WSML Deliverable

D5.1 v0.1 Inferencing Support for Semantic Web Services: Proof Obligations

WSML Working Draft – August 2, 2004

Authors:
Uwe Keller, Rubén Lara, Axel Polleres, Holger Lausen, Michael Stollberg and Michael Kifer

Editors:
Uwe Keller, Rubén Lara

This version:
http://www.wsmo.org/2004/d5/d5.1/v0.1/20040802
Latest version:
http://www.wsmo.org/2004/d5/d5.1/v0.1/
Previous version:
http://www.wsmo.org/2004/d5/d5.1/v0.1/20040621

Abstract

The *Web Service Modeling Ontology* (WSMO) provides the conceptual framework for semantically describing web services and their specific properties. Based on WSMO the *Web Service Modeling Language* (WSML) implements this conceptual framework in a formal language for annotating web services with semantic information.

In this document we present and discuss how semantic descriptions of web services based on WSMO and WSML can be exploited for various tasks in the design and dynamic composition of web-service-based software-systems.

In particular, we investigate and analyze what kind of statements need to be formally checked (resp. proven) in order to automatically support the various tasks occurring during this dynamic system-construction process and allow these processes to access the semantic information captured in the description. These formal statements are called *proof obligations*.

Together, WSMO and WSML as well as the semantic tool suite described and developed within deliverable D5 constitute the necessary semantic infrastructure that is needed for making *semantic web services* come true. Only such an infrastructure allows us to exploit the full-potential of web service technology in the context of Enterprise Application Integration (EAI) and fully-dynamic eCommerce.

Contents

1 Introduction

1.1 Motivation

The World Wide Web (WWW) has been invented in 1989 by Tim Berners-Lee [BLFD99] and since then changed the way people gather and access knowledge in everyday life. At its invention, nobody was able to foresee the effects and influences of this new promising technology. In fact, the WWW can be considered as one of the most influential technologies that has been invented in the 20th century.

Nowadays, the World Wide Web is certainly the biggest information repository, containing a huge amount of knowledge. But the major bottleneck when it comes to exploiting this information is how to find the required knowledge: the Web is still a mainly syntactically-driven information repository, where information is represented in form of web pages that actually fail to capture the semantics of their contents. As a consequence, the content of the pages is *not* easily accessible by computers but instead exclusively intended for human consumption. Given the size and the rapid growth of the WWW, this approach does not scale up.

This has led to significant research on the so-called *Semantic Web* [BLHL01, DFvH03, FHLW03], that aims at enabling computerized agents to easily gather information in the Web and exploit this knowledge on behalf of their human clients by enriching information in the WWW with machine-processable semantics.

From Web Services to Semantic Web Services. Web services add a new level of functionality on the current Web. Whereas at the moment the Web is mainly constituted by simple documents (static and dynamic web pages), in the future the Web might become a repository of electronic (web) services. Each service represents a computational entity that can be accessed and invoked over the Internet. Thus, the Web might become a huge, distributed and shared computational device.

Recently, web services have gained more and more interest in the industry as well as among academics. This is due to what they promise: a uniform infrastructure for accessing software-systems over the Internet, which can be used for integration of legacy systems – even across enterprise boundaries. Nonetheless, though web service simplify enterprise application integration, the technologies around web services – namely SOAP [W3C03], WSDL [CCMW01] and UDDI [BCE+02] – do not solve the integration problem by themselves. Again, the integration technologies are exclusively syntactical means that lack explicit, machine-understandable semantics.

Moreover, given the growing industrial interest in web services, the development and publication of a huge amount of web services for solving various tasks can be expected in the near future. Hence, the same bottleneck as for the conventional WWW arises: How can we find a web service that solves a problem at hand? SOAP, WSDL and UDDI are not sufficient for answering this question. Again, machine-processable semantics for web services is needed in order to ensure access of web services for agents without human interaction. Adding semantics to web services is the main prerequisite for an approach to scale.

This semantically enriched web services are called *Semantic Web Services*. In a sense, we can use the following equation to summarize the relationship between web services and semantic web services:

Semantic Web Service = Web Service + Semantic Annotation

And what do we do with all that Semantics? The major benefit of any form of specification is that by explicitly documenting what an artifact actually does, it becomes a lot easier for humans (who do not know that artifact before) to comprehend what a given artifact can be actually used for.

Clearly, with a formal representation of any specification, the use of an artifact becomes more precise and eventually accessible for *computerized* agents. Knowledge about the capabilities of single objects can be exploited for combining these objects to systems which solve more complex problems than each of their elements separately.

Each software system is build for some purpose: to solve a specific well-defined problem of a client or organization. Software architectures themselves are the conceptual means for describing how the functionality of a system is provided and how the given requirements are actually satisfied. In this sense software architectures [GS96, BCK03] bridge the gap between the user requirements and the level of computational units that provide some well-defined functionality and interact and communicated with each other.

In a world of rapidly changing requirements and incomplete knowledge (such as the Semantic Web with its open and heterogeneous user community), the problems to be solved most likely occur rather ad-hoc and cannot be foreseen in detail. In this case, the complete construction of a system that solves the problem at hand, in general ca not be done in advance, but has to be done on the fly instead. Thus, software-systems composed by services in the Web constitute a particularly interesting and challenging problem for the theory and practice of software architecture: how to dynamically construct systems out of existing ones in a goal-driven way.

For this purpose, agents need support for *reasoning about semantic web service descriptions* in order to effectively utilize semantic annotations.

The big picture. Together, WSMO and WSML as well as the semantic tool suite developed in the context of deliverable D5 constitute the necessary semantic infrastructure that is needed for making *semantic web services* come true. Such an infrastructure will enable the automatic discovery of services and dynamic composition of services. Thus, it is a major necessary prerequisite for enabling the exploitation of web service technology's full potential in the context of *Enterprise Application Integration* and *Electronic Commerce*[1] [Fen03].

1.2 Goal of this Document

In this document we present and discuss how semantic descriptions of web services based on WSMO and WSML can be exploited for various tasks in the design and dynamic composition of web-service-based software systems.

We identify the different phases (resp. domains) within the (dynamic) construction of a software system based on web services, where semantic information can be exploited to support a client in solving a problem at hand with a computerized agent. For the various domains, we describe what kind of reasoning task we have to cope with and what form of reasoning has to be carried out.

[1] One can imagine, for instance, the potential benefit of dynamic configuration of supply chains for manufacturing companies under consideration of the current market state.

In particular, for all these domains we investigate and analyze what kind of statements need to be formally checked (resp. proven) in order to automatize the various tasks occurring during this dynamic system construction process and allow these processes to access the semantic information captured in the Web Service descriptions. These formal statements are called *proof obligations*.

1.3 Overview of the Document

We start by identifying in Section 2 for what purposes the semantic description of web services can be exploited. Section 3 defines what a proof obligation is and why proof obligations are interesting to provide automation support. The current version of this document focuses, in Section 3.1 on the proof obligations for discovery, and their implementation, illustrated with an example. Section 4 presents our conclusions and future work. The complete implementation of the example presented for discovery can be found in Appendix A, and the semantics of rule reification in Appendix B

2 Semantic support in the context of WSMO and WSML

This section identifies the various areas in the process of dynamically building web-service-based applications where information about the semantics of a service as well as knowledge about his properties can be fruitfully applied.

Where can semantic annotation be exploited? In the process of building web-service-based software systems, we have to consider the following areas where elements of semantic descriptions of web services can be exploited. In each of these areas, we will then identify possible concrete reasoning tasks.

- Service Discovery
- Service Composition
- Service Invocation
- Service Mediation on the data, protocol and process level
- Service Execution Monitoring
- Service Compensation

Remark (Completeness of the list). Please note, that we don't consider this list to be complete in any sense. During the course of this project, we are going to extend and possibly refine it as far as this is relevant for the use cases that we aim to address.

Furthermore, in future versions of this document we will only consider those areas wherein we are able to identify substantial or important concrete reasoning task. □

Current restrictions. For the moment, we only consider WSMO-Standard [KLR04]. This excludes some important aspects that are included in the extension WSMO-Full.These aspects are mainly concerned with the business or application layer of WSMO, e.g. concepts like contract, contract templates, negotiations and protocols for the pre-negotiation phase and the post-negotiation phase etc.

Currently, WSMO-Full is in a very early and rapidly evolving stage. In the final version of WSMO-Full, it is our goal to cover all the related problems in detail. But at a first attempt, we focus here on specific reasoning tasks within WSMO-Standard e.g. goal-driven service discovery.

Moreover, in this version of the document, we focus on one specific aspect of the whole process: the service discovery task; and in particular on one specific aspect therein: the goal-capability-matching which lies in the very heart of the service discovery and is the most fundamental element of service discovery where automation is highly desired.

3 What Do We Want to Prove?

Now we want to take a closer look into each of the application areas that have been revealed by our abstract model and for which semantic annotation has to be processed and exploited. For each of these domains, we want to clarify how we can actually apply the knowledge contained in the WSMO description of semantic web services. For that purpose, we derive the corresponding proof obligations that are to be checked to automatize the tasks related to the different domains.

As mentioned in section 1, we principally aim at automated or semi-automated construction of dynamic software-systems with well-understood properties, whose architectural paradigm is exclusively based on web-services.

The question whether a given web service fulfills a requested task or whether two web services can interact with each other (and thus can actually be connected architecturally within a system) are examples for such properties of interest.

What is a proof obligation? A proof obligation is the precise formalization of a property of interest. In order to determine whether a given (concrete) system presents a given property e.g. whether a concrete web service fulfills a given task, the proof obligation associated to this property has to be proven.

3.1 Service Discovery

The web service technology, based on SOAP [W3C03], WSDL [CCMW01] and UDDI [BCE+02], provides syntactic means for seamless integration among disparate distributed applications. However, the current technology still requires significant amount of work, since all the plumbing among applications needs to be done manually. The above mentioned standards support only the syntactic aspects of the plumbing and are adequate only if the participating services are selected and hard-wired at design-time. Dynamic reconfiguration of services in order to automatically adapt to changes (e.g., when a provider goes off-line or when a cheaper provider enters the market) are not supported by the current technology.

In this context, we are interested in the automatic location of web services that can fulfill a given task specified by the requester. This scenario involves two parties between which a matching has to be established: A requester who is looking for a concrete service that solves a specific problem and a set of services that offer specific functionality to their clients. Both parties are basically interested in interacting with each other.

As an ontology for semantic web services, WSMO provides the semantic descriptions needed for dynamic location of web services that fulfill a given request. *Goals* in WSMO describe the objectives of a service requester; *Web Services* descriptions include the definition of the service *Capability* i.e. the functionality the web service offers. We define the problem of automatic service discovery as the problem of *matching the capabilities of existing web services against the goal described by the requester.*

3.1.1 Proof Obligations

Goals. Goals in WSMO describe what the requester wants to achieve; they consist of logical conditions that describe the desired state of the world and information space.

Capabilities. Service capabilities are described in WSMO by preconditions, assumptions, postconditions and effects [KLR04]. Preconditions the state of the information space the service expects prior to its invocation. Assumptions are similar but they refer to the state of the world. Postconditions describe what is guaranteed by the service to hold in the information space after its execution. Effects are similar but they refer to the state of the world.

Mediators. Mediators are modelling elements that bypass heterogeneity problems. They can link, resolving possible heterogeneities, goals to goals (*ggMediators*), ontologies to ontologies (*ooMediators*), and web services to goals (*wgMediators*). *wgMediators ooMediators* are of special interest for our discussion, as they link web services to goals and resolve heterogeneities in terms of the terminology used to describe those, respectively[2].

Formalization and scalability issues. Logic is a powerful framework for precise representation of statements about real-world objects or abstract artifacts. The most important property of a logic is that it comes with an abstract semantics and a *reasoning system*, which supports automatic ways for drawing provably correct conclusions from premises. A suitable logic is an appropriate tool to formalize goals, capabilities, mediators, and the proof obligations that are to be checked to determine a match between a user request and the functionality 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. Students have even greater difficulty 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**. Note that even clients who might be highly skilled in knowledge representation will not want to use anything more complex than the canned queries without strong incentives, since direct use of a logic requires familiarity with both the goal ontology and the ontologies used by the various services.

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 demands on the degree of sophistication of the engineers who might turn up on this type of labor market. The upshot of

[2]The role of *ggMediators* for web service discovery will be investigated in future versions of this document

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 must be designed with the above overall architecture in mind.

Proof obligations. We define the proof obligations for discovery 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 formalizations of the goal and of the service capability, respectively. The postcondition and effect parts of the goal G are denoted as G_{post} and G_{eff}. The preconditions, assumptions, postconditions and effects parts of the capability C are denoted as C_{prec}, C_{ass}, C_{post}, and C_{eff}, respectively.

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 might be needed because the goal ontology and the service ontologies might be very different.
- A mediator also needs to convert the goal into a postcondition and effect expressed in the service ontology, which is to be tested in the after-state of the service against the postconditions and effects of the service. This expressions are denoted as $Post_{wg}(G)$ and $Eff_{wg}(G)$.

Translations performed by wgMediators can be quite complex, because goals can be expressed in a very high-level syntax in order to make them usable by 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, i.e., that (after the appropriate translations) the goal is guaranteed to be true in the after-state of the service. Formally, this translates in the following proof obligation:

$$O, In_{wg}(G), C_{post}, C_{eff} \models Post_{wg}(G) \wedge Eff_{wg}(G) \quad (1)$$

The intuition behind proof obligation 1 is that we want to locate web services that can fulfill the goal when an appropriate input is provided to the service. In this way, we locate candidate services to fulfill the goal without considering the specific information e.g. login info that a specific service might require to provide its functionality.

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 more 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. Fourth, the assumptions of the service must be checked against the current state of the world (denoted by S) in order to guarantee that the service will delivered its results. Therefore, the following proof obligation has to be checked:

$$O, S, In_{wg}(G'), C_{post'}, C_{eff'} \models C_{ass} \wedge C_{prec} \wedge Post_{wg}(G') \wedge Eff_{wg}(G') \quad (2)$$

The difference between this and (1) is that possibly more complex version of the goal and of the capability postconditions and effects are used (denoted G', $C_{post'}$, and $C_{eff'}$, respectively) and that the preconditions and assumptions are checked. The proof obligation (2) can also be used for more precise discovery, which takes precondition into account. This is appropriate in situations where the user is willing to provide complete input during the discovery process.

However, the checking of the assumptions raise some problems. It might happen that some conditions on the state of the world cannot be checked with the available knowledge. This turns out in some services not being selected due to incomplete information about the state of the world. For that reason, we leave out the assumptions in our proof obligation for contracting. They will be modelled in the definition of the service with informative purposes, but not checked for contracting:

$$O, In_{wg}(G'), C_{post'}, C_{eff'} \models C_{prec} \wedge Post_{wg}(G') \wedge Eff_{wg}(G') \quad (3)$$

Proof obligations in transaction logic. In proof obligations (1) and (3) 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 none of $In_{wg}(G)$, C_{post}, and C_{eff} are part of a global knowledge base, and C_{post} and C_{eff} are different for different services. Since the effects in (1) and (3) 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 [BK95, BK98], which supports hypothetical assertions. This enables us to look at each service separately and hypothetically insert its postconditions and 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 and goal preconditions and effects relative to a service. Therefore, we will write $C_{post}(Serv)$, $C_{eff}(Serv)$, and $Post_{wg}(G, Serv)$, $Eff_{wg}(G, Serv)$, 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:

$$\begin{aligned} O \models \exists Serv \diamond (&insert\{In_{wg}(G), C_{post}(Serv), C_{eff}(Serv)\} \\ &\otimes Post_{wg}(G, Serv), Eff_{wg}(G, Serv)) \end{aligned} \quad (4)$$

The above query is looking for services such that $\diamond(insert\{In_{wg}(G), C_{post}(Serv), C_{eff}(Serv)\} \otimes Post_{wg}(G, Serv), Eff_{wg}(G, Serv))$ 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.

A similar query can be constructed for (3):

$$\begin{aligned} O \models \exists Serv \diamond (&insert\{In_{wg}(G'), C_{post'}(Serv), C_{eff'}(Serv)\} \\ &\otimes C_{prec}(Serv), Post_{wg}(G', Serv), Eff_{wg}(G', Serv)) \end{aligned} \quad (5)$$

Proof obligations (4) and (5) are the basis of our realization of the proposed framework.

3.1.2 Implementation.

For implementing the proof obligations shown before, we use $\mathcal{F}$-Flora-2. $\mathcal{F}$-Flora-2 is an implementation of a language that is closely related to *WSML*[3], the language being developed for WSMO. It supports F-logic [KLW95] and HiLog [CKW93], whose frame-based and higher-order syntax offers a simple and natural representation of the WSMO architectural components as well as the discovery query. It also supports enough of Transaction Logic [BK98] to be able to implement the proof obligations described above.

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, and the assumption represents the constraints on the initial state of the service execution. Different services can have different preconditions and assumptions and, therefore, they need to be represented as values of some attributes, which are used to specify concrete services. Preconditions and assumptions 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 [L*editors*99] and OWL [PSHH04] support a rudimentary form of reification, but not nearly enough for even simple preconditions.

Service postconditions and effects, which is the other major part of a service capability presents even greater challenge. Typically, they specify what would happen if the service were to be executed 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, which establishes that the facts in the head of the rule are true.[4] Since service postconditions and effects must also be reified in order to make it possible for a service object to refer to them, this means that the underlying logic language must be able to reify rules. This feature if provided in $\mathcal{F}$-Flora-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 (usually known under the name of *self-reference* in logic) is capable of producing logical paradoxes. An excellent introduction into the subject can be found in [Per85]. In [YK03b] it is shown that reification of queries does not cause paradoxes in a rule-based language like $\mathcal{F}$-Flora-2. However, this result only covers the precondition part of Web service capabilities. In B, we extend this result to reification of rules and thus ensure that the semantics is free of paradoxes and is adequate for handling service discovery.

3.1.3 A Walk-Through Example.

Here we show fragments of a larger running example that illustrates the $\mathcal{F}$-Flora-2 implementation of the proof obligations defined before. For the com-

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

[4] 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 $\mathcal{F}$-Flora-2 system, which serves as the platform for our implementation, is top-down.

plete example, see A.

In addition to showing concrete instances of service representation and of discovery queries, the example illustrates important architectural aspects of WSMO, such as *wgMediators*.

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.

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 most humans. However, the goal itself looks very simple: `search(germany, austria)`. Likewise, service capability descriptions are not explicitly written to support such queries either. 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 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[5] 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 (see A for the complete specification). 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 or 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[6]. 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[7] 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))
```

[5] In WSMO, our wgMediator would use a ooMediator, that would be modelled separately and included in the definition of the wgMediator. For simplicity purposes, we encode the mediation between the goal and service ontologies directly in the wgMediator

[6] This query is separated into postconditions and effects in WSMO and our proof obligations. As the need for this separation is not clear yet, and for simplicity purposes, we both together in the example

[7] Assumptions are left out in our example, and WSMO postconditions are included in the effects

```
                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 pass 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 the methods that 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 the mediator to construct the input appropriate for that 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 A

4 Conclusions

In this version of the deliverable we have only addressed the problem of service discovery in the context of WSMO. Given a description of what the requester wants to achieve (a *Goal*), and a set of descriptions of services that precisely define their functionalities (*Capabilities*), we have established the proof obligations that have to be proven in order to select services that can fulfill the requester goal.

We have distinguished two different kind of matches: *discovery*, which selects services that, given an appropriate input, can potentially fulfill the goal, and *contracting*, which given the actual and complete input for a concrete service, determines if the results of the execution of the service with such input leads to the expected results.

In addition, we have shown the implementation of the checking of the proof obligations in $\mathcal{F}$LORA-2 with a concrete example. Furthermore, such implementation can be generalized to other domains.

Our future work in discovery will focus on aligning our framework with the sublanguages of WSML that are being defined and on revisiting the modeling concepts of WSMO in light of the work presented in this document.

In addition to the work done on service discovery, future versions of this document will also include the proof obligations required for other tasks such as composition or compensation.

5 Acknowledgements

The work is funded by the European Commission under the projects DIP, Knowledge Web, Ontoweb, SEKT, SWWS, Esperonto, and h-TechSight; by Science Foundation Ireland under the DERI-Lion project; and by the Vienna city government under the CoOperate programme. The work is funded by the European Commission under the projects DIP, Knowledge Web, SEKT, SWWS and Esperonto; by Science Foundation Ireland under the DERI-Lion project; and by the Vienna city government under the CoOperate programme.

The editors would like to thank to all the members of the WSMO working group for their advice and input into this document.

References

[BCE+02] 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. http://uddi.org/pubs/uddi-v3.00-published-20020719.htm, July 2002.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley Pub Co, 2nd edition, 2003.

[BK95] Anthony J. Bonner and Michael Kifer. Transaction logic programming (or, a logic of procedural and declarative knowledge). Technical report, 1995. `ftp://ftp.cs.sunysb.edu/pub/TechReports/kifer/transaction-logic.ps.Z`.

[BK98] A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.

[BLFD99] T. Berners-Lee, M. Fischetti, and T. M. Dertouos. *Weaving the Web*. Harper, San Franscisco, 1999.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001. http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21&ref=sciam.

[CCMW01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. http://www.w3.org/TR/wsdl, March 2001.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[DFvH03] J. Davies, D. Fensel, and F. van Harmelen, editors. *Towards the Semantic Web: Ontology-Driven Knowledge Management*. Wiley, 2003.

[Fen03] D. Fensel. *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce, 2nd edition.* Springer-Verlag, Berlin, 2003.

[FHLW03] Dieter Fensel, James Hendler, Henry Liebermann, and Wolfgang Wahlster, editors. *Spinning the Semantic Web.* MIT Press, 2003.

[GS96] David Garlan and Mary Shaw. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[KLR04] Uwe Keller, Holger Lausen, and Dimitriu Roman (*editors*). Web Service Modeling Ontology – Standard (WSMO-Standard). Working draft, Digital Enterprise Research Insitute (DERI), March 2004. Look at `http://www.wsmo.org/2004/d2/v0.2`.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *JACM*, 42(4):741–843, 1995.

[L*editors*99] Ora Lassila and Ralph R. Swick (*editors*). Resource description framework (RDF) model and syntax specification. Recommendation, W3C, February 1999. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.

[Per85] D. Perlis. Languages with self-reference i: Foundations. *AI*, 25:301–322, 1985.

[PSHH04] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL web ontology language semantics and abstract syntax. W3C Recommendation, W3C, February 2004.

[W3C03] W3C. Soap version 1.2 part 0: Primer. http://www.w3.org/TR/2003/REC-soap12-part0-20030624/, June 2003.

[YK02] G. Yang and M. Kifer. Well-founded optimism: Inheritance in frame-based knowledge bases. In *International Conference on Ontologies, Databases, and Applications of Semantics (ODBASE)*, 2002.

[YK03a] G. Yang and M. Kifer. Inheritance and rules in object-oriented semantic web languages. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML)*, 2003.

[YK03b] G. Yang and M. Kifer. Reasoning about anonymous resources and meta statements on the semantic web. *Journal of Data Semantics*, 1:69–97, 2003.

A Complete example for service discovery

```
/*
** Title: SERVICE DISCOVERY WITH MEDIATORS
**
** Features:  - WG-mediators
**            - complex goals
**            - rules in effects and goals
**            - service effects can depend on input
*/

//?- debug[#check_undefined(on)]@flora(sys).  // use debug mode


/*
** A taxonomy of cities.
**       europe means European Cities
**       france - French cities, etc.
**       tyrol - Tyrolean cities, etc.
*/
usa::america.
germany::europe.
austria::europe.
france::europe.
tyrol::austria.
nystate::usa.
stonybrook:nystate.
innsbruck:tyrol.
lienz:tyrol.
vienna:austria.
bonn:germany.
frankfurt:germany.
paris:france.
nancy:france.
// regions - things like europe, germany, tyrol
europe:region.
america:region.
// any subregion of a region is also a region
Reg:region :- Reg1:region and Reg::Reg1.

/*
  Services/clients write their descriptions/queries to conform to specific
  ontologies. Most of the intelligence lies in the mediators, which are
  assumed to be written by skilled professionals.
  Client goals are assumed to be written by naive users and thus are the
  simplest.
  Service descriptions are written by knowledge engineers. They can be
  more complex, but shouldn''t require a Ph.D. in knowledge representation.

  Goal ontology:
      Clients' goals use the Goal Ontology, which provides primitives for
      discovering and contracting services.
      The primitives support a range of discovery tasks, from least specific
      to more specific.

      Discovery goals:

          searchTrip(from,to)  - from/to can be cities or regions; if a
                                 region then it means the requested service
```

```
                              must serve every city in that region that
                              is known to the KB. Assume either both are
                              cities or both are regions.
        searchCitipass(loc)  - citipass search; loc can be a city or a region

    Contracting goals:
        tripContract(serviceId,fromCity,toCity,date,creditCard)
        citipassContract(serviceId,city,date,creditCard)

Goals have the form:

    goalId[requestId->someId, query->queryType]

where
  someId      - the request Id,
  goalId      - goal Id
  queryType   - a discovery/contracting primitive described above

    Results of a search are stored in the attribute result, e.g.,

        goal1[result ->> serv1,serv2]

    The result of a contract execution is stored in the attribute
    confirmation, e.g.,

        goal2[confirmation -> info(service,confNumber,from,to,date)]
       or
        goal2[confirmation -> info(service,confNumber,city,date)]

Service ontologies:
    Services represent their inputs and outputs using ontologies that can
    possibly be different from the ontology used by the users.
    In our examples, services use two different ontologies. The ontology
    mismatch is resolved using the Web service-to-goal mediators
    (wg-mediators). The wg-mediators for the Request ontology and the
    two service ontologies are defined separately below.

    Service Ontology #1
       Inputs have the following form:
            search(requestId,fromLocation,toLocation)
            search(requestId,city)
            contract(requestId,fromLocation,toLocation,date,ccard)
            contract(requestId,city,date,ccard)

         Input is constructed from the client's goal by the mediator and
         passed to the service.

       The output produced by the service has the form

         for searches:
            itinerary(reqNumber)[from->fromCity, to->toCity]
            passinfo(reqNumber)[city->City]

                Note: for searches, services assume that the input provides
                      a specific pair of cities. Services know nothing about
                      searches by regions, so the descriptions of their
                      capabilities are relatively simple. Region-based
                      queries are constructed by mediators. This is what we
                      mean when we say that most of the intelligence is
```

```
                      in the mediators.

          for contracting:
             ticket(reqNumber)[confirmation->confNumber,
                               from->fromCity, to->toCity, date->someDate]
             pass(reqNumber)[confirmation->confNumber,
                             city->City, date->someDate]

      Service Ontology #2
          Provides basically the same information, but uses different
          representation (to illustrate the idea of different mediators).
          Services that use this ontology only sell citipasses.

  Inputs have the following form:
             discover(requestId,city)
             pay(requestId,city,date,ccard)

          The output looks like this:
             reqNumber[location->city)]  for searches
             reqNumber[confirmation->(number,city,date)] for purchases
*****************************************************************************/

/*********************** Available services ******************************
  Assume precondition/effects to be mandatory and have uniform representation
  for all services. In general, we could use mediators that the discovery and
  contracting query could invoke to reconcile the different representations.
*****************************************************************************/

// This service uses Ontology #1 and mediator1 to map client ontology
// to Ontology #1
serv1[
   // Input must be a request for ticket from somewhere in Germany to somewhere
   // in Austria  OR  a request for a city pass for a city in Tyrol
   capability->
        _#[
   precondition(Input) ->
                  $
     (Input = contract(_, From:germany, To:austria, Date, Card)
      or Input = contract(_, City:tyrol, Date, _))
     and validDate(Date) and validCard(Card)
   ,
   effects(Input) ->
  $
      // Note: repeating some preconditions, like From:germany,
      //       because precondition(Input) is not checked
      //       during discovery, but From:germany, To:austria
      //       can be relevant to discovery. However, not all
      //       of the precondition is copied here -- only what
      // is relevant for discovery.
      (itinerary(Req)[from->From,to->To] :-
Input = search(Req, From:germany, To:austria))
      and
      (passinfo(Req)[city->City] :-
Input = search(Req,City:tyrol))
      and
      // Note: precondition is checked at invocation, so
      // no need to repeat those tests here.
      (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
].

// Another Ontology #1 service
serv2[
   capability->
        _#[
   precondition(Input) -> $
      // Input must be a request for a ticket from a
                     // city in France or Germany to a city in Austria
      Input = contract(_, From:(france or germany),
                          To:austria, Date, Card)
      and validDate(Date) and validCard(Card)
   ,
   effects(Input)-> $
      (itinerary(Req)[from->From, to->To] :-
  Input = search(Req,
         From:(france or germany), To:austria))
      and
      (ticket(Req)[confirmation->Num,
                  from->From, to->To, date->Date] :-
    Input = contract(Req,From,To,Date,_CCard) and
    generateConfNumber(Num))

 ],

    usedMediators ->> med1

].

// An Ontology #2 service
serv3[
   capability->
        _#[
   precondition(Input) ->
       $
      // request for a pass for a French city
                              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
].

// Another Ontology #2 service
serv4[
   capability->
        _#[
   precondition(Input) ->
                     $
      // can do passes in any city except Paris
                              Input = pay(_,City:france,Date,Card)
      and City ~ paris
      and validDate(Date) and validCard(Card)
     ,
   effects(Input)->
                     $
       (Req[location->City] :-
Input = discover(Req,City:france)
        and City ~ paris)
       and
       (Req[confirmation->(Num,City,Date)] :-
Input = pay(Req,City,Date,_Card) and
generateConfNumber(Num))

 ],

    usedMediators ->> med2
].


/****************************** GOALS ************************************
        Goals are objects that have queries written to the Goal ontology spec
*********************************************************************************/

goal1[
   requestId -> _#,
   query -> searchTrip(bonn,innsbruck),
   result->>
].

goal2[
   requestId -> _#,
   query -> tripContract(serv1,bonn,innsbruck,'1/1/2007',1234567890),
   result->>
].

// need services that serve all cities in France and Austria
// WG-mediators will generate the appropriate queries to the services
goal3[
   requestId -> _#,
   query -> searchTrip(france,austria),
   result->>
].

goal4[
   requestId -> _#,
   query-> searchCitipass(frankfurt),
   result->>
```

```
].

goal5[
   requestId -> _#,
   query-> searchCitipass(innsbruck),
   result->>
].


goal6[
   requestId -> _#,
   query -> searchCitipass(france),
   result->>
].


goal7[
   requestId -> _#,
   query -> citipassContract(serv4,nancy,'22/2/2005',0987654321),
   result->>
].


/************************* Mediators ************************************
  A mediator needs to:
    1. Convert input - <mediatorId>[constructInput(Goal)->Input]
    2. Construct the query to be used for testing the after-state of service
       This is done by <mediatorId>[reportResult(Goal,Result)]

       This method tests that, after the appropriate translation,
       the goal is satisfied in the after-state of the service.
       Result gets bound to a formula that is appropriate for the
       representation of results in the goal ontology. That is,
       it looks like goal[result->>...] or goal[confirmation->...].
       See the header of this file for the explanations of how the goal
       ontology looks like.
*************************************************************************/

//  ******************** MEDIATOR 1 **********************
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 Query = tripContract(ServiceId,From,To,Date,CCard)
        then (
       generalizeArg(From, From1), generalizeArg(To, To1),
       Input = contract(ReqId,From1,To1,Date,CCard)
        ) else if Query = citipassContract(ServiceId,City,Date,CCard)
        then (
       generalizeArg(City, City1),
       Input = contract(ReqId,City1,Date,CCard)
        ) else fail.
```

```
med1[reportResult(Goal,Serv,Result)] :-
Goal[query->searchTrip(From:location,To:location)] and
        itinerary(_)[from->From, to->To],
Result = $Goal[result->>Serv].
med1[reportResult(Goal,Serv,Result)] :-
Goal[query->searchTrip(From:region,To:region)] and
        refresh_[doesNotServeCity(_,_)],
        not med1[doesNotServeCity(From,To)],
Result = $Goal[result->>Serv].

med1[reportResult(Goal,Serv,Result)] :-
Goal[query->searchCitipass(City:location)] and
        passinfo(_)[city->City],
Result = $Goal[result->>Serv].
med1[reportResult(Goal,Serv,Result)] :-
Goal[query->searchCitipass(Region:region)] and
        refresh_[doesNotServeCity(_)],
not med1[doesNotServeCity(Region)],
Result = $Goal[result->>Serv].

// contracting requests
med1[reportResult(Goal,Result)] :-
Goal[query->tripContract(Serv,From,To,Date,_CCard)] and
ticket(_)[confirmation->Num, from->From, to->To, date->Date],
Result = $Goal[confirmation->info(Serv,Num,From,To,Date)].
med1[reportResult(Goal,Result)] :-
Goal[query->citipassContract(Serv,City,Date,_CCard)] and
pass(_)[confirmation->Num, city->City, date->Date],
Result = $Goal[confirmation->info(Serv,Num,City,Date)].

// for region-level queries check if there is a city that is not served
med1[doesNotServeCity(FromReg,ToReg)] :-
City1:FromReg and City2:ToReg and
        not itinerary(_)[from->City1, to->City2].
med1[doesNotServeCity(Region)] :-
City:Region and
        not passinfo(_)[city->City].

//  ******************** MEDIATOR 2 **********************
med2[constructInput(Goal)->Input] :-
Goal[requestId->ReqId, query->Query] and
        if Query = searchCitipass(City)
        then (
       generalizeArg(City, City1),
       Input = discover(ReqId,City1)
        ) else if Query = citipassContract(_ServiceId,City,Date,CCard)
        then (
       generalizeArg(City, City1),
       Input = pay(ReqId,City1,Date,CCard)
        ) else fail.

med2[reportResult(Goal,Serv,Result)] :-
Goal[query->searchCitipass(City:location)] and
        _[location->City],
Result = $Goal[result->>Serv].
med2[reportResult(Goal,Serv,Result)] :-
Goal[query->searchCitipass(Region:region)] and
        refresh_[doesNotServeCity(_)],
not med2[doesNotServeCity(Region)],
```

```
Result = $Goal[result->>Serv].

// for region-level queries check if there is a city that is not served
med2[doesNotServeCity(Region)] :-
City:Region and
        not _[location->City].

// contracting request
med2[reportResult(Goal,Result)] :-
Goal[query->citipassContract(Serv,City,Date,_CCard)] and
_[confirmation->(Num,City,Date)],
Result = $Goal[confirmation->info(Serv,Num,City,Date)].


/****************** A generic service discovery query **********************
  Given a goal, find all services that match and print out their Ids.
  Represented as a transaction because it uses hypothetical updates.
  Hypothetical updates are simulated by insert/delete because Flora-2
  doesn''t support the hypotheticals.
*******************************************************************************/
#find_service(Goal) :-
Serv[usedMediators ->> Mediator[constructInput(Goal) -> Input]],
Serv.capability[effects(Input) -> Effects],
        insertruleEffects,      // hypothetically assume the effects

// Check if the goal is satisfied by the service and report result
if Mediator[reportResult(Goal,Serv,Result)] then insertResult,
// Remove the hypothetically added rules
deleteruleEffects,
fail.
#find_service(_Goal) :- true.

/************************ Service contracting ****************************
  Similar to discovery, but also checks precondition
*************************************************************************/
#contract_service(Goal) :-
// get the service to invoke: contracting queries have 4 or 5 args
(Goal.query = _(Serv,_,_,_,_) or Goal.query = _(Serv,_,_,_)),
Serv[usedMediators ->> Mediator[constructInput(Goal) -> Input]],
Serv.capability.precondition(Input)[],
Serv.capability[effects(Input) -> Effects],
        insertruleEffects,      // hypothetically assume effects
// Check if the goal is satisfied by the service and report result
if Mediator[reportResult(Goal,Result)] then insertResult,

// Remove the hypothetically added facts and rules
deleteruleEffects,
fail.
#contract_service(_Goal) :- true.


// %%%%%%%%%%%%%% MISC DEFINITIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

// use Prolog''s gensym/2 to generate a new conf number
generateConfNumber(Num) :- gensym(conf,Num)@prolog(gensym).

validDate(_). // pretending that we check dates

validCard(_). // pretending that we check credit cards
```

```
// if an arg is a region - replace with new variable
generalizeArg(In,_Out) :- nonvar(In), In:region, !.
generalizeArg(_In,_In) :- true.



// %%%%%%%%%%%%%% Sample service discovery requests %%%%%%%%%%%%%%%%%%%%%%%%%
/*
  // serv1, serv2
  ?- #find_service(goal1), goal1[result->>Serv].

  // should succeed for service 1
  ?- #contract_service(goal2), goal2[confirmation->Info].

  // serv2
  ?- #find_service(goal3), goal3[result->>Serv].

  // none
  ?- #find_service(goal4), goal4[result->>Serv].

  // serv1
  ?- #find_service(goal5), goal5[result->>Serv].

  // serv3 only. serv4 does not match because it does not serve Paris
  ?- #find_service(goal6), goal6[result->>Serv].

  // should succeed for serv4
  ?- #contract_service(goal7), goal7[confirmation->Info].

*/
```

B Semantics of rule reification

In [YK03b], Yang and Kifer extended the F-logic language with reification. However, rule reification, which is heavily relied upon in the realization of our framework presented in Section **??**, 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 [KLW95]. As before, we adopt the usual Prolog convention that capitalized symbols denote variables, while symbols beginning with a lowercase letter denote constants. 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, please see [KLW95].

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 $\leftarrow$; 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 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 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 $(A_1 \leftarrow A_2)$ is a 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, $p \leftarrow q$ is a term, while $p \vee \neg q$ is a generalized term but not a term. Therefore, $p \leftarrow q$ is not considered a shortcut for $p \vee \neg q$. The intuition behind the distinction between terms and generalized terms is that only terms can occur in the head of a rule, but the body can contain generalized terms. The reason is that we do not allow arbitrary disjunction or negation to appear in a rule head. We do allow, however, a term in the form of a rule to appear in a rule head.

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 $\phi \leftarrow \psi$ are called* **rule formulas**.[8]

[8] If the anonymous identity semantics as defined in [YK03b] is to be combined, we need to define flat formulas in that paper in the same way we define formula here.

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 (see Definition 1) constructed using the constants in $\mathcal{C}$. Such variable-free terms are called ground. Obviously $\mathcal{HU}$ is countably infinite.* [9]

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

$$\forall(\mathsf{A}_1 \wedge \ldots \wedge \mathsf{A}_m \leftarrow \mathsf{B}_1 \wedge \ldots \wedge \mathsf{B}_n) \tag{6}$$

where $\mathsf{m} \geq 1$, $\mathsf{n} \geq 0$, A_i $(1 \leq i \leq m)$ and B_j $(1 \leq j \leq n)$ are atomic formulas or rule formulas in the form of (6).

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 [YK02] and [YK03a].

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 $\phi = \psi \leftarrow \xi$, then $\mathcal{I} \models \phi$ iff $\psi \leftarrow \xi \in \mathcal{I}$ 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 $\phi = \exists \mathsf{X}\psi$, then $\mathcal{I} \models \phi$ iff there is $\mathsf{t} \in \mathcal{HU}$ such that $\mathcal{I} \models \psi[\mathsf{X}/\mathsf{t}]$, where $\psi[\mathsf{X}/\mathsf{t}]$ denotes the formula obtained from ψ substituting t for all free occurrence of the variable X.*
- *If $\phi = \forall \mathsf{X}\psi$, then $\mathcal{I} \models \phi$ iff for all $\mathsf{t} \in \mathcal{HU}$, $\mathcal{I} \models \psi[\mathsf{X}/\mathsf{t}]$.*

Note that the term $\mathsf{p} \leftarrow \mathsf{q}$ 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

[9] Note that since atomic formulas are reified, the Herbrand base in classical logic programming is the same as the Herbrand universe in our case.

a model of $\mathsf{p} \leftarrow \mathsf{q}$, 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 $\mathsf{p} \leftarrow \mathsf{p}$, 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 $\mathcal{T}_{(P)}$ maps an interpretation $\mathcal{I}$ to another interpretation $\mathcal{J}$, denoted $\mathcal{T}_{(P)}(\mathcal{I}) = \mathcal{J}$, where $\mathcal{J}$ is a set of terms A such at

- *There is an instantiated rule $H \leftarrow B_1, \wedge \ldots \wedge B_n$ in P such that A is one of the conjuncts in H; and*
- *$B_j \in \mathcal{I}$ for all B_j, $1 \leq j \leq n$.*

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

The following example illustrates the computation of the least model of a program. Let the program be:

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

we obtain:

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

$\mathcal{I}_3$ is the fixpoint. Note that c is not included in $\mathcal{I}_3$ while it would have been if we treated $\mathsf{a} \leftarrow \mathsf{b}$ 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**:

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

While one may think that the comprehension axiom is too general to be useful, the following simpler truth axiom is less esoteric:

$$\forall \mathsf{X}(\mathsf{true}(\mathsf{X}) \leftrightarrow \mathsf{X}) \tag{7}$$

Unfortunately, it turns out to be an instance of the comprehension axiom and is almost as bad. The following example is adapted from [YK03b]. Consider

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

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

In [YK03b], 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 (7) 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 X $\longleftarrow$ body, but still allows rules like X(Y) $\longleftarrow$ body or X[Y->>Z] $\longleftarrow$ 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).