Service selection by choreography-driven matching

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. The greater and greater quantity of services that are available over the web causes a growing attention to techniques that facilitate their reuse. A web service specification can be quite complex, including various operations and message exchange patterns. In this work, we focus on the problem of retrieving a web service, which can play a given choreography role, preserving at the same time a condition of interest (the goal to satisfy which the service is sought). We show that current semantic matchmaking techniques do not guarantee goal preservation. We also show an approach for overcoming these limits, which exploits the choreography definition. This work is based on an action-based representation of the operations of a service: each operation is described in terms of its preconditions and effects, without taking into account the ontology layer which is not functional to the aims of the work.

1 Introduction

Web services have a platform-independent nature, that endeavors enterprises to develop new business processes by combining existing services, retrieved over the web [19]. In the perspective of *service reuse*, the ability of retrieving services according to particular needs is crucial. Of course, it is unlikely to discover services that perfectly match a specification, some degree of flexibility is necessary. Nowadays, service retrieval is basically performed through registries like UDDI [24], where service advertisements are published. A common choice is to describe services by WSDL [29] specifications. In this context, retrieval cannot yet be accomplished automatically as well as desired because the representations used and the discovery mechanisms are semantically poor.

The need of adding a semantic layer to service descriptions brought to initiatives like the development of the language OWL-S [20] and the development of the Web Service Modeling Ontology (WSMO) [10]. In the semantic approach a richer annotation, aimed at representing the so called IOPEs (inputs, outputs, preconditions and effects of the service), is used. Inputs and outputs are usually described in terms taken from a public ontology, while preconditions and effects are often expressed by means of logic representations. The WSMO model is slightly different: here every service has also associated a logical formula, known

as the *goal* of the service, which is matched with the request during the search. The goal captures the purposes of the service while the request represents the goal of the querier: the selection is performed only when the two match.

Semantic annotation allows the discovery of services, whose descriptions *do not exactly match* with the corresponding queries. Intuitively, they allow the retrieval of services, which have been developed for a (slightly) different purpose but that can however be used for the aims of the current query. Therefore, these techniques facilitate software reuse. So-called *semantic matchmaking techniques* (e.g. [21, 16, 10]) mainly exploit forms of ontological reasoning. Many of these proposals are inspired by the seminal work of Zaremski and Wing [30] for software components match, who propose a formal specification to describe and compare software components. They define various flavors of relaxed match, that capture the notions of *generalization*, *specialization*, and *substitutability*; the best-known of these relaxed matches is the *plugin match*. Specifications are given in terms of pre- and post-conditions, written as predicates in first-order logic. Already in this work, the goal was to identify software components that could be reused in a context that was not the one for which they were originally developed.

Semantic matchmaking focuses on the discovery of single services, in the sense that a service is considered as corresponding to a *single operation*. In general, however, the use of a web service implies the execution of a *sequence of operations* in a particular *order*, which might even involve other services [1]: for instance, the clients of a supplier web service have to identify themselves, request item prices and delivery time, and so on. In order for the interaction to be successful, the message exchange must obey some constraints: if they are not satisfied the service will be unable to process the messages and will return an error. To allow the interaction, web services exhibit *interfaces* (port-types) which gather various operations that are logically related. Moreover, it is possible to specify the order in which messages are to be exchanged by means of languages like WSCI [27] and, at a lower level of detail, WSDL message exchange patterns [28].

On the other hand, the need of describing *compositions* of services, which have to interact according to (complex) patterns of interaction, ruled by conversation protocols, has lead to the development of choreography languages like WS-CDL [26]. WS-CDL is aimed at describing collaborations between any type of participant independently from the programming model used by its implementation. A WS-CDL specification can be seen as a sort of contract, that specifies the ordering conditions and constraints that rule the message exchange. The description is done from a global point of view, encompassing the expected behavior of all the participants. Each participant is supposed to use the global definition to build and test solutions that conform to it.

The task of selecting a web service, that should play a role in a choreography (rather than using the choreography as the design of a new set of services), implies verifying two things: the *conformance* of the service to the specification of a role of interest, and that the use of that service allows the achievement of the *goal*, that caused its search. *Conformance* guarantees the interoperabil-

ity of the service with the players of the other roles [23, 11, 8] by guaranteeing that the message exchange will produce correct and accepted conversations. The *goal* that caused the search of a service is a condition that should hold after the whole interaction has taken place. It is not tied to the descriptions of some service operation but it is a *global condition* that should hold in the final state, obtained after the conclusion of the conversation/interaction. The achievement of the goal depends on the operation sequence because each operation can influence the executability and the outcomes of the subsequent ones. Therefore, the matchmaking process, that is applied to discover services, should not only focus on local properties of the single operations, e.g. IOPEs, but it should also consider the global schema of execution, which is given by the choreography.

In [2] we have faced the conformance issue, proposing a conformance test that is based on a variant of bisimulation. In this work, we focus on the second problem: the selection of existing services that can play given choreography roles, preserving a condition of interest. In particular, we show that performing a match operation by operation, by applying the definitions in [30], *does not* preserve the global goal. We also show how to *overcome these limits* by exploiting the choreography definition. Actually, it is possible to extract from the choreography some information that can be used to bias the matching process so that the global goal will be preserved. To this aim, we exploit an action-based representation of the operations of a service: each operation is described in terms of its preconditions and effects, as in [3], without taking into account the ontology layer which is not functional to the aims of the work. This representation supplies the mechanisms and the tools for reasoning on compositions of services, as described in choreographies; in particular, it supplies a representation of states and an execution model that can be reasoned about.

The article is organized as follows. Section 2 introduces a simple representation for services, that is based on a declarative language, to abstract away from the details of implementation. Section 3 reports the main results of the work: we introduce the notions of conservative and of uninfluential substitution, and we show that it is possible to exploit the choreography to select services in such a way that a goal of interest is preserved. Throughout these sections we will use a same running example (introduced little by little), that is centered on the interaction ruled by the simple *purchase-flight* protocol in Figure 1. Such protocol captures the interaction between a flight ticket seller and one of its clients. In the various sections we will see the how the protocol specification is given, in the declarative language that we have adopted, as well as some implementations associated to specific services, discussing about them. Related works (Section 4) and conclusions end the paper.

2 Using a declarative language to represent services

In this section, we briefly summarize the notation that we use to represent services, introduced in [3], and we discuss the problem of verifying a global goal. The notation is based on a logical theory for reasoning about actions and change

in a modal logic programming setting. In this perspective, the problem of reasoning amounts either to build or to traverse a sequence of transitions between *states*. A state is a set of *fluents*, i.e., properties whose truth value can change over time, due to the application of actions. In general, we cannot assume that the value of each fluent in a state is known: we want to have both the possibility of representing unknown fluents and the ability of reasoning about the execution of actions on incomplete states. To explicitly represent unknown fluents, we use 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*.

2.1 Service representation

A *service description* is defined as a triple $\langle \mathcal{O}, \mathcal{G}, \mathcal{P} \rangle$, where $\mathcal{O}$ is a set of operations, $\mathcal{G}$ is a set of actions that allow to receive messages, and $\mathcal{P}$ (*policy*) is a description of the interactive behavior of the service. The name "policy" derives from the literature concerning conversation protocols [13].

- The set $\mathcal{O}$ contains the descriptions of a set of service operations. An operation is an atomic action. As such, it is described in terms of its *executability preconditions* and *effects*, the former being a set of fluents (introduced by the keyword **possible if**) which must be contained in the service state in order for the operation to be applicable, the latter being a set of fluents (introduced by the keyword **causes**) which will be added to the service state after the operation execution. Formalized in these terms, operations, when executed, trigger a revision process on the actor's beliefs. Since we describe web services from a *subjective* point of view (i.e. taking the perspective of a specific service, by representing and reasoning on the service policies), we distinguish between the case when the service is either the initiator or the servant of an operation by further decorating the operation name with a notation inspired by [5]. With reference to a specific service, $operation^{\gg}(interlocutor, content)$ denotes the fact that the service is the initiator of the operation (as in the case of "solicit-response" interactions), while $operation^{\ll}(interlocutor, content)$ denotes the fact that the service is the servant of the operation (as in the case of "request-response" interactions). Formally, an operation is represented as:

  $$operation^{d}(interlocutor, content) \ \textbf{possible if} \ \{P_1, \ldots, P_t\}$$
  $$operation^{d}(interlocutor, content) \ \textbf{causes} \ \{E_1, \ldots, E_n\}$$

  where d is either $\gg$ or $\ll$, E_i, $i \in [1, n]$, and P_j, $j \in [1, t]$, denote respectively the fluents, which are expected as effect of the execution of an operation and the precondition to its execution, while *content* denotes possible additional data that is required by the operation.

As an example, let's consider search_flight, an operation of a flight reservation service, which is offered by a *seller* and can be invoked by a *client* to search information about flights with given departure and arrival locations. From the point of view of the client, the operation search_flight is represented as:

$$\begin{array}{l}\mathsf{search\_flight}^{\gg}(seller, Date, Start, Dest)\\ \quad \textbf{possible if } \{\mathcal{B}Start, \mathcal{B}Dest, \mathcal{B}Date\}\\ \mathsf{search\_flight}^{\gg}(seller, Date, Start, Dest)\\ \quad \textbf{causes } \{\mathcal{B}will\_get\_offer\}\end{array}$$

This notation captures the preconditions and the effects of the operation: the precondition is that the departure location is known ($\mathcal{B}Start$), that the destination is known ($\mathcal{B}Dest$), and that the day of departure is also known ($\mathcal{B}Date$); the effect is that after the execution the invoker (the client) expects that an offer will be sent ($\mathcal{B}will\_get\_offer$).
Last but not least, a service can also have internal operations, which can be included in its policy but are not visible from outside. Each operation is represented again as an atomic action, specified by its *preconditions* and its *effects*. Formally, it is defined as:

$$\begin{array}{l}\mathsf{operation}(content) \ \textbf{causes} \ \{E_1, \ldots, E_n\}\\ \mathsf{operation}(content) \ \textbf{possible if} \ \{P_1, \ldots, P_t\}\end{array}$$

where E_i, $i \in [1, n]$, and P_j, $j \in [1, t]$, denote respectively the fluents, which are expected as effect of the execution of an operation and the precondition to its execution, while *content* denotes possible additional data that is required by the operation. Notice that such operations can also be implemented as invocations to other services. As an example, here is the description of the eval_offer internal operation of a possible client for the flight-purchase interaction:

$$\begin{array}{l}\mathsf{eval\_offer} \ \textbf{possible if} \ \{\mathcal{B}offer(flight)\}\\ \mathsf{eval\_offer} \ \textbf{causes} \ \{\mathcal{B}eval\_rst(Y)\}\end{array}$$

eval_offer applies when an offer for a flight is available and produces an evaluation that can be used for taking internal decision. The variable Y ranges over the set $\{business, no\_business\}$ depending on the kind of ticket that is being considered.

– Besides operations, we explicitly represent actions that allow the reception of information. We call them *get-answer* actions (set $\mathcal{G}$). The range of possible answers is supposed to be finite, in the sense that the interlocutor is supposed to use a message out of a finite and predefined set of alternatives: if a different message is sent the service is not able to handle it. We imagine the reception of a piece of information as a "one-way" operation that is invoked over the recipient. Actually, for technical reasons in our formalization, each possible alternative answer corresponds to a different operation of this kind. Formally, they are represented as:

$$\mathsf{receive\_act}(interlocutor, content) \ \textbf{receives} \ \mathcal{I}$$

where *interlocutor* is the partner in the interaction, *content* is used to store the received message, and $\mathcal{I}$ is a set of alternative action invocations each allowing the reception of one of the alternative messages. As an example get_answer, reported hereafter, allows the reception of either a not_available$^{\ll}$ answer or of an offer through the execution of one of the two actions not_available$^{\ll}$ and offer$^{\ll}$. The decision of which of the two actions will be executed is up to the interlocutor that decides which message to send.

$$\mathsf{get\_answer}(Seller) \ \textbf{receives} \ [\mathsf{not\_available}^{\ll}(Seller) \ \textbf{or} \ \mathsf{offer}^{\ll}(Seller)]$$

In this example we do not use the *content*.

– $\mathcal{P}$ encodes the behavior for the service; it is a collection of clauses of the kind:

$$p_0 \ \textbf{is} \ p_1, \ldots, p_n$$

where p_0 is the name of the procedure and p_i, $i = 1, \ldots, n$, is either an atomic action (operation), a *get-answer* action, a test action (denoted by the symbol ?), or 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 non-deterministic as in Prolog. As an instance, here we report the booking procedure:

$$\begin{array}{l}\mathsf{booking}(Seller, Date, Start, Dest) \ \textbf{is} \\ \quad \mathsf{search\_flight}^{\gg}(Seller, Date, Start, Dest), \ \mathsf{get\_answer}(Seller), \\ \quad \mathcal{B}offer(not\_avail)? \\ \mathsf{booking}(Seller, Date, Start, Dest) \ \textbf{is} \\ \quad \mathsf{search\_flight}^{\gg}(Seller, Date, Start, Dest), \ \mathsf{get\_answer}(Seller), \\ \quad \mathcal{B}offer(flight)?, \ \mathsf{eval\_offer}, \ \mathsf{finalize}(Seller)\end{array}$$

It is defined by a set of two clauses, the former capturing the case when the ticket is not available, the latter the normal situation when an offer for a ticket is actually returned. In this case, the offer is evaluated and the purchase is finalized by invoking another procedure.

A *choreography* is made of a set of interacting *roles*, a role being a subjective view of the interaction that is encoded. When a service plays a role in a choreography, its policy will contain some operations which are not of the service itself but belong to some other role of the choreography, with which it interacts. In other words, $\mathcal{O}$ can be partitioned in two sets: a set of bound operations and a set of unbound operations, that must be supplied by some counterpart(s). Until the counterpart(s) service is (are) not defined, such operations will be those specified in the choreography. We assume that they are represented in a way that is homogeneous with the representation of operations, i.e. by means of preconditions and effects. The binding will be possible only when the partner in the interaction will be found. The fact that the former service is taking a given role

in the choreography is due, in our proposal, to the fact that it knows that a certain goal condition will be true after the execution of the role. When a possible partner is identified for the latter role, after the binding has taken place, it is necessary to check if the goal condition is preserved. The reasons for which this could not happen are explained in the following section; hereafter, we formalize the notion of *substitution* that we interpret as the binding.

Let $S_d = \langle \mathcal{O}, \mathcal{G}, \mathcal{P} \rangle$ a service description, and let $\mathcal{O}_u$ be a subset of $\mathcal{O}$, containing unbound operations that are to be supplied by a same counterpart S_i. Let $\mathcal{O}_{S_i}$ be the set of operations in S_i that we want S_d to use, binding them to $\mathcal{O}_u$. We represent the binding by the substitution $\theta = [\mathcal{O}_{S_i}/\mathcal{O}_u]$ applied to S_d, i.e.: $S_d\theta = \langle \mathcal{O}\theta, \mathcal{G}\theta, \mathcal{P}\theta \rangle$, where every element of $\mathcal{O}_u$ is substituted by/bound to an element of $\mathcal{O}_{S_i}$. Notice that not all elements of $\mathcal{O}_{S_i}$ are, instead, necessarily bound. An example is reported in *Example* 3.

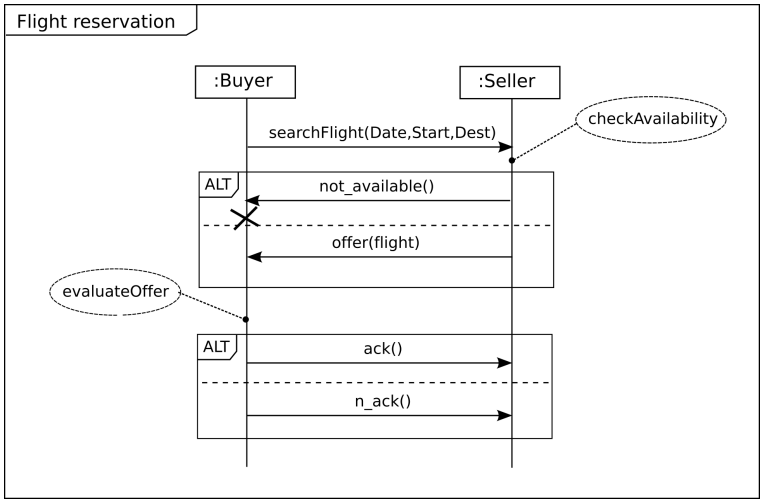

Fig. 1. An example of a simple interaction protocol, for reserving a flight, expressed as a UML sequence diagram.

Example 1. Let us introduce, as an example, a simple choreography (see Figure 1) that rules a flight reservation protocol, inspired to [22], with two roles: a *Buyer* and a *Seller*. The buyer sends a request to search for flights, specifying the departure location, the destination, and date. Depending on the seat availability, the seller can either refuse the request, or send the information regarding a specific flight. The buyer checks the offer, then, it either refuses (*n_ack*) or accepts it (*ack*). All the names on the arrows, e.g. *searchFlight* and *offer*, are specifications of the operations that the players must provide and perform.

Let us consider a service $b1$ that is conformant to the role *Buyer*. Following the proposed notation, we describe it as $\langle \mathcal{O}, \mathcal{G}, \mathcal{P} \rangle$, where $\mathcal{P} = \{\mathsf{booking}, \mathsf{finalize}\}$, $\mathcal{O} = \{\mathsf{search\_flight}^{\gg}{}_u,\ \mathsf{not\_available}^{\ll}{}_u,\ \mathsf{eval\_offer},\ \mathsf{offer}^{\ll}{}_u,\ \mathsf{ack}^{\gg}{}_u,\ \mathsf{n\_ack}^{\gg}{}_u\}$, $\mathcal{G} = \{\mathsf{get\_answer}\}$. Notice that all the operations but eval_offer are unbound and depend on the service that will actually play the other role. Instead, eval_offer is an internal action of the buyer, that is used to decide whether accepting or refusing an offer. The procedures in $\mathcal{P}$ are described by the following clauses:

$\mathsf{booking}(Seller, Date, Start, Dest)$ **is**
  $\mathsf{search\_flight}^{\gg}{}_u(Seller, Date, Start, Dest),\ \mathsf{get\_answer}(Seller),$
  $\mathcal{B}offer(not\_avail)?$
$\mathsf{booking}(Seller, Date, Start, Dest)$ **is**
  $\mathsf{search\_flight}^{\gg}{}_u(Seller, Date, Start, Dest),\ \mathsf{get\_answer}(Seller),$
  $\mathcal{B}offer(flight)?,\ \mathsf{eval\_offer},\ \mathsf{finalize}(Seller)$
$\mathsf{finalize}(Seller)$ **is**
  $\mathcal{B}eval\_rst(business)?,\ \mathsf{ack}^{\gg}{}_u(Seller)$
$\mathsf{finalize}(Seller)$ **is**
  $\mathcal{B}eval\_rst(no\_business)?,\ \mathsf{n\_ack}^{\gg}{}_u(Seller)$

The only get_message action in $\mathcal{G}$ is described by:

$\mathsf{get\_answer}(Seller)$ **receives** $[\mathsf{not\_available}^{\ll}{}_u(Seller)$ **or** $\mathsf{offer}^{\ll}{}_u(Seller)]$

Finally, the operations in $\mathcal{O}$ are described as:

$\mathsf{eval\_offer}$ **causes** $\{\mathcal{B}eval\_rst(Y)\}$
$\mathsf{eval\_offer}$ **possible if** $\{\mathcal{B}offer(flight)\}$

$\mathsf{search\_flight}^{\gg}{}_u(Seller, Date, Start, Dest)$
  causes $\{\mathcal{B}will\_get\_offer\}$
$\mathsf{search\_flight}^{\gg}{}_u(Seller, Date, Start, Dest)$
  possible if $\{\mathcal{B}start, \mathcal{B}dest, \mathcal{B}date\}$

$\mathsf{not\_available}^{\ll}{}_u(Seller)$ **causes** $\{\mathcal{B}offer(not\_available)\}$
$\mathsf{not\_available}^{\ll}{}_u(Seller)$ **possible if** $\{\}$

$\mathsf{offer}^{\ll}{}_u(Seller)$ **causes** $\{\mathcal{B}offer(flight)\}$
$\mathsf{offer}^{\ll}{}_u(Seller)$ **possible if** $\{\}$

$\mathsf{ack}^{\gg}{}_u(Seller)$ **causes** $\{\mathcal{B}booked(flight)\}$
$\mathsf{ack}^{\gg}{}_u(Seller)$ **possible if** $\{\}$

$\mathsf{n\_ack}^{\gg}{}_u(Seller)$ **causes** $\{\mathcal{B}\neg booked(flight)\}$
$\mathsf{n\_ack}^{\gg}{}_u(Seller)$ **possible if** $\{\}$

where X ranges on the set $\{not\_available,\ flight\}$ and Y ranges on the set $\{business, no\_business\}$. All the above definitions, but the one of eval_offer, are supplied by the choreography. □

2.2 Reasoning on goals

In the outlined framework, it is possible to reason about goals by means of queries of the form:

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

where Fs is the goal (represented as a conjunction of fluents), that we wish to hold after the execution of a policy p. Checking if a formula of this kind holds corresponds to answering the query: "Is it possible to execute p in such a way that the condition Fs is true in the final state?". When the answer is positive, the reasoning process returns a sequence of atomic actions that allows the achievement of the desired condition. This sequence corresponds to an execution trace of the procedure and can be seen as a plan to bring about the *goal* Fs. Such a plan can be conditional because whenever a *get-answer* action is involved, none of the possible answers from the interlocutor can be excluded. In other words, the trace will contain a different execution branch for every option.

This form of reasoning is known as *temporal projection*. Temporal projection fits our needs because, as mentioned in the introduction, in order to perform the selection we need a mechanism that verifies if a goal condition holds after the interaction with the service has taken place. Fs is the set of facts that we would like to hold "after" p.

Let $S_d = \langle \mathcal{O}, \mathcal{G}, \mathcal{P} \rangle$ be a service description. The application of temporal projection to $\mathcal{P}$ returns, if any, an execution trace, that makes a goal of interest become true. Let us, then, consider a procedure p belonging to $\mathcal{P}$, and denote by G the query Fs **after** p. Given a state S_0, containing all the fluents that we know as being true in the beginning, we denote the fact that G is successful in S_d by:

$$(\langle \mathcal{O}, \mathcal{G}, \mathcal{P} \rangle, S_0) \vdash G$$

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 contain branches, that are due, as we have mentioned, to the presence of get-message actions.

Example 2 (Flight-purchase, second part). Let us suppose that the initial state of the service $b1$ is $S_0 = \{\mathcal{B}date, \mathcal{B}start, \mathcal{B}dest, \mathcal{B}smoking\_flight\}$, (all the other fluents truth value is "unknown"). This means that $b1$ assumes a date, a departure location, an arrival location and that on the flight it is allowed to smoke. The goal of $b1$ is that the following condition holds:

$$G = \{\mathcal{B}booked(flight), \mathcal{B}smoking\_flight\} \ \textbf{after} \ \mathsf{booking}(seller, date, start, dest)$$

Intuitively, the buyer expects that, after the interaction, it will have a reservation on a smoking flight.

By reasoning on its policy and by using the definitions of the unbound operations that are given by the choreography, $b1$ can identify an execution trace, that leads to a state where G holds:

$$\sigma = \mathsf{search\_flight}^{\gg}{}_u(seller, date, start, dest); \mathsf{offer}^{\ll}{}_u(seller); \\ \mathsf{eval\_offer}; \mathsf{ack}^{\gg}{}_u(seller)$$

This is possible because in a declarative representation specifications are executable. Moreover notice that this execution does not influence the belief about the smoking flight, which persists from the initial through the final state and is not contradicted. □

3 Goal-preserving match

When the matching process is applied for selecting a service that should play a role in a (partially instantiated) choreography, the desire is that the substitution (of the service operations to the specifications contained in the choreography) preserves the properties of interest. Let us formalize this notion.

Definition 1 (Conservative substitution). *Let us consider a service* $S_i = \langle \mathcal{O}, \mathcal{G}, \mathcal{P} \rangle$ *which plays a role* R_i *in a given choreography, and a query* G *such that, given an initial state* S_0*,*

$$(\langle \mathcal{O}, \mathcal{G}, \mathcal{P} \rangle, S_0) \vdash G \ w.a. \ \sigma$$

Consider a substitution $\theta = [\mathcal{O}_{S_j} / \mathcal{O}^{\sigma}_{u(R_j)}]$*, where* $\mathcal{O}^{\sigma}_{u(R_j)} = \{o_u \in \mathcal{O} \mid o \text{ occurs in } \sigma\}$ *is the set of all unbound operations that refer to another role* R_j*,* $j \neq i$*, of the same choreography, that are used in the execution trace* σ*.* θ *is conservative when the following holds:*

$$(\langle \mathcal{O}\theta, \mathcal{G}\theta, \mathcal{P}\theta \rangle, S_0) \vdash G \ w.a. \ \sigma\theta$$

In the above definition, θ can be any kind of association between the operations of a service with the operations described in a choreography. In practice is the result of a matching process. In the literature it is possible to find many match algorithms, many of them (in the case of semantic web services) are grounded into the work by Zaremski and Wing [30], mentioned in the introduction.

Zaremski and Wing propose a formal specification to describe the behavior of software components, and to determine if two components match. Each software component has precondition $\mathsf{Precs}(s)$ and postcondition $\mathsf{Effs}(s)$. Their specifications are matched against a requirement, coherently specified as having precondition $\mathsf{Precs}(r)$ and postcondition $\mathsf{Effs}(r)$. Five kinds of relaxed match between r and s are defined, that we rephrase hereafter, to adapt them to our framework:

- EM (*Exact Pre/Post Match*): $\mathsf{Precs}(r) = \mathsf{Precs}(s) \wedge \mathsf{Effs}(r) = \mathsf{Effs}(s)$
- PIM (*Plugin Match*): $\mathsf{Precs}(r) \supseteq \mathsf{Precs}(s) \wedge \mathsf{Effs}(s) \supseteq \mathsf{Effs}(r)$
- POM (*Plugin Post Match*): $\mathsf{Effs}(s) \supseteq \mathsf{Effs}(r)$
- GPIM (*Guarded Plugin Match*): $\mathsf{Precs}(r) \supseteq \mathsf{Precs}(s) \wedge ((\mathsf{Precs}(s) \cup \mathsf{Effs}(s)) \supseteq \mathsf{Effs}(r))$
- GPOM (*Guarded Post Match*): $((\mathsf{Precs}(s) \cup \mathsf{Effs}(s)) \supseteq \mathsf{Effs}(r))$

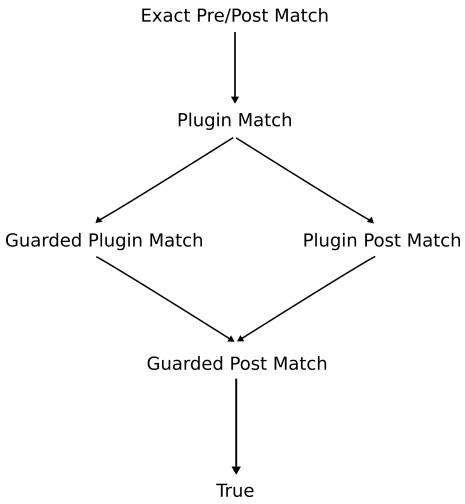

Fig. 2. The lattice of the different local matches: on top the strongest.

Exact pre/post match states the equivalence of r and s. *Plugin match* is weaker: s must only be behaviorally equivalent to r when plugged-in to replace r. *Plugin post match* relaxes the former: only the postcondition is considered. *Guarded matches* focus on guaranteeing that the desired postcondition holds when the precondition of s holds, not necessarily in general. The different matches can be organized according to a lattice [30], that we have reported in Fig. 2. For short, we will respectively denote by θ_{EM}, θ_{PIM}, θ_{POM}, θ_{GPIM}, θ_{GPOM}, the substitutions obtained by applying the five degrees of match.

It is immediate to see that any substitution, obtained by applying the exact pre/post match, satisfies Definition 1. However, this is not true for the other kinds of match. Let us show this with the help of a simple example.

Example 3. The buyer service $b1$ (see previous examples) is looking for another service, which can play the role of the *Seller*, to reserve a flight seat. This service must provide a set of operations that will substitute the unbound operations of the buyer role. Let us choose the *plugin match* as matching rule. Let us consider the candidate $s1$, a service that is conformant to the protocol w.r.t. the role *Seller*. The set of operations of the seller represented in the knowledge base of the buyer includes the following definition for operation search_flight$^{\gg}$:

search_flight$^{\gg}(Seller, Date, Start, Dest)$
  possible if $\{\mathcal{B}start,\ \mathcal{B}dest,\ \mathcal{B}date\}$
search_flight$^{\gg}(Seller, Date, Start, Dest)$
  causes $\{\mathcal{B}will\_get\_offer,\ \mathcal{B}\neg smoking\_flight\}$

while all the other operations are defined exactly as in Example 1.

By applying the *plugin match*, we obtain the substitution θ_{PIM}, which includes, among the others, also $[\mathsf{search\_flight}^{\gg}/\mathsf{search\_flight}^{\gg}{}_u]$ [1]. By applying the substitution θ_{PIM} we obtain the set of policies $\mathcal{P}\theta_{PIM}$:

$\mathsf{booking}(Seller, Date, Start, Dest)$ **is**
  $\mathsf{search\_flight}^{\gg}(Seller, Date, Start, Dest)$, $\mathsf{get\_answer}(Seller)$,
  $\mathcal{B}offer(not\_avail)?$
$\mathsf{booking}(Seller, Date, Start, Dest)$ **is**
  $\mathsf{search\_flight}^{\gg}(Seller, Date, Start, Dest), \mathsf{get\_answer}(Seller)$,
  $\mathcal{B}offer(flight)?; \mathsf{eval\_offer}; \mathsf{finalize}(Seller)$

$\mathsf{finalize}(Seller)$ **is**
  $\mathcal{B}eval\_rst(business)?; \mathsf{ack}^{\gg}(Seller)$
$\mathsf{finalize}(Seller)$ **is**
  $\mathcal{B}eval\_rst(no\_business)?; \mathsf{n\_ack}^{\gg}(Seller)$

By using this policy, the query $(\langle \mathcal{O}\theta_{PIM}, \mathcal{G}\theta_{PIM}, \mathcal{C}, \mathcal{P}\theta_{PIM}\rangle, S_0) \vdash G$ fails: in fact, the additional effect $\mathcal{B}\neg smoking\_flight$ of the service $\mathsf{search\_flight}^{\gg}$ prevents the buyer to achieve a part of its goal, i.e. to book a *smoking* flight. □

Theorem 1. *The class of PIM, POM, GPIM and GPOM substitutions are not conservative.*

Proof. The proof is given by the counterexample in Example 3. In fact, θ, besides being a PIM substitution, is also an instance of all the other kinds of substitution that we have listed, i.e. it is also a POM, a GPIM, and a GPOM substitution.

In order for a substitution to be conservative, it must take into account also the *overall structure*, encoded by the choreography. The locality of the matches used in the matchmaking phase, indeed, seriously limits the possibility of re-using services by selecting and composing them in an automatic way.

In the remainder of this section, we focus on the *plug-in match*. The plugin match is one of the most used matches and it immediately follows the exact match in the lattice (it is the strongest of the flexible matches). We show how to enrich it so to allow the construction of conservative substitutions. To this aim, we take into account the *dependencies* between actions, which produce as effects fluents, that are used as preconditions by subsequent action. Intuitively, the idea is to verify that the "causal chain" which allows the execution of the sequence of actions, is not broken by the differences between capabilities/services and requirements, as instead happens in the example. The obvious hypothesis is that we have a choreography and that we know that it allows to achieve the goal of interest, i.e. that there is an execution σ of the role specification, which allows the achievement of the goal. We will use this trace for defining the additional properties for the match.

[1] For the sake of brevity, we omit to specify the substitutions when the operations exactly match the specifications.

Let us start by introducing the notions that define dependencies between actions and dependency sets for fluents. Consider a service description $S = \langle \mathcal{O}, \mathcal{G}, \mathcal{P} \rangle$, and suppose that, given the initial state S_0, the goal $G = Fs$ **after** p succeeds, thus obtaining as answer the successful sequence of actions $\sigma = a_1; a_2; \ldots; a_n$, which is an execution trace of p.[2] We denote by $\overline{\sigma}$ the sequence of actions $a_0; a_1; a_2; \ldots; a_n; a_{n+1}$, where a_0 and a_{n+1} are two *fictitious* actions that will be used respectively to represent the initial state S_0 and the set of fluents Fs, which must hold after σ. That is, we assume a_0 has no precondition and $\mathsf{Effs}(a_0) = S_0$, and that a_{n+1} has no effect but $\mathsf{Precs}(a_{n+1}) = Fs$.

Consider two indexes i and j, such that $j < i$, $i, j = 0, \ldots, n+1$. We say that *in* $\overline{\sigma}$ the action a_i *depends on* a_j *for the fluent* $\mathcal{B}l$, written $a_j \rightsquigarrow_{\langle \mathcal{B}l, \overline{\sigma} \rangle} a_i$, iff $\mathcal{B}l \in \mathsf{Effs}(a_j)$, $\mathcal{B}l \in \mathsf{Precs}(a_i)$, and there is not a k, $j < k < i$, such that $\mathcal{B}l \in \mathsf{Effs}(a_k)$. Given a fluent $\mathcal{B}l$ and a sequence of actions σ, we can, therefore, define the *dependency set* of $\mathcal{B}l$ as $\mathsf{Deps}(\mathcal{B}l, \sigma) = \{(j, i) \mid a_j \rightsquigarrow_{\langle \mathcal{B}l, \overline{\sigma} \rangle} a_i\}$.

Let $[s/o_u]$ be a specific substitution of a service operation s to an unbound operation o_u, that is contained in θ_{PIM}, we say that a fluent $\mathcal{B}l \in \mathsf{Effs}(s) - \mathsf{Effs}(o_u)$ (i.e. an additional effect of s w.r.t. the effects of o_u) is an *uninfluential fluent* w.r.t. the sequence $\sigma\theta_{PIM}$ iff for all pairs $(j, i) \in \mathsf{Deps}(\mathcal{B}\neg l, \sigma)$, identifying by k the position of o_u in σ, we have that $k < j$ or $i \leq k$, Intuitively, this means that the fluent will not break any dependency between the actions which involve the inverse fluent because either it will be overwritten or it will appear after its inverse has already been used. Note that σ and $\sigma\theta_{PIM}$ have the same length and are identical as sequences of actions but for the fact that in the latter the selected service operations substitute unbound operations. For this reason, we can reduce to reasoning on σ for what concerns the action positions.

A substitution θ_{PIM} is called *uninfluential* iff for any substitution $[s/o_u]$ in θ_{PIM}, all beliefs in $\mathsf{Effs}(s) - \mathsf{Effs}(o_u)$ are uninfluential fluents w.r.t. σ. Now we are in position to prove that a substitution which exploits the *plugin match* and which is also *uninfluential*, is conservative.

Theorem 2. *Let us consider a service $S_i = \langle \mathcal{O}, \mathcal{G}, \mathcal{P} \rangle$ which plays a role R_i in a given choreography, and a query G such that, given an initial state S_0,*

$$(\langle \mathcal{O}, \mathcal{G}, \mathcal{P} \rangle, S_0) \vdash G \text{ w.a. } \sigma$$

Consider an uninfluential substitution $\theta_{PIM} = [\mathcal{O}_{S_j}/\mathcal{O}^{\sigma}_{u(R_j)}]$, where $\mathcal{O}^{\sigma}_{u(R_j)} = \{o_u \in \mathcal{O} \mid o \text{ occurs in } \sigma\}$ is the set of all unbound operations that refer to another role R_j, $j \neq i$, of the same choreography, that are used in the execution trace σ. Then, the following holds:

$$(\langle \mathcal{O}\theta_{PIM}, \mathcal{G}\theta_{PIM}, \mathcal{P}\theta_{PIM} \rangle, S_0) \vdash G \text{ w.a. } \sigma\theta_{PIM}$$

Proof. The proof is by absurd and it uses the proof theory introduced in [4]. Let us assume that $(\langle \mathcal{O}, \mathcal{G}, \mathcal{P} \rangle, S_0) \vdash G$ w.a. σ but $(\langle \mathcal{O}\theta_{PIM}, \mathcal{G}\theta_{PIM}, \mathcal{P}\theta_{PIM} \rangle, S_0) \nvdash$

[2] In the following we focus on linear plans. Conditional plans can be tackled by considering each path separately.

G w.a. $\sigma\theta_{PIM}$. Since, by hypothesis, for any substitution $[o/o_u]$ in θ_{PIM}, $\mathsf{Effs}(o) \subseteq \mathsf{Effs}(o_u)$ holds, there exists a fluent F such that $a_0, a_1, \ldots, a_{i-1} \vdash F$ but $(a_0, a_1, \ldots, a_{i-1})\theta_{PIM} \nvdash F$, where $\sigma = a_0, a_1, \ldots, a_{i-1}, a_i, \ldots, a_n$ and $F \in \mathsf{Precs}(a_i)$. Now, since $a_0, a_1, \ldots, a_{i-1} \vdash F$, there exists $j \leq i-1$, such that $a_0, a_1, \ldots, a_j \vdash F$ and $F \in \mathsf{Effs}(a_j)$ but $(a_0, a_1, \ldots, a_j)\theta_{PIM} \nvdash F$, that is $F \notin \mathsf{Effs}(a_j\theta_{PIM})$. This is absurd due to the hypothesis that θ_{PIM} is an uninfluential substitution. □

Example 4. Let us now consider the goal and the service description specified in Example 1, and let us also consider another service $s2$, that is conformant to the role *Seller*. Let $s2$ be equivalent to the role specification but for the service that implements $\mathsf{search\_flight}^{\gg}{}_u$, which is specified as:

$\mathsf{search\_flight}^{\gg}(Seller, Date, Start, Dest)$
  possible if $\{\mathcal{B}start,\ \mathcal{B}dest,\ \mathcal{B}date\}$
$\mathsf{search\_flight}^{\gg}(Seller, Date, Start, Dest)$
  causes $\{\mathcal{B}will\_get\_offer,\ \mathcal{B}veg\_meals\}$

Differently than the one of $s1$, this service does not compromise the achievement of the goal, even though it provides some additional information ($\mathcal{B}veg\_meals$). This information is not used in the interaction that we are considering but we must take into account the fact that $s2$ might be conformant also to other protocols, in which this information is relevant. It is realistic that the service will not be re-implemented each time if not strictly necessary. □

The verification that a substitution is uninfluential involves the derivation σ, and it is based on checking whether the chains of dependencies between actions for the various fluents are not interrupted by some opposite fluent. Obviously, if the domain is such that no fluent, once asserted, can be negated, any θ_{PIM} will be conservative. This can be verified statically on the choreography and the set of unbound operations, by checking that every fluent (that appears as effect of some action) is always positive or negative, including the initial state and the goal in the verification. Indeed, the application domains in which actions produce *knowledge* are of this kind.

4 Conclusion and related works

In this work we have studied the relation between the matchmaking rules and the achievement of a goal in a choreography, within the process of selecting a service for playing a role. We have shown that, when the adoption of a role is due to the desire of reaching a goal, the matches performed on single operations (but the exact match) are not adequate and it is necessary to introduce a verification that takes into account the context given by the choreography. Afterwards, we have presented an extension of the plugin match that takes into account also the choreography. To the best of our knowledge, the selection of a service based on a kind of match that takes into account *context of application* of the sought services (i.e. the choreography in which it will be immersed) has not been yet tackled in the literature, with the only exception of the work by Biswas [6]. Biswas proposes

to enrich service descriptions with constraints, i.e. conditions that hold during the execution of the service. Given a specification of a desired composed service, in BPEL or in OWL-S, a discovery process is enacted to identify the services to assemble. The constraints associated to them are used to build the overall constraints of the composition, which is then checked against the constraints given by the user, to see if the composition satisfies the user's needs. This is a bottom-up approach, aimed at verifying some properties of the composition which are not captured by the IOPE analysis.

The literature related to matchmaking is wide and it is really difficult to be exhaustive. The matches proposed in [30] have inspired most of the semantic matches for web service discovery. Amongst them, Paolucci et al. [21] propose four degrees of match (exact, plugin, subsumes, and fail) that are computed on the ontological relations of the outputs of an advertisement for a service and a query. This approach tackles DAML-S representations, in which services are described by means of inputs and outputs. This approach is refined in [16], a work that describes a service matchmaking prototype, which uses a DAML-S based ontology and a Description Logic reasoner to compare ontology-based service descriptions, given in terms of input and output parameters. The matchmaking process, like in [21], produces a discrete scale of degrees of match (Exact, PlugIn, Subsume, Intersection, Disjoint).

WSMO (Web Service Modeling Ontology) [10] is an organizational framework for semantic web services. As such, it does not suggest a specific matching rule, which is up to the specific implementations. However, the authors propose in [15] an approach that is based on [30] and on [16]. More recently, a WSMO matchmaker has been proposed in [14], which combines several aspects: type matching, relation matching, constraint matching, parameter matching, intentional matching. Last but not least, in [18] a multi-level evaluation model is proposed, for deciding whether two services are composable. This is done through four levels of control (quality, dynamic semantics, static semantics, and syntax). Dynamic semantics is the name given to the matches of [30].

The idea of synthesizing a policy from an abstract specification is also stated in [9], where it is observed that services are often conceived so as to be delivered individually, while there is a growing need of reusing this software, either by composing services or by tailoring a composition to some specific client. This direction has been suggested in [19], where a UML specification of a business process was used to abstract the description of a composition away from the specification of the composed services. This abstract specification defines a *model*, used for driving the retrieval and the composition task.

Works like [22, 7] propose approaches for goal-driven service composition based on planning. However, the task is accomplished without reference to any choreography. In particular, in [22] the composition and the semantic reasoning phases (carried on on inputs and outputs) are separated and the latter is performed on a local basis only. In [12, 17] web services are composed by composing their interaction protocols in a social framework, by means of a temporal logic.

The next step of this research will be to test the presented method, by implementing it in a real system and applying it to real cases. In particular, for what concerns the representation we mean to explore the use of SAWSDL [25], an extension of WSDL that enables the use of semantic annotations. Moreover, so far we have not yet tackled the integration of ontological reasoning in our work. This is surely an interesting extension that we will face soon; actually, many proposals for semantic matchmaking base upon the same relaxed match that we have used, and we expect similar results.

Acknowledgment

This research has partially been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6^{th} FP project REWERSE number 506779 (cf. http://rewerse.net), and by MIUR PRIN 2005 "Specification 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

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services*. Springer, 2004.
2. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. A priori conformance verification for guaranteeing interoperability in open environments. In *Proc. of ICSOC 2006*, volume 4294 of *LNCS*, pages 339–351. Springer, 2006.
3. 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*, 70(1):53–73, 2007.
4. 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.
5. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Synthesis of Underspecified Composite e-Service bases on Atomated Reasoning. In *Proc. of ICSOC04*, pages 105–114. ACM, 2004.
6. D. Biswas. Web services discovery and constraints composition. In *Proc. of the 1st Int. Conf. on Web Reasoning and Rule Systems, RR 2007*, volume 4524 of *LNCS*, pages 73–87. Springer, 2007.
7. J. Bryson, D. Martin, S. McIlraith, and L. A. Stein. Agent-based composite services in DAML-S: The behavior-oriented design of an intelligent semantic web. In *Agent-Based Composite Services in DAML-S: The Behavior-Oriented Design of an Intelligent Semantic Web, Web Intelligence*. Springer-Verlag, 2002.
8. 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.
9. F. Casati and M. C. Chien. Dynamic and adaptive composition of e-services. *Information Systems*, 26:143–163, 2001.
10. D. Fensel, H. Lausen, J. de Bruijn, M. Stollberg, D. Roman, and A. Polleres. *Enabling Semantic Web Services : The Web Service Modeling Ontology*. Springer, 2007.

11. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based analysis of obligations in web service choreography. In *Proc. of IEEE International Conference on Internet&Web Applications and Services 2006*, 2006.
12. L. Giordano and A. Martelli. Web Service Composition in a Temporal Action Logic. In *Proc. of 4th Int. Work. on AI for Service Composition*, 2006.
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. F. Kaufer and M. Klusch. Wsmo-mx: A logic programming based hybrid service matchmaker. In *ECOWS '06: Proc. of the European Conference on Web Services*, pages 161–170, Washington, DC, USA, 2006. IEEE Computer Society.
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. L. Li and I. Horrocks. A software framework for matchmaking based on semantic technology. In *Proc. of WWW Conference*. ACM Press, 2003.
17. A. Martelli and L. Giordano. Reasoning About Web Services in a Temporal Action Logic. In *Reasoning, Action and Interaction in AI Theories and System*, number 4155 in LNAI, pages 229–246. Springer, 2006.
18. B. Medjahed and A. Bouguettaya. A multilevel composability model for semantic web services. *IEEE Trans. on Knowledge and Data Engineering*, 17(7):954–968, 2005.
19. B. Örrens, J. Yang, and M.P. Papazoglou. Model driven service composition. In *ICSOC 2003*, 2003.
20. OWL-S Coalition. http://www.daml.org/services/owl-s/.
21. M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic matching of web services capabilities. In *Proc. of ISWC'02*, pages 333–347. Springer, 2002.
22. M. Pistore, L. Spalazzi, and P. Traverso. A minimalist approach to semantic annotations for web processes compositions. In *ESWC*, pages 620–634, 2006.
23. S. K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *Proc. of CAV 2002*, volume 2404 of *LNCS*, pages 166–179. Springer, 2002.
24. UDDI, Universal Description, Discovery and Integration. http://www.uddi.org/.
25. W3C. Semantic Annotations for WSDL Working Group. http://www.w3.org/2002/ws/sawsdl/.
26. WS-CDL. http://www.w3.org/tr/ws-cdl-10/.
27. WSCI, Web Service Choreography Interface. http://www.w3.org/tr/wsci.
28. WSDL Message Exchange Patterns. http://www.w3.org/tr/2004/wd-wsdl20-patterns-20040326/.
29. WSDL, Web Service Description Language. http://www.w3.org/tr/wsdl.
30. A. Moormann Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. on Software Engineering and Methodology*, 6(4):333–369, 1997.