Integrating Abductive Logic Programming and Description Logics in a Dynamic Contracting Architecture

Marco Alberti[1], Massimiliano Cattafi[2], Federico Chesani[3], Marco Gavanelli[2], Evelina Lamma[2], Marco Montali[3], Paola Mello[3], Paolo Torroni[3]

[1]CENTRIA, Universidade Nova de Lisboa, Portugal
[2]ENDIF, Università di Ferrara, Italy
[3]DEIS, Università di Bologna, Italy

Abstract

In Semantic Web technologies, searching for a service means to identify components that can potentially satisfy the user needs in terms of outputs and effects (discovery), and that, when invoked by the customer, can fruitfully interact with her (contracting). In this paper, we present an application framework that encompasses both the discovery and the contracting steps, in a unified search process. In particular, we accommodate service discovery by ontology-based reasoning, and contracting by automated reasoning about policies published in a formal language. To this purpose, we consider a formal approach grounded on Computational Logic, and Abductive Logic Programming in particular. We propose a framework, called $\mathcal{S}CIFF$ Reasoning Engine, able to establish, by ontological and abductive reasoning, if a semantic web service and a requester can fruitfully inter-operate, taking as input the behavioural interfaces of both the participants, and producing as output a sort of a contract.

1. Introduction

Service Oriented Architecture (SOA) and Web Services are emerging as standard architectures for distributed application development. The adoption of well-known network protocols and communication standards has solved interoperability and heterogeneity issues. Eventually, the use of off-the-shelf solutions/services is becoming possible, although concerns about the adoption of such components have been raised. In particular, the search of software components on the basis of the functionality they provide, rather than on some syntactical property, is still an open research matter. To this end, some authors identify Semantic Web technologies as a promising way to address this issue [9, 12]. The idea is to augment web service descriptions by semantic information that can be used to search for *Semantic Web Services* (SWS, for short, in the following).

In our view, searching for a service means to identify such components that $i)$ can potentially satisfy the user needs in terms of outputs and effects, and $ii)$ can be invoked by the customer and interact with her without violating her interaction policies. An example of interaction policy could be a user constraint that prevents providing a credit card number to a service which is not certified, or a service constraint that prevents to accept credit card payments for items out of stock or with more than 30 % discount. Hence, a user request contains not only a description (given in semantic terms) of the user desires, but also the user policies, which constitute a declarative "behavioural interface."

We consider the search of a SWS as the process of identifying, among a given set of services, those components that both satisfy the ontological requirements (i.e., they provide the requested functionality), and the constraints on interaction (i.e., they support the requested behaviour). Following [9], we propose a two-fold search process. A first phase, called *discovery*, considers a requester's desires, and, using ontology-based reasoning, produces a selection of services that can potentially satisfy a request of such a kind. A second step, called *contracting*, matches the requester's interaction policies with those of every selected service, and establishes whether an interaction can be effectively achieved, and if the result matches the user goals.

The contracting phase requires reasoning about policies, that should be provided in a web-friendly language. Rule-based languages, such as RuleML (`http://www.ruleml.org/`) or the RIF (`http://www.w3.org/2005/rules`), have been advocated to enhance the semantic information associated to web content. Once the rules are published in a formal language, one can adopt the results obtained by the community of computational logics, that developed a plethora of languages, tools, and proof-methods for formal reasoning.

In previous work [1], we focused on the contracting step only, exploring the possibility of using a computational logic language to specify both the interaction policies of the requester and of the service. Then, by exploiting abductive reasoning, we showed how our approach could be a feasible solution for reasoning upon interactions. The solution proposed, called SCIFF Reasoning Engine (SRE), is a framework able to establish if a given SWS and a requester can fruitfully inter-operate, taking as input the behavioural interfaces of both the participants, and producing as output a sort of a contract (a plan).

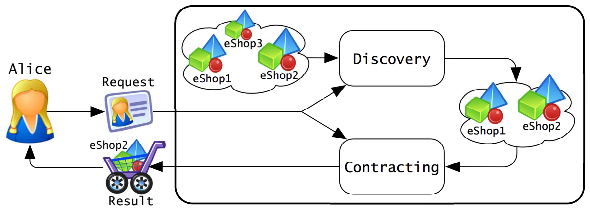

Figure 1. Framework overview

Building on the contracting framework extensively described in [1], in this paper we present the implemented application framework that encompasses both the discovery and the contracting steps, in a unified search process. In particular, we accommodate service discovery by ontology-based reasoning, specifically by matching the user inputs, outputs and effects, with "similar" inputs, outputs and effects of potentially suitable services. Similarity here can be intended as subsumption. Once a set of software components has been shortlisted from all the available services, in a second step the SRE evaluates which services can successfully interact with the user. Ontology-based reasoning can also be performed in this second stage, if necessary.

2 The description of a Semantic Web Service

Several, different solutions have been proposed in order to describe a Semantic Web Service, and a vast literature is available on the topic. However, up to now no solution has been widely accepted, and a proper standard for defining the semantics of a Web Service is still a matter of research. This is indeed, in our opinion, one of the biggest obstacles to the adoption of SWS standards.

The two major proposals, the Web Service Modeling Ontology (WSMO [18]) and the Semantic Markup for Web Services (OWL-S [11]) address both the ontological aspects and the behavioral issues, when describing a SWS. However, WSMO proposes a rigid structure, and the behavioural aspects are mainly defined on abstract state machines semantics. OWL-S instead is more flexible, and can be extended by the user: behavioural aspects are supported by allowing their definition using at least two languages (Knowledge Interchange Format, KIF [4] and Semantic Web Rule Language, SWRL [6]), plus the possibility of adding any required language. WSMO offers a complete suite of tools for editing, developing and testing SWS descriptions, while OWL-S comes as a general ontology, not associated with specific dedicated tools.

Our framework implements two strata of the semantic web cake: the ontology and the logical reasoning. They are internally represented with two different sets of information and stored in two different files. Ontological aspects are represented by means of an OWL-S 1.1 profile, while behavioural properties are defined using the $\mathcal{S}$CIFF language [2]. This allows us to keep the architecture open to other SWS description solutions, without giving up the powerful $\mathcal{S}$CIFF formalism for representing the interaction issues.

3 An example scenario

User *alice* forgot to buy her brother a Christmas present and now she is desperately searching the Internet for an on-line shop that sells the last crime fiction novel featuring detective Montalbano. She is particularly worried because she needs to find a shop that can deliver the book to Italy within 3 days. She can pay by cash or by credit card. She is also worried about frauds, so she will not provide her credit card number to any electronic shop, but only to those belonging to a Better Business Bureau (BBB).

$eShop1$ is the biggest Internet book seller, and through its semantic web services it provides every type of books. Its services are advertised with the generic term "book". Moreover, it has some policies about the delivery: fast delivery (one day) is allowed only if payment is performed by means of credit card; otherwise, standard delivery (one week) is the default option.

$eShop2$ is a small Internet seller, specialized in crime fiction books only. Its service advertisements use again the generic term "book," and it accepts "credit card" payment and "cash" payment. The shop delivers in two days but delivers only to customers in the European Union. It will prove its membership to the BBB on request.

$eShop3$ is a huge consumer electronics chain, which advertises its Internet service as "selling hardware." It accepts any payment method, supports delivery in 1 day, provides its membership to BBB each time the selected payment method is credit card.

We envisage a scenario in which *alice* queries a search engine, which will perform a discovery step; in this phase $eShop1$ and $eShop2$ are selected as possible services ($eShop3$ is discarded as it does not sell items related to the concept of "book"). However, this does not guarantee that an interaction is possible. Due to *alice*'s policy, the credit card number is provided only to those that are members of BBB, so only $eShop2$ remains viable. Feasibility of delivery, based on geographical criteria, is also to be checked.

In the next section, we briefly recall the $\mathcal{S}$CIFF-based contracting framework [1], and show how the policies of *alice* and $eShop2$ (called simply *eShop* in the following) can be represented in it. The policies of $eShop1$ and $eShop3$ can be represented in an analogous way.

4 The $\mathcal{S}$CIFF framework

A web service's policy is defined, in the $\mathcal{S}$CIFF language, as an abductive logic program (ALP). An ALP is defined as the triplet $\langle KB_S, \mathcal{A}, IC\rangle$. KB_S is a logic program in which the clauses can contain special atoms, that belong to the set $\mathcal{A}$ and are called *abducibles*. Such atoms are not defined in KB_S, and, as such, they cannot be proven: their truth value can be only hypothesized. In order to avoid unconstrained hypotheses, a set of *integrity constraints* (IC) must always be satisfied. Integrity constraints, in our language, are in the form of implications, and can relate abducible literals, defined literals, as well as constraints with Constraint Logic Programming [8] semantics.

We represent interaction as message sequences, by means of special abducible predicates: *events* **H**/4 represent message exchanges that one observes or controls (such as the sending of her own messages), and *expectations* **E**/4 represent messages that are expected to happen, such as another entity's messages. In both cases, the arguments are the message's sender, receiver, content and time.

An SWS' policies can be represented with integrity constraints, which relate the web services' messages with the expected input from peer web services. The possible relations can be of various types, including temporal relations (e.g., deadlines), linear constraints, disequalities and inequalities, all defined by means of constraints. Such definitions are then used to make assumptions on the possible evolutions of the interaction.

For example, *alice*'s policy states that if a shop asks to pay cash, *alice* will proceed with the payment:

$$\begin{array}{l}\mathbf{H}(S, alice, ask(pay(Item, cash)), T_a) \rightarrow \\ \mathbf{H}(alice, S, pay(Item, cash), T_r) \wedge T_a < T_r.\end{array}$$

As in all event and expectation atoms, the last parameter represent the time at which the event happens or is expected. In this case, the constraint $T_a < T_r$ imposes that the pay event follow the ask event.

If, instead, the payment is with credit card, then *alice* will require evidence of the shop's affiliation to the BBB.

$$\begin{array}{l}\mathbf{H}(S, alice, ask(pay(Item, cc)), T_a) \rightarrow \\ \mathbf{H}(alice, S, request\_guar(BBB), T_{rg}) \wedge T_{rg} > T_a \wedge \\ \mathbf{E}(S, alice, give\_guar(BBB), T_g) \wedge T_g > T_{rg}.\end{array} \quad (1)$$

$$\begin{array}{l}\mathbf{H}(S, alice, ask(pay(Item, cc)), T_a) \wedge \\ \mathbf{H}(S, alice, give\_guar(BBB), T_g) \rightarrow \\ \mathbf{H}(alice, S, pay(Item, cc), T_p) \wedge T_p > T_a \wedge T_p > T_g.\end{array} \quad (2)$$

The policy of the $eShop2$ is also represented through integrity constraints. "If an acceptable customer requests an item, then I expect the customer to pay for the item with an acceptable payment method. If the customer is not acceptable, I will inform him/her of the failure. If an acceptable customer pays with an acceptable means of payment, I will deliver the item within two days. If a customer requests evidence of my affiliation to the BBB, I will provide it."

$$\begin{array}{l}\mathbf{H}(C, eShop, request(Item), T_r) \\ \rightarrow accepted\_customer(C) \wedge accepted\_pay(How) \\ \quad \wedge \mathbf{H}(eShop, C, ask(pay(Item, How)), T_a) \wedge T_a > T_r \\ \quad \wedge \mathbf{E}(C, eShop, pay(Item, How), T_p) \wedge T_p > T_a \\ \vee\, rejected\_customer(C) \\ \quad \wedge \mathbf{H}(eShop, C, inform(fail), T_i) \wedge T_i > T_r.\end{array} \quad (3)$$

$$\begin{array}{l}\mathbf{H}(C, eShop, pay(Item, How), T_p) \\ \wedge accepted\_customer(C) \wedge accepted\_pay(How) \\ \rightarrow \mathbf{H}(eShop, C, deliver(Item), T_d) \wedge T_p < T_d < T_p + 2.\end{array}$$

$$\begin{array}{l}\mathbf{H}(C, eShop, request\_guar(BBB), T_{rg}) \\ \rightarrow \mathbf{H}(eShop, C, give\_guar(BBB), T_g) \wedge T_g > T_{rg}.\end{array} \quad (4)$$

The notion of acceptability for customers and payment methods from *eShop*'s viewpoint, given with the *accepted_customer/1* and *accepted_pay/1* predicates, can be defined in *eShop*'s knowledge base, as we proposed in a previous work [1]. The fact that only EU residents are accepted customers, can be defined by the following clauses:

$$\begin{array}{l}accepted\_customer(C) : -resident\_in(C, L), \\ \quad accepted\_destination(L). \\ rejected\_customer(C) : -resident\_in(C, L), \\ \quad not\ accepted\_destination(L). \\ accepted\_destination(european\_union). \\ accepted\_pay(cc).\ accepted\_pay(cash).\end{array} \quad (5)$$

On her side, *alice* knows she is resident in the EU:

$$resident\_in(alice, european\_union). \quad (6)$$

Operationally, the $\mathcal{S}$CIFF reasoning engine (SRE) joins the knowledge bases of the customer with that of the candidate shops. It starts with *alice*'s goal, namely obtaining a *book* by interacting with a shop: *alice* will start an interaction by $request$ing the book, and will expect the shop to $deliver$ it:

$$\mathbf{H}(alice, eShop, request(book), 0) \wedge$$
$$\mathbf{E}_{alice}(eShop, alice, deliver(book), T_d), 0 \le T_d \le 3$$

SRE reasons about events and expectations, and tries to match them in order to find a successful arrangement. For this reason, it tags each expectation with its holder: in this example, *alice* is the entity that holds the expectation of *eShop* delivering the book.

Now, the happened $request$ event triggers new integrity constraints. In particular, it activates the IC (3) of the *eShop*'s policy, that can be satisfied in two alternative ways: either the transaction succeeds or fails. The $\mathcal{S}$CIFF proof-procedure generates a proof tree; typical implementations adopt a depth-first search strategy. In the first branch, $\mathcal{S}$CIFF verifies that *alice* is an acceptable customer (that can be proven by joining *alice* and *eShop* knowledge bases), then abduces that *eShop* will ask for payment with one of the accepted payments. Let us consider the case in which payment with cc is assumed; in this case, *eShop* will ask for the payment and expect *alice* to perform it:

$$\mathbf{H}(eShop, alice, ask(pay(book, cc)), T_a)$$
$$\mathbf{E}_{eShop}(alice, eShop, pay(book, cc), T_p).$$

The new abduced event makes the body of rule (2) true: SRE assumes that *alice* will follow her own policy, by requesting the guarantee and expecting a reply. The request event will activate rule (4) that forces *eShop* to provide it.

In this way, a set of events and one of expectations are generated by abduction. If the expectations are matched by corresponding events, the current branch succeeds, otherwise it fails, and another alternative will be selected (if there exists one). In this way, the $\mathcal{S}$CIFF proof-procedure is able to find if there exists at least a set of events that satisfies a given ALP, and provides in output both the abduced events and expectations. A successful computation yields a sequence of actions that satisfies all the parties' policies.

Representing customer acceptability with KB clauses, however, is prolematic. For example, it would not work in case the customer declared `resident_in(alice,italy)`, as the term `italy` does not syntactically unify with `european_union`. One could, of course, add to the knowledge base `accepted_destination/1` facts for all the EU members, but such knowledge should be updated locally when new countries join the EU. In other cases, acceptability could be defined by a transitive, symmetric relation. For example, a service could accept requests from a set of trusted peers, and also from customers that are trusted by them, in a transitive fashion. If the abductive proof procedure adopts a depth-first search, symmetric and/or transitive relations can lead to loops. A solution (which, as shown in the following, also brings performance improvement) is to use ontological reasoning, as described in the next section.

4.1. Representing domain knowledge with ontologies

An alternative way to represent (part of) the domain specific knowledge is to use technology and concepts developed, with focus on this very purpose, in the Knowledge Representation field, and to rely, in particular, on ontologies. Ontologies are one of the layers of the semantic web cake, are more and more used on the web, and, in significant cases, they support reasoning in polynomial time. The W3C recommendation for ontology representation on the Web is the Web Ontology Language (OWL) [3] based on the well established semantics of Description Logics [10] and on XML and RDF syntax. Using OWL for domain knowledge representation improves expressivity (with such features as stating subclassing relations, constructing classes on property restrictions or by set operators, defining transitive properties and so on) yet keeping decidability (if using OWL Lite or OWL DL) in a straightforward and domain modeling-oriented notation. Moreover, since OWL is tailored for the Web, it provides support for expressing knowledge in distributed contexts (identified by URIs) and its recognized standard status is a warranty on interoperability and reusability issues. As a plus, it can be mentioned that community driven development of Semantic Web tools provides already good support for OWL ontology management tasks such as editing [14] also for not KR-skilled users.

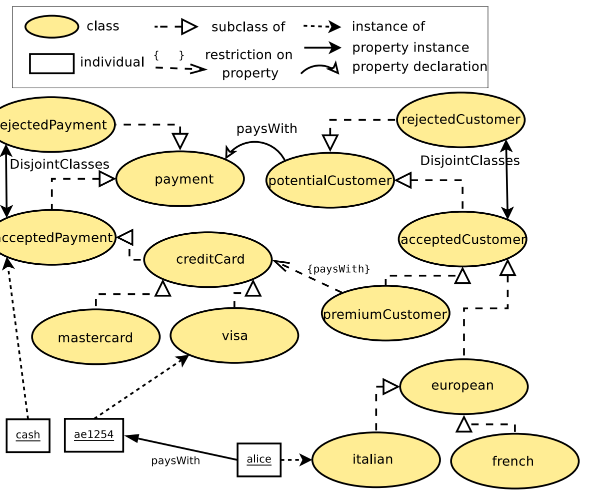

Figure 2. A graphical representation of the ontology

In Fig. 2 we show a possible ontological representation of *eShop*'s policies concerning acceptable customers and means of payments, merged with *alice*'s own knowledge. For example, we can state that `acceptedCustomer` is a subclass of the `potentialCustomer` class, and that it is disjoint from the `rejectedCustomer` class with the following OWL syntax:

```
<owl:Class rdf:about="#acceptedCustomer">
 <rdfs:subClassOf
     rdf:resource="#potentialCustomer" />
 <owl:disjointWith
      rdf:resource="#rejectedCustomer" />
</owl:Class>
```

The following assertion states that `cash` is an instance of the `acceptedPayment` class:

```
<owl:Thing rdf:about="#cash">
 <rdf:type
      rdf:resource="#acceptedPayment" />
</owl:Thing>
```

The following is the declaration of the `paysWith` property:

```
<owl:ObjectProperty rdf:ID="paysWith">
 <rdfs:domain
   rdf:resource="#potentialCustomer" />
 <rdfs:range rdf:resource="#payment" />
</owl:ObjectProperty>
```

The following assertion states that `alice` is an instance of `italian`, with value `ae1254` for the `paysWith` property:

```
<owl:Thing rdf:about="#alice">
 <rdf:type rdf:resource="#italian" />
 <paysWith rdf:resource="#ae1254" />
</owl:Thing>
```

Now *alice* no longer needs to express explicitly that she is resident in the European Union. Simply declaring that she is from Italy, and providing her ontology (or, possibly, the official ontology of the EU, containing all the member states), ontological reasoning is able to infer that she is European.

Another interesting feature of Description Logic (and thus OWL) ontologies is the definition of classes using restrictions on properties. For instance we could define a class, `premiumCustomer`, representing the accepted customers who pay with a credit card. It could be then used to add refinements to policies (for instance providing to customers belonging to this class a faster delivery or a lower price) and since `alice` is an accepted customer and pays with her credit card, the ontological reasoning would automatically recognize her as a `premiumCustomer`.

5. Discovery and Contracting with $\mathcal{S}$CIFF

As in [9], we distinguish between the discovery step and the contracting step. During the discovery step, the user request for a service is compared with each SWS description, and possible services are selected. In this phase the main problem is that the terms/concepts used in the request could differ from those used in the service description. In particular, two apparently different terms could actually refer to the same concept, or to different but related concepts (e.g., one concept could be subsumed by the other). These are typical ontological problems, and several different ontological reasoners are available to cope with such issues. At the end of the discovery phase, a set of services that might fulfil the user requirements is provided as result. Such a set will be the input for the next phase.

The contracting phase then focuses on the interaction policies, i.e., on the set of rules that each partner has declared as representing its behavioural interface. Here the problem consists of deciding whether an interaction can effectively happen, achieving at the same time the user goals.

5.1. Discovery

In our framework the discovery phase is implemented following the algorithm by Paolucci et al. [15]. Given a client's request, discovery is conceived as the problem of selecting those services that might satisfy the client's needs. The client publishes her needs in term of information she is willing to provide as input to the service, and in term of outputs she expects from the service. Similarly, each service advertises its own capabilities as a list of information she requires in input, paired with the information she will provide as output. Each piece of information represents a parameter, and the discovery problem can be intended as looking for those services whose input (output) parameters *match* the input (output) parameters of the client. The client's request is confronted with every service description available, and a set of candidate services is returned to the client.

Paolucci et al. assume that each parameter is described by means of ontological propositions (in our approach the parameters are defined in terms of OWL-S concepts; we assume that, if providers refer to different ontologies, equivalences between concepts have already been established). The parameters of each available service profile are checked against the parameters in the client's request. To decide if two parameters *match*, we use an ontological criterion, namely *subsumption*, as in [15]; however, the matching algorithm could use other criteria, for example based on semantic contexts, as long as a degree of similarity is provided by the criteria, rather than a yes/no answer.

Paolucci and colleagues propose four different matching levels, depending on the subsumption relation: *exact*, if it is possible to establish that two parameters defined with different terms refer both to the same concept; *plugin* and *subsume* if a parameter is subsumed/subsumes the other; and *fail* if no subsumption relation can be identified among two parameters.

Our implementation differs slightly from the algorithm in [15], in order to provide more flexibility. The original algorithm requires the number of input/output parameters to be exactly the same in the client request and in the service profile, in order for a service to be discovered. In our system instead, if a service provides more outputs or requires less inputs than those stated in the client's query, such a service is selected anyway. The reason for this choice is to give more choices to the following contracting phase. The contract could be satisfactory for both parties even if the client discards some of the outputs of the service, or if it does provide an input disregarded by the service. Suppose, e.g., that $eShop2$ is sponsoring a marketing campaign to attract new customers: together with each book bought on its web site, $eShop2$ will provide also a free voucher of 10$ valid for the next order. The algorithm in its original form would disregard $eShop2$ since it provides both a book and a voucher, while $alice$ is looking for a book only. We include $eShop2$ in the set of discovered services, following the intuition that $alice$ can freely decide to use the voucher or not.

5.2. Contracting

The contracting phase is performed through the $\mathcal{S}$CIFF reasoning engine. As explained in Section 4, SRE tries to establish if there exists a possible sequence of events (exchanged messages) that respect the constraints of both the service and the user. If such interaction is possible, SRE comes up with a sort of a plan indicating the messages that should be exchanged. Note that the reasoning process is driven by the user goals: not all the possible interactions are of interest, but only those that satisfy the user's needs.

In this work, the SRE framework [1] is extended to cope with the ontological layer of the semantic web. In fact, we do not limit the use of ontologies to the discovery phase, but extend it also to the contracting phase. This is useful both to provide precise meaning to the terms used in the contract, and to exploit efficient, polynomial reasoners for the parts regarding terms. In particular, we intertwine the $\mathcal{S}$CIFF proof procedure (that deals with the rules part) and the Pellet reasoner (that deals with ontologies).

5.3 Interfacing $\mathcal{S}$CIFF and ontological reasoners

A possible approach to let $\mathcal{S}$CIFF access ontological knowledge is to exploit the common root of logic programming and description logics in first order logic, by finding their intersection and translating ontologies to LP clauses, following Grosof et al. [5] and Hustadt et al. [7]. For example, it is possible to use dlpconvert [13] to translate domain knowledge described in OWL to $\mathcal{S}$CIFF clauses. Reasoning is then performed by $\mathcal{S}$CIFF in the usual way. However, this solution limits ontological expressivity, since the DLP fragment covered by dlpconvert is a proper subset of DL, and some OWL axioms are not included. Moreover, some axioms translations are not suitable for reasoning with goal-driven operational semantics, such as as resolution or unfolding, employed in $\mathcal{S}$CIFF, because it leads to loops.

A different (and more effective) approach consists of interfacing $\mathcal{S}$CIFF with an external specific ontology-focused component which can be queried by $\mathcal{S}$CIFF and which performs the actual ontological reasoning. As represented in Fig. 3, this solution involves a Prolog meta-predicate which invokes the ontological reasoning on desired goals, an intercommunication interface from $\mathcal{S}$CIFF to the external component (which incorporates a query and results translation schema) and the actual reasoning module. Both modules can access both local and networked knowledge.

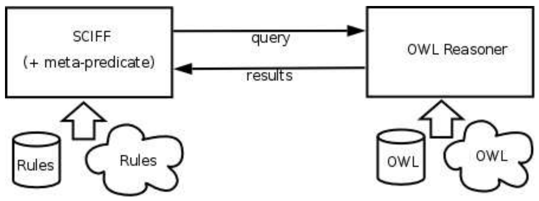

Figure 3. Integration architecture

Goals given to the meta-predicate are handled, as suggested in [7, 20], considering single arity predicates as "belongs to class (with same name of predicate)" queries and double arity ones as "are related by property (with same name of predicate)" queries. To reduce the overhead caused by external communication, we implemented a caching mechanism: the meta-predicate first checks if a similar query (i.e., involving the same predicate) has been issued before and, only if not, it invokes the external reasoner and stores its answers as Prolog facts. The OWL reasoning module uses the Pellet [16] API, while the communication interface uses the Jasper Prolog-Java library [19]. This solution provides full OWL(-DL) expressivity, including features such as equivalence of classes and properties, transitive properties, declaration of classes on property restriction and property-based individual classification.

We tested the approach in simple contracting scenarios, where the overhead was hardly measurable. However, to assess scalability, we experimented with randomly generated ontologies. Each ontology, composed of N classes, was built starting from its root node, and recursively trying, for each node, five attempts of child generation, each with probability $1/3$. Table 1 compares the implementation based on the meta-interpreter with one in which the whole ontology was converted into $\mathcal{S}$CIFF syntax and then imported into the $\mathcal{S}$CIFF knowledge base (through `dlpconvert` [13]). For both approaches, we report the time spent for loading

	Ontology import			Interface with Pellet		
N	Load	Query	Total	Load	Query	Total
100	3.4	~ 0	3.4	~ 0	~ 0	~ 0
500	5.8	~ 0	5.8	1.0	~ 0	1.0
1000	8.2	~ 0	8.2	1.0	~ 0	1.0
5000	14.9	~ 0	14.9	2.0	1.2	3.2
10000	26.6	~ 0	26.6	4.0	2.8	6.8

Table 1. Performance comparison

the ontology into the reasoner[1] and for the actual query (PC with Intel Celeron 2.4 GHz CPU, times in seconds, average over 50 runs). Both approaches appear to scale reasonably. In both cases, the loading time is higher than the query time. On the other hand, importing is possible only for those ontologies that can be expressed in DLP [5]. In conclusion, the interface approach appears to dominate the importing approach both in expressivity and in performance.

6. Architecture

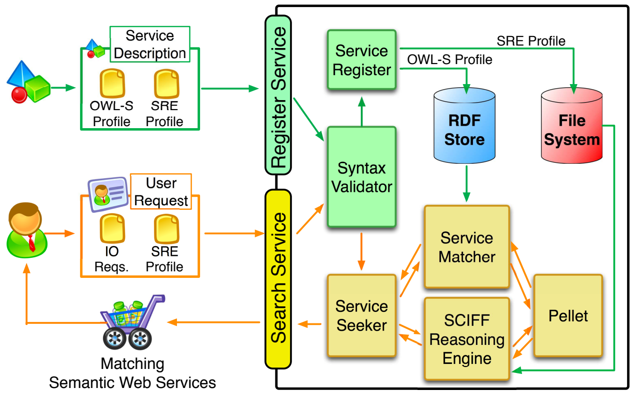

Figure 4. The Framework architecture

The framework is organized as a set of web services. The aim was a modular architecture, with the intention of using the framework as a proof of concept of the SRE approach. Our framework provides two facilities: registering and querying. The first is used by service providers, which register by providing a service description in terms of the pair (OWL-S profile, SRE profile). The second facility accepts requests from the users, and returns a list of SWS fulfilling the requirements.

The web services composing the system are shown in Figure 4 (where arrows represent data flows). A Web Service can register at our application, by providing an OWL-S description together with a SRE description of the provided functionality and its behavioural policies. After a syntactic validation performed by the *Syntax Validation* module, the service description is sent to the *Service Register* component which manages the storing procedures. It stores OWL-S profiles by means of a RDF store, while SRE profiles are directly stored in the file system. OWL-S profiles are preprocessed, and a summary of the profile is extracted for each SWS; in this way, some specific cases can be directly identified and handled, simplifying the matching algorithm described in Section 5.1. If the storing procedure terminates successfully, an acknowledgement is returned to the service asking for registration.

A user starts the process with a request, composed of a description of the functionality she is looking for and her own behavioural policies. The desired service is described in terms of inputs and outputs: however we assume such lists as a sort of “indication” of the needs of the user, and a certain flexibility is adopted, as explained in Section 5.1. After a syntactic validation step, the request is passed to the *Service Seeker* component, which coordinates the search process orchestrating the other components. The input/output list is passed to the *Service Matcher* component that selects, among the registered services, the ones that could satisfy the user request, with the algorithm explained in Section 5.1. The ontology subsumption relation is evaluated by the *MatchMaker* component, a simple wrapper for the Pellet reasoner [16].

The list of selected services is returned to the *Service Seeker*, that in turn gives it to the SCIFF Reasoning Engine module. Such module reasons about the existence of a possible interaction that could satisfy the user needs, as explained in Section 5.2. The result is a restricted list of services , for each selected service, a possible interaction plan that justifies why that service has been selected, and shows how the user can successfully interact with the service. To perform ontological reasoning, also the SCIFF Reasoning Engine module uses the *MatchMaker* and its integrated Pellet reasoner. Finally, the list of selected services is returned to the user by the *Service Seeker* module.

7. Related Work

Many authors tackled the service matching problem, considering both ontological and interaction perspectives. Kifer et al. [9] propose a comprehensive solution to the discovery and contracting problem. Our proposal resembles the solution proposed by Kifer and colleagues: both solutions are based on declarative approaches (they rely on F-Logic while we build up on extended logic programming); both use hypothetical reasoning to solve the contracting problem. However, in [9] the contracting problem is solved by taking into account only the user’s goal, while in our ap-

[1]For the importing approach, the load time is the time spent for translating the ontology and parsing the resulting clauses and ICs, while for the Pellet-based approach it is the time spent for parsing ICs and loading the ontology into a persistent OWLOntology object.

proach the user can also specify policies that constrain how the goal can be achieved. The use of SRE, and in particular of the underlying $\mathcal{S}$CIFF language and proof procedure, provides a great expressivity when defining the policies, with the typical advantages of declarative approaches and of a solid underlying computational counterpart. Ragone et al. [17] use the idea of Concept Covering and Concept Abduction to overcome some of the limits of previous matching approaches, and to address also the composition problem. In this work we focus on discovering a SWS able to satisfy the user requests, and we concentrate our efforts instead on reasoning about the interaction aspects: in this perspective, we understand a SWS as a complex agent, for which the interaction aspects play an important role.

8 Discussion and Conclusion

The architecture introduced in this paper is a first prototype, suffering from many limitations we introduced to support flexibility and extensibility. E.g., we currently store service profiles using a relational DBMS. This enables a certain flexibility, but does not permit to take advantage of ontology-aware storing systems. We assumed also that all the service profiles and the user requests refer to the same set of ontologies, and that no ontology alignment is required.

Another limit of the current implementation is that the discovery phase mostly relies upon the algorithm in [15], where only input and output parameters are considered while looking for a match. However, OWL-S profiles and other proposals let the user specify a service profile also in terms of pre-conditions and effects.

Nevertheless, the results of our preliminary test using OWL-S profiles at `projects.semwebcentral.org` are encouraging. They show that our proposal for the service discovery and contracting problem is a viable solution, with the advantages of offering a powerful yet simple, declarative language for expressing service policies.

In the future, we intend to run a more comprehensive comparison with other solutions, by considering both expressive power and computational performance. We also plan to extend our application by providing support to other service description languages, such as WSMO, SAWSDL and WSMO-Lite. Finally, we plan to encode SRE rules using emerging standards such as the Rule Interchange Format (RIF) and its Framework for Logic Dialects (RIF-FLD).

Acknowledgments

This work has been partially supported by the FIRB RBNE05BFRK project TOCAI.it.

References

[1] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, M. Montali, and P. Torroni. Web service contracting: specification and reasoning with SCIFF. In E. Franconi, M. Kifer, and W. May, editors, *ESWC*, volume 4519 of *LNAI*, 2007.

[2] M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logics*, 9(4), 2008.

[3] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. OWL web ontology language reference. W3C Recommendation, 2004.

[4] M. Genesereth and R. Fikes. Knowledge interchange format version 3.0 reference manual.

[5] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: combining logic programs with description logic. In *WWW '03*, pages 48–57. ACM, 2003.

[6] I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL. W3C submission, 2004.

[7] U. Hustadt, B. Motik, and U. Sattler. Reducing $\mathcal{SHIQ}^-$ description logic to disjunctive datalog programs. In D. Dubois, C. Welty, and M.-A. Williams, editors, *KR2004*, June 2004.

[8] J. Jaffar and M. Maher. Constraint logic programming: a survey. *J. of Logic Programming*, 19-20:503–582, 1994.

[9] M. Kifer, R. Lara, A. Polleres, C. Zhao, U. Keller, H. Lausen, and D. Fensel. A logical framework for web service discovery. In D. Martin, R. Lara, and T. Yamaguchi, editors, *SWS*, volume 119 of *CEUR Workshop Proc.*, 2004.

[10] C. Lutz. Description logic resources. `dl.kr.org`, 2008.

[11] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara. Bringing semantics to web services: the OWL-S approach. In J. Cardoso, A. Sheth, L. Kalinichenko, and F. Curbera, editors, *Proc. of SWSWPC*, 2004.

[12] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[13] B. Motik, D. Vrandečić, P. Hitzler, Y. Sure, and R. Studer. Dlpconvert - Converting OWL DLP statements to logic programs. System Demo at the 2nd ESWC, May 2005.

[14] N. F. Noy, M. Sintek, S. Decker, M. Crubézy, R. W. Fergerson, and M. A. Musen. Creating semantic web contents with Protégé-2000. *IEEE Int. Systems*, 16(2):60–71, 2001.

[15] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic matching of web services capabilities. In I. Horrocks and J. A. Hendler, editors, *ISWC*, 2002.

[16] B. Parsia and E. Sirin. Pellet: An OWL DL reasoner. In F. van Harmelen, editor, *ISWC 2004*, 2004.

[17] A. Ragone, T. Di Noia, E. Di Sciascio, F. Donini, S. Colucci, and F. Colasuonno. Fully automated web services discovery and composition through concept covering and concept abduction. *Int. J. Web Service Res.*, 4(3):85–112, 2007.

[18] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M.Stollberg, A.Polleres, C.Feier, C.Bussler, and D. Fensel. Web service modeling ontology. *Appl. Ontology*, 1(1), 2005.

[19] SICStus Prolog. `http://www.sics.se/sicstus`.

[20] D. Vrandečić, P. Haase, P. Hitzler, Y. Sure, and R. Studer. DLP-an introduction. Tech.Rep., Univ. Karlsruhe, 2006.