A Logical Framework for Web Service Discovery[⋆]

Michael Kifer[2], Rubén Lara[1], Axel Polleres[1], Chang Zhao[2],
Uwe Keller[1], Holger Lausen[1], and Dieter Fensel[1]

[1] Digital Enterprise Research Institute (DERI) Galway, Ireland and Innsbruck, Austria
{ruben.lara, axel.polleres, uwe.keller, holger.lausen, dieter.fensel}@deri.org

[2] Department of Computer Science University at Stony Brook Stony Brook, New York, USA
{kifer, changz}@cs.sunysb.edu

Abstract. Current technologies for Web Services are based on syntactical descriptions and, therefore, lend themselves to only limited amount of automation. Research efforts in Semantic Web Services, such as WSMO, try to overcome this major deficiency by providing a complete semantic description for Web Services and their related aspects. In this paper we present a logical framework which exploits such formal descriptions in order to dynamically discover Web Services that match requester goals. We consider two kinds of user goals: *discovery* and *contracting*. Based on the WSMO conceptual model, we define proof obligations that formalize the concepts of a match in these two cases. We also describe a concrete realization of this framework in the F-Logic reasoning engine $\mathcal{F}$LORA-2. Such a realization requires an extension of F-Logic in order to support rule reification. With this extension, F-logic becomes a suitable framework for describing and reasoning about Semantic Web Services and their capabilities.

1 Introduction

The current Web service technology, based on the emerging standards of SOAP [20], WSDL [6] and UDDI [1], targets mainly the syntax needed for seamless integration among distributed applications. This technology simplifies *manual* plumbing among applications, but still requires significant amount of custom work in each case. This approach is adequate only if the participating services are selected and hard-wired at design-time and provided that enough manpower is available to integrate the distributed applications. This technology does not support dynamic reconfiguration of services, which can adapt to changes (e.g., when a provider goes off-line or when a cheaper provider enters the market).

Semantic Web is a promising vision that is based on the idea that adding machine-understandable semantic information to Web resources will facilitate automation of many tasks, including integration of distributed applications. Adding semantics can

[⋆] This work has been supported by the SFI (Science Funds Ireland) under the DERI-Lion project and the European Commission under the projects DIP, Knowledge Web, SEKT, SWWS, and Esperonto. The work of M. Kifer and C. Zhao was supported by the NSF grants CCR-0311512 and IIS-0072927 and by U.S. Army Medical Research Institute under a subcontract through Brookhaven National Lab.

also enable a host of new applications, such as automatic location of Web Services that provide a particular functionality, automated contracting for services, and on-the-fly composition of services aimed to achieve goals that cannot be achieved by individual services separately [14].

In this paper, we present a framework for automated Web Service discovery that uses the *Web Service Modeling Ontology* (WSMO) [12] as the conceptual model for describing Web Services, requester goals, and related aspects. The capability, i.e., the functionality of a given Web Service and the requester goals are formalized using F-Logic [10] based on the conceptual framework of WSMO. Since WSML, the service description language of WSMO, is still in development we illustrate the main concepts of the proposed framework using $\mathcal{F}$LORA-2 [8] — an F-logic based system whose main features are very close to and have inspired WSML.

As an ontology for Semantic Web services, WSMO provides the semantic descriptions needed for dynamic location of Web Services that fulfill a given request. The conceptual model of WSMO consists of four major components: *Goals*, which describe the objectives of a service requester; *Web Services*, whose description includes service *Capabilities* (i.e., functionality); *Ontologies*, which formally define the terminology used by the other WSMO elements; and *Mediators*, which serve as adaptors that reconcile the differences among the representations used by the various actors and enable their inter-operation.[3] The problem of automatic service discovery is that of matching the capabilities of existing Web Services against the goal described by the requester.

All four major components of WSMO are specified by F-Logic expressions—a frame-based logic that provides higher-oder features, which we found to be very convenient and natural to use. The $\mathcal{F}$LORA-2 system used in this paper also supports Transaction Logic [3, 4], which plays a role in the proposed discovery framework.

Our framework differentiates between two stages in the process of searching for an appropriate service. The first stage is what we call *discovery*. Here, the requester states only the things that are desired (e.g., a flight ticket from Munich to Dublin). All the services that can potentially satisfy this kind of request are matched.

The second stage is what we call *contracting*. Here the requester provides complete input (including, for example, a credit card number) for an already selected service. The job here is to verify that the input provided will lead to a desired state that satisfies the requester goal. No execution takes place at the contracting stage. Instead, the parties formally verify that the service can fulfill the user request with the given input. However, adding execution to our framework is straightforward.

This paper is organized as follows. Section 2 presents a formalization of the requester goals, the Web Service capabilities, and of the proof obligations that have to be checked for different types of discovery. Section 3 provides a concrete realization of the framework in $\mathcal{F}$LORA-2. Since our framework relies on the ability to reify logical rules, Section 4 extends the semantics of F-logic with support for rule reification. Finally, Section 5 describes related work and presents conclusions.

[3] For a detailed definition of the WSMO core components, we refer the reader to [12] and subsequent versions of this document that may appear at http://www.wsmo.org/

2 Proof Obligations and Formalization

Goals and capabilities. First we briefly review the relevant concepts from WSMO [12], which will be used in our service discovery framework.

We define the Web Service discovery problem as a problem of matching formally described user requests with service functionality satisfying these requests. In WSMO, user requests are called **goals** and service functionalities are called **capabilities**. Both goals and capabilities are defined in terms of their respective **ontologies**. In addition, **mediators** are used as distinct modeling elements that ensure interoperability between Web Services by bridging heterogeneity among the different WSMO elements.

A goal describes what the requester wants to achieve; it consists of logical conditions that describe the desired state of the world and information space.

Service capabilities are described by preconditions and effects of the service. Preconditions specify what the service expects to be fulfilled prior to its invocation. This includes constraints on the input of the service and conditions on the world state before the execution. Effects define what is guaranteed to hold in the state after the service execution. This guarantee typically depends on the input to the service.

A special kind of mediators, the **wgMediators**, are used in WSMO to link Web Service and goals, resolving the possible heterogeneity between them. In this paper, wgMediators are used to resolve possible differences in terminologies that are used by requesters to define goals and by providers to define service capabilities.

Formalization and scalability issues. Logic has long been used for precise representation of statements about real-world objects or abstract artifacts. A suitable logic can be used to formalize goals, capabilities, mediators, as well as the proof obligations that must be established in order to determine whether a match exists between a user request and the functionalities of available services. Unfortunately, experience shows that proficient use of even simple kinds of logic is beyond the capabilities of most programmers. Most students have great difficulty even with translating simple English statements into SQL queries when the statements involve implication or universal quantification. Even greater difficulty exists in translating such statements directly into first-order logic.

Therefore, for a Web service discovery framework to scale in terms of human resources, the underlying architecture must rely on a relatively small number of professionals who are highly skilled in logic and knowledge representation. With this in mind, we envision three categories of people who would be in direct contact with the logical mechanisms of Semantic Web Service discovery:

1. *Customers* who have no training in knowledge representation. These users will have access to pre-selected service discovery queries, which they can choose from a menu or construct using simple graphical tools. These queries would be the main components of the *goals* introduced earlier. The ontology that defines the terms used in these queries is called the **goal ontology**.
2. *Service providers.* These users might not necessarily be more skilled logicians than the rest of the public, but they can hire skilled knowledge engineers. Still the number of businesses who might want to share in the Semantic Web infrastructure can be potentially large, and it is unlikely that sufficient number of highly skilled engineers will be available to meet the demand. Therefore, the Semantic Web Service

infrastructure should impose only modest requirements to the degree of sophistication of the engineers who might turn up in this type of labor market. The upshot of this is that Web service capabilities should be written to relatively simple ontologies and use relatively simple types of rules.

3. *Mediation providers.* The bulk of logical expertise will reside with companies whose business will be to provide ontology mediation. Mediators will bridge the gap between the ultimate simplicity of goal ontologies used by the clients of semantic Web services and the relative simplicity of the service descriptions supplied by service providers. Since mediators link ontologies rather than customers and businesses, the number of skilled workers required to support such an infrastructure can be low enough to make the infrastructure scalable in terms of human resources.

The proof obligations for service discovery, which we introduce next, are designed with the above overall architecture in mind.

Proof obligations. A **proof obligation** is a logical entailment that needs to be established in order for a service to be considered a "match" for a discovery goal. A proof obligation is defined in terms of a set of imported ontologies O, a goal G, a service capability C, and a wgMediator wg. Here, G and C are logical formulas for the goal and the service capability, respectively. The effects and the precondition parts of the capability C are denoted as C_{eff} and C_{prec}. C_{eff} is a logical formula that states what the service guarantees to be true after the execution. C_{prec} is a formula that must be true before the service execution; typically it contains predicates on the input provided by the requester and predicates on the state of the world right before the execution.

A wgMediator wg performs two main functions:

– It takes a goal, G, and constructs input, $In_{wg}(G)$, suitable for the services that are mediated by this particular mediator. This is needed because the goal ontology and the service ontologies might be very different.
– A mediator also needs to convert the goal into a postcondition expressed in the service ontology, which is to be tested in the after-state of the service against the effects of the service. This expression is denoted as $Post_{wg}(G)$.

Translations performed by wgMediators can be quite complex, because goals can be expressed in a very high-level syntax in order to make them palatable to naive users and service capabilities can be rather simple in order to make it inexpensive to specify them by a knowledge engineer.

We consider two different notions of a match. In one, which we call **service discovery**, the user supplies a general goal G and wants to check if a service can execute in a way such that the requester goal will be achieved. This means that (after the appropriate translations) the goal is guaranteed to be true in the after-state of the service. This is formally stated as the following proof obligation:

$$O, In_{wg}(G), C_{eff} \models Post_{wg}(G) \qquad (1)$$

Service contracting comes into play after a potentially suitable service has been discovered. In contracting, given an *actual* input to a *specific* service, we want to guarantee that this input does indeed lead to the results expected by the requester.

This goes beyond the proof obligation for discovery. First, at this stage concrete input may be required (e.g., a credit card number). Second, this input needs to be checked against the precondition specified in the service capability. Third, the specification of the effects of the service and the requester's goal might be more complex. Therefore, the following proof obligation has to be checked:

$$O, In_{wg}(G'), C_{eff'} \models C_{prec} \wedge Post_{wg}(G') \qquad (2)$$

The difference between (2) and (1) is that more complex versions of the goal and effects might be used for contracting (denoted G' and $C_{eff'}$) and that the precondition is checked. The proof obligation (2) can also be used for more precise discovery, which takes precondition into account. This may be appropriate in situations where the user is willing to provide complete input during the discovery process.

The discovery query. The proof obligations (1) and (2) are not quite what is needed for service discovery. In both cases it is assumed that we are dealing with a *particular* service and just need to test if it matches the goal. In practice, we need to go over all the services and test which ones match. The problem with this is that neither $In_{wg}(G)$ nor C_{eff} are part of a global knowledge base, and C_{eff} is different for different services. Since the effects in (1) and (2) are different for different services being tested, what is a general discovery query that could yield all the matching services?

The answer is provided by Transaction Logic [4, 3], which supports hypothetical assertions. This enables us to look at each service separately and hypothetically insert the effects into the knowledge base. The goal can then be tested in the new hypothetical state. If it is true, the service is declared a match. To be able to refer to different services in the same proof obligation, we change our notation to make service effects and goal postconditions relative to a service. Therefore, we will write $C_{eff}(Serv)$ and $Post_{wg}(G, Serv)$, where *Serv* is a variable that represents a service. This idea is logically expressed as follows, where $\diamond$ is the hypothetical operator in Transaction Logic:

$$O \models \exists Serv \ \diamond(insert\{In_{wg}(G), C_{eff}(Serv)\} \otimes Post_{wg}(G, Serv)) \qquad (3)$$

A similar query can be constructed for (2). The above query is looking for services such that $\diamond(insert\{In_{wg}(G), C_{eff}(Serv)\} \otimes Post_{wg}(G, Serv))$ holds in the models of the imported ontologies O. The symbol $\otimes$ is a sequence operator, which says that first the effects and the input must be asserted and then the goal must be tested. Since the assertion is hypothetical, it is "rolled back" after the test is done. Query (3) is the basis of our realization of the framework, which we describe in the next section.

3 A Concrete Realization of the Framework for Discovery

This section shows fragments of a larger running example that illustrates the $\mathcal{F}$LORA-2 implementation of the proof obligations defined in the previous section.[4] In addition to showing concrete instances of service representation and of discovery queries, the example illustrates important architectural aspects of WSMO, such as wgMediators.

[4] The complete example is at http://www.wsmo.org/2004/d5/d5.1/floradiscovery/discovery.flr

The chosen example shows typical elements of a travel reservation system. However it is important to realize that the discovery query is *completely generic* and does not depend on the concrete problem instance or problem domain. It will work as well for any domain provided that service descriptions conform to the WSMO ontology.

Why $\mathcal{F}$LORA-2? $\mathcal{F}$LORA-2 is an implementation of a language that is closely related to *WSML*,[5] the language being developed for WSMO. The main difference is that WSML uses mnemonic terms where $\mathcal{F}$LORA-2 uses more concise symbols.

$\mathcal{F}$LORA-2 supports F-logic [10] and HiLog [5], whose frame-based and higher-order syntax offers a simple and natural representation of the WSMO architectural components as well as of the discovery query. It also supports enough of Transaction Logic [3] to be able to implement the proof obligations of Section 2.

One of the hardest issues in developing a logical framework for Web service discovery is the representation of the capabilities of a service. For instance, the precondition of a service is a logical formula that acts as a constraint on the service's input as well as the initial state of the service execution. Different services can have different preconditions and therefore they need to be represented as values of some attributes of concrete services that are represented as objects. Preconditions can be quite complex formulas and, therefore, the language must support *reification* of complex formulas (i.e., a way to represent such formulas as objects in the language). RDF [11] and OWL [16] support a rudimentary form of reification, but not nearly enough for even simple preconditions.

Service *effects*, which is the other major part of a service capability presents even greater challenge. Typically, service effects specify what would happen *if* the service were to execute with a given input. A natural way to represent this kind of relationships is by using rules, in the style of logic programming and deductive databases, which are parameterized by the *Input* variable. When the input variable is instantiated with concrete input, the body of the rule can be applied to the knowledge base provided by the ontology. If the body evaluates to true, the facts in the head of the rule are established to be true as well.[6] Since service effects must also be reified in order to make it possible for a service object to refer to them via an attribute, this means that the underlying logic language must be able to reify rules. This feature is provided in $\mathcal{F}$LORA-2, but goes well beyond the abilities of other implemented logical platforms that we are aware of.

The main challenge in building a language that supports reification of complex formulas is the well-known fact that reification (in logic usually known under the name of *self-reference*) is capable of producing logical paradoxes. An excellent introduction into the subject can be found in [17]. In [23] it is shown that reification of queries does not cause paradoxes in a rule-based language like $\mathcal{F}$LORA-2 (and therefore WSML). However, this result only covers the precondition part of Web service capabilities. In Section 4, we extend this result to reification of rules and thus ensure that WSML's semantics is free of paradoxes and is adequate for handling service discovery.

[5] http://www.wsmo.org/wsml/

[6] This is an informal description of the semantics of a rule. Although it may sound as if the semantics mandates a bottom-up evaluation, it does not. In fact, the operational semantics of the $\mathcal{F}$LORA-2 system, which serves as the platform for our implementation, is top-down.

A Walk-through the Example

The concrete realization of the logical framework for discovery developed in this section is built around the ideas presented in the Section 2. Our examples show a small number of simple logical expressions that a typical client can use, a number of relatively simple service capabilities, and examples of the mediators that can bridge between the ontologies underlying these two worlds.

It is important to realize that some of the goals in our goal ontology are *quite sophisticated* — it is only the logical expression that represents them that is simple! For instance, the goal of finding travel services that can book a ticket from *everywhere* in Germany to *everywhere* in Austria requires the use of universal quantifiers and is often beyond the ability of naive users. However, the goal itself looks very simple: `search(germany, austria)`. Likewise, service capability descriptions are not explicitly written to support such queries. All that they can seemingly do is to tell whether a trip can be booked for a pair of *specific* cities. However, a mediator is capable of translating the goal into input and the results produced by the services into output that together ensure that the user goal is answered correctly.

A geographic ontology. We start with a simple ontology that represents geographic regions and cities. In $\mathcal{F}$LORA-2, the symbols that begin with a lowercase letter are constants that represent objects, and capitalized symbols (and symbols beginning with a "_" are variables. In our taxonomy, `europe`, `germany`, `usa`, `america`, etc., denote classes of cities. Thus, `europe` is a class whose members are all the cities in Europe, `usa` is a class whose members are U.S. cities, and so on. The subclass relationship is denoted using "::", i.e., `austria :: europe` states that `austria` is a subclass of `europe` (which implies that all Austrian cities are also European cities). To specify that an object is a member of a class, we use the symbol ":". For instance, `paris : france` states that Paris is a city in France. A fragment of such a geographic taxonomy is shown below:

```
germany :: europe.   stonybrook : nystate.   frankfurt : germany.
austria :: europe.   innsbruck : tyrol.      paris : france.
france :: europe.    lienz : tyrol.          nancy : france.
tyrol :: austria.    vienna : austria.       usa :: america.
bonn : germany.      nystate :: usa
```

F-logic classes are also viewed as objects and therefore they can be members of other classes. For instance, `europe` is a region, and so is `america`. In the above statements these two symbols played the role of classes, but in the following statements they play the role of objects that are members of class `region`.

```
europe : region.    america : region.
```

USA, Austria, and Germany are also regions and so is Tyrol. Rather than listing all of them explicitly as members of class `region`, we use a rule to define all regions:

```
Region : region  :-  AnotherRegion : region and Region :: AnotherRegion.
```

Service descriptions. In accordance with the conceptual framework of WSMO, a service description in our example includes a specification of the service capability and of the mediators used by the service. In our example, each service uses only one wgMediator to tell how to convert the goal ontology into the ontology used by the service. We also assume that there is a single goal ontology and two service ontologies.

The goal ontology and the service ontologies are not specified explicitly for the lack of space. Instead, we assume that goals have the form

```
goalId[requestId -> someId, query -> someQuery]
```

This means that goals are represented as objects with certain properties. In F-logic, a statement of the above form means that *goalId* is a symbol that represents the object Id of a goal (it can look, for example, like `g123`) and that goal-objects have attributes `requestId` and `query`. The attribute `requestId` represents the Id of the request in case it is desirable to have it separate from the Id of the goal (for instance, if goals are intended to be reused). The attribute `query` represents the query that corresponds to the goal. The symbol `->` means that these attributes are functional; the symbol `->>` (used in service descriptions below) means that the attribute is set-valued. Our use case assumes four types of queries:

```
searchTrip(from, to)        tripContract(servId, from, to, date, crCard)
searchCitipass(loc)         citipassContract(servId, city, date, crCard)
```

The first two queries are used to discover services that can sell tickets from one location to another and citipasses for various cities. The last two queries are used to make a contract with a specific service for purchase of a ticket or a citipass. This is why the Id of a concrete service is part of the query.

A description of the service `serv1` is shown below. Preconditions and effects are specified as reified formulas, which is indicated with ${...} in $\mathcal{F}$LORA-2. In addition, the effects of the service are specified via rules, which tell how the input supplied at service invocation affects what will be true in the after-state of the service.

```
serv1[capability->
      // Request for a ticket from somewhere in Germany to somewhere
      // in Austria OR a request for a citipass for a city in Tyrol
      cap1[ precondition(Input)->${
                (Input = contract(_, From : germany, To : austria, Date, Card)
                  or Input = contract(_, City : tyrol, Date, _))
                and validDate(Date) and validCard(Card)  }
            effects(Input)->${
                (itinerary(Req)[from->From, to->To] :-
                    Input = search(Req, From : germany, To : austria))
                and
                (passinfo(Req)[city->City] :-Input = search(Req, City : tyrol))
                and
                (ticket(Req)[confirmation->Num, from->From, to->To, date->Date] :-
                      Input = contract(Req, From, To, Date, _CCard),
                      generateConfNumber(Num))
                and
                (pass(Req)[confirmation->Num, city->City, date->Date] :-
                    Input = contract(Req, City, Date, _CCard),
                    generateConfNumber(Num)) }
     ],
usedMediators->>med1 ].

serv3[capability->
     // request for a citipass for a French city
     cap3[ precondition(Input)->${
                Input = pay(_, City : france, Date, Card)
                and validDate(Date) and validCard(Card) },
            effects(Input)->${
                (Req[location->City] :- Input = discover(Req, City : france))
                and
                (Req[confirmation->(Num, City, Date)] :-
                      Input = pay(Req, City, Date, _Card) and
                      generateConfNumber(Num)) }
     ],
usedMediators->>med2 ].
```

Notice the differences in the input that the two services expect and in the form of their output, which is due to the fact that the two services use *different ontologies*. For instance, serv1 expects search(Req, City : tyrol) as one of the possible inputs, while serv3 wants discover(Req, City : france). Likewise, serv1 yields objects of the form passinfo(Req)[city->City] in response, while serv3 yields objects of the form Req[location->City]. Due to the differences in the ontologies, serv1 and serv2 tell the world that different mediators must be used to talk to them. In the first case, this is mediator med1 and in the second it is med2. Mediators are represented as objects that possess methods for performing the mediation tasks. The first mediator is shown in some detail later.

Goals. Goals are objects that have two main attributes, requestId and query, as described earlier. The third attribute, result (not shown), represents the set of items returned by the discovery/contracting process. Here are examples of some goals:

```
goal3[requestId->g123, query->searchTrip(france, austria)].
goal2[requestId->g321,
      query->tripContract(serv1, bonn, innsbruck, '1/1/2007', 12345)].
```

The first goal is quite interesting, because none of the services expects regions as input. Thus, without mediation, goal3 cannot be answered. Specifying mediators between this kind of queries and the input expected by the services is quite nontrivial and cannot be expected of a common user.

Mediators. The job of a mediator in our scenario is to bridge between goals and services. More specifically, a wgMediator performs two functions:

1. It takes a goal and constructs the input to the service, which is appropriate for that goal; and
2. It takes the result produced by the service and converts it to the format specified by the goal ontology.

Part of the mediator med1 is shown below.

```
med1[constructInput(Goal)->Input] :-
        Goal[requestId->ReqId, query->Query] and
        if Query = searchTrip(From, To)
        then ( generalizeArg(From, From1), generalizeArg(To, To1),
              Input = search(ReqId, From1, To1) )
        else if Query = searchCitipass(City)
        then ( generalizeArg(City, City1), Input = search(ReqId, City1) )
        else if ... ... ...
        else fail.
med1[reportResult(Goal, Serv, Result)] :-
        Goal[query->searchTrip(From : region, To : region)] and
        not med1[doesNotServeCity(From, To)]
        and Result = ${Goal[result->>Serv]}.
```

The above rules define methods to perform the two main tasks mentioned above: constructing the input and converting the service results into the format suitable for the goal ontology. The definition of the method constructInput checks the form of the user goal and yields appropriate input for the service. The predicate generalizeArg (not shown here, but defined in the full example) replaces the arguments that are objects corresponding to geographical regions with universal variables, because the mediator "knows" that this corresponds to the query with the quantifier "for all cities in the region." The method reportResult is defined by several rules of which we show only the one that corresponds to region-level requests, i.e., requests for services that sell tickets from/to every city in a pair of regions. If the user query is a region-level request, the rule checks if the service serves every city in the specified regions and then constructs the result expected by the service ontology. This result is then inserted into the knowledge base by the discovery query — see next.

Discovery. The discovery query is shown below. It examines each available service one by one. For each service, it obtains the mediator specified by the service and uses that mediator to construct the input appropriate for the service. Next we can use the input to obtain the effects of the service. Then the effects are hypothetically assumed and the goal is tested in the resulting state. If the goal is true in that state, the result (which contains the identification for the service) is inserted into the knowledge base.

```
find_service(Goal) :-
      Serv[usedMediators->>Mediator[constructInput(Goal)->Input]],
      Serv.capability[effects(Input)->Effects],
      insertrule{Effects},        // hypothetically assume the effects
      if Mediator[reportResult(Goal, Serv, Result)] then insert{Result},
      deleterule{Effects}.        // Remove the hypothetical effects
```

The query for verifying a service contract is essentially similar except that it also tests the precondition. Details can be found in the full example at http://www.wsmo.org/2004/d5/d5.1/floradiscovery/discovery.flr.

4 Semantics of Rule Reification

In [23], Yang and Kifer described an extension of F-logic with support for reification. However, rule reification, which we rely heavily on in our realization presented in Section 3, was not considered. In this section, we define a model theory for F-logic extended with rule reification.

Before giving the model theory, let us briefly define the new F-logic syntax which extends the syntax defined in [10]. For simplicity, we will focus on F-logic atoms in the form of o[m->>v], which correspond to multi-valued attributes. For a complete definition of F-logic atom syntax, see [10].

An F-logic language $\mathcal{L}$ consists of a set of *constants*, $\mathcal{C}$; a set of *variables*, $\mathcal{V}$; the *connectives* $\neg$, $\vee$, $\wedge$, and $\longleftarrow$; the *quantifiers* $\exists$ and $\forall$; and *auxiliary symbols*, such as comma, parentheses, and brackets. We will assume that the language $\mathcal{L}$ is fixed.

Definition 1 (Terms and Generalized Terms). *Given an F-logic language $\mathcal{L}$, the* **terms** *and* **generalized terms** *are defined inductively as follows:*

- *Any constant $c \in \mathcal{C}$ is a $\boldsymbol{term}$.*
- *Any variable $X \in \mathcal{V}$ is a term.*
- *If t is a term and $t_1,\ldots,t_n$ are terms, then $t(t_1,\ldots,t_n)$ is a term.*
- *Any term in any of the above forms is called a HiLog term.*
- *If o, m, and v are terms, then o[m->>v] is a term, also called an F-logic term.*
- *If A_1 and A_2 are terms, then $A_1 \wedge A_2$, is a term.*

- *Any term t is also a $\boldsymbol{generalized\ term}$.*
- *If A_1 and A_2 are generalized terms, then $A_1 \vee A_2$, is a generalized term.*
- *If A is a generalized term, then $\neg A$ is also a generalized term.*
- *If A_1 is a term and A_2 is a generalized term, then $(A_1 \longleftarrow A_2)$ is a term (not just a generalized term!).*
- *If A is a generalized term, then $\exists X(A)$ and $\forall X(A)$ are generalized terms, where $X \in \mathcal{V}$ is a variable.*

Note that according to the definition, $p \longleftarrow q$ is a term (and thus also a generalized term), while $p \vee \neg q$ is a generalized term, but not a term. In particular, $p \longleftarrow q$ is not considered to be equivalent to $p \vee \neg q$ (as usual in logic programming). The rationale behind the distinction between terms and generalized terms is that we need to distinguish which terms can occur as literals in the rule heads and which in the rule bodies. In our extension to F-logic, only terms can occur in the head of a rule, while the rule body can also contain generalized terms. This allows us to prevent explicit negation from appearing in the rule heads and, as will be seen shortly, to avoid logical paradoxes related to the use of reification. Note that we *do* allow terms in the form of a rule to appear in the rule heads.

Definition 2 (Formula). *Any generalized term is a* **formula**. *In particular, any HiLog term or F-logic term is an atomic (HiLog or F-logic) formula. Terms of the form $\phi \longleftarrow \psi$ are called* **rule formulas**.

By Definitions 1 and 2, atomic formulas, rules, and conjunctions of such formulas are terms. Since terms are first-class objects in the language and variables can range over them, we have a higher-order syntax that supports reification, including rule reification.

Definition 3 (Augmented Herbrand Universe). *Let $\mathcal{L}$ be an F-logic language and $\mathcal{C}$ be the set of constants in $\mathcal{L}$. The* **augmented Herbrand universe** *of $\mathcal{L}$, denoted $\mathcal{HU}$, is the set of all terms (according to Definition 1) constructed using the constants in $\mathcal{C}$. Such variable-free terms are called* **ground**. *Clearly, $\mathcal{HU}$ is countably infinite.* [7]

An **F-logic program** is a finite collection of rules where all variables are universally quantified. The program amounts to the conjunction of all its rules. A rule has the following form:

$$\forall(\mathsf{A}_1 \wedge \ldots \wedge \mathsf{A}_\mathsf{m} \ \longleftarrow\ \mathsf{B}_1 \wedge \ldots \wedge \mathsf{B}_\mathsf{n}) \tag{4}$$

where $\mathsf{m} \geq 1$, $\mathsf{n} \geq 0$, A_i $(1 \leq \mathsf{i} \leq \mathsf{m})$ and B_j $(1 \leq \mathsf{j} \leq \mathsf{n})$ are atomic formulas or rule formulas in the form of (4).

Note that, for simplicity, we do not allow negation in the rule body. However, our model theory can be readily extended to F-logic programs with negation in rule bodies as well as inheritance by combining it with the semantics described in [21] and [22].

When defining the semantics of a program, we are actually considering its *Herbrand instantiation* which is the set of rules obtained by substituting terms in the augmented Herbrand universe $\mathcal{HU}$ for variables in every possible way.

Definition 4 (Interpretations). *Given an F-logic language $\mathcal{L}$, an interpretation $\mathcal{I}$ is a subset of $\mathcal{HU}$, which contains only atomic formulas and rule formulas. Intuitively, $\mathcal{I}$ represents true statements about some possible world.*

Definition 5 (Models). *Let $\mathcal{I}$ be an interpretation and ϕ be a formula. We say that $\mathcal{I}$ is a model of ϕ, denoted $\mathcal{I} \models \phi$, if the following holds. Note that the definition is similar to the classical one, except for the case of rule formulas.*

- *If ϕ is an atomic formula, then $\mathcal{I} \models \phi$ iff $\phi \in \mathcal{I}$.*
- *If $\phi = \psi \longleftarrow \xi$, then $\mathcal{I} \models \phi$ iff $\psi \longleftarrow \xi \in \mathcal{I}$ and $\mathcal{I} \models \psi \vee \neg\xi$.*
- *If $\phi = \neg\psi$, then $\mathcal{I} \models \phi$ iff it is not the case that $\mathcal{I} \models \psi$.*
- *If $\phi = \psi \wedge \xi$, then $\mathcal{I} \models \phi$ iff $\mathcal{I} \models \psi$ and $\mathcal{I} \models \xi$.*
- *If $\phi = \psi \vee \xi$, then $\mathcal{I} \models \phi$ iff either $\mathcal{I} \models \psi$ or $\mathcal{I} \models \xi$.*
- *If $\phi = \exists X\psi$, then $\mathcal{I} \models \phi$ iff there is $\mathsf{t} \in \mathcal{HU}$ such that $\mathcal{I} \models \psi[X/t]$, where $\psi[X/t]$ denotes the formula obtained from ψ by substituting t for all free occurrence of the variable X.*
- *If $\phi = \forall X\psi$, then $\mathcal{I} \models \phi$ iff for all $\mathsf{t} \in \mathcal{HU}$, $\mathcal{I} \models \psi[X/t]$.*

[7] Note that since atomic formulas are reified, Herbrand bases used in classical logic programming are the same as Herbrand universes in our setting.

Note that the term $\mathsf{p} \longleftarrow \mathsf{q}$ and the generalized term $\mathsf{p} \vee\neg\mathsf{q}$ are different: the former belongs to the augmented Herbrand universe while the latter does not. The above definition also says that these terms have different semantics when they are viewed as formulas. More specifically, if an interpretation $\mathcal{I}$ is a model of $\mathsf{p} \longleftarrow \mathsf{q}$, then it is also a model of $\mathsf{p} \vee\neg\mathsf{q}$, but not the other way around. For example, the empty set is a model of $\mathsf{p} \vee\neg\mathsf{p}$ but not a model of p $\longleftarrow$ p, since any model of a rule must contain that rule.

We are now ready to develop a fixpoint semantics for F-logic programs with rule reification. Analogous to classical theory of logic programming, we define a program consequence operator.

Definition 6 (Program Consequence Operator). *Let* **P** *be an F-logic program. The program consequence operator* $\mathcal{T}_{\mathbf{P}}$ *maps an interpretation* $\mathcal{I}$ *to another interpretation* $\mathcal{J}$*, denoted* $\mathcal{T}_{\mathbf{P}}(\mathcal{I}) = \mathcal{J}$*, where* $\mathcal{J}$ *is a set of terms* A *such that there is an instantiated rule* $\mathsf{H} \longleftarrow \mathsf{B}_1, \wedge \ldots \wedge \mathsf{B}_\mathsf{n}$ *in* **P**, *where*

– *A is one of the conjuncts in H; and*
– *$\mathsf{B}_\mathsf{j} \in \mathcal{I}$ for all B_j, $1 \leq \mathsf{j} \leq \mathsf{n}$.*

Theorem 1. *Let P be an F-logic program. Then the least fixpoint of* $\mathcal{T}_{\mathbf{P}}$*, denoted lfp*$(\mathcal{T}_{\mathbf{P}})$*, is the least model for P.*

The following example illustrates the computation of the least model of a program. Here the lowercase letters a, b, etc., denote ground atoms. Let the program be:

$$\mathsf{a} \longleftarrow . \quad \mathsf{c} \longleftarrow (\mathsf{a} \longleftarrow \mathsf{b}). \quad \mathsf{d} \longleftarrow . \quad (\mathsf{e} \longleftarrow \mathsf{d}) \longleftarrow \mathsf{a}. \quad \mathsf{f} \longleftarrow (\mathsf{e} \longleftarrow \mathsf{d}).$$

we obtain:

$\mathcal{I}_1 = \{\mathsf{a}, \mathsf{d}, \mathsf{c} \longleftarrow (\mathsf{a} \longleftarrow \mathsf{b}), (\mathsf{e} \longleftarrow \mathsf{d}) \longleftarrow \mathsf{a}, \mathsf{f} \longleftarrow (\mathsf{e} \longleftarrow \mathsf{d})\}$
$\mathcal{I}_2 = \mathcal{T}_{\mathbf{P}}(\mathcal{I}_1) = \{\mathsf{a}, \mathsf{d}, \mathsf{c} \longleftarrow (\mathsf{a} \longleftarrow \mathsf{b}), (\mathsf{e} \longleftarrow \mathsf{d}) \longleftarrow \mathsf{a}, \mathsf{f} \longleftarrow (\mathsf{e} \longleftarrow \mathsf{d}), \mathsf{e} \longleftarrow \mathsf{d}\}$
$\mathcal{I}_3 = \mathcal{T}_{\mathbf{P}}(\mathcal{I}_2) = \{\mathsf{a}, \mathsf{d}, \mathsf{c} \longleftarrow (\mathsf{a} \longleftarrow \mathsf{b}), (\mathsf{e} \longleftarrow \mathsf{d}) \longleftarrow \mathsf{a}, \mathsf{f} \longleftarrow (\mathsf{e} \longleftarrow \mathsf{d}), \mathsf{e} \longleftarrow \mathsf{d}, \mathsf{e}, \mathsf{f}\}$
$\mathcal{I}_4 = \mathcal{T}_{\mathbf{P}}(\mathcal{I}_3) = \{\mathsf{a}, \mathsf{d}, \mathsf{c} \longleftarrow (\mathsf{a} \longleftarrow \mathsf{b}), (\mathsf{e} \longleftarrow \mathsf{d}) \longleftarrow \mathsf{a}, \mathsf{f} \longleftarrow (\mathsf{e} \longleftarrow \mathsf{d}), \mathsf{e} \longleftarrow \mathsf{d}, \mathsf{e}, \mathsf{f}\}$

$\mathcal{I}_3$ is a fixpoint. Note that c is not included in $\mathcal{I}_3$ while it would have been if we treated $\mathsf{a} \longleftarrow \mathsf{b}$ as a shortcut for $\mathsf{a} \vee\neg\mathsf{b}$.

Reification and Logical Paradoxes The well-known inconsistency of Frege's comprehension axioms is a result of the ability to reify logical sentences and make statements about these sentences. Now that F-logic is extended with reification, is it free of paradoxes? Consider the following **comprehension axiom schema**:

$$\exists \mathsf{P} \forall \mathsf{X}(\mathsf{P}(\mathsf{X}) \longleftrightarrow \phi(\mathsf{X}))$$

While one might think that the comprehension axiom is too general to be useful (and thus unlikely to occur in user specifications), the following simpler truth axiom is less esoteric:

$$\forall \mathsf{X}(\mathsf{true}(\mathsf{X}) \longleftrightarrow \mathsf{X}) \tag{5}$$

Unfortunately, it turns out to be an instance of the comprehension axiom and is almost as bad. The following example is adapted from [23]. Consider

$\mathsf{true}(\neg \mathsf{p}) \longleftarrow \mathsf{p}. \quad \mathsf{p} \longleftarrow \neg \mathsf{p}.$

Together with the truth axiom, this program implies $\mathsf{p} \leftrightarrow \neg\, \mathsf{p}$.

In [23], it is proved that Horn programs in reified F-logic are consistent with the truth axiom. However, that version of F-logic did not reify rules. Can rule reification cause paradoxes? The answer is, fortunately, *no*.

Theorem 2. *Horn programs in reified F-logic augmented with the truth axiom (5) are consistent.*

Reified F-logic avoids paradoxes through the following restrictions:

- No negation is allowed in the rule head, and
- Reification of negation of any fact or any rule is not permitted.

The $\mathcal{F}$LORA-2 system prohibits reification of negative rules, but allows reification of negative facts. So the second condition above does not hold. $\mathcal{F}$LORA-2 closes the loophole with the following restriction: *The head of a rule cannot be a variable.*

This excludes rules of the form $\mathsf{X} \longleftarrow \mathsf{body}$, but still allows rules like $\mathsf{X(Y)} \longleftarrow \mathsf{body}$ or $\mathsf{X[Y{-}{>}{>}Z]} \longleftarrow \mathsf{body}$. This restriction eliminates the truth axiom. Since negation of facts or rules cannot occur in the rule heads, it becomes impossible to derive negative information by $\mathcal{F}$LORA-2 programs (except through the closed world assumption).

5 Related Work

Several existing approaches to dynamic Web Service discovery rely on OWL-S. For instance, [15] and [13] propose an OWL-S-based approach to matching service advertisements and requests. Both approaches rely on OWL-style subsumption reasoning [16] to determine if a match exists between the information defined in the service profile and the information given in the request. Other approaches such as [9], [19] or [18] do not use OWL-S service descriptions but still rely on subsumption reasoning. The work reported in [2] also uses OWL-S. However, the service discovery problem is not formulated as a subsumption reasoning problem but rather as a rewriting problem where requests are attempted to be rewritten in terms of available services.

All these approaches are limited by the lack of rules in the OWL language. This prevents the provider from describing exactly and explicitly how the effects of the service relate to the inputs provided.[8] Therefore, the type of inputs and outputs (or effects) can be semantically annotated but the relation between them cannot be captured. Therefore, no formal guarantee can be given that the service will actually fulfill the requester's goal. For this reason, in the terminology of Section 2, these approaches can only do *discovery* but not *contracting*.

In [24], an F-Logic-based approach for the description of Web Services is presented. However, this work uses simple query answering and, therefore, the description of the services is limited to ground facts, which considerably reduces the expressivity allowed for describing the service functionality.

[8] Recently OWL-S tried to partially rectify the problem by introducing the notion of *conditional effects*.

6 Conclusions

We presented a logical framework and a concrete realization for dynamic discovery of Web services and for verification of their contractual statements. By exploiting the WSMO conceptual model and, in particular, *wgMediators*, we shield the requesters and providers from the complexity of the logical formalization of goals and capabilities and thus ensure the scalability of the framework in terms of human resources. To properly formalize our framework, we introduced the notion of rule reification and used it for reasoning about the formal descriptions of goals, capabilities, and *wgMediators*.

As discussed in Section 5, our framework is able to accurately capture the functionality that is offered by services and sought by requesters, thus enabling more accurate discovery than the approaches proposed so far. Moreover, our framework can be easily extended to include service invocation. Although Section 3 uses a particular use case to illustrate our framework, the framework itself is domain-independent.

Our future work will focus on aligning the framework with the sublanguages of WSML that are currently being defined and on revisiting the modeling concepts of WSMO in light of this work. Testing of industrial use-cases specified by the SDK cluster[9] will receive special attention. Extending our approach with (semi)automatic composition of Web Services will also be a subject for future research in the context of the SDK cluster.

References

1. T. Bellwood, L. Clment, D. Ehnebuske, A. Hately, Maryann Hondo, Y.L. Husband, K. Januszewski, S. Lee, B. McKee, J. Munter, and C. von Riegen. Uddi version 3.0, July 2002. http://uddi.org/pubs/uddi-v3.00-published-20020719.htm
2. B. Benatallah, M-S. Hacid, C. Rey, and F. Toumani. Request rewriting-based web service discovery. In *The Semantic Web - ISWC 2003*, pp. 242–257, Oct. 2003.
3. A.J. Bonner and M. Kifer. A logic for programming database transactions. In *Logics for Databases and Information Systems*, chapter 5, pp. 117–166. Kluwer, March 1998.
4. A. J. Bonner, M. Kifer. Transaction logic programming (or, a logic of procedural and declarative knowledge). Tech. report, 1995.
5. W. Chen, M. Kifer, D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, Feb. 1993.
6. E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web services description language (wsdl) 1.1, March 2001. http://www.w3.org/TR/wsdl
7. D. Connolly, F. van Harmelen, I. Horrocks, D.L. McGuinness, P.F. Patel-Schneider, L.A. Stein. DAML+OIL reference description. W3C note, W3C, Dec. 2001.
8. FLORA-2. The $\mathcal{F}$LORA-2 web site. http://flora.sourceforge.net
9. J. Gonzlez-Castillo, D. Trastour, C. Bartolini. Description logics for matchmaking of services. In *KI-2001 Workshop on Applications of Description Logics*, Sept. 2001.
10. M. Kifer, G. Lausen, J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, July 1995.
11. O. Lassila, R.R. Swick (*eds*). Resource description framework (RDF) model and syntax specification. Rec. W3C, Feb. 1999. www.w3.org/TR/1999/REC-rdf-syntax-19990222/

[9] http://www.sdkcluster.org/

12. H. Lausen, D. Roman, U. Keller (*eds*). Web service modeling ontology - standard (WSMO-Standard). Working draft, Digital Enterprise Research Institute (DERI), March 2004. http://www.wsmo.org/2004/d2/v0.2/
13. L. Li, I. Horrocks. A software framework for matchmaking based on semantic web technology. 12th Int. Conf. on the World Wide Web, Budapest, Hungary, May 2003.
14. S. McIlraith, T.C. Son, H. Zeng. Semantic web services. *IEEE Intelligent Systems, Special Issue on the Semantic Web*, 16(2):46/53, March/April 2001.
15. M. Paolucci, T. Kawamura, T. Payne, K. Sycara. Semantic matching of web services capabilities. *1st Int. Semantic Web Conference (ISWC)*, pp. 333–347, 2002.
16. P.F. Patel-Schneider, P. Hayes, I. Horrocks. OWL web ontology language semantics and abstract syntax. W3C recommendation, W3C, February 2004.
17. D. Perlis. Languages with self-reference i: Foundations. 25:301–322, 1985.
18. K. Sivashanmugam, K. Verma, A. Sheth, J. Miller. Adding semantics to web services standards. In *1st Int. Conf. on Web Services (ICWS'03)*, pp. 395–401, June 2003.
19. K. Verma, K. Sivashanmugam, A. Sheth, A. Patil. Meteor-s wsdi: A scalable p2p infrastructure of registries for semantic publication and discovery of web services. *Journal of Information Technology and Management*, 2004.
20. W3C. Soap version 1.2 part 0: Primer, June 2003. http://www.w3.org/TR/2003/REC-soap12-part0-20030624/
21. G. Yang, M. Kifer. Well-founded optimism: Inheritance in frame-based knowledge bases. In *Int. Conf. on Ontologies, Databases, and Applications of Semantics (ODBASE)*, 2002.
22. G. Yang, M. Kifer. Inheritance and rules in object-oriented semantic web languages. In *Int. Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML)*, 2003.
23. G. Yang, M. Kifer. Reasoning about anonymous resources and meta statements on the semantic web. *Journal of Data Semantics*, 1:69–97, 2003.
24. O.K. Zein, Y. Kermarrec. An approach for describing/discovering services and for adapting them to the needs of users in distributed systems. 2004.