an inquiry and sending back responses and quotes. But, in fact, there is no annotation to the interactions and transitions in the XML code to specify that it has to be a seller who is receiving the inquiries and that there is another buyer to whom the seller responds. Thus, if the seller was sending the same catalogue information,"CatRS", to multiple buyers, we would not be able to differentiate between various buyers and their responses.

Third, conversations are specified from the view-point of one of the participants (usually the service provider). Thus the client has to interpret the specification by inverting the message direction. For example, a conversation is published in a service directory from the perspective of the provider and so the first interaction is a `Receive` or `ReceiveSend` interaction. Thus, an initiator (a customer) has to derive its version by converting the actions, for example `Receive` into `Send`, `ReceiveSend` into `SendReceive`, `Send` into `Receive` and `SendReceive` into `ReceiveSend`. Inbound documents from a provider's perspective should be interpreted as outbound documents from the customer's viewpoint and outbound documents for a provider as inbound documents for the customer. Such inversion is not always straightforward in complex protocols and although two participants can successfully interact if their protocol are duals of each other, this does not scale to more than two agents.

Fourth, the conversation specifications in WSCL remain at the level of exchanging documents and do not support more interaction and negotiation prior to sending or receiving a document. For example, the simple `CatalogInquiry` transition involves more than just passing a document. It may be that the agents negotiate over the ability, time and price to access a catalogue or its subsections. However, protocols for such negotiations cannot be expressed in the current WSCL schema because it does not include any concept of bargaining, bidding or the set of attributes over which to negotiate.

Fifth, there is no method specified for correlating a transition from a `SourceInteraction` with the inbound or outbound document field. For example, in fig-

ure 3 and its counterpart representation in WSCL, it is not clear which of the outbound document "CatRS" or "QRQ" in `CatalogInquiry`, or both, is associated with the transition from `CatalogInquiry` to `Quote`.

2.3 WS-Agreement

WS-Agreement specifies an XML-based language for creating contracts, agreements and guarantees from offers between a service provider and a client. In this case, an agreement may involve multiple services and includes fields for the parties, references to prior agreements, service definitions and guarantee terms. Here the service definition is part of the terms of the agreement and is established prior to the agreement creation. An agreement is defined as being composed of:

(1) *Name* identifies the agreement and is used for reference in other agreements.
(2) *Context* includes parties to an agreement, reference to the service provided and possibly other related or prior agreements.
(3) *Service Description Terms* provide information to instantiate or identify a service to which the agreement pertains.
(4) *Guarantee Terms* specify the service levels that the parties are agreeing to and may be used to monitor and enforce the agreement. They consist of: 1) the list of services it applies to, 2) the list of variables representing domain-specific concepts (e.g. response time or bandwidth), 3) optional conditions that have to be met for the guarantee to be enforced, 4) conditions to satisfy the guarantee and 5) one or more business values (e.g. the penalty upon failure to meet the objective, the strength of a commitment by a service provider or the importance and confidence of meeting an objective).

An agreement template follows the above structure. A service provider offers an agreement template describing the service and its guarantees. Negotiation then involves a service consumer retrieving the template of agreement for a particular service from the provider and filling in the appropriate fields. The filled template is then sent as an offer to the provider. The provider decides whether to accept or reject the offer, depending on its resources. Although offers and agreements have mostly the same fields, an offer contains choices for an agreement from the service customer for the service provision. In an agreement, the choices in an offer are modified by the service provider to finalise the agreement.

2.4 Weaknesses of WS-Agreement Contracts

The strength of WS-Agreement lies in a well-defined template for specifying agreements. The template or part of the template, such as the service description terms and the guarantee terms, can be used in the content of exchanged messages. Moreover, generally speaking, this template is suitable in cases where interactions are concerned with reaching agreements and drawing up contracts.

The first significant weakness lies in the fact that messages in WS-Agreement are limited to two types – *offer* and *agree*, according to a template published by a service provider. The WS-Agreement specification is only used at the last stage in a transaction where the parties are closing their interaction with a contract specified as a WS-Agreement. Different situations requires different types of interactions, where *offer* and *agree* messages may not be sufficient or appropriate. For example, in a collaborative interaction, messages are specified for informing, confirming or dis-informing other parties about the state of the world. Another type of interaction would be competitive situations such as auctions where messages can be bids, call for bids and announcing the winning bid. Another situation would be when the provider and customer have non-matching preferences, negotiation type interactions include messages for making offers, counter-offers, and proposals for helping the parties to learn about each other preferences, to revise their offers and proposals and to eventually come to a mutually acceptable agreement. In the multi-agent systems field, such actions such as proposals and call for proposals are referred to as speech-acts and are specified through an agent communication language [10]. In this context a speech-act conveys a special meaning to a receiver (for example, to inform the receiver about the state of the world). Speech-acts can be considered as classes of asynchronous messages modeled on the Theories of Speech-Acts enunciated by Austin [11] and Searle [12]. Their semantics may be similar to that defined in agent communication languages such as FIPA ACL [10], but in this paper we do not enforce any style of semantics. Specifying speech-acts, such as making the call for proposals, generating the proposals, accepting proposals, and informing a web service of a fact all require changes in the WS-Agreement specification. To do so, we re-use how speech-acts are modeled in multi-agent systems to specify speech-acts as WS-Agreement and as a result increasing the set of messages that can be exchanged as WS-Agreement.

The second significant weakness lies in the fact that there is no interaction protocol between parties specified in WS-Agreement. There is only a two step conversation, an *offer* followed by an *agree*. Without an adequate set of speech-acts and specification of how to construct interaction protocols, the usefulness of a WS-Agreement exchange is limited to cases such as buying from catalogues, with take-it or leave-it offers from the seller or buyer. For example the

Contract Net protocol is probably the most widely used interaction protocol in the multi-agent systems field and it cannot be expressed solely through the WS-Agreement specification. In the Contract Net protocol, there is no offer-accept situation, but rather call for proposals are made by a manager for contractor to carry out a task. Here a manager can be a web service making a call for proposals for other web services to execute some task. After a call for proposal, the contractors, send proposals to the manager. Even if we increase the WS-Agreement schema with various speech-acts, there is no concept of how to sequence messages to form a valid conversation. Another example of the inadequacy of WS-Agreement becomes apparent when we consider interactions based on auctions in which there is a sequence of sellers posting their item, bidders making bids, and auctioneers announcing winning items. The sequencing of these actions are missing from the WS-Agreement specification, but this shortcoming can be remedied by combining the WS-Agreement and the WSCL schemas.

3 Speech-Acts in WS-Agreement

In this section we first specify speech-acts as WS-Agreement and then, in section 4, we show how such specifications can be sequenced through WSCL schemas to form interaction protocols and conversations. As mentioned in the previous section, we can list a number of actions, such as *offer, accept, inform, request, bid, sell, propose*, and *call for proposal*, that are commonly used in interaction protocols [1]. Given this, we re-use and extend the structure for WS-Agreement to specify such speech-acts within the `wsag` tag, taking care to remain compatible with the WS-Agreement specification. Thus, let an agreement consist of the context and both the service definition and guarantee terms (as per section 2.3). Whilst re-using the structure of a WS-Agreement schema, we nevertheless need to extend it to include the participants of an interaction and any action to be executed (to overcome the weaknesses discussed in section 2.3). These modified schemas are given in section 3.1 and are then used to define the necessary speech-acts as WS-Agreement schemas (in sections 3.2 to 3.6).

[1] In the definition of all of the speech-acts, we assume sincerity and ignore Gricean conditions [10]. Gricean conditions express the belief that the sender does not believe the receiver already believes the proposition, or is uncertain about it. We make these assumptions because we believe that Gricean conditions introduce inessential complexity and can be expressed as an axiom holding in the framework.

3.1 Extensions to WS-Agreement

We add two types to the WS-Agreement schema to express perpetrators and recipients of (1) exchanged speech-acts and (2) process executions. We do this by extending the `Context` field to include a speech-act. In figure 4, it can be seen that a speech-act is defined as a complex type, called `Speech-Act`, with attributes the sender, the list of recipients and any action to be executed. For example, if the web service s sends a speech-act $sa(r,\alpha)$ to web service r, the details about the participants (s and r) and the action (α) in sa are added to the context field. Thus, there is one initiator as the sender s of the speech-act, a list of recipients r, and the action (α) as an attribute of the speech-act which is of type WSCL Process (`wscl:Process`) (as later defined in section 4).

We also extend the `ServiceDescriptionTerms` field in WS-Agreement to include the action (α) that may be sent in a speech-act (as defined in figure 5) in order to specify details about which service executes the action and which service should be notified about the action's execution. In this case, it can be seen that an action has a name, which web service should execute it, the receivers of the execution results, and any pre-conditions for executing the action.

```
<xs:complexType> xs:Name="Speech-Act "
<xs:attribute> Name="xs:NCName" </xs:attribute>
  <xs:sequence>
    <xs:element name="Initiator" type="xs:NCName"/>
    <xs:simpleType name="Respondents" use="optional">
      <xs:list item Type="xs:NCName"/>
    </xs:simpleType>
    <xs:element name="Process" type="wscl:Process"/>
  </xs:sequence>
</xs:complexType>
```

Fig. 4. Definition of Speech-Act Type in the `Context` field

```
<xs:complexType> xs:Name="Action "
  <xs:sequence>
    <xs:element name="Executor" type="xs:NCName"/>
    <xs:element name="Action-PreCondition" type="xs:boolean" defaultvalue="false"
                                                          use="optional" />
    <xs:element name="Process" type="wscl:Process"/>
  </xs:sequence>
```

Fig. 5. Definition of Action Included in `ServiceDescriptionTerms`

Using the above types and the WS-Agreement schema, we can now specify the different speech-acts that can be part of our negotiation interaction protocols.

3.2 The Offer Speech-Act

The *offer* speech-act is a take-it or leave-it offer and precedes an agreement or a rejection to terminate an interaction. It thus resembles the WS-Agreement

offer of an agreement, with the terms, service description and guarantee of the offer. However, we also allow either the service provider or the service client to make an offer. Thus, an agreement template may not only be published from the service provider, but a client may also devise its own template and an offer can be sent by either the provider or the client. Here an offer γ from a sender s is sent to a receiver r, declaring amongst other things and the terms of an eventual agreement. An offer is expressed as *s.offer(r,γ)* and its XML representation, which is compatible with an `AgreementOffer` in WS-Agreement, is as follows:

```
<wsag:Offer>
<wsag:Name> "xsd:NCName" </wsag:Name>
<wsag:Context>
   <wsag:AgreementInitiator> xs:Name="s" </wsag:AgreementInitiator>
   <wsag:AgreementProvider> xs: Name="r" </wsag:AgreementProvider>
   <wsag:Speech-Act> wscl:Process=γ </wsag:Speech-Act>
   <wsag:TerminationTime> xs:Time </wsag:TerminationTime>
</wsag:Context>


<wsag:Terms>
     <wsag:ServiceDescriptionTerm wsag:Name="xs:NCName" wsag:ServiceName="xs:NCName">
     <xs:element name="offer" type="wsag:AgreementOffer"
      xs:value= r can agree or reject to do action to satisfy offer γ/>
    </wsag:ServiceDescriptionTerm>
    <wsag:GuaranteeTerm wsag:Name="xs:NCName" wsag:ServiceScope="wsag:ListofServiceNames">
     <wsag:Variables wsag:Name="xsd:NCName" wsag:Metric="xsd:QName"> </wsag:Variables>
     <wsag:QualifyingCondition> ...</wsag:QualifyingCondition>
     <wsag:ServiceLevelObjective> ...</wsag:ServiceLevelObjective>
     <wsag:BusinessValueList>
       <wsag:Importance> xsd:integer </wsag:Importance>
       <wsag:Penalty> xsd:integer </wsag:Penalty>
       <wsag:Reward> xsd:integer </wsag:Reward>
      </wsag:BusinessValueList>
    </wsag:GuaranteeTerm>
</wsag:Terms>
</wsag:Offer>
```

3.3 The Agreement Speech-Act

An *agree* action follows an *offer* which is not rejected. Here an agreement is compatible to that defined in WS-Agreement and includes the context, the service description and the terms of the agreement. As for an offer, either a service provider or the client may make the agreement. In addition, we make explicit that there is an agreement to perform an action, which to this end is included in the service description. Thus, the sender s informs the receiver

r that it will perform an action given a precondition. An agreement may be expressed as *s.agree(r,γ)*, where the agreement γ expresses the fact that a specific web service (either s or r or a third party) will perform an agreed action α when the conditions in the `ServiceLevelObjective` become true. We show below the salient points (`ServiceDescriptionTerm`) in an XML representation of an agree (again this is compatible to an Agreement in WS-Agreement). Below we denote the service or the agent performing α by *Executor* (which can be the sender or receiver of the speech-act) and the condition for executing the action by *Cond*.

```
<wsag:Agreement>
<wsag:Name> xs:γ </wsag:Name>
   <wsag:ServiceDescriptionTerm wsag:Name="Perform-Action" wsag: ServiceName="xs:NCName">
    <xs:element name="agree" type="wsag:Agreement"
      xs:value <!--Call procedure γ e.g. /bin/Perform-Act(Executor, γ)/>
    <wsag:ServiceDescriptionTerm wsag:Name="xs:NCName" wsag:ServiceName="xs:NCName"/>
   <wsag:GuaranteeTerm wsag:Name="xs:NCName" wsag:ServiceScope="wsag:ListofServiceNames">
    <wsag:ServiceLevelObjective> Cond holds </wsag:ServiceLevelObjective>
<!--same as offer, i.e. variables, conditions, penalties and rewards.-->
   </wsag:GuaranteeTerm>
```

3.4 The Inform Speech-Act

The *inform* speech-act is a basic one that that can be used to define others. Here the meaning of an *inform* is that the sender informs the receiver that a given proposition is true. The XML representation of *s.inform(r,ϕ)* is shown below:

```
<wsag:Inform>
<wsag:Name> NCName <!--e.g. s.inform(r,φ)> </wsag:Name>
<wsag:Context>
  <Participants:Initiator> Sender "s" </Participants:Initiator>
  <Participants:Respondents> Receiver "r" </Participants:Respondents>
</wsag:Context>
<wsag:Terms>
   <wsag:ServiceDescriptionTerm wsag:Name="inform" wsag:ServiceName="xs:NCName">
       <xs:element name="inform" type="xs:boolean" value="φ" minOccurs="1"/>
   </wsag:ServiceDescriptionTerm>
</wsag:Terms>
</wsag:Inform>
```

The inform action may be expressed as *s.inform(r,ϕ)*, where sender s informs receiver r that ϕ holds. This means s may inform r about the state of a

service or an agreement. In our XML representation of inform, the proposition ϕ is included in the `ServiceDescription-Terms` because informing about a service (for example, whether it is accessible or it requires payment) is in fact some form of service description. The inform parameters can permeate to the Guarantee terms – `ServiceLevelObjective` and `QualifyingCondition` – if the condition within the *inform* expresses some guarantee condition to be true.

3.5 The Proposal and Call for Proposals Speech-Acts

A *propose* speech-act means that sender s proposes receiver r to do an action γ. This can be expressed as *s.propose(r,*γ*)*. Here s may be a web service which advertises (sends proposals) for executing specific operations and sending particular results back. On the other hand, r may be a web service outsourcing a task and accepting proposals from other web services. Thus, in general, proposals are a means for web services to collaborate and form an agreement about task execution. As for *inform*, proposals are defined in the `Context` and the `ServiceDescriptionTerms` fields, as shown below. In this case, the condition for s to execute the action in a proposal is that it receives an *accept* proposal from r, requiring *received(r.accept(s,*γ*))* to be true.

```
<wsag:Propose>
<wsag:Name> NCName <!--e.g. s.propose(r,γ)> </wsag:Name>
<wsag:Context>
   <Participants:Initiator> Sender "s" </Participants:Initiator>
   <Participants:Respondents> Receiver "r" </Participants:Respondents>
   <wsag:Speech-Act> wscl:Process=γ </wsag:Speech-Act>
   <wsag:TerminationTime> xs:Time </wsag:TerminationTime>
</wsag:Context>
<wsag:Terms>
    <wsag:ServiceDescriptionTerm wsag:Name="propose" wsag:ServiceName="xs:NCName">
       <Action:Executor> sender s </Action:Executor>
       <Action:Process> γ </Action:Executor>
       <Action:Action-PreCondition> received(r.accept(s,γ) </Action:Action-PreCondition>
       <Action:Operation> do(r,γ) <!--call procedure γ on service r) </Action:Operation>
     </wsag:ServiceDescriptionTerms>
</wsag:Terms>
</wsag:Propose>
```

A call for proposal is normally broadcasted from a sender to a number of agents or services. Here, let such a call be denoted by *s.cfp(r,*γ*)* where sender s sends a call for proposal to (one or more services) r to do γ. This can be considered as a request for r to respond with a proposal or a refusal to execute

γ. Thus, a call for proposal is specified as a request speech-act embedding as action to be carried out by the receiver to be either a proposal or a refusal. A sender requesting a receiver to perform some action is given as *s.request(r,γ)* where s requests r to do γ. Here the condition for r to do γ in the call for proposal specification is that there has been a proposal and an acceptance of the proposal between s and r. The action-precondition field in a call-for-proposal schema thus includes *received(r.propose(s,γ))* and *received(s.accept(r,γ))*. In a call for proposal, there is a choice between sending a proposal or a refusal in the `ServiceDescriptionField`.

3.6 The Accept Proposal Speech-Act

The speech-act *s.accept(r,γ)* is read as sender s sending an accept proposal to receiver r for r to execute γ. As for the proposal speech-act, the `Context` and `ServiceDescriptionTerms` express an accept proposal action. Thus, we provide only the fields in the `ServiceDescriptionTerms` below. The condition is that there has been a prior proposal.

```
<wsag:Accept>
<wsag:Name> NCName <!--e.g. s.accept(r,γ)> </wsag:Name>
<wsag:Terms>
   <wsag:ServiceDescriptionTerm wsag:Name="accept" wsag:ServiceName="xs:NCName">
      <Action:Executor> receiver r </Action:Executor>
      <Action:Process> γ </Action:Executor>
      <Action:Action-PreCondition> received(r.propose(s,γ) </Action:Action-PreCondition>
      <Action:Operation> do(r,γ) <!--call procedure γ on service r) </Action:Operation>
    </wsag:ServiceDescriptionTerms>
</wsag:Terms>
</wsag:Accept>
```

4 Schema for Protocol Construction

Given that we have defined the speech-act content of the messages, as schemas compliant with the WS-Agreement standard, we now focus on the form of a web services conversation. This primarily involves the sequencing of the messages in order that a conversation that follows a protocol leads to a desired state. Such sequencings are given by the allowable transitions in a protocol. Now in the WSCL schema, allowable sequences are defined in the `Interaction` and `Transition` fields, but they are not bound to a web service or an action (as discussed in section 2.2). Therefore, we increase the expressiveness of the WSCL schema for web conversations by extending the WSCL representation

for sequenced document exchange to also represent *sequenced speech-act exchanges* in addition to the interaction and transition elements.

As described in section 2.1, a conversation has `ConversationInteractions` and `ConversationTransitions`. Thus, we add states to `Interactions` and actions to `Transitions`. In the next section, we define the type `state` as part of an `Interaction` and the complex type `action` as part of a `ConversationTransition`. States added to the `Interaction` element are named propositions and include optional fields indicating which service triggered that state. Atomic actions and complex processes that are added to `ConversationTransitions` are of type `wscl:Process` and explicitly name the transitions. This results in an interaction protocol where actions (such as offer) lead to specific states that are propositions that hold after an offer. Currently, in the WSCL specification, transitions that lead from `Source` to `Destination` Interaction (states) are not named and deal only with document exchanges. However as these transitions are distinguished only by their `Source` and `Destination` states, there is significant scope for ambiguity since there can be more than two transitions in or out of an `Interaction` (state). Moreover, we need to name actions and we also need to express the fact that a transition can also be a speech-act.

4.1 States

`ConversationInteractions` list a sequences of `Interaction` elements that reference the documents exchanged and the types of exchanges (see section 2.1). We add a `State` element to the `Interaction` element in the WSCL Schema definition. A `State` has a name, a boolean attribute (whether the state holds or not), and optionally includes the service that triggered the state, the recipients and any action needed.

```
<xsd:element name="Interaction>
   <xsd:element name = "State"/>
    <xsd:complexType>
     <xsd:attribute name="xs:Name" type="xs:boolean" use="required"/>
     <xsd:sequence>
      <xsd:element name="Initiator" type="wsag:Initiator" use="optional"/>
      <xsd:element name="Respondents" type="wsag:Respondents" use="optional"/>
      <xsd:element name="Process" type="wscl:Process" use="optional"/>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
</xsd:element>
```

By way of illustration, we add a state to one interaction in the example of figure 3. Specifically, the state *offered(s,r, α)* is added to the "CatalogInquiry" state.

```
<Interaction interactionType="Receive" id="CatalogInquiry">
    <InboundXMLDocument hrefSchema ="http://conv123.org/CatalogRQ.xsd" id ="CatalogRQ"/>
    <State Name="offered" value="true" >
       <Initiator> s </Initiator>
       <Respondents> r </Respondents>
       <Process> α </Process>
    </State>
</Interaction>
```

4.2 Transitions

In WSCL, the type `ConversationTransitions` includes `Transition` elements, which list the `SourceInteraction` and `DestinationInteraction` and any conditions for the transition. The `wscl:Process` type is thus defined in XML as follows:

```
<xsd:element name ="Process">
   <xsd:attribute name="Name" type="xsd:ID" use="required"/>
   <xsd:complexType>
    <xsd:choice minOccurs="0" maxOccurs="1" >
     <xsd:element name="SA" type="wsag:specific-speech-act"/>
     <xsd:element name="Atomic" type="wsdl:operation"/>

     <xsd:element name="Sequential"/>
      <xs:sequence>
       <xsd:element name ="Process" type="wscl:Process">
       <xsd:element name ="Process" type="wscl:Process">
      </xs:sequence>

     <xsd:element name="Alternative"/>
      <xs:choice>
       <xsd:element name ="Process" type="wscl:Process">
       <xsd:element name ="Process" type="wscl:Process">
      </xs:choice>

     <xsd:element name="Iterative"/>
        <xsd:element name ="Process" type="wscl:Process">*


    </xsd:choice>
   </xsd:complexType>
</xsd:element name >
```

In our extensions, we additionally allow transitions to be actions other than exchanging documents (such as exchanging speech-acts or carrying out some operation on a web service). To this end, we first define a type called `wscl:Process`, which can be a speech-act defined as a WS-Agreement schema (see section 3, for example `wsag:offer`, `wsag:agreement` or `wsag:inform`). Here, an enumer-

ated type listing speech-acts specified in WS-Agreement schemas is defined as `wsag:specific-speech-act`. A WSCL process can also be an `Empty` process or an atomic operation (for example, `/bin/file/open<filename>`) or a complex WSCL process that is a sequence, an alternation or an iteration of WSCL processes.

The above definition of `wscl:Process` has been used in our WS-Agreement schema extensions (in section 2.3) and the definition of states (in section 4.1). In addition, transitions are typed as `wscl:Process` processes. Thus, we add an element called `Trigger` in the WSCL definition of transitions. A trigger has a name, is of type `wscl:Process` and is associated with the service performing the `wscl:Process`. A trigger also includes the recipients being affected by the WSCL process, the action which is an attribute in the trigger (for example, when sending a *request* to do α, the trigger is *request* and the action attribute is α) and the conditions that need to hold in order to perform the action. An example trigger is the service s sending a `wsag:offer` to r to perform α if *Cond* is true (that is, *s.offer(r,*α*)*). The element trigger, added to transitions (after `SourceInteractionCondition`) is defined as follows:

```
<xsd:element name ="Trigger">
   <xsd:attribute trigger-process="Name" type="wscl:Process use="required"/>
   <xsd:complexType>
    <xsd:element name ="Sender" type="wsag:Initiator">
    <xsd:element name ="Recipients" type="wsag:Respondents">
    <xsd:element name ="Action" type="wscl:Process">
    <xsd:element name ="Condition" type="xs:boolean" defaultvalue="false">
   </xsd:complexType>
</xsd:element name >
```

5 Contract Net Protocol in WSCL/WS-Agreement

Given the above extensions to WSCL and WS-Agreement, it is now possible to describe richer interactions between web services. To illustrate this, in this section we discuss the Contract Net protocol since it involves multiple participants and allows some form of collaboration between web services when executing a task. In addition, this protocol shows realistic interactions for enabling transactions and negotiations and illustrates the use of several of the speech-acts that we defined in section 3.

The interactions in the Contract Net protocol can be enumerated as follows: 1) A manager m issues a *call for proposals (cfp)* to group of services, G, to do process α, *m.cfp(G,* α*)*. 2) Potential contractors c respond with *proposals*, *c.propose(m,* c_α*)*. 3) The manager either rejects or accepts the proposal or can-

cels the call for proposal (respectively through *m.reject(c, α)*, *m.accept(c, α)*, *m.cancel(c, α)*). 4) Contractors inform the manager of success or failure of their execution, *c.inform(m, done(α))*.

The speech-acts *propose*, *inform*, *call for proposal* and *accept* have all been defined as WS-Agreement schemas in section 3 and therefore can be called directly in the `Transition` and `Interaction` elements in the WSCL specification. Below we show the *cfped* (after a call for proposal) and *proposed* (after a proposal) states and the *call for proposal* and *accept* transitions of the Contract Net protocol as a WSCL conversation. The `Conversation` is named "ContractNetProtocol" and has the standard start and end interactions defined. In the `ConversationInteractions` of the fragment of the protocol, the interactions *cfped* and *proposed* are defined as `ReceiveSend` interaction types. The `Interaction` *cfped* specifies a call for proposal as an inbound document and a proposal as an outbound one. It is the `State` field in the `Interaction` that specifies that the call for proposal is received by the `Initiator c` (a contractor) and that the proposal is sent to respondent `Initiator m` (the manager) to perform process α . Similarly, the *proposed* interaction specifies that a proposal is received as an inbound document and outbound documents are agreements or rejections. Again, the state field specifies that the initiator m (manager) receives a proposal and sends out to c (contractors) agreements or rejections for doing process c_α.

The second part of the fragment specifies the `ConversationTransitions`, where the conversation is started by making a call for proposal from the *start* interaction leading to the *cfped* interaction. The call for proposal is defined as `wsag:CallForProposal` process, which is a WSCL process and has been defined in section 3.5. Within the trigger field, we also specify that the sender m (manager) sends out the call for proposal to receivers G (contractors) to perform action α. Similarly, from the *proposed* `SourceInteraction`, there is a transition to *accepted*, meaning that an acceptance may follow a proposal. The trigger field specifies the accept as of type `wsag:accept`, sent by m to c for doing process c_α.

```
<Conversation name="ContractNetProtocol"
<!-- WSCL specification for the Contract Net protocol; import WSCL and WS-Agreement-->
   initialInteraction="Start" finalInteraction="End" >
 <ConversationInteractions>

<Interaction interactionType="ReceiveSend" id="cfped">
<!-- receives a call for proposal and sends out a proposal document -->
    <InboundXMLDocument hrefSchema="http://conv.org/call-for-proposal-1" id="cfp-1"/>
    <OutboundXMLDocument hrefSchema="http://conv.org/proposal" id="proposal"/>
    <State Name="cfped" >
    <!-- c receives the call for proposal and proposes to m-->
```

```
        <Initiator> c </Initiator>
        <Respondents> m </Respondents>
        <Process> α </Process>
    </State>
</Interaction>
<Interaction interactionType="ReceiveSend" id="proposed">
<!-- receives a proposal, sends out an agreement or rejection document -->
    <InboundXMLDocument hrefSchema="http://conv.org/proposal" id="proposal"/>
    <OutboundXMLDocument hrefSchema="http://conv.org/agreement" id="agreement"/>
    <OutboundXMLDocument hrefSchema="http://conv.org/rejection-reason" id="rejection"/>
    <State Name="proposed" >
    <!-- m receives proposals and rejects or agrees with c-->
        <Initiator> m </Initiator>
        <Respondents> c </Respondents>
        <Process> c_α </Process>
    </State>
</Interaction>
</ConversationInteractions>
<!-- actions triggering interaction-->
<ConversationTransitions>
<Transition>
   <!-- conversation started by a CallForProposal-->
  <SourceInteraction href="start"/>
  <DestinationInteraction href="cfped"/>
    <Trigger trigger-process"wsag:CallForProposal">
   <!-- CallForProposal is of type wsag and sent by m to G to perform α-->
        <Sender> m </Sender>
        <Recipients> G </Recipients>
        <Action> α </Action>
    </Trigger>
</Transition>
<Transition>
   <!-- The accept transition from a proposed state-->
  <SourceInteraction href="proposed"/>
  <DestinationInteraction href="accepted"/>
    <Trigger trigger-name="wsag:accept">
   <!-- accept is of type wsag and sent by m to G-->
        <Sender> m </Sender>
        <Recipients> c </Recipients>
        <Action> c_α </Action>
        <Condition> time < deadline </Condition>
    </Trigger>
</Transition>
</ConversationTransitions>
</Conversation>
```

6 Visualisation of Protocols

As can be seen from the example of the previous section, the XML representation of a protocol can be relatively long. Now this may be not be a problem when programming the conversations because it is possible to re-use tools for specifying web services [13], [14] to code WSCL/WS-Agreement conversations. On the other hand, these conversation templates have to be shared between services and agents, and implemented by their developers. One developer may need to implement a protocol proposed by some other web services or agent for them to follow it together. A developer unfamiliar with another's protocol may misundersand it. Therefore, misunderstandings about allowable states and transitions must be prevented for successful conversations and agreements. To alleviate this, we propose a visual representation of the WSCL protocol to facilitate comprehension and to provide a high-level yet precise, language which allows protocol designers to express and reason about interaction concepts at their natural level of abstraction. Specifically, we choose statecharts [15] (which are part of UML diagrams) as a visual representation since in our previous work we have found them to be intuitive and sufficiently expressive for expressing multi-agent interaction protocols [16]. Moreover, since a web service behaviour is a partially ordered set of operations, then an XML protocol can be mapped onto a statechart. In particular, `Interactions`, `States` and `Transitions` in a WSCL protocol are analogous to states and transitions in a statechart as we show in section 6.3. First, however, we show how the contract net protocol is expressed as a statechart (section 6.1), then we provide a translation between statecharts and WSCL components (section 6.3). We also introduce the notion of sub-states which allow more concise specifications of conversations for both WSCL and statechart representations (section 6.2).

6.1 The Contract Net Protocol in Statecharts

Figure 6 shows an abstraction of the Contract Net protocol in statecharts (see section 5 for a description of the protocol). In particular, let the manager be denoted by m, the group of contractor web services as G and one specific contractor web service as c. As can be seen, there is a hierarchical representation of states with parent and substates (see section 6.2). The outermost parent state is *contract_net* which has *open* and *closed* as sub-states. This means that a Contract Net can either be *open* or *closed*, but not both. The interaction is started with a call for proposal *m.cfp(G,*p_m*)* from manager m, to the group of web services G for task p_m. The resulting state is *cfped(m,G,*p_m*)*, parameterised with the manager as sender, the receivers and the task requested for execution. Contractors may refuse a call for proposal, ultimately leading to a *closed* and *refused* state if no proposals are received. However, if a proposal

is received from a contractor, the state becomes *proposed(c,m,p)*, and remains so as other proposals occur, until the manager either accepts (*m.accept(c,p_c)*) or rejects (*m.reject(c,p_c)*) the proposals. Acceptances have the terms of agreements and service descriptions included. Those web services having received accepts eventually send *c.inform(m,p_c)* to the manager indicating the success or failure of executing the task. The interactions terminate in a sub-state of *closed*.

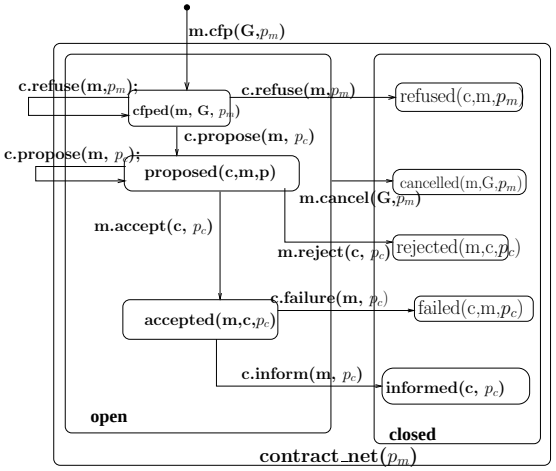

Fig. 6. Contract Net Protocol in Statecharts

It can be seen that the statechart representation of the Contract Net protocol is more concise and intuitive than the corresponding XML protocol from section 5. Therefore for the reasons outlined at the beginning of section 6 we believe it would be useful to accompany the protocols defined in WSCL/WS-Agreement with a statechart counterpart. Given this, we investigate the ease with which translations from XML to statecharts (and vice versa) can be made. Such translations are needed if we are to automatically generate the WSCL conversation from the design of a protocol in statecharts. The reverse translation is also important because it can help one developer to understand another developer's protocol. The first step of the translation requires the definition of *sub-state* in WSCL because sub-states are a fundamental part of statecharts as can be seen in the Contract Net protocol statechart in figure 6.

6.2 Sub-States in WSCL

Part of the conciseness of the Contract Net protocol in figure 6 is due to the use of sub-states. Here, sub-states are states embedded within other states (for example, the *contract_net* parent state in figure 6 has the sub-states *open* and *closed*). Such representations are particularly useful because any transition from a parent state is also a transition from its sub-state, without needing to explicitly show the transition from the sub-state. Thus, hierarchical states

allow the redundancy in expressing the same transition from all the sub-states of a particular parent state to be removed. Thus, in figure 6, we need only to show the *cancel* transition from the *open* parent state and it can be inferred that a *cancel* transition can also occur from the sub-states *cfped*, *proposed* and *accepted*.

As such, neither WSCL nor WS-agreement allow sub-states to be expressed although sub-states would reduce the length of the specification of conversations. To this end, we introduce the type `substate` in the WSCL schema. In so doing, we impose the constraint that transitions from the parent state also hold for all of its sub-states. Thus, the element `substate` is a list and is defined inside the scope of the element `State` in an `Interaction`. The XML definition of `substate` is given as follows:

```
<xsd:element name="State">
    <xsd:simpleType name="substate" use="optional">
        <xs:list item Type="Interaction:State"/>
    </xs:simpleType>
</xsd:element>
```

In the definition of sub-states, only the nearest sub-states need to be defined from the perspective of a parent state. For example, in the Contract Net protocol, *accepted* is the sub-state of *open*, which itself is the sub-state of *contract_net*. This means there are three levels of nesting. However, in the declaration of the *contract_net*, only *open* and *closed* have to be included (i.e. the three sub-states of *open* are included only in the declaration of *open* and not the *contract_net* state).

6.3 Translation between WSCL and Statecharts

The bidirectional translation between the WSCL/WS-Agreement protocols and statecharts is straightforward. There are three components in the statecharts to translate:- the states, their sub-states and the transitions. States that have only an incoming transition in statecharts are `Receive` WSCL `Interaction` types. States that have only an outgoing transition in statecharts are `Send` WSCL `Interaction` types. States that have both an incoming and outgoing transition in statecharts are `ReceiveSend` WSCL `Interaction` types. The information about inbound and outbound documents may be visually represented inside the states as in figure 2. Given this, figure 7 shows a general translation from a statechart to a WSCL/WS-Agreement protocol (the translation from an XML protocol to a statechart is just the reverse process).

Translating that $State_\beta$ is a sub-state of $State_\alpha$ from a statechart requires identifying, in the WSCL specification, the field specifying $State_\alpha$ and embed-

Fig. 7. Translation of a Protocol from a Statechart to a WSCL/WS-Agreement Specification

ding in it a field `substate` with name *State*$_\beta$. Translating, from statechart to WSCL, a transition *Action*$_\alpha$ that initialises a conversation in state *State*$_\alpha$ requires that there is a `Transition` with `SourceInteraction` *Start*, trigger process *Action*$_\alpha$ of type `wscl:Process` and `DestinationInteraction` *State*$_\alpha$. A transition *Action*$_\beta$ from *State*$_\alpha$ to *State*$_\beta$ in statecharts is translated to a `Transition` with `SourceInteraction` *State*$_\alpha$, trigger process *Action*$_\beta$ of type `wscl:Process` and `DestinationInteraction` *State*$_\beta$. A terminating transition *Action*$_\beta$ into *State*$_\beta$ is translated from statecharts into WSCL in two steps. First the translation of the transition leading from the previous state into *State*$_\beta$ through action *Action*$_\beta$ is translated. Then, there is a `Transition` with `SourceInteraction` *State*$_\beta$, trigger process *empty* of type `wscl:Process` and `DestinationInteraction` *end*.

7 Conclusions and Future Work

In this paper, we have focussed on flexible interactions between web services because they are fundamental if web services are to reach their full potential in future networked environments. In such environments, there are a number of limitations on the applicability of the current versions of the web service agreement and conversation languages stemming from the fact that interactions between service providers and clients require more than just requesting simple tasks. Specifically, this paper addresses the need for developers to code client applications that can bind to and interact with services of a specific

type, according to an interaction protocol. To achieve this , we have specified speech-acts as WS-Agreement schemas for richer messages in conversations than just offer and agree. We have combined WS-Agreement and WSCL schemas in order to specify sequences of messages from the perspective of WS-Agreement and, from the perspective of WSCL, to exchange speech-acts and obtain more flexible and richer conversations. We have also extended the structure of WS-Agreement to include sender and recipients of messages for the specification of speech-acts. In turn, the Web Services Conversation Language has also been extended to include states, sub-states, transitions and WSCL processes. These extensions allow who is sending which message to whom to be expressed. Consequently, as we have shown, protocols of realistic expressiveness (such as the Contract Net protocol) can be specified in our WSCL/WS-Agreement extended language. Finally, the statecharts notation has been proposed as a visual counterpart to facilitate comprehension of the protocols and we have shown that translation between a statechart protocol and its XML representation is straightforward.

As future work, we intend to verify the Contract Net protocol in WSCL/WS-Agreement in order for it to be sharable without leading to any misunderstandings amongst participants. In particular, model checking will be investigated since it automates the verification of properties of finite-state concurrent systems. Moreover, much of the work on web services and agent interactions remains to be tested in real open environments and we intend to test our work in these environments, where network communication are not perfect. More specifically, the reliability of web services is decreased by the fact that they use HTTP (which is a best effort delivery service). Given this, we can bring to here our existing work on multi-agent interaction protocols in fallible communication domains [17]. In this, we investigated synchronisation of messages depending on the features of the communication layer, such as delayed in messages.

8 Acknowledgements

We thank Professor David de Roure and Dr. Nick Gibbins for their helpful comments on reading the paper.

References

[1] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard, Web Services Architecture, World-Wide-Web Consortium (W3C), `http://www.w3.org/TR/ws-arch` (2003).

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

[3] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H. Nielsen, SOAP Version 1.2 Part 1: Messaging Framework, World-Wide-Web Consortium (W3C), http://www.w3.org/TR/soap/ (2003).

[4] N. R. Jennings, An agent-based approach for building complex software systems, Comms. of the ACM 44 (4) (2001) 35–41.

[5] A. Banerji, C. Bartolini, D. Beringer, *et al.*, Web Services Conversation Language (WSCL) 1.0, World-Wide-Web Consortium (W3C), http://www.w3.org/TR/wscl10 (2002).

[6] A. Andrieux, K. Czajkowski, A. t. Dan, Web Services Agreement Specification (WS-Agreement), World-Wide-Web Consortium (W3C), http://www.w3.org/XML, http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf (2004).

[7] R. G. Smith, The contract net protocol: High-level communication and control in a distributed problem solver, IEEE Transactions on Computers C-29 (12) (1981) 1104–1113.

[8] Object Management Group, Common Object Request Broker Architecture, http://www.corba.org.

[9] MicrosoftCom Technologies, Distributed Component Object Model, http://www.microsoft.com/com/tech/DCOM.asp.

[10] Foundation for Intelligent Physical Agents, FIPA Communicative Act Library Specification, http://www.fipa.org (2002).

[11] J. L. Austin, How to Do Things with Words., Oxford University Press, 1962.

[12] J. R. Searle, Speech acts: An essay in the philosophy of language, Cambridge University Press, 1969.

[13] IBM Corporation, IBM Web Services Toolkit 2.1, http://www.alphaworks.ibm.com/tech/webservicestoolkit.

[14] BEA Systems, Inc., Introduction to WebLogic Web Services, http://e-docs.bea.com/wls/docs81/webserv/overview.html#1071587.

[15] D. Harel, M. Politi, Modeling reactive systems with statecharts, McGraw-Hill, 1998.

[16] S. Paurobally, R. Cunningham, N. R. Jennings, Developing agent interaction protocols using graphical and logical methodologies, in: Programming MAS, languages, frameworks, techniques and tools workshop, 2003, pp. 124–131.

[17] S. Paurobally, R. Cunningham, N. R. Jennings, Ensuring consistency in joint beliefs of interacting agents, in: Proc. 2nd Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems, 2003, pp. 662–669.