Reasoning on choreographies and capability requirements

M. Baldoni*, C. Baroglio, A. Martelli,
V. Patti and C. Schifanella
Dipartimento di Informatica,
Università degli Studi di Torino
c.so Svizzera, 185 — Torino (italy)
E-mail: {baldoni,baroglio,mrt,patti,schi}@di.unito.it
*Corresponding author

Abstract: A typical problem of research in the area of Service-Oriented Systems is the composition of a set of services for executing a complex task. In this paper we face an instance of this problem in which a set of parties (be they peers, agents or systems) have to interact according to a given choreography. In particular, we mean to exploit the role definitions contained in the choreography for realizing interaction policies that can be executed by the involved parties. In this case it is necessary that the choreography captures not only the interactive behavior of the system as a whole but that the role definitions contain also a set of *requirements* of capabilities that the parties should exhibit, where by the term "capability" we mean the skill of doing something or of making some condition become true. Such capabilities have the twofold aim of connecting the interactive behavior to be shown by the role-player to its internal state and of making the policy executable.

Keywords: choreographies and interaction protocols; capabilities; reasoning; semantic matchmaking.

Reference to this paper should be made as follows: Baldoni, M., Baroglio, C., Martelli, A., Patti, V. and Schifanella, C. (2007) 'Reasoning on choreographies and capability requirements', Int. J. High Performance Computing and Networking, Vol. ??, Nos. ???, pp.??–??.

1 INTRODUCTION

In various application contexts there is a growing need of being able to compose sets of heterogeneous and independent entities with the general aim of executing a task, whose complexity cannot be handled by a single component. In this framework, it is mandatory to find a flexible way for gluing components, a highly complex problem which encompasses various skills, pursued by different scientific disciplines (describing the goal to be achieved, describing the solution in terms of involved entities and of their interactions, identifying within the pool of available entities those which can solve subproblems, etc.). One solution, which is being explored both in the Web services research area and in the multi-agent systems (MAS) research area, is to compose entities on the basis of "dialogue". In the case of Web services, ad hoc languages (e.g. WS-BPEL by OASIS), have been proposed for building executable composite services based on a description of the flow of information, in terms of the messages that are exchanged by the composed services (e.g. Srivastava and Koehler (2003); Giordano and Martelli (2006)). On the other hand, the problem of aggregating communicating agents into (open) societies is well-known, and 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. a global communication protocol, see Dignum (2004).

As recently observed by van der Aalst et al. (2005) and Baldoni et al. (September, 2005), the MAS and Web services research areas show convergences in the approach by which systems of agents, on a side, and composite services, on the other, are designed, implemented and verified. In both cases it is in fact possible to distinguish two levels: a global and abstract view of the system as a whole, which is independent from the specific agents/peers which will take part to the interaction (the design of the system), and the implementation of the system in which the specific entities that will interact are identified. In the case of MASs, as

Copyright © 200x Inderscience Enterprises Ltd.

Huget and Koning (2003) point out, the design level often corresponds to a shared interaction protocol. In a services oriented scenario this level corresponds to a general *choreography* of the system, in which a set of roles are captured together with their interactions by means of ad hoc representation languages (e.g. WS-CDL). On the other hand, the interactive behavior of a specific party/peer involved in the interaction is to be given in some executable language (e.g. WS-BPEL).

In this proposal, we will consider choreographies as *shared knowledge* among the parties, which will also share a common communication language. We will, then, refer to choreographies as *public* and non-executable specifications. The same assumption *cannot* be made for what concerns the interactive behavior of specific parties (be they service oriented entities, peers or agents). The actual behavior of a peer will be considered as being *private*, i.e., non-inspectable from outside. Nevertheless, if we are interested in coordinating the interaction of a set of parties as specified by a given choreography, we need them to play specific roles. For instance, if an entity publishes the fact that it acts according to the role "seller" of a public choreography, for interacting with it, it will be necessary to play the role "customer" of the specified choreography. For playing a role described in a choreography a peer must own a *policy* that is conformant to that role. The so-called *conformance checking* test, of a policy w.r.t. a choreography, guarantees that the owner of the policy will be able to interoperate with the other role players within the choreography, the interesting point being that such verifications can be performed locally, player by player, w.r.t. the public specification.

Let us, nevertheless, focus on the case when a peer *does not* have any conformant policy for playing a certain role (that is contained in some choreography specification). Anyway, it needs to take part to an interaction, that is ruled by *that* choreography. A possible solution is to define a method for generating, in an automatic way, a conformant policy from the role specification. The role specification, in fact, contains all the necessary information about what sending/receiving to/from which peer at which moment. An example that will be further described in Section 2.1, is this: consider an e-travel company that allows users to buy flight tickets on-line. The company will need to interact with various flight companies. A viable solution, alternative to forcing each flight company to develop its own conformant policy, is to produce such policies automatically, when the need arises, exploiting the choreography description as a guideline.

As a first approximation, we can, then, think of translating the role, as expressed in the specification language, in a policy (at least into a policy *skeleton*) given in an executable language. This is, however, not sufficient. In fact, it is necessary to bind the interactive (observable) behavior that is encoded by the role specification with the internal (unobservable) behavior that the peer must anyway have and with its internal state. For instance, the peer must have some means for retrieving or building the information that it sends. This might be done in several ways, e.g. by querying a local data base or by querying another peer. The way in which this operation is performed is not relevant, the important point is to be sure that the peer has the *skill* of preparing the necessary information. For completing the construction of the policy, it is necessary to have a means for checking whether the peer can actually play the policy, in other words, if it has the *required capabilities*. This can only be done if we have a specification of which capabilities are required in the choreography itself. The capability verification can be accomplished role by role by the specific party willing to take part to the interaction.

Notice that capability verification is not just a question of matchmaking. In fact, matchmaking is a *local* kind of verification, limited to a single software component, while capabilities must fit in a predefined interaction schema (given by the choreography), and their selection must not compromise, on the one hand, the interoperability of the involved parties, and on the other, the executability of the interaction itself. Working at the level of matchmaking only, as already observed in works about software component retrieval by Zaremski and Wing (1997), does not necessarily preserve these properties.

Starting from this result, the proposal of this article is to *combine* the matchmaking process with a reasoning process that works at the higher level of the *specification of interaction*. In particular, we formally express the relations between the matchmaking process, applied to the skills that the entity has againt the skills required by the choreography, with the global behaviour of a peer that wants to play a role, with the purpose of achieving a goal of interest. To do so, we will introduce the concepts of "capability requirement" and of "capability" respectively in the specifications of the global choreography and of the specific entities, that should interact according to it, in such a way that capabilities can be accounted for during automatic reasoning processes. We will see how such processes allow to dynamically build customized policies for the interacting entities. The reasoning techniques that we take into account are goal-driven, and derive from the experience of the authors in the field of reasoning about actions and change, Baldoni et al. (2007).

The article is organized as follows. Section 2 defines the setting of the work. Moreover, it reports two examples: the first is a simple representation of the well-known FIPA Contract Net protocol, the second (that we call the *flight company example*) is more complex and concerns a realistic application domain. We will use contract net in Section 5, where a possible extension of the choreography language WS-CDL is presented. The flight company example is, instead, used along Section 4, where our approach is described and some reasoning techniques are introduced. This section is preceded by Section 3, where the problem of checking capabilities is introduced. The article ends with conclusions and perspectives on future works.

2 INTRODUCING CAPABILITIES

In the introduction we have sketched a scenario in which a system of interacting parties is described by a choreography at an abstract level, in which specific peers do not yet appear. A choreography is a schema, a set of rules according to which interaction should occur. In this context the problem of verifying whether a party's interaction policy respects a given role specification is extremely relevant. This problem is known as *conformance* test (see the work by Alberti et al. (2003), Guerin and Pitt (2003), and Baldoni et al. (2006a)). As already proposed by Baldoni et al. (2006a, September, 2005); Busi et al. (2005), the conformance test can be a means for guaranteeing a priori the interoperability of a set of peers, each playing one of the roles described by a given choreography.

In this work we will focus on the case in which a peer is interested in playing a role in an interaction ruled by a choreography, but it does not have a conformant policy. Thus, in order to allow for the interaction to occur, 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 by Ancona and Mascardi (2004). 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 Web services technologies, leading Bozzo et al. (2005) to the development of CooWS agents. Another recent work in this line of research is the one by S. Costantini (2005). 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. Agents have reasoning techniques that enable them to evaluate how useful the new knowledge is. Unfortunately, these techniques 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 entities in a service oriented scenario 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 entity 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 Martelli and Mascardi (2003). 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 as a basis for policy skeletons, we need to specify some more information, i.e. actions that allow the access to the peer's internal state. Throughout this work we will refer to such actions as *capability requirements*. For what concerns the specific peers, it is necessary to have a representation of their capabilities. In general in an open environment a peer can be any kind of entity (an agent, a service or some other kind of system), due to this heterogeneity we will adopt a logic-based representation that abstracts away from the details of the specification. In the particular case of Web services, we could annotate each service by means of a description of its capabilities expressed in terms of some standard language for semantic Web services, e.g. OWL-S (OWL-S Coalition) or WSMO (Keller et al. (2004)). Indeed, all these proposals allow for the representation of inputs, outputs, *preconditions*, and *effects*. This issue is discussed in Section 3.

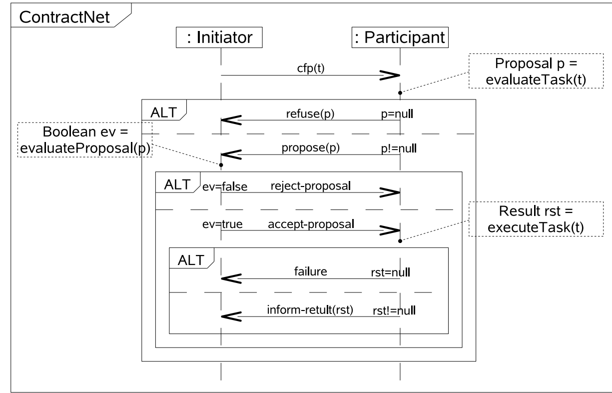

Figure 1: The FIPA ContractNet Protocol, represented by means of UML sequence diagrams, and enriched with capability specifications.

For better explaining our ideas, let us consider, as a simple choreography example, the well-known FIPA ContractNet Protocol, by the Foundation for Intelligent Physical Agents, 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 (dotted rectangles represent capabilities).

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 ac-

cept (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 capability requirements, 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 *evaluateTask* 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, it can be meaningful to specify a capability by its name, a description of its inputs and a description of its outputs (see fig 1). However 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 (or goals). We will come back to this issue shortly.

2.1 The flight company example

Let us now go back to the scenario briefly mentioned in Section 1 and introduce an example choreography (enriched with capability requirements) in the e-travelling application domain, that calls for the interaction of two peers. We will also show a few situations, that can be tackled by reasoning on this schema and on the actual capabilities that a candidate role player has.

Let us, then, consider an e-travel organization company, of the kind of well-known Expedia (URL: http://www.expedia.com), focussing, in particular, on the issue of having some seats reserved on some flight. A company of this kind offers a sort of portal, through which a user has access to a wide variety of flight (and other kinds of) companies, retrieves information about, e.g., flights, compares the various offers and, in the end, can buy tickets. In a traditional view, each flight company, which has its own information system, must adhere to a common interface for allowing information about its products to be shown on demand and, eventually, bought. In this way, only companies which explicitly agree on the use of the common interface can be advertised through the portal.

The adoption of choreographies, annotated with capability requirements, would allow to smoothen this limit and allow also flight companies, which do not explicitly take part to the "portal association" to be enquired about flight offers and to sell their products through the portal. In the remainder of the work we will call the extended choreographies, which account not only for the interaction among the parties but also for capability requirements, *enriched choreographies*.

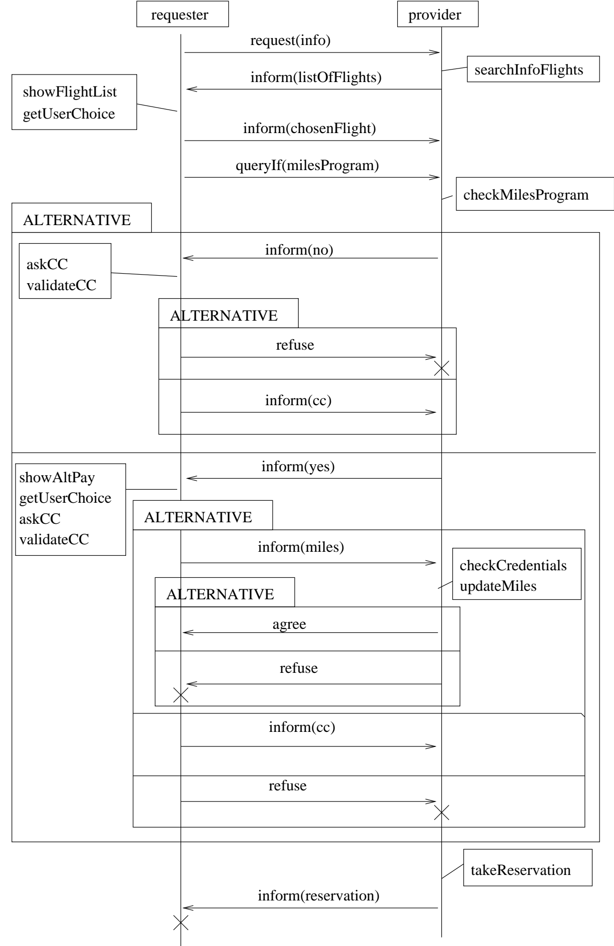

Figure 2: E-flight reservation: an example of choreography, annotated with capability requirements. For graphical reasons we denote each required capability simply by a label. Their full representation is in the text.

Figure 2 reports an enriched choreography, in the form of a sequence diagram. This choreography is an example of the interaction that one might reasonably expect to occur between the e-travel company and each involved flight company, when a user wants to buy a ticket. If the choreography is made public, any flight company can download the role *provider* and reason about it to decide (1) if it can play it and (2) if it can play it *as it likes to*. The role *requester* is supposed to be played by the e-travel company, instead. We could also think to an opposite scenario in which a similar choreography is published by a flight company, rather than by an e-travel agency, stating the interaction protocol required by an interlocutor to deal with it. In this case the difference would reside in the fact that the validation of credit card information should be performed by the provider and not by the requester.

The role *provider* includes five capability requirements:

1. *listOfFlights = searchInfoFlight(info)*: the capability to search for the requested flights for some user;
2. *checkMilesProgram()*: the capability to check if a miles program can be applied, it returns a boolean value and can be used in conditions);
3. *checkCredentials(miles, chosenFlight)*: the capability to check the credentials of a user who would like to pay by miles; it returns a boolean value that can be used in conditions;
4. *updateMiles(miles)*: the capability to update the miles amount when this option of payment is chosen;
5. *reservation = takeReservation(info, chosenFlight)*: the capability to reserve the chosen seats for the user.

Figure 2 shows also the capabilities required to the *requester*:

1. *showFlightList(listOfFlights)*: showing a list of flights;
2. *chosenFlight = getUserChoice()*: getting the user's choice;
3. *cc = askCCinfo()*: asking for information about a credit card;
4. *validateCC(cc)*: validating a credit card, it returns a boolean that can be used in conditions;
5. *showAltPay()*: showing alternative forms of payment;
6. *choice = getUserPayChoice()*: getting the user's the desired form of payment, it returns a value to be used internally by the requester.

By taking a look at the choreography definition, it is easy to see that some capabilities are necessary in all alternative execution paths, while some others are used only in some of the paths. One of them is *checkCredentials*, another one, on the *requester*'s side is *showAltPay*. Indeed, a flight company which does not offer a miles program is very unlikely to have a capability that matches *checkCredentials* because it would be meaningless. A question that one would like to answer is whether this company can anyway succeed in selling tickets by playing the role. In our example this is possible, given that the company has capabilities that match other requirements. In fact, there is an execution path leading to success that does not require this capability.

Notice that it is not possible to drive the same conclusion in the case of a peer willing to play the role *requester* and not having a capability matching with *showAltPay*. In fact, the choice of taking the path on which that capability is required is not up to the requester but to the provider.

Another problem that can easily be solved by using enriched choreographies concerns the enactment of privacy policies. This case is a variation of the previous one, in which a capability is available or unavailable depending on a decision of the candidate player. To better explain this point let us suppose that the flight company has a capability matching with *checkCredentials* but its use is constrained by a precondition: the capability can be used only if the interlocutor is member of the "Super Alliance" group. Since the interlocutor's identity is known, it becomes possible to decide, before actually taking part to the interaction, if it is the case of using this capability or not. So if the *requester* is not member of the alliance, the paths containing this requirement will be disabled. The flight company will anyway be able to complete with success the selling process by using the credit card payment option.

A third case that we consider is that in which the peer has various implementations of a same requirement, and the reasoning process selects the properest one by reasoning on the role definition and on the previously selected capabilities. To this aim, let us suppose that the capability matching with *searchInfoFlight* has as an effect the prebooking of seats on the list of flights that are returned as an answer. This is done in order to allow the user to make its choice being sure that a seat, that has been declared available at that stage, will still be available when the transaction will be concluded with the real reservation. Let us also suppose that the flight company service has two implementations that match *takeReservation*. One is thought for being used in case of prebooking, the other one in case no prebooking has been done. If preconditions and effects of each capability are specified, the reasoning process finds out automatically that only the first capability is actually applicable, because at the point when it is requested the fact "prebooking" will have been added to the state of execution.

If preconditions and effects were anyway represented, it would also be possible to apply forms of *goal-driven* reasoning. For instance, let us suppose that the flight company, willing to play the role of the *provider*, does not desire to use its miles program when interacting with a mediator like our e-travel agency, although it would have the necessary capabilities. Checking if it can take part to the interaction avoiding, at the same time, to be paied in miles, can be translated in the problem of finding a plan, seen as a subset of the possible executions of the role specification, that allows for it to reach a state where a ticket has been sold and no miles have been used. If, for instance, the capability matching *updateMiles* has the effect of stating that miles have been used, the paths including this capability will be cut out. In the case of the specification reported in Figure 2 the flight company will actually find out that it is necessary to cut the whole subtree starting with the answer "yes" to *queryIf(milesProgram)*.

2.2 Related works

The term "capability" has been used by Padgham and Lambrix (2000) (a work inspired by JACK, by Busetta et al. (1999) and extended by Padmanabhan et al. (2001)), 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 capabilities. Such requirements specify "actions" that peers, willing to play specific roles in the interaction schema, should exhibit. In order for a peer to play a role, some verification 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, by CSELT, 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 does nothing. The task of the programmer is to implement the specified methods, whose name and signature is, however, fixed in the partial policy. For example, the class *ContractNetResponder* supplies the policy implementation for the role *partecipant* of the contract net protocol shown in Figure 1. That class provides, among the others, the methods *handleCfp* and *handleAcceptProposal*. The default implementation for such methods does nothing and returns null, programmers have to override it to react to these events. These methods represent the capabilities *evaluatedTask* and *exacuteTask* in Figure 1. It is worth noting that it is not mandatory to handle all events but only the events the programmer believes relevant in his implementation.

Another example is powerJava, Baldoni et al. (2006c,b), 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.

Checking whether a peer has the capability corresponding to a requirement 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, focusing 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.

3 CHECKING CAPABILITIES

The idea that we mean to explore is to exploit a description of the required capabilities defined at the level of the choreography (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 (see Section 2.2), for instance, there is no real capability test because policies already supply empty methods corresponding to the capabilities, which the programmer redefines. In powerJava the check is performed by the compiler, which verifies the implementation of a given interface representing the requirements. For further details see the work by Baldoni et al. (2006c), 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*, like in classical programming languages and in powerJava, as well 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. In fact, semantic matchmaking supports 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 peer 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 peer 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.

Semantic matchmaking has been thoroughly studied and formalized also in the Semantic Web community, in particular in the context of the DAML-S (Paolucci et al. (2002)) and WSMO initiatives (Keller et al. (2004)). Paolucci et al. (2002) propose a form of semantic matchmaking concerning the input and output parameters. The ontological reasoning is applied to the parameters of a semantic Web service, which are compared to a query. The limit of this technique is that it is not possible to perform the search on the basis of a goal to achieve.

A different approach is taken in the WSMO initiative

by Keller et al. (2004), where services are also described on the base of their *assumptions* (another term for "preconditions", a word which in the WSMO framework has a different meaning, concerning the structure of the request) and *effects*. Indeed, even though working on the inputs and the outputs of actions has the advantage of allowing static checks, it is limiting for many reasons. A simple and intuitive example, taken from the literature about software components (Zaremski and Wing (1997)), is this. Consider two services which respectively perform addition and subtraction of two integer numbers. Their signatures, in terms of inputs and outputs, will perfectly match, however, their semantics is totally different and can only be captured in terms of a *specification* of what the service actually does. This specification can be given as a logical formula stating what must hold because the service can be executed (the *precondition*), and what will hold after the execution of the service (the *postcondition* or *effect*). Either of the two can be empty, in this case they will be assumed as corresponding to the constant *true*.

Having preconditions and effects enables further forms of verification, e.g., verification based on *goal-driven* forms of reasoning. When we deal with entities which interact with one another or with some human user, it is sometimes important that part of the information that is exchanged is protected, so it can be disclosed only if some conditions hold. Moreover, the entity could have (or have adopted) some goals of its own, besides those concerning the interaction for which the role is taken into account. These goals can be seen as constraints which must hold all along the possible executions, which could prevent the use of some capabilities, force the use of other capabilities or prevent the disclosure of some information. These goals are internal to the entity and they require forms of reasoning about the execution of policies. To this aim, describing the entity's functionalities by means of inputs and outputs is not enough: it is necessary to introduce next (or in place of) them a representation of the *preconditions* and *effects* of functionalities. In other words, it is useful to model capabilities as *actions*.

Enacting a form of reasoning that involves the use of goals requires the simulation of the execution of the received policy, therefore, it is necessary to introduce a notion of *state*. Last but not least, it is worth to mention that the matchmaking process and the reasoning about goals process can be fruitfully combined. In fact, when an entity has a set of alternative capabilities which match a capability requirement, it is necessary to make a choice. Nevertheless, the chosen capability not necessarily will lead to the goal satisfaction. In these cases forms of backtracking are to be considered. For instance, a flight company could have a miles program and, at the same time, it could decide to accept only payments by credit card. In this situation, the company is very likely to have a capability matching with *updateMiles*. Using this capability, however, would not allow for it to reach the goal. If the path that foresees the use of this capability is being explored, then, it will be necessary to backtrack and see if there are other viable solutions.

Moreover, the choice of a capability could prevent the application of another capability. In our e-travelling example, using a capability exactly matching with *searchInfoFlight* causes *prebooking* as an effect. Prebooking prevents the choice of one of the two implementations of *takeReservation* that we mentioned. What if this implementation were the only one owned by the candidate player?

4 REASONING ON CAPABILITIES

As we have seen in the previous section with the help of little examples, the answers of a matchmaker have a *local scope* because they take into account only a capability requirement and the capabilities of an entity. It does not consider the context given by the choreography. In order to perform even the choice of one capability to be used in an interaction ruled by a given protocol, it is important to have the ability of verifying the consequences of our choice. In this case, (semantic) matchmaking is to be used for associating to each capability requirement a set of capabilities that can possibly match that requirement. The choice of the specific capabilities to use is, however, demanded to a goal-driven reasoning process.

Such kind of reasoning could be done by describing the ideal complete policy for an entity aiming at implementing a given role in a declarative language that supports *a-priori* reasoning on the policy executions. In fact, if a *declarative representation* of the complete policy were given, it would be possible to perform a rational inspection of the policy, in which the execution is simulated.

A choreography defines a global execution context, in which the various capability requirements are immersed, while capabilities are single software components: intuitively, the selection of a capability should preserve the properties of the interaction encoded by the choreography specification or, at least, it should preserve some properties of interest to the candidate player. In other words, a peer could be interested in playing a given role of some choreography because it is sure that it will allow it to achieve its purpose (e.g. selling a ticket); moreover, it could verify that the interaction can occur in a way that satisfies some properties of its own interest (e.g. the payment will not be done in miles). After the decision to play the role has been taken, the player should be guaranteed that the substitution of capabilities to capability requirements will preserve its goal and the properties that it checked.

In order to make our presentation formal, though keeping it simple, let us show how it is possible to represent both *policies* and *roles* by means of a *declarative language*, interpreting capabilities and capability requirements as *actions*, and interpreting reasoning about capabilities and capability requirements as *reasoning about actions*. We will use the language DyLOG, extended with a communication kit, and already used for customizing Web service composition, adopting an *English*-like notation for the sake of readability. This language is fully described by Baldoni

et al. (2004, 2007), the interested reader can refer to that publication for having details about it. Since our focus is to study how refine the matchmaking process, taking into account also the choreography in which the selected capabilities are to be used, we will assume that the sets of terms used for representing speech acts and capabilities in the choreography and in the peer descriptions are the same.

DyLOG has been developed as a language for programming agents and is based on a logical theory for reasoning about actions and change in a modal logic programming setting. An agent's behavior is described in a nondeterministic way by giving the set of actions that it can perform. Each action can have preconditions to its application (that decide if the action is executable) and effects due to its application. Moreover, effects can be subject to further conditions in order to become true. For instance, the executability precondition to the action "paying by credit card" is that I hold a valid credit card. A conditional effect of this action could be "to be notified by SMS about payments". This effect will become true only if I subscribed the service (precondition to the effect).

Given this view of actions, we can think to the problem of reasoning as the act of building or of traversing a sequence of transitions between *states*. Technically speaking, a state is a set of *fluents*, i.e., properties whose truth value can change over time. In our application domain such properties encode the information that flows during the execution of a policy: for instance, if a requester communicates to pay by miles, this information will be included in the state of the provider as a fluent.

In general, we cannot assume that the value of each fluent in a state is known, so we want to have both the possibility of representing that some fluents are unknown and the ability of reasoning about the execution of actions on incomplete states. To explicitly represent the unknown value of some fluents, we introduced an epistemic operator $\mathcal{B}$, to represent the beliefs an entity has about the world: $\mathcal{B}f$ means that the fluent f is known to be true, $\mathcal{B}\neg f$ means that the fluent f is known to be false. A fluent f is undefined when both $\neg\mathcal{B}f$ and $\neg\mathcal{B}\neg f$ hold ($\neg\mathcal{B}f \wedge \neg\mathcal{B}\neg f$). For expressing that a fluent f is undefined, we write $u(f)$. Thus each fluent in a state can have one of the three values: *true*, *false* or *unknown*.

Actually, in the *implementation* of DyLOG (and, for the sake of simplicity, also in the following description) we do not explicitly use the epistemic operator $\mathcal{B}$ with the following assumption. If a fluent f (or its negation $\neg f$) is present in a state, it is intended to be believed and to be unknown otherwise.

Complex behaviors can be specified by means of *procedures*, Prolog-like clauses built upon other actions. The interaction between the peers will be defined in terms of *speech acts*, *get-message actions*, and *procedures*.

A *speech act* is an atomic action of form *performative(sender, receiver, content)*, where *performative* is a kind of speech act, *sender* and *receiver* are the interacting peers, while *content* is the piece of information that is passed by its execution. The set of all performatives is supposed to be shared by the two parties. Being an action, a speech act will have its preconditions (that decide if the action is executable) and effects. A communicative action usually can modify not only the beliefs of the executor about the world but also its beliefs about the interlocutor's mental state (see Baldoni et al. (2007)). For instance, the act of sending a piece of information, *inform(me, other, fact)*, will modify my mental state by adding the belief that my counterpart now knows *fact*. As a difference in our model non communicative actions modifies only the executor's believes about the world.

Besides speech acts, it is necessary to specify also actions that allow to represent the reception of information. This is done by means of *get-message* actions. Notice that the range of possible answers is supposed to be finite, in the sense that the interlocutor is supposed to use a performative out of a finite set to produce its answer.

Capabilities and *capability requirements* are both represented as atomic actions. They could be both communicative and non communicative actions.

Each choreography is made of a set of interacting roles. It can be described as a set of subjective views of the interaction that is encoded, each corresponding to one of the roles. We will call the implementation of each role a *policy*, where the point of view is that of the role player. In our formal setting, the interactive behavior of both roles and policies can be represented by DyLOG procedures. Intuitively we will describe a *role* as a procedure that combines speech acts, get-message acts, *capability requirements* and procedure calls, and a *policy* as a procedure combining speech acts, get-message acts, *capabilities* and procedure calls.

A *role* in a choreography can, therefore, be specified as a quadruple of the form $R_d = \langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{CR}, \mathcal{P} \rangle$, where:

1. $\mathcal{S_A}$ is a set of *speech acts*, they are represented as:

   $$\begin{array}{l}\mathsf{performative}(sender, receiver, l)\ \textbf{causes}\ E_1\\ \qquad \textbf{if}\ Cond_i\\ \ldots\\ \mathsf{performative}(sender, receiver, l)\ \textbf{causes}\ E_n\\ \qquad \textbf{if}\ Cond_n\\ \mathsf{performative}(sender, receiver, l)\ \textbf{possible if}\ P\end{array}$$

   where E_i, P_E and $Cond_i$ are respectively: fluents that can be obtained as effects of the speech act, and conjunctions of fluents that represent the set of conditions that further bias each effect and the preconditions to the executability of the speech act. P is the precondition to the execution of the performative.

2. $\mathcal{G_A}$ is a set of *get-message actions*, they are represented as:

   $$\mathsf{receive_act}(receiver, sender, l)\ \textbf{receives}\ \mathcal{I}$$

   where $\mathcal{I}$ is a set of alternative speech act, that can be received by the executor of receive_act.

3. $\mathcal{CR}$ is a set of capability requirements, they are modelled as atomic actions and are represented as:

   c **causes** E_1 **if** $Cond_1$
   ...
   c **causes** E_m **if** $Cond_m$
   c **possible if** P

   where c is the name of the required capability and the semantics of the clauses is the same as above.

4. $\mathcal{P}$ is a procedure that encodes the behavior for the role, it is represented as a collection of clauses of the kind:

   $$p_0 \textbf{ is } p_1, \ldots, p_n \ (n \geq 0)$$

   where p_0 is the name of the procedure and p_i, $i = 1, \ldots, n$, is either an atomic action, a *get-message* action, a test action, or a procedure name (i.e., a procedure call). Procedures can be recursive and are executed in a goal-directed way, similarly to standard logic programs, and their definitions can be nondeterministic as in Prolog.

Let us call $\mathcal{C}$ the set of capabilities of a peer, aiming at playing a role, then, we define *policy* for the peer the quadruple $P_d = \langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{C}, \mathcal{P} \rangle$, where capabilities are modelled as atomic actions.

As mentioned above, the problem the we face is to *build* a policy description, for a peer, *in an automatic way*, starting from a *role* description and substituting the capability requirements with a proper set of the capabilities that the peer has. In particular, we assume that the communication language used in the role description is the same as the one used by the peer (i.e. speech acts and get-message actions are described by the same DyLOG clauses).

The procedure in P_d will be obtained by applying a proper transformation of the procedures in $\mathcal{P} \in R_d$. This transformation is achieved by applying a substitution of the capability requirements, replacing them by some of the peer capabilities. Let us call θ a mapping between the set $\mathcal{CR}$ of a role description $R_d = \langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{CR}, \mathcal{P} \rangle$ and the set $\mathcal{C}$ of capabilities, contained in the peer description (i.e., a function $\theta : \mathcal{CR} \longrightarrow \mathcal{C}$). We will denote $\theta = [\mathcal{C}/\mathcal{CR}]$, following the common notation for substitutions. The DyLOG policy, synthesized for the peer, will be a quadruple $P_d = \langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{C}, \mathcal{P}\theta \rangle$, where $\mathcal{P}\theta$ is obtained by substituting capability requirements with capabilitites, following the mapping given by θ.

The construction of the policy for the peer is the result of a *reasoning process*. Generally speaking, in DyLOG, given a role description, a state description and a goal of interest, it is possible to perform a form of reasoning known as *temporal projection*, by means of *existential* queries of the form:

$$Fs \ \textbf{after} \ p$$

where p is a policy name and Fs is a conjunction of fluents. Checking if a formula of this kind holds corresponds to answering the query "Is there an execution trace of p that leads to a state in which Fs is true?". By execution trace we mean a sequence of atomic actions (speech acts and capability requirements), such sequence is a plan to bring about Fs. This plan can be *conditional* because whenever a *get-message* action is involved none of the possible answers from the interlocutor can be excluded. In other words, we will have a different execution branch for every option.

Now, let us consider a role description $R_d = \langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{CR}, \mathcal{P} \rangle$. $\mathcal{P}$ is a set of procedures, therefore we can think of applying temporal projection to find out if there is an execution trace of one of its policies, that makes a goal of interest become true. Let us, then, consider a procedure p belonging to $\mathcal{P}$, and denote by G the DyLOG query: Fs **after** p, where Fs is the set of fluents that we want to be true after the execution of p. Given a state S_0, containing all the fluents that we know as being true in the beginning, we will denote the fact that G is successful in R_d by:

$$(\langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{CR}, \mathcal{P} \rangle, S_0) \vdash G$$

The execution of the above query returns as a side-effect an *execution trace* σ of p. The execution trace σ can either be *linear*, i.e. a terminating sequence $a_1, \ldots, a_n$ of atomic actions, or it can be *conditional*, when the procedure contains get-message actions. In this case it will have the form of a tree with alternative branches. An example of conditional plan is highlighted in Figure 3. Circles represent speech acts, while squares represent capability requirements; *rec* and *send* are used to express the fact that the communication act, passed as their argument, is received or sent by the provider. The conditional plan is an *and-or* tree where the *or*-branches represent alternative speech acts that the entity (the provider in the example) can send. The *and*-branches represent get-message actions, and include all the possible incoming messages which the entity must be able to handle, because it cannot foresee what it will receive. For details see Baldoni et al. (2004, 2007). The part of *and-or* tree highlighted by the dashed line is a *conditional plan*. There are other two conditional plans: one containing the traces (3), (4) and (5), the other containing (3), (4) and (6). With reference to the flight company example, Figure 3 reports the case when the company does not have a miles program. The interaction can be completed with the sale of a ticket only if the requester accepts to pay by credit card. Otherwise, it will send a refusal. The two alternative branches are both to be included in the plan because the choice of the requester cannot be foreseen at planning time.

Intuitively, it is possible to verify, by reasoning about the choreography, if the role allows for an execution after which a condition of interest holds.

Analogously, given a policy description for a peer $P_d = \langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{C}, \mathcal{P} \rangle$, a $G = Fs$ **after** p, and an initial state S_0 we can verify if G is successful in P_d by:

$$(\langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{C}, \mathcal{P} \rangle, S_0) \vdash G$$

Intuitively, it is possible to verify, by reasoning about the

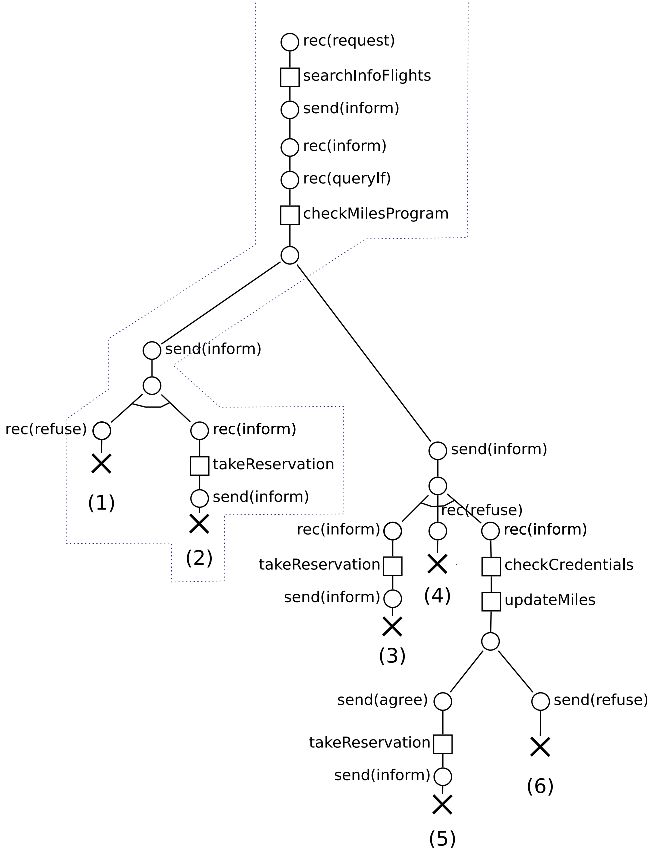

Figure 3: Execution traces for the role *provider* of the flight company example (Section 2.1).

peer description, if its policy allows for an execution, after which a condition of interest holds.

Now we are in position to formally express the relation between the mapping θ, that matches capability requirements with capabilities, and the behavior of a peer that wants to play a role achieving a goal. Generally, the matchmaking process will result in a set of alternative θ_i because each capability requirements will have a set of matching capabilities. The selected θ not only must satisfy the matching rules but it must also guarantee the achievement of the goal. In other words, those goals that can be achieved by reasoning on the role specification must hold also after the substitution. Then, the following implication should hold:

$$\begin{aligned}&\exists\theta = [\mathcal{C}/\mathcal{CR}]\ s.t.\\&(\langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{CR}, \mathcal{P}\rangle, S_0) \vdash G \Rightarrow\\&\qquad(\langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{C}, \mathcal{P}\theta\rangle, S_0) \vdash G\end{aligned}$$

This implication means that the choice of θ *does not* depend only on the *local* aspects taken into account by the matchmaking process, instead, it also depends on the *overall structure* of the interaction process, encoded by the choreography. Therefore, our proposal is to integrate the matching process with a reasoning process that takes into account the procedure in which the capabilities will be immerged.

As an example, let us consider the *provider* role of Figure 2. One goal that we could wish to prove is that, at the end of the execution, some ticket will be sold. Reasoning at the level of the role, this can be done using all the five capability requirements contained in the role description. Let us consider a candidate player for the role, that has two capabilities that match with *takeReservation*: one requiring as a precondition that some prebooking has been done, the other requiring that no prebooking has been done. In both cases the effect is to sell tickets. The choice of which of the two is to be used, will depend on the choice taken for other capability requirements. If, for instance, the capability chosen for *searchInfoFlight* has prebooking as an effect, only the first of the two capabilities can be selected. The six possible execution traces are reported in Figure 3.

In the above case, the substitution θ involves all the capability requirements contained in the role description (the function θ is *total*). In some situations, it is interesting to restrict the attention to a subset of capability requirements, focussing on a slightly different problem and formulate the following relation:

$$\begin{aligned}&\exists\sigma, \theta' = [\mathcal{C}/\mathcal{CR}_\sigma],\ \mathcal{CR}_\sigma \subseteq \mathcal{CR}\ s.t.\\&(\langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{CR}, \mathcal{P}\rangle, S_0) \vdash G \text{ w.a. } \sigma \Rightarrow\\&\qquad(\langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{C}, \mathcal{P}\theta'\rangle, S_0) \vdash G \text{ w.a. } \sigma\theta'\end{aligned}$$

where G stands for Fs **after** p, σ is an execution trace of p which makes the goal true when reasoning at the level of the choreography specification, and $\theta' : \mathcal{CR}_\sigma \longrightarrow \mathcal{C}$ where $\mathcal{CR}_\sigma \subseteq \mathcal{CR}$, $\mathcal{CR}_\sigma = \{c \in \mathcal{CR} \mid c \text{ occurs in } \sigma\}$. Finally, $\sigma\theta'$ is obtained by applying the substitution θ' to σ. Differently than the previous one, the above relation states that we are interested in a substitution θ' that involves only the capability requirements contained in σ, which is, therefore, used to select the requirements to be matched.

As a simple example of this case, let us suppose that a peer wishes to play the role of "provider" with the aim of selling some flight tickets, and that it can only handle payments by credit card. This goal can actually be seen as a constraint on the possible interactions. If the policy implementing the complete role allows this form of payment among the others (see Figure 3, traces (1) and (2)) the candidate provider will be very likely to continue the interaction because some of the alternatives allow reaching the goal of selling tickets in a way that can be handled. It can, then, customize the policy by deleting the undesired path. If some of the capabilities are to be used *only* along the discarded execution path (e.g. *checkCredentials*), it is not necessary for the candidate customer to have them.

The set of capabilities of a peer could be not completely predefined but it could depend 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. In this case, a third kind of relation emerges (a variant of the previous one):

$$\begin{aligned}&\exists\sigma, \theta'' = [\mathcal{C}'/\mathcal{CR}_\sigma],\ \mathcal{C}' \subseteq \mathcal{C},\ \mathcal{CR}_\sigma \subseteq \mathcal{CR}\ s.t.\\&(\langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{CR}, \mathcal{P}\rangle, S_0) \vdash G \text{ w.a. } \sigma \Rightarrow\\&\qquad(\langle \mathcal{S_A}, \mathcal{G_A}, \mathcal{C}', \mathcal{P}\theta''\rangle, S_0) \vdash G \text{ w.a. } \sigma\theta''\end{aligned}$$

where σ is an execution trace of p which makes the goal true when reasoning at the level of the choreography specification, $\mathcal{C}'$ is a subset of the capabilities of the peer, those which I am enabled to use in that context, and $\theta'' : \mathcal{CR}_\sigma \longrightarrow \mathcal{C}'$. One example that we have seen was that of the flight company, that has a miles program but is not willing to accept payments in miles when dealing with mediators. The difference with the previous case is that now the capability is inhibited, while in the previous case it was not available. In this perspective, our work is related to researches that concern the notion of *opportunity*, proposed by Padmanabhan et al. (2001), in connection with the concept of capability (but with the meaning proposed by Padgham and Lambrix (2000), see Section 1).

5 USING CAPABILITY REQUIREMENTS

The most important formalism used to represent interaction protocols in the WS domain is WS-CDL (Web Services Choreography Description Language) WS-CDL: 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 first step toward the extension to the WS-CDL definition where requirement of capabilities are added in order to enable the automatic synthesis of policies described in the previous sections. We will call this extension WS-CDL+C. Capability requirements are expressed in a general way, in order to enable the of both semantic matchmaking algorithms and reasoning about actions techniques. In particular, we introduce semantic annotations for *input* and *output* parameters, that can be exploited by semantic matchmaking, like the techniques described by Paolucci et al. (2002) for performing the capability checking. Moreover, we introduce annotations for expressing *goals* and *preconditions* of capabilities. Goals could be used by a WSMO-like matchmaker, as suggested in Section 2. Alternatively, they could be interpreted as effects and, together with preconditions, they could be used by a reasoner for performing the kind of reasoning sketched in Section 4. The schema that defines this extension is here omitted for sake of brevity, but can be found at `http://www.di.unito.it/~alice/WSCDL_Cap_v1.2/`. Since XML-based encodings tend to be quite long, in this section we will use, as a running example, the FIPA Contract Net Protocol. A representation of the choreography published by the e-travel company can be found at the same URL written a few lines above.

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*, which has been defined somewhere. Coherently, we can think of representing the concept of capability requirement in the WS-CDL+C by two new tag elements: the tag *capabilityRequirement* (see Figure 4), and the tag *capabilityRequirementInstance* (see Figure 5). The former defines a required capability specifying its inputs, outputs, preconditions and effects.

```
1 <cdl:capabilitySection>
2
3  <cdl:capabilityRequirement
        name="evaluateTask">
4       <cdl:input>?task</cdl:inputs>
5       <cdl:output>?proposal</cdl:outputs>
6       <cdl:preconditions>
            executable(?task)
7       <cdl:effects>
            proposed(?proposal,?cost,?time)
8       </cdl:effects>
9  </cdl:capabilityRequirement>

10 <cdl:capabilityRequirement
        name="executeTask">
       .......
11 </cdl:capabilityRequirement>

12 <cdl:capabilityRequirement
        name="evaluateProposal">
       ........
13 </cdl:capabilityRequirement>

14 </cdl:capabilitySection>
```

Figure 4: Definition of capability requirements in a choreography written in WS-CDL+C.

The latter is used along the description of the policy skeleton, to be followed when playing a given role, and defines the bindings, for that specific call, of the variables in the IOPEs of the requirement with the variables used to capture the flow of information. All capability requirement definitions are gathered in a so-called *capabilitySection*.

For instance, Figure 4 reports the definition of the capability requirement *evaluateTask*. In particular, we can see that contains, besides one input (*task*: the task to accomplish) and one output (*proposal*: an evaluation that will allow the initiator to make a choice), two formulas, that express which conditions must hold because the capability is applicable, and which conditions will be caused after the execution of the capability. In the specific case of the example, the precondition is made of a single predicate, *executable(?task)*, which amounts to determine whether the task can actually be executed by the peer. On the other hand, the effect will be *proposed(?proposal,?cost,?time)*, which means that the peer will know to have done a proposal with a given estimate of execution cost and execution time.

Figure 5 reports an example invocation of a capability requirement at some point of the choreography. The notation `variable="tns:task"` is a reference to a variable, according to the definition of WS-CDL. As you can notice, each parameter in the capability requirement specification (tag *capabilityVariable*) is bound (tag *variableBind*) to a variable (or a token) of the choreography specification (tag *cdlVariable*). In this manner the values represented in the

```
1<cdl:silentAction roleType="tns:Participant">
2   <cdl:capabilityRequirementInstace
         name="evaluateTask">
3     <cdl:variableBind>
4       <cdl:cdlVariable>
             tns:task
        </cdl:cdlVariable>
5       <cdl:capabilityVariable>
             ?task
        </cdl:capabilityVariable>
6     </cdl:variableBind>
7     <cdl:variableBind>
8       <cdl:cdlVariable>
             tns:proposal
        </cdl:cdlVariable>
9       <cdl:capabilityVariable>
             ?proposal
        </cdl:capabilityVariable>
10    </cdl:variableBind>
11  </cdl:capabilityRequirementInstace>
12</cdl:silentAction>
```

Figure 5: Representing a capability requirement call in a choreography written in WS-CDL+C.

elements of the definition can be used in the whole WS-CDL document in standard ways (like Interaction, Workunit and Assign activities). In particular variables can be used in guard conditions of Workunits inside a Choice activities in order to choose alternative paths (see below for an example). Figure 6 shows an example, where the branch to follow within a choice operator depends upon the value of a variable *tns:proposal* that is returned as output by a capability requirement specification.

Choreographies not only list the set of capabilities that a peer should have but they also identify the points of the interaction at which such capabilities are to be used, i.e., invoked. A capability requirement invocation is an operation 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 (1999), 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 [1]. Returning to Figure 5, as an instance, it reports the call of the capability requirement *evaluateTask* for the role *Participant* of the Contract Net protocol. More precisely, the *evaluateTask* call is defined within a silent action and its definition comprises its name plus the bindings for its input and output variables. The tags *capabilityRequirementInstance*, *cdlVariable*, *capabilityVariable*, and *variableBind* are defined in WS-CDL+C. It is relevant to observe that each variable in this description refers to a variable that has been defined in the choreography. 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 6 shows, for example, a piece of a choreography code for the role *Participant*, containing a *choice* operator. It expands the code reported at lines 16-63 from Figures 7 and 8, skipping the part between lines 21-60. The *choice* operator allows two alternative executions, that depend on the outcome of *evaluateTask*: one is followed when the participant returns a proposal, which will be evaluated by the initiator, the other when it will return *null*, meaning that it cannot execute the requested task. The selection is done on the basis of the outcome, previously associated to the variable *proposal*, of the capability *evaluateTask*. Only when such variable has a non-null value the interaction will be continued. The guard condition at line 3 in Figure 6 amounts to determine this.

```
1  <choice>
2    <workunit name="proposeWorkUnit"
3      guard="cdl:getVariable('tns:proposal',
        '', '',
        'tns:Participant') != 'null' ">
4      <interaction
        name="proposeInteraction">
5       ...
6      </interaction>
7    </workunit>
     ...
8    <interaction
       name="evaluateTaskRefuseInteraction">
9      ...
10   </interaction>
11 </choice>
```

Figure 6: Example of how variables in capability requirements can be used in a *choice* operator of a choreography.

To complete the example we sketch in Figures 7 and 8 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 4-15, Figure 7), returning a *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 35-42, Figure 8), returning a *result*. On the other side, the *Initiator* must have the capability *evaluateProposal*, for choosing a proposal out of those sent by the participants (lines 21-28, Figure 7).

As discussed before, we can start from a representation of this kind for performing the capability test and checking if a party can play a given role. Afterwards it will be possible to synthesize the policy skeleton, possibly customized

[1] Since in WS-CDL there is not the concept of observable action, capability requirements can describe only silent actions

```
1  <sequence>
2    <interaction name="callForProposalInteraction"> ...
3    </interaction>
4    <silentAction roleType="Participant">
5      <capabilityRequirementInstace
         name="evaluateTask">
6        <cdl:variableBind>
7          <cdl:cdlVariable>
               tns:task
           </cdl:cdlVariable>
8          <cdl:capabilityVariable>
               ?task
           </cdl:capabilityVariable>
9        </cdl:variableBind>
10       <cdl:variableBind>
11         <cdl:cdlVariable>
               tns:proposal
           </cdl:cdlVariable>
12         <cdl:capabilityVariable>
               ?proposal
           </cdl:capabilityVariable>
13       </cdl:variableBind>
14     </capabilityRequirementInstace>
15   </silentAction>
16   <choice>
17     <workunit name="proposeWorkUnit" guard=...  >
18       <sequence>
19         <interaction name="proposeInteraction">
20         </interaction>
21         <silentAction roleType="Initiator">
22           <capabilityRequirementInstace
               name="evaluateProposal">
23             <cdl:variableBind>
24               <cdl:cdlVariable>
                     ...
                 </cdl:cdlVariable>
25               <cdl:capabilityVariable>
                     ...
                 </cdl:capabilityVariable>
26             </cdl:variableBind>
               ...
27           </capabilityRequirementInstance>
28         </silentAction>
29 ...
```

Figure 7: A representation of the FIPA ContractNet Protocol in the extended WS-CDL: part 1. Notice the two capability requirement invocations *evaluateTask*, for the role *Participant* and *evaluateProposal*, for the role *Initiator*, respectively at lines 4-15 and 21-28.

```
29 ...
30         <choice>
31           <workunit name="acceptProposalWorkUnit"
                       guard=...>
32             <sequence>
33               <interaction
                    name="proposeInteraction">
34               </interaction>
35               <silentAction roleType="Initiator">
36                 <capabilityRequirementInstace
                     name="executeTask">
37                   <cdl:variableBind>
38                     <cdl:cdlVariable>
                           ...
                       </cdl:cdlVariable>
39                     <cdl:capabilityVariable>
                           ...
                       </cdl:capabilityVariable>
40                   </cdl:variableBind>
                     ...
41                 </capabilityRequirementInstance>
42               </silentAction>
43               <choice>
44                 <workunit
                     name="informResultWorkUnit"
45                   guard=...  >
46                   <interaction
47                    name="informResultInteraction">
48                   </interaction>
49                 </workunit>
50                 <interaction
51                   name="failureExecuteInteraction">
52                 </interaction>
53               </choice>
54             </sequence>
55           </workunit>
56           <interaction
                name="rejectProposalInteraction">
57           </interaction>
58         </choice>
59       </sequence>
60     </workunit>
61     <interaction
          name="evaluateTaskRefuseInteraction">
62     </interaction>
63   </choice>
64 </sequence>
```

Figure 8: A representation of the FIPA ContractNet Protocol in the extended WS-CDL: part 2. Notice the capability requirement invocation *executeTask*, for the role *Initiator* at lines 35-42.

w.r.t. the capabilities and the goals of the party that is going to play the role. To this aim, a translation algorithm for turning the XML-based specification into an equivalent schema in the execution language of interest is needed.

6 CONCLUSIONS

We have proposed the introduction of the notions of “capability requirement” and of “capability”, in the context of service-oriented systems, respectively in the specification of global choreographies and of single interacting entities. A capability requirement specifies a skill to be owned by the player of a role in the choreography, *at some point of*

the interaction. The set of capability requirements associated to a choreography, therefore, specifies all those actions that are, in a way, private to the player of the role, which are, however, necessary for being in condition to actually playing the role. Such actions do not concern the exchange of messages in the context of the represented interaction: they can basically be described as those actions that allow for gathering or producing the exchanged information and for dealing with the world. The actual implementation of these actions depends very much on the player of the role. Capabilities, on the other hand, are the actual skills that an interacting party (be it a service or an agent) has.

An entity that must decide whether taking part to an interaction ruled by a choreography can download a *policy skeleton* (see also Busi et al. (2005)), that is automatically produced from the choreography specification of the role of interest. Then, it can apply a reasoning process aimed at deciding, in the simplest case, if it has the right capabilities. This decision process includes, as we have seen, a matchmaking process aimed at associating to each capability requirement a set of capabilities shown by the candidate player (notice that some of the sets could be empty). The matchmaking process alone is, however, not sufficient because it concerns single capabilities without considering the effects of using a capability within the schema of interaction given by the role specification. As we have seen, in fact, a capability could have effects that inhibit the applicability of a following one. In this case, it would be necessary to backtrack the decision process and choose, if available, some alternative.

An assumption, that we have implicitly taken so far, in the line of the seminal work by Zaremski and Wing (1997) (recently retaken into account in the application field of spatial data infrastructures by Klien and Kuhn (2006)), is that the representation of capabilities includes preconditions and effects, following a classical representational model for actions. One could wonder whether this choice is really necessary. There are, for instance, many semantic matchmaking techniques which account only for the inputs and outputs of the matched services. Indeed, the use of preconditions and effects is a means for going one step beyond the pure use of service signatures, which guarantees the correspondence of inputs and outputs (of the implementation w.r.t. the specification) but not that the way in which the implementation processes the inputs to produce the outputs is as it is meant to be, in the specification.

A representation that includes preconditions and effects has the further advantage of well-integrating the matchmaking process with the application of goal-driven reasoning techniques. We have presented, with the help of examples, different decisional tasks that can be accomplished by means of goal-driven reasoning techniques, showing how all of them can be reported to a same global definition. The reasoning task will be accomplished independently by each service/entity, when the need of playing a role arises. In this way it will not be necessary for the service to communicate its internal behavior (which would very likely be preferable to keep private). The only knowledge to make public is, in fact, the choreography itself.

Another, partly implicit, assumption that we have made, with the aim of focussing on the reasoning task, is that the choreography is written in the same language used to represent the candidate players and their capabilities. In other words, we have assumed that the set of communicative actions that appear in the choreography are the same used by the candidate players, and that the predicates used to represent preconditions and effects of capability requirements are part of the knowledge of the candidate players. In a general setting, this assumption could not hold, the speech act ontology and the predicate ontology used to specify the choreography could differ from those used by a possible role player. This general setting, indeed, raises the need of ontology matching techniques (also known as ontology mediation techniques, see, for instance, the work by Campbell and Shapiro (1998) and the WSMO approach). We have not yet faced this problem, which could be part of an extension to the current proposal.

Another possible extension concerns, instead, the kinds of reasoning that are applied to the policies. In this article we have seen a goal-driven reasoning technique that has an existential nature: "is it possible to execute, with my capabilities, the given policy, in such a way that some condition holds afterwards?", in other words "is there an execution trace of the given policy that allows me to achieve a goal of interest by using my capabilities?". A different reasoning problem is the one which is aimed at answering to universal queries of the kind: "which are all the execution paths of a given policy, that I can follow, given my capabilities?". At present, this is a task that we cannot perform.

Last but not least, we have focussed on a particular notion of "capability". It is mandatory to say, before concluding, that the term "capability" is often considered in the literature as an intuitive notion, which is not given a precise definition. However, besides the acceptations that are closer to our notion of capability, that have already been cited in the article, e.g. the one by Padgham and Lambrix (2000), it is worth to mention at least another definition for the term. Klusch and Sycara (2001) report the definition of capability used in InfoSleuth (Giampapa et al. (2000)) as being represented at four distinct levels: (1) the agent conversations, used to communicate about the service, (2) the interface to the service, (3) the information a service operates over, (4) the semantics of what the service does. This notion is more complex that the one used in this work. In a way it includes many features we have tackled in the article although it does not fully match the notion of choreography nor that of interaction protocol.

Acknowledgements

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 "Specifi-

cation and verification of agent interaction protocols" national project. Claudio Schifanella is partially supported by the fellowship program "Fondazione CRT - Progetto Lagrange" (cf. http://www.progettolagrange.it).

REFERENCES

M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Specification and verification of agent interactions using social integrity constraints. In *Proc. of the Workshop on Logic and Communication in Multi-Agent Systems, LCMAS 2003*, volume 85(2) of *ENTCS*. Elsevier, 2003.

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, *Proc. of the 1st Declarative Agent Languages and Technologies Workshop (DALT'03), Revised Selected and Invited Papers*, pages 109–134. Springer-Verlag, 2004. LNAI 2990.

M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Programming Rational Agents in a Modal Action Logic. *Annals of Mathematics and Artificial Intelligence, Special issue on Logic-Based Agent Implementation*, 41(2-4):207–257, 2004. URL `http://www.kluweronline.com/issn/1012-2443`.

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 6th Int. Workshop on Computational Logic in Multi-Agent Systems, CLIMA VI*, volume 3900 of *LNCS State-of-the-Art Survey*, pages 265–283. Springer, 2006a.

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 2006b. ACM.

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, 2006c.

M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about interaction protocols for customizing web service selection and composition. *J. of Logic and Algebraic Programming, special issue on Web Services and Formal Methods*, 70(1):53–73, 2007.

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.

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.

P. Busetta, N. Howden, R. Ronquist, and A. Hodgson. Structuring bdi agents in functional clusters. In *Proc. of the 6th Int. Workshop on Agent Theories, Architectures, and Languages (ATAL99)*, 1999.

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.

A. E. Campbell and S. C. Shapiro. Algorithms for ontological mediation. Technical Report 98-02, 23, 1998.

CSELT. http://jade.cselt.it/.

F. Dignum, editor. *Advances in agent communication languages*, volume 2922 of *LNAI*, 2004. Springer-Verlag.

Foundation for Intelligent Physical Agents. http://www.fipa.org.

J. A. Giampapa, M. Paolucci, and K. Sycara. Agent interoperation across multiagent system boundaries. In *Agents-2000 conference on autonomous agents*. ACM Press, 2000.

L. Giordano and A. Martelli. Web Service Composition in a Temporal Action Logic. In *Proc. of 4th International Workshop on AI for Service Composition (held in conjunction with ECAI 2006)*, Riva del Garda, August 2006.

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.

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.

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.

E. Klien and M. Lutz W. Kuhn. Ontology-based discovery of geographic information services - an application in disaster management. *Computers, Environment and Urban Systems*, 30(1):102–123, 2006.

M. Klusch and K. Sycara. *Brokering and Matchmaking for Coordination of Agent Societies: A Survey*, chapter 8. Springer, 2001.

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.

R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

OASIS. Business process execution language for web services. URL `ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf`.

OWL-S Coalition. http://www.daml.org/services/owl-s/.

L. Padgham and P. Lambrix. Agent capabilities: Extending BDI theory. In *AAAI/IAAI*, pages 68–73, 2000. URL `citeseer.ist.psu.edu/625805.html`.

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.

M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *First International Semantic Web Conference*, 2002.

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. ISBN 88-371-1590-3.

B. Srivastava and J. Koehler. Web service composition - current solutions and open problems. In *ICAPS 2003, Workshop on Plannuing for Web Services*, pages 28–35, 2003.

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.

WS-CDL. http://www.w3.org/tr/ws-cdl-10/.

A. Moormann Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.