Exploiting Semantic Technology in Computational Logic-based Service Contracting

Marco Alberti, Massimiliano Cattafi, Marco Gavanelli, and Evelina Lamma

ENDIF, University of Ferrara
Via Saragat, 1 - 44100 Ferrara (Italy)
marco.alberti@unife.it
massimiliano.cattafi@student.unife.it
marco.gavanelli@unife.it
evelina.lamma@unife.it

Abstract. Dynamic composition of web services requires an automated step of contracting, i.e., the computation of a possibly fruitful interaction between two (or more) services, based on their policies and goals. In previous work, the $\mathcal{S}$CIFF abductive logic language was used to represent the services' policies, and the associated proof procedure to perform the contracting.
In this paper, we build on that work in order to exploit the results of the Description Logics research area to represent domain specific knowledge, either by importing the knowledge encoded in an ontology into a $\mathcal{S}$CIFF knowledge base, or by interfacing the $\mathcal{S}$CIFF proof procedure to an existing ontological reasoner.

1 Introduction

Service Oriented Architectures, exploiting the Internet as a means of communication, are emerging as a simple and effective paradigm for distributed application development. Heterogeneous entities, in terms of hardware and software settings, can interoperate effectively following well established communication standards.

Service providers can expose their policies, in order for potential customers to evaluate the feasibility of a fruitful interaction. Such an evaluation can be performed manually, if the service's interface is bound not to change over time.

However, a more promising approach is for customers to choose the appropriate service provider at run-time, based on the service provider's exposed policies.

This approach requires a reasoning engine that reasons on the provider and customer's goals and policies, in order to devise a sequence of actions that lets both achieve their goals, while respecting their policies.

In [1], Alberti et al. proposed a contracting architecture based on the $\mathcal{S}$CIFF abductive logic framework [2]. In that work, the policies were expressed by means of integrity constraints, and the domain knowledge was expressed in $\mathcal{S}$CIFF's logic clauses. The sound and complete $\mathcal{S}$CIFF proof procedure was employed to find, if possible, a course of action that was satisfactory for both.

However, in a practical perspective, we envisage a possible improvement in representing the domain specific knowledge exploiting the vast body of results from the Knowledge Representation field and, in particular, Description Logics. In this way, we can address the Ontology layer of the Semantic Web stack.

In this paper, we propose two different approaches to let $\mathcal{S}$CIFF access existing knowledge, expressed by means of an ontology:

- by translating (part of) an ontology into $\mathcal{S}$CIFF clauses, which can subsequently be handled by the $\mathcal{S}$CIFF proof procedure in the usual way;
- by interfacing $\mathcal{S}$CIFF with ontological reasoners, distinguishing syntactically the ontology-related predicates and delegating their computation to the external reasoners.

This approach bears two main practical advantages: first, it makes the knowledge encoded in ontologies available to $\mathcal{S}$CIFF, and second (with the second approach) it takes advantage of the formal properties enjoyed by the existing ontological reasoners for the associated computational tasks, in particular concerning decidability and efficiency.

Our approach differs from the one proposed by Matzner and Hitzler in PrOWLog [3] which integrates OWL into Prolog by means of the any-world-assumption, in that we do not propose an embedding of the OWL language into a logic programming framework.

An integration of ontological reasoning into $\mathcal{S}$CIFF, following the ontology importing approach, is also proposed by Chesani et al. [4].

The paper is structured as follows: in Sect. 2 we briefly recall the $\mathcal{S}$CIFF approach to contracting, in Sect. 3 we show how the relevant domain knowledge can be expressed by means of an ontology, and in Sect. 4 we describe the two different approaches to access ontologies from $\mathcal{S}$CIFF: importing an ontology into a $\mathcal{S}$CIFF knowledge base (Sect. 4.1), and interfacing the $\mathcal{S}$CIFF proof procedure with the Pellet [5] ontological reasoner (Sect. 4.2). Conclusions follow.

2 Contracting with $\mathcal{S}$CIFF

In this section, we briefly recall the $\mathcal{S}$CIFF-based contracting framework [1].

2.1 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 $\langle KB_S, \mathcal{A}, IC \rangle$, where 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 by means of clauses in the 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 [6] semantics.

In particular, the integrity constraints can relate the web services' information exchanges with the expected input from peer web services. The possible relations can be of various types, and include temporal relations, such as deadlines, linear constraints, disequalities and inequalities, all defined by means of constraints. The definitions stated in this way are then used to make assumptions on the possible evolutions of the interaction.

A web service can reason about *happened* events, denoted with $\mathbf{H}(S, R, M, T)$, meaning that a Sender S sends message M to a Receiver R in some time T. All the parameters can be logical terms, or variables, possibly subject to constraints. Also, the policy might describe that a peer should provide some input, by sending a message. Such *expectation* of the incoming message is represented as $\mathbf{E}(S, R, M, T)$, and it will be satisfied in case the message, in the form of $\mathbf{H}$ atom, actually arrives. In some cases, the *expectation* could be *negative*, as in $\mathbf{EN}(S, R, M, T)$: the policy says in this case that a matching $\mathbf{H}$ event would be unwelcome, as it would contradict other assumptions.

For example, an integrity constraint can state that, if a peer web service sends me a request, then I will accept:

$$\mathbf{H}(S, me, request(X), T) \rightarrow \mathbf{H}(me, S, accept(X), T_a).$$

Another rule can say that, if I send a request, I expect the peer either to accept or to refuse, within 5 time units:

$$\begin{aligned} \mathbf{H}(me, S, request(X), T) \rightarrow\ & \mathbf{E}(S, me, accept(X), T_a), T_a < T + 5 \\ & \mathbf{E}(S, me, refuse(X), T_r), T_r < T + 5 \end{aligned}$$

Once the specification of a policy is defined by means of integrity constraints and expectations, they are processed by the $\mathcal{S}$CIFF proof procedure, the operational counterpart of the language. The proof-procedure performs abductive reasoning, i.e., it can make assumptions and generate hypotheses. In particular, it can hypothesise that some message will be sent, or that some expectation will be raised. In case the expectations are matched by corresponding events, the abductive process succeeds, otherwise the current branch will fail, 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.

In contracting applications, the potential interacting services expose their policies (in a format resulting from the extension of RuleML), and $\mathcal{S}$CIFF operates on the merging of the policies. A successful computation yields a sequence of actions that satisfies all the parties' policies.

2.2 A contracting scenario

In [1] the $\mathcal{S}$CIFF framework is applied to contracting in an e-commerce scenario, which we extend in this paper in order to demonstrate our approach to integration.

The two actors are *eShop* (which wants to sell a device) and *alice* (a potential customer for that device), each with policies expressed as $\mathcal{S}$CIFF ICs. $\mathcal{S}$CIFF is used as a reasoner in a component that acts as a mediator, considering the actors' policies and trying to devise a course of action that will let both reach their goals. The two actors' policies, expressed in $\mathcal{S}$CIFF integrity constraints, are as follows.

alice*'s policy.* "If the shop asks me to pay cash, I will, but if the shop asks me to pay by credit card, I will require evidence of the shop's affiliation to the Better Business Bureau."

$$\begin{array}{l}
\mathbf{H}(tell(Shop, alice, ask(pay(Item, cc))), Ta) \rightarrow \\
\mathbf{H}(tell(alice, Shop, request_guar(BBB)), Trg) \wedge \\
\mathbf{E}(tell(Shop, alice, give_guar(BBB)), Tg) \wedge Tg > Trg \wedge Trg > Ta. \\
\\
\mathbf{H}(tell(Shop, alice, ask(pay(Item, cc))), Ta) \wedge \\
\mathbf{H}(tell(Shop, alice, give_guar(BBB)), Tg) \rightarrow \\
\mathbf{H}(tell(alice, Shop, pay(Item, cc)), Tp) \wedge Tp > Ta \wedge Tp > Tg. \\
\\
\mathbf{H}(tell(Shop, alice, ask(pay(Item, cash))), Ta) \rightarrow \\
\mathbf{H}(tell(alice, eShop, pay(Item, cash)), Tr) \wedge Ta < Tr.
\end{array} \tag{1}$$

eShop*'s policy.* "If an acceptable customer requests an item from me, then I expect the customer to pay for the item with an acceptable means of payment. 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. If a customer requests evidence of my affiliation to the Better Business Bureau (BBB), I will provide it."

$$\begin{array}{l}
\mathbf{H}(tell(Customer, eShop, request(Item)), Tr) \wedge accepted_payment(How) \rightarrow \\
accepted_customer(Customer) \wedge \mathbf{H}(tell(eShop, Customer, ask(pay(Item, How))), Ta) \wedge \\
\mathbf{E}(tell(Customer, eShop, pay(Item, How)), Tp) \wedge Tp > Ta \wedge Ta > Tr \\
\vee\, rejected_customer(Customer) \wedge \mathbf{H}(tell(eShop, Customer, inform(fail)), Ti) \wedge Ti > Tr. \\
\\
\mathbf{H}(tell(Customer, eShop, pay(Item, How)), Tp) \\
\wedge\, accepted_customer(Customer) \wedge accepted_payment(How) \rightarrow \\
\mathbf{H}(tell(eShop, Customer, deliver(Item)), Td) \wedge Td > Tp. \\
\\
\mathbf{H}(tell(Customer, eShop, request_guar(BBB)), Trg) \rightarrow \\
\mathbf{H}(tell(eShop, Customer, give_guar(BBB)), Tg) \wedge Tg > Trg.
\end{array} \tag{2}$$

The notion of acceptability for customers and payment methods from *eShop*'s viewpoint, defined by the *accepted_customer/1* and *accepted_payment/1* predicates, is defined in *eShop*'s knowledge base. In [1], only EU residents are accepted customers, as defined by the following clauses:

```
accepted_customer(Customer):-
    resident_in(Customer, Location),
    accepted_destination(Location).
rejected_customer(Customer):-
    resident_in(Customer, Location),
    not accepted_destination(Location).
accepted_destination(european_union).
accepted_payment(cc).
accepted_payment(cash).
```

This knowledge is merged with that provided by the customer, for example

```
resident_in(alice,european_union).
```

so that in the resulting knowledge base, on which $\mathcal{S}$CIFF operates, `accepted_customer(alice)` is true.

However, this method would not work in case the customer declared `resident_in(alice,italy)`, as `italy` does not unify with `european_union`. One could, of course, add to the knowledge base `accepted_destination/1` facts for all the current EU members, but such knowledge should be updated when new countries join the EU. In other cases, acceptability could be defined by a transitive, symmetric relation, which could introduce cycles in the knowledge base, possibly leading to loops.

3 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. The W3C recommendation for ontology representation on the Web is the Web Ontology Language (OWL) [7] based on the well established semantics of Description Logics [8] 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 straight-forward 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 reuse issues. As a plus, it can be mentioned that community driven development of Semantic Web

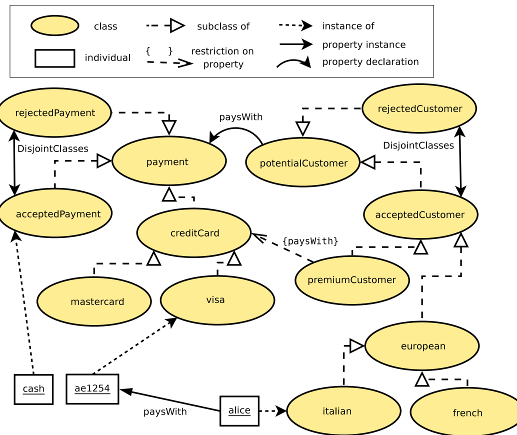

Fig. 1. A graphical representation of the ontology

tools already provides good support for OWL ontology management tasks such as editing [9] also for not KR-skilled users.

For instance, in Fig. 1 we show a possible ontological representation of *eShop*'s policies concerning acceptable customers and means of payments, merged with *alice*'s own knowledge.

The following OWL clause says that the `acceptedCustomer` class is a subclass of the `potentialCustomer` class, and that it is disjoint from the `rejectedCustomer` class:

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

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

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

The following clause declares the `paysWith` property:

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

The following clause 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>
```

It can be noticed that the ontological notation for the KB makes it possible to infer that `alice` is `European` (and therefore an accepted customer) even if she just states being `Italian`, while if we had put `resident_in(alice, italy)` instead of `resident_in(alice, europe)` in the KB at the end of Sect. 2.2 she would not have been recognized as such.

Moreover, we defined a class `premiumCustomer` representing the accepted customers who pay with a credit card, and which we could use to add refinement to policies (for instance providing a faster delivery). Since `alice` is an accepted customer and pays with her credit card, the ontological reasoning allows to recognize her as a `premiumCustomer`.

4 Handling semantic knowledge with $\mathcal{S}$CIFF

We implemented the $\mathcal{S}$CIFF access to OWL ontologies in two different ways, that we describe in this section. In both cases it should be noticed that even if OWL relies on the Open World Assumption, when $\mathcal{S}$CIFF comes to reasoning involving it, the logic programming peculiar Closed World Assumption is reintroduced, thus assuming to have all (relevant) information on the domain available at that time and providing usual features such as negation as failure. In this section we present the two approaches, and their experimental evaluation.

4.1 Importing ontologies into the $\mathcal{S}$CIFF KB_S

Considering that Logic Programming (LP) and Description Logics (DL) have a common root in First Order Logic, a first approach is to find their intersection and to translate ontologies to LP clauses. These two problems have already been addressed by Grosof et al. [10] who named this intersection DLP (Description Logics Program) and by Hustadt et al. [11] who proposed a method for translation. On these basis, the dlpconvert [12] tool was developed and made available by the Karlsruhe University, which converts (the DLP fragment of) OWL ontologies to datalog clauses. We used dlpconvert to traslate domain knowledge described in OWL documents to $\mathcal{S}$CIFF clauses. Reasoning is then performed by $\mathcal{S}$CIFF in the usual way. However, this solution limits ontological expressivity. First of all, since the DLP fragment is a proper subset of DL, some OWL axioms are not included. For instance, out of the features mentioned as available in [13] by Vrandecic et al., DisjointClasses and the important DL (and OWL) feature of class definition by restriction on properties are missing. Moreover, some axioms' translation is not actually suitable for reasoning with goal-driven operational semantics, such as as resolution or unfolding, employed in $\mathcal{S}$CIFF, because it leads to loops. As an example, consider

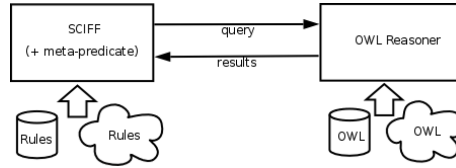

Fig. 2. Integration architecture

```
<owl:Class rdf:about="#danaan">
     <owl:equivalentClass rdf:resource="#achaean"/>
</owl:Class>
```

whose translation is

```
danaan(X) :- achaean(X).
achaean(X) :- danaan(X).
```

and

```
<owl:ObjectProperty rdf:ID="ancestorOf">
     <rdf:type rdf:resource="&owl;TransitiveProperty" />
     <rdfs:domain rdf:resource="#person" />
     <rdfs:range rdf:resource="#person" />
</owl:ObjectProperty>
```

whose translation is

```
ancestorOf(X, Z) :- ancestorOf(X, Y), ancestorOf(Y, Z).
```

For all these reasons the e-commerce ontology must be modified, in particular removing the premiumCustomer class, and the DisjointClasses axioms.

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

An approach that aims to overcome the limitations described in Sect. 4.1 consists of interfacing $\mathcal{S}$CIFF with an external specific ontology-focused component which can be, when necessary, queried by $\mathcal{S}$CIFF and which performs the actual ontological reasoning and gives back results. As represented in Fig. 2, the architecture for 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.

Goals given to the meta-predicate are handled like suggested in [11] and [13] 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

	Ontology import			Interface with Pellet		
N	Load time	Query time	Total	Load time	Query time	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 (all times are in seconds, average over 50 runs)

of predicate)" queries. To reduce the overhead caused by external communication, our implementation of the meta-predicate provides a caching mechanism: it is first checked if a similar query (i.e., involving the same predicate) has been performed before and, only if not, the external reasoner is invoked and answers are cached by storing them as Prolog facts (by means of `asserta/1}).`The OWL reasoning module uses the Pellet [5] API, while the communication interface uses the Jasper Prolog-Java library [14]. This solution enables access to the 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.

4.3 Experimental evaluation

Both approaches were tested successfully in simple contracting scenaria.

Performance comparisons of the two approaches are hardly significant, as they mainly differ in expressive power. However, to assess their scalability, we experimented with randomly generated ontologies. Each ontology, composed of N classes, was built starting from its root node, and recursively attempting, for each node, five attempts of child generation, each with probability 1/3.

The reasoners were queried about the belonging of an entity to the hierarchy root class. For both approaches, we report in Tab. 1 the time spent for loading the ontology into the reasoner[1] and for the actual query. Both approaches appear to scale reasonably (on PC equipped with an Intel Celeron 2.4 GHz CPU). In both cases, the loading time is higher than the query time; for the importing approach, this is particularly noticeable, and encouraging, as in real applications the ontology would be imported only once.

5 Conclusions

In this paper, we proposed an integration of the previously developed $\mathcal{S}$CIFF reasoning framework for service contracting with Description Logics standards.

[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.

We proposed two approaches: first, to translate ontological knowledge into the $\mathcal{S}$CIFF formalism; second, to interface the $\mathcal{S}$CIFF reasoner to ontological reasoners, delegating the ontological reasoning to them. In this way, we make the results (in terms of representation capability and computational efficiency) obtained in the Description Logics area available to the $\mathcal{S}$CIFF contracting framework.

References

1. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Torroni, P.: Web service contracting: specification and reasoning with sciff. In Franconi, E., Kifer, M., May, W., eds.: Proceedings of the 4th European Semantic Web Conference (ESWC). Volume 4519 of Lecture Notes in Artificial Intelligence., Innsbr uck, Springer-Verlag (2007) 68–83
2. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: the SCIFF framework. ACM Transactions on Computational Logics **9** (2008)
3. Matzner, T., Hitzler, P.: Any-World Access to OWL from Prolog. In: KI 2007: Advances in Artificial Intelligence, 30th Annual German Conference on AI, KI 2007, Osnabrück, Germany, September 10-13, 2007, Proceedings. Volume 4667 of Lecture Notes in Computer Science., Springer (2007) 84–98
4. Chesani, F., Mello, P., Montali, M., Torroni, P.: Integrating ontological reasoning and abductive logic programming for service discovery and contracting. Submitted to SWAP'08 (2008)
5. : Pellet: the open source OWL DL reasoner. http://pellet.owldl.com/ (2008)
6. Jaffar, J., Maher, M.: Constraint logic programming: a survey. Journal of Logic Programming **19-20** (1994) 503–582
7. : W3c recommendation: Owl, web ontology language. http://www.w3.org/TR/owl-guide/ (2004)
8. Lutz, C.: Description logic resources. http://dl.kr.org (2008)
9. Noy, N.F., Sintek, M., Decker, S., Crubézy, M., Fergerson, R.W., Musen, M.A.: Creating semantic web contents with protégé-2000. IEEE Intelligent Systems **16** (2001) 60–71
10. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: combining logic programs with description logic. In: WWW '03: Proceedings of the 12th international conference on World Wide Web, New York, NY, USA, ACM (2003) 48–57
11. Hustadt, U., Motik, B., Sattler, U.: Reducing $\mathcal{SHIQ}^-$ description logic to disjunctive datalog programs. In Dubois, D., Welty, C., Williams, M.A., eds.: Proceedings of the 9th International Conference on Knowledge Representation and Reasoning (KR2004), AAAI Press (2004) 152–162
12. Motik, B., Vrandecic, D., Hitzler, P., Sure, Y., Studer, R.: Dlpconvert - Converting OWL DLP statements to logic programs. System Demo at the 2nd European Semantic Web Conference, Iraklion, Greece (2005)
13. Vrandecic, D., Haase, P., Hitzler, P., Sure, Y., Studer, R.: Dlp - an introduction. http://www.aifb.uni-karlsruhe.de/WBS/phi/pub/dlpintro.pdf (2006)
14. : SICStus prolog user manual, release 4.0.4 (2008) http://www.sics.se/isl/sicstus/.