IRS-II: A Framework and Infrastructure for Semantic Web Services

Enrico Motta[1], John Domingue[1], Liliana Cabral[1], and Mauro Gaspari[2]

[1] Knowledge Media Institute, The Open University, Milton Keynes, UK
{E.Motta, J.B.Domingue, L.S.Cabral}@open.ac.uk
[2] Dipartimento di Scienze dell'Informazione, University of Bologna, Italy
gaspari@cs.unibo.it

Abstract. In this paper we describe IRS-II (Internet Reasoning Service) a framework and implemented infrastructure, whose main goal is to support the publication, location, composition and execution of heterogeneous web services, augmented with semantic descriptions of their functionalities. IRS-II has three main classes of features which distinguish it from other work on semantic web services. Firstly, it supports one-click publishing of standalone software: IRS-II automatically creates the appropriate wrappers, given pointers to the standalone code. Secondly, it explicitly distinguishes between tasks (what to do) and methods (how to achieve tasks) and as a result supports *capability-driven* service invocation; flexible mappings between services and problem specifications; and dynamic, knowledge-based service selection. Finally, IRS-II services are web service compatible – standard web services can be trivially published through the IRS-II and any IRS-II service automatically appears as a standard web service to other web service infrastructures. In the paper we illustrate the main functionalities of IRS-II through a scenario involving a distributed application in the healthcare domain.

1 Introduction

Web services promise to turn the web of static documents into a vast library of interoperable running computer programs and as such have attracted considerable interest, both from industry and academia. For example, IDC [8] predicts that the Web Services market, valued at $416 million in 2002, will be worth $2.9 billion by 2006.

Existing web service technologies are based on a manual approach to their creation, maintenance and management. At the centre of the conceptual architecture is a registry which stores descriptions of published web services. Clients query the registry to obtain relevant details and then interact directly with the deployed service. The descriptions, represented in XML based description languages, such as WSDL [17] and UDDI [16], mostly focus on the specification of the input and output data types and the access details. These specifications are obviously not powerful enough to support automatic discovery, mediation and composition of web services. A software agent cannot find out what a web service actually does, by reasoning about a WSDL specification. Analogously the same agent cannot locate the appropriate service in a

UDDI registry, given the specification of a target functionality. As a result, existing web service infrastructures by and large support a manual approach to web service management: only manual discovery is supported and only 'static', manually configured web applications are possible.

The above issues are being addressed by ongoing work in the area of *semantic web services* [3, 5, 14]. The basic idea here is that by augmenting web services with rich formal descriptions of their competence many aspects of their management will become automatic. Specifically, web service location, composition and mediation can become dynamic, with software agents able to reason about the functionalities provided by different web services, able to locate the best ones for solving a particular problem and able to automatically compose the relevant web services to build applications dynamically. Research in the area is relatively new and although a number of approaches have been proposed, such as DAML-S [3] and WSMF [5], no comprehensive tool infrastructures exist, which support the specification and use of semantic web services.

In this paper we describe IRS-II (Internet Reasoning Service) a framework and implemented infrastructure which supports the publication, location, composition and execution of heterogeneous web services, augmented with semantic descriptions of their functionalities. IRS-II has three main classes of features which distinguish it from other work on semantic web services.

Firstly, it supports *one-click publishing* of 'standard' programming code. In other words, it automatically transforms programming code (currently we support Java and Lisp environments) into a web service, by automatically creating the appropriate wrappers. Hence, it is very easy to make existing standalone software available on the net, as web services.

Secondly, the IRS-II builds on knowledge modeling research on reusable components for knowledge-based systems [2, 6, 9, 10], and as a result, its architecture explicitly separates task specifications (the problems which need to be solved), from the method specifications (ways to solve problems), from the domain models (where these problems, which can be generic in nature, need to be solved). As a consequence, IRS-II is able to support *capability-driven* service invocation (find me a service that can solve problem X). Moreover, the clean distinction between tasks and methods enables the specification of flexible mappings between services and problem specifications, thus allowing a n:m mapping between problems and methods and a dynamic, knowledge-based service selection.

Finally, IRS-II services are web service compatible – standard web services can be trivially published through the IRS-II and any IRS-II service automatically appears as a standard web service to other web service infrastructures.

The paper is organized as follows: in the following section we outline our overall approach. We then describe the IRS-II framework in detail and illustrate its main components through a scenario involving a distributed healthcare application. The final section of the paper contains our conclusions.

2 The IRS-II Approach

Work on the IRS-II began in the context of the IBROW project [1], whose overall goal was to support on-the-fly application development through the automatic configuration of reusable knowledge components, available from distributed libraries on the Internet. These libraries are structured according to the UPML framework [6], which is shown in figure 1. The UPML framework distinguishes between the following classes of components:

- *Domain models*. These describe the domain of an application (e.g. vehicles, a medical disease).
- *Task models*. These provide a generic description of the task to be solved, specifying the input and output types, the goal to be achieved and applicable preconditions. Tasks can be high-level generic descriptions of complex classes of applications, such as Classification or Scheduling, as well as more 'mundane' problem specifications, such as Exchange Rate Conversion.
- *Problem Solving Methods (PSMs)*. These provide abstract, implementation-independent descriptions of reasoning processes which can be applied to solve tasks in specific domains. As in the case of task models, these can be high-level, generic PSMs such as Heuristic Classification [2] and Propose&Revise [9], or they can be specialized methods applicable to fine-grained tasks, such as Exchange Rate Conversion.
- *Bridges*. These specify mappings between the different model components within an application. For example, the refinement process in heuristic classification may be mapped onto a taxonomic hierarchy of attributes within some domain, in order to construct a specific application.

Each class of component is specified by means of an appropriate *ontology* [7].

The main advantage of this framework from an epistemological point of view is that it clearly separates the various components of knowledge-based applications, thus providing a theoretical basis for analyzing knowledge-based reasoners and an engineering basis for structuring libraries of reusable components, performing knowledge acquisition, and carrying out application development by reuse [10].

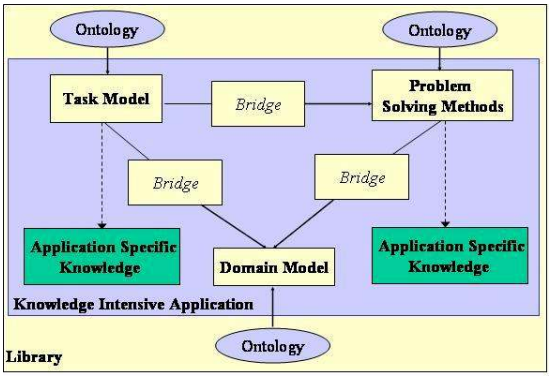

Fig. 1. The UPML framework.

The application of the UPML framework to semantic web services also provides a number of advantages and in our view our framework compares favorably with approaches such as DAML-S, where services are arranged in hierarchies and no explicit notion of task is provided – tasks are defined as service-seeking agents. In DAML-S tasks are always application specific, no provision for task registries is envisaged. In contrast in our approach, tasks provide the basic mechanism for aggregating services and it is possible to specify service types (i.e., tasks), independently of specific service providers. In principle this is also possible in DAML-S. Here a task would be defined as a service class, say S, and its profile will give the task definition. However, this solution implies that all instances of S will inherit the task profile. This approach is not very flexible, given that it makes it impossible to distinguish (and to reason about) the differences between the profile of a task (service class) and the profile of a method (specific service provider) – attributes are inherited down is-a hierarchies. In particular, in some cases, a method may only solve a weaker form of a task, and it is therefore important for a brokering agent to be able to reason about the task-method competence matching, to decide whether it is OK to use the weaker method in the given scenario. For instance, in a currency conversion scenario, a task specification may define currency conversion rates in terms of the official FT quotes, but different service providers may adopt other conversion rates. By explicitly distinguishing between tasks and methods we provide a basic framework for representing these differences and for enabling *matchmaking agents* [15] to reason about them.

The separation between tasks and methods also provides a basic model for dealing with ontology mismatches. While in DAML-S subscribing to a Service Class implies a strong ontological commitment (i.e., it means to define the new service as an instance of the class), the UPML framework assumes that the mapping between methods and tasks may be mediated by *bridges*. In practice this means that if task T is specified in ontology A and a method M is specified in ontology B, which can be used to solve T, it is still possible to use M to solve T, provided the appropriate bridge is defined.

Finally, another advantage of our approach is that the task-method distinction also enables capability-driven service invocation. While this is also possible in principle in approaches such as DAML-S, as discussed above, our approach provides both an

explicit separation between service types and service providers and more flexibility in the association between methods and tasks.

3 IRS-II Architecture

The overall architecture of the IRS-II is shown in figure 2. The main components are the IRS Server, the IRS Publisher and the IRS Client, which communicate through a SOAP-based protocol [13].

3.1 IRS Server

The IRS server holds descriptions of semantic web services at two different levels. A knowledge level description is stored using the UPML framework of tasks, PSMs and domain models. These are currently represented internally in OCML [10], an Ontolingua-derived language which provides both the expressive power to express task specifications and service competencies, as well as the operational support to reason about these. Once rule and constraint languages are developed for OWL [12], we will provide the appropriate import/export mechanisms. In addition we have also special-purpose *mapping mechanisms* to connect competence specifications to specific web services. These correspond to the notion of *grounding* in DAML-S.

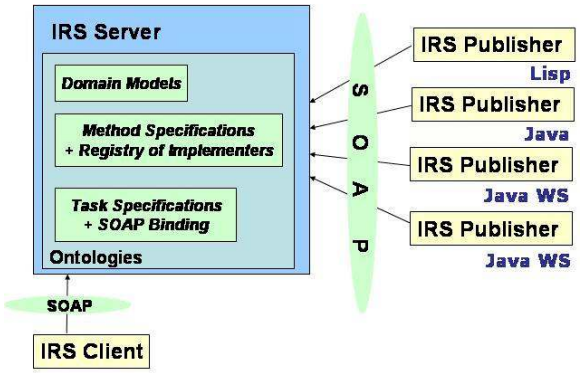

Fig. 2. The IRS-II architecture

3.2 Task Descriptions

An example task description, `exchange_rate_provision`, is shown in figure 3. As can be seen in the figure the task has two input roles, a `source_currency` and a `target_currency`, and one output role, the `exchange_rate`. The supporting definitions, such as `currency` and `positive_number`, are defined in the task ontology associated with this task, or in ontologies included by it.

```
(def-class exchange_rate_provision (goal-specification-task)
  ?task
  ((has-input-role   :value has_source_currency
                     :value has_target_currency)
   (has-output-role :value has_exchange_rate)
   (has_source_currency :type currency :cardinality 1)
   (has_target_currency :type currency :cardinality 1)
   (has_exchange_rate :type positive_number)
 (has-goal-expression
    :value (kappa (?psm ?sol)
                  (= ?sol (the_official_exchange_rate
                           (role-value ?psm has_source_currency)
                           (role-value
                            ?psm has_target_currency)))))))
```

Fig. 3. Definition of the `exchange_rate_provision` task.

Web service mediation and composition are supported by task preconditions and goal expressions. No precondition is specified for this task, although the specifications of the input roles implicitly state that one (and no more than one) source and target currency have to be specified. The goal expression states that the output for the task must be compliant with the "official exchange rate", as specified in the relevant ontology.

```
(def-irs-soap-bindings
 exchange_rate_provision_ontology        ;;ontology name
 exchange_rate_provision                 ;;task name
 ((has-source-currency "xsd:symbol")     ;;source currency
  (has-target-currency "xsd:symbol"))    ;;target currency
 "xsd:float")                            ;;output
```

Fig. 4. The soap-bindings for the `exchange_rate_provision` task.

The integration of semantic specifications and web service descriptions is achieved at the task level by means of *SOAP bindings*. A SOAP binding maps the input and output roles onto SOAP types - the soap binding for the `exchange_rate_provision` task is shown in figure 4. The binding specifies that the input roles, `source_currency` and `target_currency`, are mapped to the SOAP type `xsd:symbol` and the output role is mapped to the SOAP type `xsd:float`. The relation between SOAP types and ontological input and output types is analogous to the distinction between knowledge and symbol level in knowledge-based systems [11]. The ontology specifies what knowledge is required and produced; the SOAP types specify the way this knowledge is effectively encoded in the symbol-level communication mechanism. Hence, any web service which solves a particular task must comply with both knowledge and symbol level requirements, or alternatively, bridges need to be defined to ensure interoperability.

3.3 Problem Solving Methods

The IRS server holds both the method descriptions (PSMs) and a registry of web services, which implement them. An example PSM, which tackles the `exchange_rate_provision` task, is shown in figure 5. We can see that the type of the input roles has been constrained from currency to `european_currency`. Also pre and post conditions have been introduced.

The precondition states that the bank must have available stock of the target currency, whilst the post-condition states that the rate provided is the one supplied by the European Central Bank (ECB). Hence, this particular service may or may not 'solve' the exchange provision task, depending on whether the exchange rate provided by ECB is the same as the one required by the task, or whether the matchmaking agent is happy to consider them as 'close enough'. The explicit distinction between tasks and PSMs makes it possible to precisely specify, by means of ontologies, both the problems to be addressed and the different ways to address them and provides a basis to matchmaking agents to reason about the method-to-to-task mapping and to mediation services to try and 'bridge the gap' between service requirements and service providers.

In a similar fashion to tasks, web service discovery is supported by the pre and post conditions. For instance, the conditions formulated in the `MM_Bank_exchange_rate_provider` PSM can be used to answer agent queries such as "which exchange rate services focus on European currencies" and "which exchange rate services are able to change 250K pounds into euros?".

```
(def-class MM_Bank_exchange_rate_provider (primitive-method)
 ?psm
 ((has-input-role
   :value has-source-currency
   :value has-target-currency)
  (has-output-role
   :value has-exchange-rate)
  (has-source-currency :type european_currency :cardinality 1)
  (has-target-currency :type european_currency :cardinality 1)
  (has-exchange-rate :type positive_number)
  (has-precondition
   :value (kappa (?psm) (stock-available
                         (role-value ?psm has-target-currency))
  (has-postcondition
   :value (kappa (?psm ?sol)
           (= ?sol (the-European-Central-Bank-exchange-rate
                    (role-value ?psm has-source-currency)
                    (role-value ?psm has-target-currency)))))))
```

Fig. 5. A PSM which addresses the `exchange_rate_provision` task.

3.4 IRS Publisher

The IRS Publisher plays two roles in the IRS-II framework. Firstly, it links web services to their semantic descriptions within the IRS server. Note that it is possible to have multiple services described by the same semantic specifications (i.e., multiple implementation of the same functionality), as well as multiple semantic specifications of the same service. For instance, the same exchange rate converter can be described using two different ontologies for the financial sector.
Secondly, it automatically generates a set of wrappers which turn standalone Lisp or Java code into a web service described by a PSM. Standalone code which is published on the IRS appears as a standard java web service. That is, a web service endpoint is automatically generated.

Fig. 6. The IRS-II form based interface for publishing a web service.

Web services can be published using either the IRS Java API or the Publisher form based interface. Figure 6 shows an IRS-II user publishing a web service which implements the `MM_Bank_exchange_rate_provider`. As it can be seen from the figure, publishing a standard web service through the IRS is very easy. All the web service developer has to do to is:

1. Specify the location of the IRS server via a host and port number.
2. Indicate the PSM implemented by the service by providing its name and ontology. The menu of available PSMs is generated automatically once the location of the IRS server has been specified.
3. Specify the endpoint for the web service. If the 'service' in question is a piece of java code, specified as <java class, java method>, then the appropriate wrapper and an end-point are automatically generated by the IRS publisher

Once the 'Publish Web Service' button has been pressed, a SOAP message encoding the information in the form is sent to the IRS server where an association between the PSM and the web service endpoint is stored in the registry of implementers. A Java API, which replicates the functionality of the form, also exists.

As we mentioned earlier the IRS Publisher also allows standalone Java and Lisp code to be turned into a web service and associated with a PSM through a simple API. For Lisp a macro `irs-method-registration` is used - an example for the `MM_Bank_exchange_rate_provider` PSM is given in figure 7. When the form in figure 7 is executed, a set of wrappers are generated which make the function `mm-exchange-rate available` as a web service. Executing a second IRS form (`publish-all-services`) sends the description and location of all the newly created web services to the IRS server. The IRS Server automatically generates an endpoint, which enables the Lisp function to be accessed as a standard web service.

```
(irs-method-registration
  MM_Bank_exchange_rate_provider_ontology   ;; the ontology
  MM_Bank_exchange_rate_provider            ;; the PSM
  mm-exchange-rate)                         ;; the Lisp function
```

Fig. 7. Registering the lisp function `mm-exchange-rate` as an implementation of the `MM_Bank_exchange_rate_provider` PSM.

A similar API is provided for Java. Figure 8 below shows how a Java method implementing the `MM_Bank_exchange_rate_provider` PSM could be published through the IRS publisher.

```
IRSPublisher irsPublisher =
   new IRSPublisher
         ("http://137.108.24.248:3000/soap");        //IRS server URL

irsPublisher.PublishPSM(
    "MM_Bank_exchange_rate_provider",                //PSM Name
    "MM_Bank_exchange_rate_provider_ontology",       //PSM Ontology
    "MM_Bank",                                       //Class name
    "exchangeRate");                                 //method name
```

Fig. 8. The `exchangeRate` method of the Java class `MM_bank` published as an implementation of the `MM_Bank_exchange_rate_provider` PSM through the IRS Publisher.

3.5 IRS Client

A key feature of IRS-II is that web service invocation is *capability driven*. The IRS Client supports this by providing an interface and a set of APIs which are task centric. An IRS-II user simply asks for a task to be achieved and the IRS-II broker locates an appropriate PSM and then invokes the corresponding web service - see section 4 for

an example. The same functionality can also be invoked programmatically, through appropriate APIs associated with the current client platforms, currently Lisp and Java.

4 The Patient Shipping Healthcare Scenario

To illustrate how the IRS can be used to develop applications in terms of a number of co-operating, distributed semantic web services, we will describe a scenario taken from the health-care domain. This scenario covers a UK health care policy which was introduced in 2002. The policy was to reduce waiting lists for the treatment of some non-urgent medical problems by giving patients who were expected to wait more than 6 months for an operation the option to be treated in mainland Europe. Figure 9 graphically illustrates how we have implemented the scenario, which we dub "patient shipping", within the IRS-II. To limit the scope of the application we focused on the medical condition of arthritis which can sometimes require surgery.

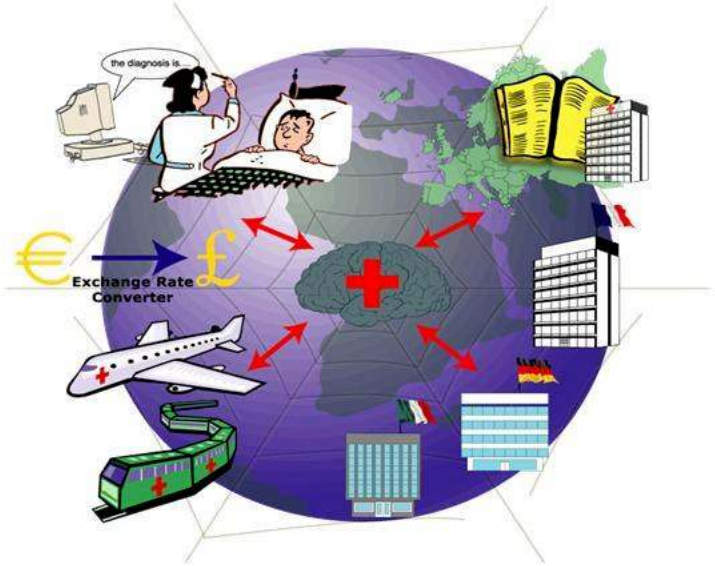

Fig. 9. A graphical overview of the patient shipping scenario

As can be seen in figure 9 five main types of web services are supported. Starting from the top left of the figure and proceeding clockwise these are:

- A diagnostic and recommender service able to diagnose a condition, for example a type of arthritis, from a set of symptoms, and to recommend a therapy such as a particular kind of surgery.
- A yellow pages service able to indicate which hospitals around Europe perform specific medical services.

- Services associated with individual hospitals able to answer queries about the availability and cost of the specific medical treatments they offer.
- Ambulance services able to provide prices for shipping patients from one hospital to another across international boundaries.
- An exchange rate service for converting prices into local currencies.

Task and PSM descriptions were created for the above services within the IRS server, using our knowledge modelling tool, WebOnto [4]. The services were then implemented in a mixture of Java Web Services (exchange rate, ambulance services and a number of the hospitals) and Lisp (all the remaining), and published using the IRS Publisher. Finally, a patient shipping web service which integrates the above services was implemented and published.

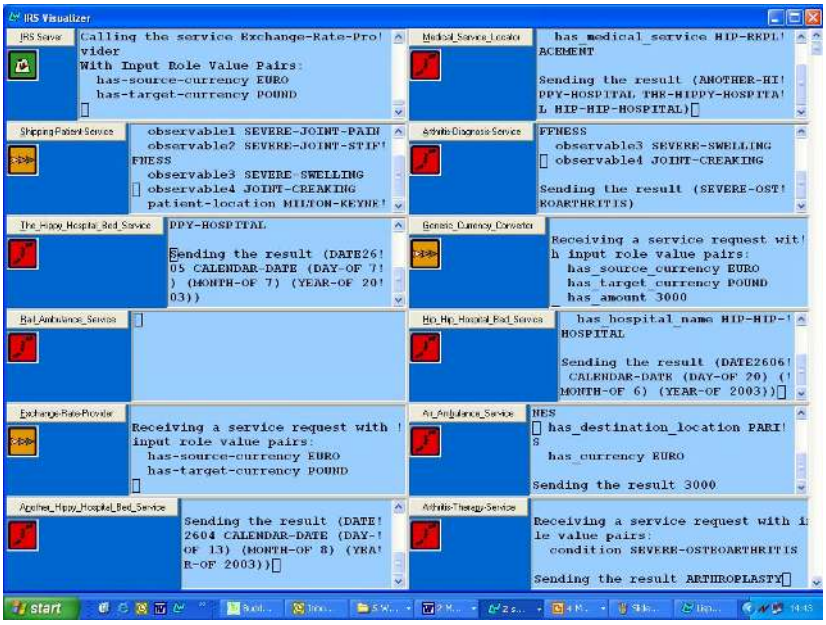

Fig. 10. A visualization of the patient shipping web service in mid execution.

The patient shipping task has five input roles. The first four are the symptoms which the patient displays and the fifth is the location of the patient.

Figure 10 shows a visualization of the distributed application during the execution. The visualization is composed of two columns showing the IRS server and eleven published services. Each published web service is displayed in a panel containing a) the name of the PSM, b) an iconic representation of the status of the web service, and c) a log of the messages the web service sends and receives. The meanings of the icons are:

 - the web service is currently idle.

 - the web service is currently processing.

 - the web service is sending a message.

We can see in figure 10 that a number of services have been called with the following results:

- The patient has been diagnosed with *severe osteoarthritis* by the `Arthritis-Diagnosis-Service`.
- The `Arthritis-Therapy-Service` recommends that the patient is treated by means of Arthroplasty, a synonym for hip-replacement.
- The `Medical_Service_Locator` service has found three hospitals which offer hip-replacement as a medical service, specifically `Another-Hippy-Hospital`, `The-Hippy-Hospital`, and the `Hip-Hip-Hospital`.
- The `Hip-Hip-Hospital` can treat the patient first (on the 20th of June, 2003).
- The `Air_Ambulance_Service` can move the patient from Milton Keynes to Paris, the location of the `Hip-Hip-Hospital`, for 3000 Euros.

We can also see from figure 10 that three web services are currently running: the `Shipping-Patient-Service`; the `Generic_Currency_Converter` and the `Exchange-Rate-Provider`. The IRS server has just sent a message to the `Exchange-Rate-Provider` requesting an exchange rate between the Euro source currency and the Pound target currency. Three more steps will occur before the application terminates. First, the `Exchange-Rate-Provider` will send an exchange rate to the `Generic_Currency_Converter`. Second, the `Generic_Currency_Converter` will convert the 3000 Euros to 1920 pounds. Finally, the `Shipping-Patient-Service` will send the result back to the client interface (shown in figure 10).

This application illustrates some of the advantages of semantic web services in general and our approach in particular. Service discovery is carried out using semantic descriptions. For instance, once a need for hip replacement has been ascertained, the appropriate hospitals are identified, which can provide hip replacement, using a directory of hospitals and interrogating each hospital agent in turn. Thanks to the availability of semantic descriptions, it is not necessary to invoke hospital web services directly. Instead, a semantic query for hospitals providing hip replacement services is sent to the IRS and the IRS broker is then able to match this query against the semantic descriptions of the various hospital service providers. The other important aspect is the use of capability-driven service invocation. For instance, once a hospital has been identified, which can treat the patient in Paris, the application client simply sends an "achieve-task" message to the IRS server, asking the latter to find the cheapest provider of ambulance services between Milton Keynes and Paris.

5 Related Work

The framework used by the IRS-II has much in common with the *Web Service Modelling Framework (WSMF)* [5], as both the IRS-II and WSMF build on research in knowledge modelling and in particular on the UPML framework. As a result both approaches emphasize the importance of separating goal and service descriptions to ensure flexibility and scalability. The main difference between IRS-II and WSMF is that while the latter is exclusively a framework, the IRS-II is also an implemented infrastructure, providing publishing support, client APIs, brokering and registry mechanisms.

The IRS-II also differs from the DAML-S work in a number of ways, as already discussed in section 2. In particular, DAML-S does not include flexible tasks-to-methods mappings and relies instead on hierarchies of services, thus limiting the possibilities for flexible, n:m mediation between problems and services. Indeed no service-independent notion of problem type is present in DAML-S. Another difference is that IRS-II represents descriptions in OCML, while DAML-S uses DAML+OIL. This is likely to be a temporary difference, given that both approaches plan to move to OWL-based representations in the near future.

Regarding W3C Web Services standards, there are differences in the approach we take towards application development and in the roles of architecture components. For example, unlike UDDI registries, when a service description is published to IRS, the code for service invocation is automatically generated and later used during task achievement.

6 Conclusions

In this paper we have described IRS-II, a framework and an infrastructure which supports the publication, discovery, composition and use of semantic web services. IRS-II provides one-click publishing support for different software platforms, to facilitate publishing and semantic annotation of web services. Like WSMF, IRS-II capitalizes on knowledge modelling research and is based on a flexible framework separating service from problem specifications.

Future work on IRS-II will improve error handling, which at the moment is very basic. We also want to facilitate automatic mediation, in order to exploit the separation of tasks and methods more fully. Another important goal is to move away from a built-in matchmaking facility and generalize this to a matchmaking infrastructure, essentially providing hooks for different matchmaking approaches to be integrated. Finally, we plan to OWL-ify the infrastructure, to ensure its compliance with emerging semantic web standards.

Acknowledgements

This work has been partially supported by the Advanced Knowledge Technologies (AKT) Interdisciplinary Research Collaboration (IRC), which is sponsored by the UK Engineering and Physical Sciences Research Council under grant number GR/N15764/01. The AKT IRC comprises the Universities of Aberdeen, Edinburgh, Sheffield, Southampton and the Open University.

References

1. Benjamins, V. R., Plaza, E., Motta, E., Fensel, D., Studer, R., Wielinga, B., Schreiber, G., Zdrahal, Z., and Decker, S. An intelligent brokering service for knowledge-component reuse on the World-Wide Web. In Gaines, B. and Musen, M. (Editors), 11th Workshop on Knowledge Acquisition, Modeling and Management, Banff, Canada, 1998. http://ksi.cpsc.ucalgary.ca/KAW/KAW98/benjamins3/
2. Clancey W. J. (1985). Heuristic Classification. Artificial Intell*igence, 27*, pp. 289-350.
3. DAML-S 0.7 Draft Release (2002). DAML Services Coalition.. Available online at http://www.daml.org/services/daml-s/0.7/.
4. Domingue, J. (1998) Tadzebao and WebOnto: Discussing, Browsing, and Editing Ontologies on the Web. 11th Knowledge Acquisition for Knowledge-Based Systems Workshop, April 18th-23rd. Banff, Canada.
5. Fensel, D., Bussler, C. (2002). The Web Service Modeling Framework WSMF. Available at http://informatik.uibk.ac.at/users/c70385/wese/wsmf.bis2002.pdf
6. Fensel, D. and Motta, E. (2001). Structured Development of Problem Solving Methods. IEEE Transactions on Knowledge and Data Engineering, 13(6), pp. 913-932.
7. Gruber, T. R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, *5(2)*.
8. IDC (2003) IDC direction 2003 Conference, Boston, March 2003
9. Marcus, S. and McDermott, J. (1989). SALT: A Knowledge Acquisition Language for Propose and Revise Systems. *Journal of Artificial Intelligence, 39(1)*, pp. 1-37.
10. Motta E. (1999). Reusable Components for Knowledge Modelling. IOS Press, Amsterdam, The Netherlands.
11. Newell A. (1982). The knowledge level. *Artificial Intelligence, 18(1)*, pp. 87-127.
12. Semantic Web. W3C Activity. Available online at http://www.w3.org/2001/sw/
13. Simple Object Access Protocol (SOAP) (2000). W3C Note 08. Available online at http://www.w3.org/TR/SOAP/.
14. McIlraith, S., Son, T. C., and Zeng, H. Semantic Web Services. IEEE Intelligent Systems, Mar/Apr. 2001, pp.46-53.
15. Sycara, K., Lu, J., Klusch, M. and Widoff, S. Matchmaking among Heterogeneous Agents on the Internet. Proceedings of the 1999 AAAI Spring Symposium on Intelligent Agents in Cyberspace, Stanford University, USA, 22-24 March 1999.
16. UDDI Specification. Available online at http://www.uddi.org/specification.html
17. Web Services Description Language (WSDL) (2001). W3C Note 15. Available online at http://www.w3.org/TR/wsdl