Interaction Protocols and Capabilities: a preliminary report*

Matteo Baldoni, Cristina Baroglio, Alberto Martelli,
Viviana Patti, and Claudio Schifanella

Dipartimento di Informatica — Università degli Studi di Torino
C.so Svizzera, 185 — I-10149 Torino (Italy)
{baldoni,baroglio,mrt,patti,schi}@di.unito.it

Abstract. A typical problem of the research area on Service-oriented Architectures is the composition of a set of existing services with the aim of executing a complex task. The selection and composition of the services are based on a description of the services themselves and it often exploits an abstract description of the system that we wish to build. A distinction is made between the global and the individual points of view of the interaction between the services. Interaction protocols (or choreographies) capture the interaction as a whole, defining the rules that entities should respect in order to guarantee the interoperability; they do not refer to specific services but they specify the roles and the communication among the roles. Policies (behavioral interfaces in web service terminology), instead, focus on communication from the point of view of the individual services. We can find many works that aim at verifying if a service can take part in a specific interaction, that is if its behavioral interface is conformant to the global protocol. The idea of focussing on a representation that captures solely the message exchange is, however, not sufficient. It is not sufficient when one means to exploit the protocol description for synthesizing an executable policy, that is to be supplied to a service which, by itself, cannot take part to the interaction. In this case it is necessary to take care of the interface between the new policy and the service. For being executable, the policy must, in fact, have access to the internal state of the service, for instance, for building the contents of the messages that will be exchanged with others. This can be done by associating to the protocol description a description of a set of "actions" (in a broad sense) that are not necessarily communicative actions but that are necessary to tie the policy to the player. We will call such actions *capabilities*. In this paper we present a preliminary study of the concept of *capability* and an extension of WS-CDL, the choreography language by W3C. We also introduce the notion of *capability test*.

* This research has partially been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net), and it has also been supported by MIUR PRIN 2005 "Specification and verification of agent interaction protocols" national project.

1 Introduction

In various application contexts there is a growing need of being able to compose a set of heterogeneous and independent entities with the general aim of executing a task, which cannot be executed by a single component alone. In an application framework in which components are developed individually and can be based on various technologies, it is mandatory to find a flexible way for gluing components. The solution explored in some application scenarios is to compose entities based on dialogue. This happens, for instance, in both the cases of web services and of multi-agent systems (MAS). In the case of web services, for instance, the language BPEL4WS [19] has become the *de facto* standard for building executable composite services on top of already existing services by describing the flow of information in terms of exchanged messages. On the other hand, the problem of aggregating communicating agents into (open) societies is well-known in the research area about MASs, where a lot of attention has been devoted to the issues of defining interaction policies, verifying the interoperability of agents based on dialogue, and checking the conformance of policies w.r.t. global communication protocols [26, 16, 10].

The problem is highly complex and encompasses various skills such as: the ability of describing the goal to be accomplished, the ability of describing the solution (in terms of involved entities and their interactions), that of identifying in the pool of available entities those which can solve subproblems, etc. From the point of view of the scientific disciplines which are involved, a non- exhaustive list includes: AOSE, MAS, WS, and software engineering.

As observed in recent work [25, 5], the MAS and WS research areas show convergences in the methodology according to which systems of agents, on a side, and composite services, on the other, are designed, implemented and verified. In both cases it is possible to distinguish the design of a system, which is independent from the specific agents/services which will take part to the interaction, and the implementation or the identification of the specific entities that will interact. The former level gives a global view of the system as a whole, in which roles are identified and their interactions specified, so that it is possible to verify global properties of the system. This high-level design can, for instance, be done in UML, automata or Petri Nets in the case of agents, while in the case of services it can be done in WS-CDL. The other level concerns the specification of the interaction policy of the single entity. Differently than in the previous case, now the perspective is that of the single agent/service, which might interact with only a subset of the involved entities. This specification can coincide with a real implementation (in BPEL, for services, or in some declarative language, for agents). A lot of effort is being devoted to the problem of exploiting these two levels of description for deciding in some automatic way whether a service/agent respects some given interaction schema (conformance verification) [12, 9] or whether a set of services/agents will be able to interact with each other (interoperability) [3], without getting stuck in some deadlock condition.

The idea of focussing on a representation that captures solely the message exchange is, however, not sufficient (in our opinion) in all those cases in which

we mean to exploit this description for synthesizing an executable policy to be supplied to an agent/service that wishes to take part to the interaction. In other words, when a policy is to be supplied to an already existing party that does not, itself, have one (policy) allowing it to take part to the interaction. In this case it is necessary to take care of the interface between the new policy and the agent/service. For being executable, the policy must, in fact, have access to the internal state of the agent/service, for instance, for building the contents of the messages that will be exchanged with others. This can be done by associating to the protocol specification a description of a set of "actions" (in a broad sense) that are not necessarily communicative actions but that are necessary to tie the *policy* to the *player*. We will call such actions *capabilities*.

The term capability has recently been used by Padgham et al. [21] (further extended in [22]), in the BDI framework, for identifying the "ability to react rationally towards achieving a particular goal". More specifically, an agent has the capability to achieve a goal if its plan library contains at least one plan for reaching the goal. The authors incorporate this notion in the BDI framework so as to constrain an agent's goals and intentions to be compatible with its capabilities. This notion of capability is orthogonal w.r.t. what proposed in our work. In fact, we propose to associate to a choreography (or protocol) specification, aimed at representing an interaction schema among a set of yet unspecified peers, a set of *requirements* of abilities. Such requirements (that we call capabilities) specify "actions" (in a broad sense) that peers, willing to play specific roles in the interaction schema, should exhibit. In order for a peer to play a role, some reasoning must be performed for deciding if it matches the requirements.

In this perspective, our notion of capability resembles more closely (sometimes unnamed) concepts, that emerge in a more or less explicit way in various frameworks/languages, in which there is a need for defining interfaces. One example is Jade [14], the well-known platform for developing multi-agent systems. In this framework policies are supplied as partial implementations with "holes" that the programmer must fill with code when creating agents. Such holes are represented by methods whose body is not defined. The task of the programmer is to implement the specified methods, whose name and signature is, however, fixed in the partial policy.

Another example is powerJava [6, 7], an extension of the Java language that accounts for roles and institutions. Without getting into the depths of the language, a role in powerJava represents an interlocutor in the interaction schema. A role definition contains only the implementation of the interaction schema and leaves to the role-player the task of implementing the internal actions. Such calls to the player's internal actions are named "requirements" and are represented as method prototypes.

This paper presents a preliminary report about a work aimed at introducing the concept of capability in the global/local system/entity specifications, in such a way that capabilities can be accounted for during the reasoning processes that are applied for dynamically building and customizing policies.

The paper is organized as follows. Section 2 defines the setting of this work and introduces the problem that we face. Moreover, a first example of protocol (the well-known FIPA Contract Net protocol), that is enriched with capabilities, is reported. Section 3 introduces the notion of *capability test*, it discusses an approach to this verification making a comparison with systems in which this notion is implicit. It also contains a description of various reasoning techniques that can be associated with the capability test for performing a customization of the policy being constructed. In order to make this proposal more concrete we sketch in Section 4 a possible extension of WS-CDL [27] aimed at introducing capability descriptions in web service choreographies. Conclusions follow.

2 Interaction protocols and capabilities

Multi-agent systems often comprise heterogeneous agents, that differ in the way they represent knowledge about the world and about other agents, as well as in the mechanisms used for reasoning about it. In general, every agent in a MAS is characterized by a set of actions and/or a set of behaviors that it uses to achieve a specific goal. In order to interact with the others, an agent specification must describe also the communicative behavior. Concerning the specification of interaction, according to Agent-Oriented Software Engineering [13], a distinction is made between the global and the individual points of view of the interaction between the various agents. The *global* viewpoint is captured by an *abstract protocol* that contains the rules that must be followed by the society and it is expressed by formalisms like AUML sequence diagrams [20]. In general, each specification language allows the definition of a list of *roles*, which will be played by some agents, and a set of communicative acts that they will exchange. The *local* viewpoint expresses communication from the perspective of an agent, that plays one of the roles and it is captured by the agent's communication policy, which is usually written in some executable language. In this context the problem of verifying whether an agent's interaction policy respects a given protocol is extremely relevant. This problem is known as *conformance* test [1, 12, 3]. The conformance test can be a means for guaranteeing a priori the interoperability of a set of agents, each playing one of the roles described by a given protocol [3].

The framework that we have briefly outlined above shows convergences with the research carried on in Service-oriented Computing. In this latter context the role played by interaction protocols is in a way played by *choreographies*, while the role of agents is played by *web services* (or peers). In order for a peer to take part to a choreography it is necessary to check whether it is conformant to the latter, in a very similar way to what is done in the case of agents. Also in this application context choreography should entail interoperability [5, 9].

In this work we will, however, focus on the case in which a peer does not have a valid policy but, despite this fact, we would like the interaction to take place anyway. Of course, in order for this to happen, it is necessary that the peer *adopts* a new interaction policy. If this scenario were set in an agent-framework, one might think of enriching the set of behaviors of the agent, which failed the

conformance test, by asking other agents to supply a correct interaction policy. This solution has been proposed from time to time in the literature; recently it was adopted in Coo-BDI architectures [2]. CooBDI extends the BDI (*Belief, Desire, Intention*) model in such a way that agents are enabled to cooperate through a mechanism, which allows them to exchange plans and which is used whenever it is not possible to find a plan, for pursuing a goal of interest, by just exploiting the local agent's knowledge. The ideas behind the CooBDI theory have been implemented by means of WS technologies, leading to CooWS agents [8]. Another recent work in this line of research is [24]. Here, in the setting of the DALI language, agents can cooperate by exchanging sets of rule that can either define a procedure, or constitute a module for coping with some situation, or be just a segment of a knowledge base. Moreover, agents have reasoning techniques that enable them to evaluate how useful the new information is.

These techniques, however, cannot be directly imported in the context of Service-oriented Computing. The reason is that, while in agent systems it is not a problem to find out *during* the interaction that an agent does not own all the necessary actions, when we compose web services it is fundamental that the analogous knowledge is available before the interaction among the peers takes place.

Going back to the situation in which a peer failed the conformance test, one might think of using the protocol definition for supplying the service with a new policy that is obtained directly from the definition of the role that the peer would like to play. A policy skeleton could be directly synthesized in a semi-automatic way from the protocol description. A similar approach has been adopted, in the past, for synthesizing agent behaviors from UML specifications in [17]. In this perspective, a problem arises: protocols only concern communication patterns, i.e. the interactions of a peer with others, abstracting from all references to the internal state of the player and from all actions/instructions that do not concern communication. Nevertheless, in our framework we are interested in a policy that the peer will *execute* and, for permitting the execution, it is necessary to express to some extent also this kind of information. The conclusion is that if we wish to use protocols for synthesizing policy skeletons, we need to specify some more information, i.e. actions that allow us the access to the peer's internal state. Throughout this work we will refer to such actions as *capabilities*. In [17], that we cited just above, it was up to the programmer to write the code of such actions; the methodology that is described in that work is, in fact, aimed at helping the *design* of a whole agent system. Differently than Mascardi *et al.*, for what concerns the design of the system, we consider only the specification of interaction protocol *but* we would like the synthesis to produce an executable policy without human intervention. To this aim it is necessary to perform a different verification, i.e. that the peer has the capabilities which are required by the role specification.

Checking whether a peer has the desired capabilities is, in a way, a complementary test w.r.t. checking conformance. With a rough approximation, when I check conformance I abstract away from the behavior that does not concern the

communication described by the protocol of interest, focussing on the interaction with a set of other peers that are involved, whereas checking capabilities means to check whether it is possible to tie the description of a policy to the execution environment defined by the peer.

2.1 An example: the contract net protocol

For better explaining our ideas, in this section we consider as a choreography the well-known FIPA ContractNet Protocol [11], pinpointing the capabilities that are required to a peer which would like to play the role of *Participant*. Figure 1 reports a UML version of the protocol, enriched with dotted rectangles that represent capabilities.

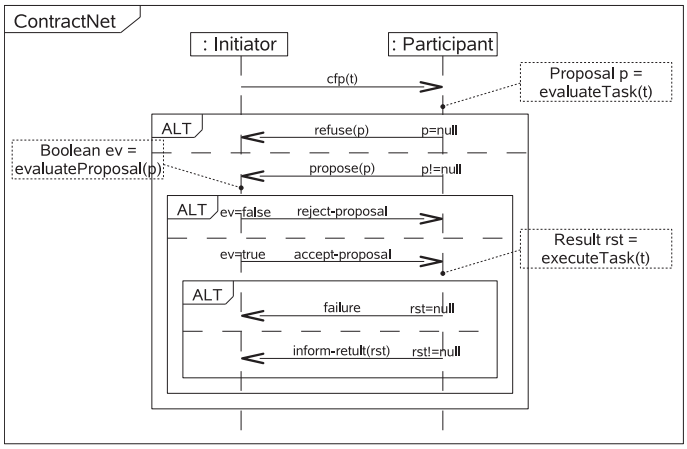

Fig. 1. The FIPA ContractNet Protocol, represented by means of UML sequence diagrams, and enriched with capability specifications.

ContractNet is used in electronic commerce and in robotics for allowing entities, which are unable to do some task, to have it done. The protocol captures a pattern of interaction, in which the initiator sends a *call-for-proposal* to a set of participants. Each participant can either accept (and send a proposal) or refuse. The initiator collects all the proposals and selects one of them. Figure 1 describes the interactions between the *Initiator* and one of the *Participants*.

In this example we can detect three different capabilities, one for the role of Initiator and two for the Participant. Starting from an instance of the concept Task, the Participant must be able to evaluate it by performing the *evaluate-Task* capability, returning an instance of the concept Proposal. Moreover, if its proposal is accepted by the *Initiator*, it must be able to execute the task by using the capability *executeTask*, returning an instance of concept Result. On the

other side, the Initiator must have the capability *evaluateProposal* that chooses a proposal among those received from the participants.

In order to play the role of Participant a peer will, then, need to have the capabilities *evaluateTask* and *executeTask*, whereas it needs to have the capability *evaluateProposal* if it means to play the role of Initiator. As it emerges from the example, a capability identifies an action (in a broad sense) that might require some inputs and might return a result. This is analogous to defining a method or a function or a web service. So, for us, a capability will be specified by its name, a description of its inputs and a description of its outputs. This is not the only possible representation, for instance if we interpret them as actions, it would make sense to represent also their preconditions and effects.

3 Checking capabilities

In a conformance test we exploit a schema of interaction, the choreography or the protocol, given a priori. The idea that we mean to explore for checking capabilities is to do something analogous for what concerns the internal behavior. In particular, we propose to exploit a description of the required capabilities (see the previous section), which act as connecting points between the external, communicative behavior of the peer and its internal behavior.

The capability test obviously depends on the way in which the policy is developed and therefore it depends on the adopted language. In Jade [14] there is no real capability test because policies already supply empty methods corresponding to the capabilities, the programmer can just redefine them. In powerJava the check is performed by the compiler, which verifies the implementation of a given interface representing the requirements. For further details see [6], in which the same example concerning the ContractNet protocol is described.

In the scenario that we have outlined in the previous section, the capability test is done a priori w.r.t. all the capabilities required by the role specification, however, the way in which the test is implemented is not predefined and can be executed by means of different matching techniques. We could use a simple *signature matching* technique, like the one used in classical programming languages and also by powerJava, as well as techniques that perform more flexible forms of matching. We consider particularly promising to adopt *semantic matchmaking* techniques proposed for matching web service descriptions with queries, based on *ontologies* of concepts. The use of semantic matchmaking techniques allows the matching of capabilities with different names, though connected by an ontology, and with different numbers (and descriptions) of input/output parameters.

For instance, let us consider the *evaluateProposal* capability associated to the role *Initiator* of the ContractNet protocol (see Figure 1). This capability has an input parameter (a proposal) and is supposed to return a boolean value, stating whether the proposal has been accepted or refused. A first example of flexible, semantics-based matchmaking consists in allowing a service to play the part of *Initiator* even though it does not have a capability of name *evaluateProposal*. Let us suppose that *evaluateProposal* is a concept in a shared ontology. Then,

if the service has a capability *evaluate*, with same signature of *evaluateProposal*, and *evaluate* is a concept in the shared ontology, that is more general than *evaluateProposal*, we might be eager to consider the capability as matching with the description associated to the role specification.

Other forms of semantic matchmaking concern the input or output parameters. For instance, in [23] the ontological reasoning is applied to the parameters of a semantic web service, which are compared to a query. In the WSMO initiative [15], instead, different degrees of matching are formalized, which concern only the output parameters of a service, that are compared to those in the query.

In the following we will exemplify our proposal by introducing, in an explicit way, capabilities in WS-CDL. One way for performing the capability test in this enriched WS-CDL could be to exploit already developed techniques for the semantic matchmaking, such as those introduced above. Please, notice that at this stage of the work our focus is not to define some new language and that the enriched WS-CDL is just aimed at grounding our proposal to the reality of web services. A first step in this research.

3.1 Reasoning on capabilities

In the previous section, we have described the capability test, showing how it is possible to implement flexible forms of matching. This is, however, just a starting point and further customization can be achieved.

A policy can be seen as a procedure with different *execution traces*. Each execution trace corresponds to a branch in the policy. So an observation that we can make is that it is quite likely that only a part of the capabilities associated to a role will be used in a given execution trace. As an example, Figure 2 shows three alternative execution traces for a given policy, which contain references to different capabilities. In particular, one of the two highlighted traces exploits capability *C1* and capability *C3*, the other instead exploits *C1* and *C4*. The third possible execution trace contains only capability *C2*.

As a first consequence, we can think of a simplification of the capability test in which only the execution traces concerning the specific call, that the service would like to enact, are considered. This set, which will probably consist of a single trace, will tell us which capabilities are actually necessary in our execution context (i.e. given the specified input parameter values). In this perspective, it is not compulsory that the service has all the capabilities associated to the role but it will be sufficient that it has those used in this set of execution traces. For instance and with reference to Figure 2, suppose that for some given input values, only the first execution trace (starting from left) might become actually executable. This trace relies on capabilities *C1* and *C3* only: it will be sufficient that our service owns such capabilities for making the *policy call* executable by it. If a *declarative representation* of the policy were given, e.g. see [4], it would be possible to perform a rational inspection of the policy, in which the execution is simulated. During the simulation we could focus on the execution traces that allow the service to complete the interaction for the inputs of the given call. As a last step we could, then, collect the capabilities used in these traces only (*C1*,

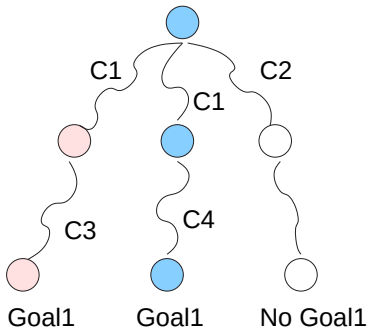

Fig. 2. Execution traces for a policy skeleton: two traces allow to reach a final state in which *goal1* is true but exploiting different capabilities.

C3, and *C4* but not *C2*). In this way, we could restrict the capability test to the subset of capabilities associated to the role, which have been identified during the simulation.

On the other hand, it is also possible to generalize this approach and determine the set of all the execution traces that can possibly be engaged by a given service, independently from its specific inputs. In fact, having the possibility of inspecting the possible evolutions of a policy, it would be possible to single out those execution traces that require only the subset of capabilities that the service can execute. In this way, the policy can be customized w.r.t. the peculiarities of the service, guaranteeing the success under determined circumstances.

A third possible reasoning task consists on focussing on those execution traces that, after the execution, make a certain condition become true in the service internal state. For instance, with reference to Figure 2, two out of the three possible executions lead to a final situation in which *goal1* holds. Out of this set the reasoning engine could, then, single out those interactions that can actually be executed given the set of capabilities that the service has available.

Last but not least, it is possible that the set of capabilities of a service is not completely predefined but it depends on the context and on privacy or security policies defined by the user. Therefore, I might have a capability which I do not want to use in that circumstance. Also this kind of reasoning can be integrated in the capability test. In this perspective, it would be interesting to explore the use of the notion of *opportunity* proposed by Padmanabhan et al. [22] in connection with the concept of capability (but with the meaning proposed in [21], see Section 1).

4 A case study: introducing capabilities in WS-CDL

As a case study, we introduce an example in the web services scenario. In this field, the most important formalism used to represent interaction protocols is

WS-CDL (Web Services Choreography Description Language) [27]: an XML-based language that describes peer-to-peer collaborations of heterogeneous entities from a global point of view. In this section, we propose a little extention to the WS-CDL definition by adding capability specifications in order to enable the automatic synthesis of policies described in the previous sections. The schema that defines this extension can be found at `http://www.di.unito.it/~alice/WSCDL_Cap_v1/`.

```
1  <silentAction roleType="Participant">
2    <capability name="evaluateTask">
3      <input>
4        <parameter variable="cdl:getVariable('tns:t','','')"/>
5      </input>
6      <output>
7        <parameter variable="cdl:getVariable('tns:p','','')"/>
8      </output>
9    </capability>
10 </silentAction>
```

Fig. 3. Representing a capability in the extended WS-CDL. The tag *input* is used to define one of the input parameters, while *output* is used to define one of the output parameters.

In this scenario an operation executed by a peer often corresponds to an invocation of a web service, in a way that is analogous to a *procedure call*. Coherently, we can think of representing the concept of capability in the WS-CDL extension as a new tag element, the tag *capability* (see for instance Figure 3), which is characterized by its *name*, and its *input* and *output parameters*. Each parameter refers to a variable defined inside the choreography document. The notation `variable="cdl:getVariable('tns:t','','')"` used in Figure 3 is a reference to a variable, according to the definition of WS-CDL. In this manner inputs and outputs can be used in the whole WS-CDL document in standard ways (like Interaction, Workunit and Assign activities). In particular parameters can be used in guard conditions of Workunits inside a Choice activities in order to choose alternative paths (see below for an example).

A capability represents an operation (a call not a declaration) that must be performed by a role and which is non-observable by the other roles; this kind of activity is described in WS-CDL by *SilentAction* elements. The presence of silent actions is due to the fact that WS-CDL derives from the well-known *pi-calculus* by Milner *et al.* [18], in which silent actions represent the non-observable (or private) behavior of a process. We can, therefore, think of modifying the WS-CDL definition by adding capabilities as child elements of this kind of activity. Returning to Figure 3, as an instance, it defines the capability *evaluateTask* for the role *Participant* of the Contract Net protocol. More precisely, *evaluateTask* is defined within a silent action and its definition comprises its name plus a list

of inputs and outputs. The tags *capability*, *input*, and *output* are defined in our extension of WS-CDL. It is relevant to observe that each parameter refers to a variable that has been defined in the choreography.

```
1  <choice>
2    <workunit name="informResultWorkUnit"
3      guard="cdl:getVariable('tns:rst', '', '', 'tns:Participant') !=
                                                          'failure' ">
4      <interaction name="informResultInteraction">
5       ...
6      </interaction>
7    </workunit>
8    <interaction name="failureExecuteInteraction">
9      ...
10   </interaction>
11 </choice>
```

Fig. 4. Example of how output parameters can be used in a *choice* operator of a choreography.

Choreographies not only list the set of capabilities that a service should have but they also identify the points of the interaction at which such capabilities are to be used. In particular, the values returned by a call to a capability (as a value of an output parameter) can be used for controlling the execution of the interaction. Figure 4 shows, for example, a piece of a choreography code for the role *Participant*, containing a *choice* operator. The *choice* operator allows two alternative executions: one leading to an inform speech act, the other leading to a failure speech act. The selection of which message will actually be sent is done on the basis of the outcome, previously associated to the variable *rst*, of the capability *executeTask*. Only when such variable has a non-null value the inform will be sent. The guard condition at line 3 in Figure 4 amounts to determine whether the task that the *Participant* has executed has failed.

To complete the example we sketch in Figure 5 a part of the ContractNet protocol as it is represented in our proposal of extension for WS-CDL. In this example we can detect three different capabilities, one for the role of *Initiator* and two for the role *Participant*. Starting from an instance of the type *Task*, the *Participant* must be able to evaluate it by performing the *evaluateTask* capability (lines 4-9), returning an instance of type *Proposal*. Moreover, it must be able to execute the received task (if its proposal is accepted by the *Initiator*) by using the capability *executeTask* (lines 26-31), returning an instance of type *Result*. On the other side, the *Initiator* must have the capability *evaluateProposal*, for choosing a proposal out of those sent by the participants (lines 15-20).

As we have seen in the previous sections, it is possible to start from a representation of this kind for performing the capability test and check if a service can play a given role (e.g. *Initiator*). Moreover, given a similar description it is

```
1  <sequence>
2    <interaction name="callForProposalInteraction"> ...
3    </interaction>
4    <silentAction roleType="Participant">
5      <capability name="evaluateTask">
6        <input> ...  </input>
7        <output> ...  </output>
8      </capability>
9    </silentAction>
10   <choice>
11     <workunit name="proposeWorkUnit" guard=...  >
12       <sequence>
13         <interaction name="proposeInteraction">
14         </interaction>
15         <silentAction roleType="Initiator">
16           <capability name="evaluateProposal">
17             <input> ...  </input>
18             <output> ...  </output>
19           </capability>
20         </silentAction>
21         <choice>
22           <workunit name="acceptProposalWorkUnit" guard=...  >
23             <sequence>
24               <interaction name="proposeInteraction">
25               </interaction>
26               <silentAction roleType="Initiator">
27                    <capability name="executeTask">
28                      <input> ...  </input>
29                      <output> ...  </output>
30                    </capability>
31               </silentAction>
32               <choice>
33                 <workunit name="informResultWorkUnit"
34                   guard=...  >
35                   <interaction name="informResultInteraction">
36                   </interaction>
37                 </workunit>
38                 <interaction name="failureExecuteInteraction">
39                 </interaction>
40               </choice>
41             </sequence>
42           </workunit>
43           <interaction name="rejectProposalInteraction">
44           </interaction>
45         </choice>
46        </sequence>
47     </workunit>
48     <interaction name="evaluateTaskRefuseInteraction">
49     </interaction>
50   </choice>
51 </sequence>
```

Fig. 5. A representation of the FIPA ContractNet Protocol in the extended WS-CDL.

also possible to synthesize the skeleton of a policy, possibly customized w.r.t. the capabilities and the goals of the service that is going to play the role. To this aim, it is necessary to have a translation algorithm for turning the XML-based specification into an equivalent schema expressed in the execution language of interest.

5 Conclusions

This work presents a preliminary study aimed at allowing the use of public choreography specifications for automatically synthesizing executable interaction policies for peers that would like to take part to an interaction but that do not own an appropriate policy themselves. To this purpose, as we have explained also by examples, it is necessary to link the abstract, communicative behavior, expressed at the protocol level, with the internal state of the role player by means of actions that might be non-communicative in nature (capabilities). It is important, in an open framework like the web, to be able to take a decision about the possibility of taking part to a choreography before the interaction begins. This is the reason why we have proposed the introduction of the notion of capability at the level of choreography specification. A capability is the specification of an action in terms of its name, and of its input and output parameters. Given such a description it is possible to apply matching techniques of various kind in order to decide whether a service has the capabilities required for playing a role of interest. In particular, we have proposed the use of semantic matchmaking techniques, such as those developed by WSMO or those proposed by Sycara *et al.* [23], for matching web service descriptions to queries. This can be done because in the application scenarios for which such techniques were developed, services are represented as procedures with a name and with a list of input and output parameters.

We have, furthermore, shown how, given a (possibly) declarative representation of the policy skeletons, obtained from the automatic synthesis process, it is possible to apply further reasoning techniques for customizing the implemented policy to the specific characteristic of the service that will act as a player or for personalizing the interaction according to the user's desires. This goal can, for instance, be achieved by applying techniques like procedural planning, that we have already used in previous work concerning the personalization of the interaction with a web service [4].

Presently, we are working at more thorough formalization of the proposal that will be followed by the implementation of a system that turns a role represented in the proposed extension of WS-CDL into a executable composite service, for instance represented in BPEL. BPEL is just a possibility, actually any programming language by means of which it is possible to develop web services could be used. We plan to face the problem of matchmaking in an incremental way, starting from a simple syntactic matchmaking and passing to forms of semantic matchmaking in subsequent steps.

References

1. M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Specification and verification of agent interactions using social integrity constraints. In W. van der Hoek, A. Lomuscio, E. de Vink, and M. Wooldridge, editors, *Proc. of the Workshop on Logic and Communication in Multi-Agent Systems, LCMAS 2003*, volume 85(2) of *ENTCS*, Eindhoven, the Netherlands, 2003. Elsevier.
2. D. Ancona and V. Mascardi. Coo-BDI: Extending the BDI Model with Cooperativity. In J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Proceedings of the First Declarative Agent Languages and Technologies Workshop (DALT'03), Revised Selected and Invited Papers*, pages 109–134. Springer-Verlag, 2004. LNAI 2990.
3. M. Baldoni, C. Baroglio, A. Martelli, and Patti. Verification of protocol conformance and agent interoperability. In F. Toni and P. Torroni, editors, *Post-Proc. of Sixth International Workshop on Computational Logic in Multi-Agent Systems, CLIMA VI*, volume 3900 of *LNCS State-of-the-Art Survey*, pages 265–283. Springer, 2006.
4. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about interaction protocols for customizing web service selection and composition. *Journal of Logic and Algebraic Programming, special issue on Web Services and Formal Methods*, 2006. To appear.
5. M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: a first step. In M. Bravetti and G. Zavattaro, editors, *Proc. of 2nd Int. Workshop on Web Services and Formal Methods, WS-FM 2005*, volume 3670 of *LNCS*, pages 257–271. Springer, Versailles, France, September, 2005.
6. M. Baldoni, G. Boella, and L. van der Torre. Bridging Agent Theory and Object Orientation: Importing Social Roles in Object Oriented Languages. In R. H. Bordini, M. Dastani, J. Dix, and A. Seghrouchni, editors, *Post-Proc. of the International Workshop on Programming Multi-Agent Systems, ProMAS 2005*, volume 3862 of *Lecture Notes in Computer Science (LNCS)*, pages 57–75. Springer, 2006.
7. M. Baldoni, G. Boella, and L. van der Torre. powerjava: Ontologically Founded Roles in Object Oriented Programming Languages. In D. Ancona and M. Viroli, editors, *Proc. of 21st ACM Symposium on Applied Computing, SAC 2006, Special Track on Object-Oriented Programming Languages and Systems (OOPS 2006)*, Dijon, France, April 2006. ACM.
8. L. Bozzo, V. Mascardi, D. Ancona, and P. Busetta. CooWS: Adaptive BDI agents meet service-oriented computing. In *Proceedings of the Int. Conference on WWW/Internet*, pages 205–209, 2005.
9. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration: a synergic approach for system design. In *Proc. of 4th International Conference on Service Oriented Computing (ICSOC 2005)*, 2005.
10. F. Dignum, editor. *Advances in agent communication languages*, volume 2922 of *LNAI*. Springer-Verlag, 2004.
11. Foundation for Intelligent Physical Agents. http://www.fipa.org.
12. F. Guerin and J. Pitt. Verification and Compliance Testing. In H.P. Huget, editor, *Communication in Multiagent Systems*, volume 2650 of *LNAI*, pages 98–112. Springer, 2003.
13. M. P. Huget and J.L. Koning. Interaction Protocol Engineering. In H.P. Huget, editor, *Communication in Multiagent Systems*, volume 2650 of *LNAI*, pages 179–193. Springer, 2003.

14. Jade. http://jade.cselt.it/.
15. U. Keller, R. Laraand A. Polleres, I. Toma, M. Kifer, and D. Fensel. D5.1 v0.1 wsmo web service discovery. Technical report, WSML deliverable, 2004.
16. A. Mamdani and J. Pitt. Communication protocols in multi-agent systems: A development method and reference architecture. In *Issues in Agent Communication*, volume 1916 of *LNCS*, pages 160–177. Springer, 2000.
17. M. Martelli and V. Mascardi. From UML diagrams to Jess rules: Integrating OO and rule-based languages to specify, implement and execute agents. In F. Buccafurri, editor, *Proceedings of the 8th APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP'03)*, pages 275–286, 2003.
18. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.
19. OASIS. Business process execution language for web services.
20. J. H. Odell, H. Van Dyke Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *Agent-Oriented Software Engineering*, pages 121–140. Springer, 2001. http://www.fipa.org/docs/input/f-in-00077/.
21. L. Padgham and P. Lambrix. Agent capabilities: Extending BDI theory. In *AAAI/IAAI*, pages 68–73, 2000.
22. V. Padmanabhan, G. Governatori, and A. Sattar. Actions made explicit in bdi. In *Advances in Artificial Intelligence*, number 2256 in LNCS, pages 390–401. Springer, 2001.
23. M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *First International Semantic Web Conference*, 2002.
24. Arianna Tocchio S. Costantini. Learning by knowledge exchange in logical agents. In F. Corradini, F. De Paoli, E. Merelli, and A. Omicini, editors, *Proc. of WOA 2005: Dagli oggetti agli agenti, simulazione e analisi formale di sistemi complessi*, Camerino, Italy, november 2005. Pitagora Editrice Bologna.
25. W. M. P. van der Aalst, M. Dumas, A. H. M. ter Hofstede, N. Russell, H. M. W. Verbeek, and P. Wohed. Life after BPEL? In *Proc. of WS-FM'05*, volume 3670 of *LNCS*, pages 35–50. Springer, 2005. Invited speaker.
26. Michael Wooldridge and Simon Parsons. Issues in the design of negotiation protocols for logic-based agent communication languages. In Frank Dignum and Ulises Cortés, editors, *Agent-Mediated Electronic Commerce III, Current Issues in Agent-Based Electronic Commerce Systems (includes revised papers from AMEC 2000 Workshop)*, volume 2003 of *Lecture Notes in Computer Science*. Springer, 2001.
27. WS-CDL. http://www.w3.org/tr/ws-cdl-10/.