Semantic Markup for Semantic Web Tools: A DAML-S Description of an RDF-Store

Debbie Richards[1] and Marta Sabou[2]

[1] Div. of I.C.S., Macquarie University, Sydney, Australia
richards@ics.mq.edu.au
[2] Dept. of AI, Vrije Universiteit, Amsterdam, The Netherlands
marta@cs.vu.nl

Abstract. Easy integration of available tools will be a key success factor for the future of the Semantic Web. We envision that semantic descriptions of the tools themselves will play an important part in addressing this issue. Motivated by this vision we used DAML-S (a key initiative to support automated management of Web services) to describe Sesame (an RDF(S) storage and query engine). This paper discusses the major problems that we encountered as well as suggested solutions. This work is relevant to other Semantic Web research groups who wish to annotate their tools or to use annotated tools. Also, we hope to offer helpful input to the DAML-S coalition in the further development of their standard.

1 Introduction

Interest in the Semantic Web (SW) and Web services is increasing. SW technology promises to support a second generation of the World Wide Web by turning it from a repository for human consumption to a source of information and services that can be utilized by software agents. The need for easy discovery and integration of web-accessible services has fuelled the rapid development of the Web Services field. While many standards have been offered to support inter-operability of Web services, SW technology promises to increase automation in service discovery and integration on the basis of semantic markup. A major initiative in this direction is DAML-S [7], a DAML+OIL ontology which allows describing Web services in a machine interpretable way. It is claimed that DAML-S descriptions will automate the discovery, invocation, composition, interoperation and monitoring of web services [7].

As a result, an increasing number of efforts have been reported which involve the usage of DAML-S, however many of them do not use the whole language. Several projects use only certain parts of the DAML-S ontology, eg. matchmaking research tends to focus on the Profile ontology ([6], [14], [11]) while composition only requires the Process ontology [17]. Often, these parts are heavily extended rather then reused: [4] enriches the Process ontology, [18] extends the Profile ontology with bio-informatics related properties, [13] extends the specification of conditions. Finally, some papers mention use of complete DAML-S as is but give no details about their experiences.

The variety of application domains where DAML-S is used is also increasing. The first described services were fictitious (ex. the examples offered on the DAML-S site), just

D. Fensel et al. (Eds.): ISWC 2003, LNCS 2870, pp. 274–289, 2003.

like the services used for demonstrating solutions for complex tasks such as composition or matchmaking ([17], [11], [12]). However, recently different communities have started to experiment with describing their own real-life services, such as the merging of agent technology with DAML-S ([9], [15]). Important developments are reported in the field of bioinformatics as well [18]. Also, wishing to describe mathematical services, [3] concludes that DAML-S offers a good base, however they define their own description language - Mathematical Service Description Language (MSDL). Last but not least, e-commerce services are described: a currency converter and a description of Amazon[1]. Another class of services consists of Semantic Web services, such as ontology storage facilities (Sesame [5]), validators[2], annotation tools[3]. Surprisingly however, even if many Semantic Web related tools are made available as web-services, to our knowledge there is no work for describing such services.

We address this lack by using the entire DAML-S[4] ontology and WSDL to describe a typical SW Tool, Sesame [5][5]. We found that Sesame exhibits a set of characteristics which were not previously discussed for any particular group of services and therefore made its modelling non-trivial. First, it has a modular architecture composed of independent components which can be used in arbitrary combinations to fulfill certain tasks. In contrast, Web services described to-date, even if modular, have a well-defined workflow model and their components are not usable stand-alone. Second, it uses complex data types requiring a shift from the current practice of perceiving inputs and outputs as atomic values. Third, there are several constraints between its Input/Output parameters.

Naturally, the question arises: *Is DAML-S expressive enough to model these characteristics?* This paper attempts to answer this question since it is likely that such characteristics exist for other (SW) tools as well. Therefore the problems that we encountered are likely to be typical of the problems other people will encounter with publishing their tools as a web-service. Thus we provide our solutions and recommendations in terms of usage guidelines and language extensions to DAML-S.

We start with two introductory sections about the languages used (2) and about Sesame (3). Our findings relate to three main topics and are grouped in the next three sections (4, 5, 6). We conclude and suggest topics for future work in 7.

2 Languages for (Semantic) Service Description

2.1 DAML-S

The DAML-S ontology is conceptually divided into three sub-ontologies for specifying *what a service does*, *how the service works* and *how the service is implemented*. Accordingly, each DAML-S description has three major parts.

The *Profile*[6] describes what the service does so that it can be discovered at matchmaking time. It contains the contact information of providers, an extensible set of features

[1] http://www.daml.org/services/.
[2] DAML Validator at http://www.daml.org/validator/.
[3] Annotea – http://annotest.w3.org/annotations.
[4] This paper was written for DAML-S v.0.7. and updated for v0.9.
[5] Latest version available at http://www.cs.vu.nl/~marta/services/sesame/owl/.
[6] http://www.daml.org/services/daml-s/0.7/Profile.daml

that specify characteristics of the service and a functionality description by specifying the inputs, outputs, preconditions and effects (IOPE's) of the service.

The *Process*[7] presents the internal working of the service in terms of the internal processes, their process model and the internal data-flow. This information can be used during matchmaking but is aimed to assist execution monitoring. The Process describes the IOPE's of the service from a different perspective, but naturally links between the Profile and the Process of the model exist. Taken together they represent the conceptual level description of the service.

The *Grounding*[8] specifies the operational level details of the service by linking the conceptual level descriptions to the WSDL (Web Services Description Language) description of the service. The WSDL file contains the implementation details of the service such as message exchange formats and network protocols so that automatic invocation of the service is possible.

This layered description is used in the following way. First the Profile is inspected to see if the service has the desired capability. Further, the Process Model can give more detailed information about how the service works internally. This description can also be inspected during matchmaking. Finally, if the service conforms to the requirements, the Grounding specifies the implementation details needed for executing the service.

2.2 WSDL

WSDL is an industry standard for describing web-accessible services. WSDL has considerable support from industry and increasing tool-support (WSDL generators, editors). As an XML-based language, WSDL is machine processable: it is a structured (and standardized) way to describe web-interfaces. However the intended meaning (semantics) of the interface elements is interpretable only by human readers.

In WSDL a service is seen as a collection of network endpoints which operate on messages. Each WSDL document specifies an abstract description of the main components of the service and then binds this interface to concrete implementation details.

At the abstract description level types, messages, operations and portTypes are defined. First, one can define complex data types using XML Schema. Then a set of messages are specified by describing the name and the type of their parts. These messages represent either an input parameter set or the output of some abstract operations. Operations are conceptually grouped together in a PortType.

Through the binding part of the description the elements of the abstract interface are bound to concrete network protocols and message formats (SOAP, HTTP) and the actual network address of the service is specified.

3 Sesame

This section briefly introduces Sesame (3.1) and states some issues that we want our modelling to reflect (3.2).

[7] http://www.daml.org/services/daml-s/0.7/Process.daml
[8] http://www.daml.org/services/daml-s/0.7/Grounding.daml

3.1 Description

Sesame is an RDF(S) repository and query engine [5]. As such it can provide valuable support for ontology-based applications. It can be used either as a part of an application or it can be accessed via a web-interface. Currently both HTTP and SOAP communication protocols are available. Sesame's HTTP interface[9] makes available six different functionalities at six different URLs. We shortly present these functionalities including in brackets the URL extension where they are published.[10]

A Sesame server contains a set of password-protected repositories. So called public repositories are accessible without any login information. Through the server's HTTP interface one can request a list of all repositories which are available for a specified login, including the public ones (*listRepositories*). As a storage facility Sesame offers functionality to upload data and to interact with it. The data upload functionality adds data with a specified URL or sent as a string (*addData*). The content of a repository can be deleted completely (*clearRepository*). Further, deletion at the statement level is also supported (*removeStatements*). The user can retrieve the whole content of the repository (*extractRDF*) or alternatively only the schema information, only the instance information or both. A more refined data extraction method, the query method (*performRQLQuery* and *performRDQLQuery*), transforms Sesame into a query engine. There is no predefined way in which these functionalities should be used. In fact, they are highly self-contained and can be used stand-alone rather than in combination with other functionalities.

3.2 Modelling Requirements

We expect that our service markup will be used by intelligent agents which will reason with the provided semantic data. Also, for operational level integration all needed technical information must be properly captured. This requirement is fulfilled by DAML-S which allows both conceptual and syntactic level specifications (through WSDL).

At the conceptual level the ***semantics of the offered functionality*** has to be specified. There are multiple ways to do this. First, one can specify a service by *describing its parameters*. This issue is treated throughout the paper but mostly in section 6. Second, the meaning of a service depends on its relation to other services. It is desirable to describe how these components relate to each other so that an intelligent service could determine usage patterns for fulfilling certain tasks. One dependency between services is that of *composition*, when a certain service is composed out of multiple other services. Since this is the case of Sesame we investigate this issue in section 4. Another way to describe interdependencies is to specify *algebraic properties*. Examples are:

- DeleteStatement is idempotent: doing it twice is the same as doing it once. (Same for addStatement).
- Querying for a statement after having added it gives the statement back as a result.
- Querying for a statement after having deleted it gives a null result.
- Removing a statement after having added it is a null operation.

[9] See detailed description at http://sesame.aidministrator.nl/publications/users/ch06s02.html.

[10] A base URL of the form "http://HOSTNAME/SESAMEDIR/servlets" proceeds each extension – capitalized words depend on the server installation.

These are the real semantics of the operations, i.e. these properties really specify what it means to delete a statement, or to add one. We do not treat them in this paper.

Another issue is ***linking between the semantic and syntactic descriptions*** of a service. It is desirable that a single conceptual description can be mapped to descriptions of different technical implementations of the same service. In Sesame's case, it should be possible to ground its semantic description to the syntactic descriptions of all its different interfaces (HTTP, SOAP, RMI). We only experimented with linking to a WSDL description of an HTTP interface, however we encountered some problems (5.2).

Still related to grounding, technical descriptions used in combination with semantic ones should undergo minor changes so that they are still usable by tools which do not require semantic data. More on this in section 6.2.

4 Specifying Service Semantics

4.1 Splitting up Sesame

In order to satisfy the various needs of service providers, DAML-S does not impose a particular modelling style leaving a gap between conceptual considerations and the actual modelling. This happens with the very notion of service. Conceptually, they define web services as "Web sites that [...] allow one to effect some action or change in the world" [7]. Further they differentiate between simple and complex services. A simple service invokes a single web-accessible program and there is no interaction with the user during its execution. Request-response services are regarded as simple services. A complex service is composed of multiple simple services.

In the light of these definitions Sesame and each of its functionalities are web-services, where the functionalities are simple services and Sesame is a complex service. Deciding on the actual modelling was not so trivial. We present two of a number of alternatives we tried, underlining their benefits and disadvantages.

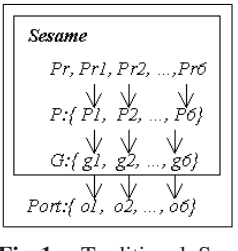

Fig. 1. Traditional Service Modelling

We first adopted the modelling style used in the existing DAML-S examples (see schematic representation in fig. 1). Accordingly, we have created a single *Service* instance, *Sesame*. The component functionalities were modelled as *AtomicProcesses (P1, ...P6)* within the *ProcessModel (P)*. Because, there is no fixed workflow between the components of Sesame, i.e. the six functionalities can be used in arbitrary ways to solve tasks, the workflow model that was closest to our needs was *process:choice*. This expresses very little about the relationships between the components and thus its semantic value is very weak. We tried to compensate this at the *Profile* (*Pr*) level where we have associated multiple *Profile* instances with the service, one for each functionality *(Pr1, ...Pr6)* and one for the global functionality of Sesame (*P* as an *OntologyStore*). Finally, the *Grounding (G)* of Sesame contains six *AtomicProcessGrounding* entities *(g1, ..g6)*, one for each internal process and its corresponding WSDL operation *(o1, ...o6)*.

This approach has the following limitations. First, several of the simple services offered alternative interfaces and implementations. Because only the *Service* concept

can have different groundings (and not atomic processes), a different implementation of an atomic service results in a different grounding for the whole service. Therefore we face a combinatorial explosion of groundings for the general service whenever a component offers a new implementation. Second, at the *Profile* level, there is no indication of the relationship between the seven services. Third, even if all services are visible at matchmaking time (since their *Profile* is explicitly mentioned), one needs to interpret the full description of the complex service when using just a simple service.

Note that this style of description works fine for web services composed of simple services which (1) do not have meaning outside the service and (2) are used according to a fixed workflow. In such cases a single Profile suffices to describe the task their execution can achieve. There is no need to advertise all of them since they cannot be used stand-alone anyway. However, none of these assumptions apply to Sesame: its functionalities can be used in any combination and also just alone.

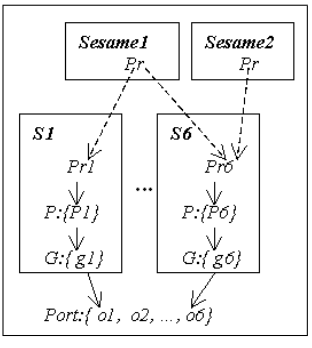

Fig. 2. Our Modelling

Due to these considerations we decided to model each functionality as a *Service (S1,...S6)*, as represented in fig. 2. Two of the limitations of the previous model are solved: (1) the overhead of declaring a new implementation of a simple service is reduced to declaring a single atomic grounding and (2) when using a service only its own description needs to be interpreted. However, the issue of expressing how these services inter-relate is still open. Optimally we would like to express that sets of these services can be used upon a certain Sesame server. Depending on the exposed services, different Sesame servers can have different type: some can offer only storage functionality (*OntologyStore - Sesame2*), some offer querying facilities (*QueryEngine*), some offer both (*OntologyStore and QueryEngine - Sesame1*). We want to express composition at the *Profile* rather than the *Process* level. We implemented this by building a domain ontology in which we enrich some of the DAML-S concepts as discussed in the next section.

4.2 The Role and Use of Domain Knowledge

Externally defined knowledge plays a major role in each DAML-S description. DAML-S offers a generic framework to describe a service, but to make it truly useful, domain knowledge is required. As DAML-S supports the inclusion of external knowledge in the service descriptions we are not addressing a problem in this section. Instead, since ontology acquisition is itself a difficult and time-consuming task we are offering the ontology we developed to support the description of Sesame and our reasoning leading to its development. Also, we show how this domain knowledge allowed us to model Sesame in the desired way.

The Domain Ontology. Matchmaking and integration of a set of web-services becomes easy if they are defined using the same rich domain ontology. A widely used domain ontology (1) should reflect terms accepted by a large community while (2) being easy to

extend with new knowledge and (3) straightforward to integrate in the service descriptions. We tried to achieve all this in our domain ontology[11].

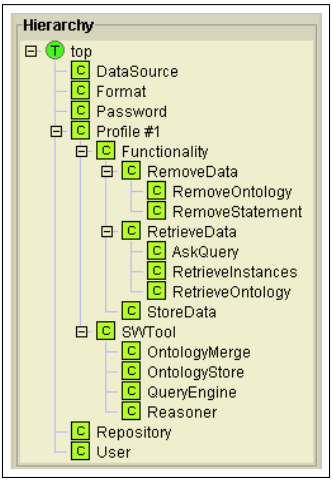

Fig. 3. The Domain Ontology[12]

Previously (section 4.1) we have motivated the need to express composition of services not at the *Process* but rather at the *Profile* level. To achieve this, in our domain ontology we make a conceptual difference between tools and their functionalities and define tool-profiles in terms of the functionality-profiles they provide. In terms of modelling this involved two inter-related constructions.

First, we specialized the DAML-S *profile:Profile* into *SWTool* and *Functionality*. The subclasses of the first concept reflect the main tool categories described by a thorough survey of existing Semantic Web Tools[10]. The second concept is the superclass for all functionalities. For now we have declared the functionality types relevant for Sesame and grouped them under more generic concepts. For example, we consider *RetrieveOntology* and *AskQuery* as methods for data retrieval. Their superclass, *RetrieveData*, can be extended with other methods specific to other tools which fulfill the same function.

Second, we have declared the new property, (*hasFunctionality*), which allows us to specify the kind of functionalities that a certain tool offers (depicted as part of the DomainOntology in fig.4). This construction allows us to define SWTool-s by imposing constraints on the type of functionalities they have. For example, an *OntologyStore* might allow a functionality of type *StoreData* and *must* offer a *RetrieveOntology* functionality. Also, a *QueryEngine* must provide an *AskQuery* functionality. Further, the ontology contains a set of terms needed to describe Sesame, such as *Repository, User, Password* etc. See the hierarchy of the main concepts in fig. 3.

We fulfilled the requirements for a widely used domain ontology. First, we used concepts from the largest survey about Semantic Web tools within the SW community and extended the concepts of DAML-S, which is growing into a standard. Second, our model is easy to extend: new functionality types can be added and new tool types can be defined using existing or newly added functionality. The third issue, that of easy use for service description is demonstrated in the next section.

Using Domain Knowledge in DAML-S Descriptions. Domain concepts are used for several reasons in a DAML-S description. First, they express the meaning of the offered functionality at the *Profile* level. This section is devoted to this first issue. Second, domain knowledge can be used to describe the IOPE's both at the *Process* and the *Profile* level, as will shortly be demonstrated in section 5.1. Finally, during the grounding process, WSDL descriptions are enriched with domain knowledge (as shown in section 6.2).

[11] http://www.cs.vu.nl/~marta/services/swTools.owl
[12] Image created with OilEd3.5, http://oiled.man.ac.uk/

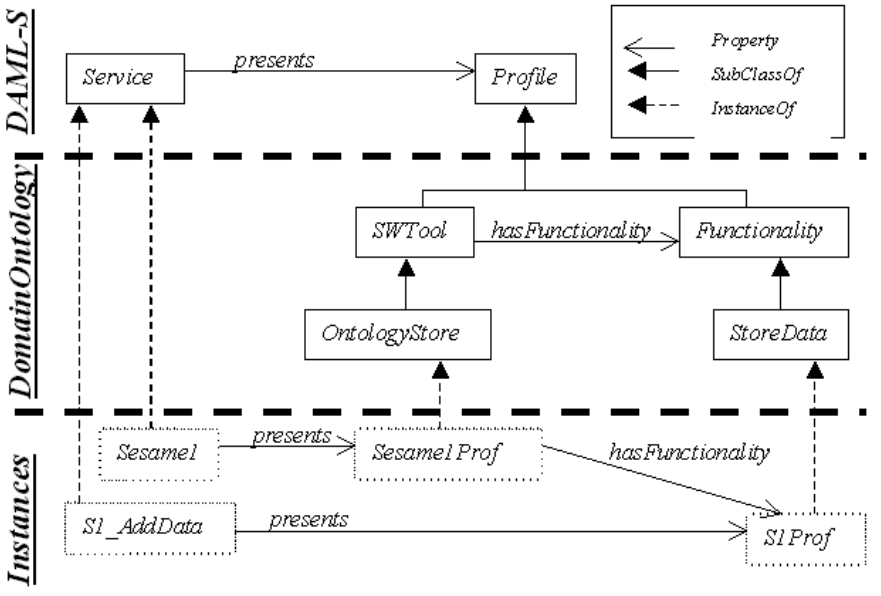

Fig. 4. Use of Domain Knowledge in Profiles

One indicates the overall functionality of a service by declaring its *Profile* as being of a type documented in the Domain Ontology. Figure 4 demonstrates this process. *Sesame1* is a *Service* instance and its *Profile*, *Sesame1Prof* is of the type *OntologyStore* which was declared in the domain ontology. Similarly, each individual functionality is declared as a service. For example, for the AddData functionality, we declared a *Service* instance, *S1AddData*, and associated with it a *Profile* instance of type *StoreData*.

By declaring Sesame's *Profile* of type *OntologyStore* we inherit the *hasFunctionality* property defined for any *SWTool*. At the instance level, we used this property to associate *Sesame1Prof* with *S1Prof*. This indicates that the functionality offered by the *Sesame1* service relies on the functionality of the *S1AddData* service.

It is very easy to define different Sesame instances with different combinations of these functionalities. This is useful in practical scenarios when different user groups can access different functionality. For example, a company which stores its data in Sesame may wish to make it available both for its customers and employees. Customers will only be allowed to read the stored data. Therefore, that particular service instance of Sesame will only advertise a data reading capability. On the contrary, employees can both add and read data, therefore their service instance will point to both functionalities.

We managed to express composition (interdependency) at the *Profile* rather than just at the *Process* level, as originally possible with DAML-S. At matchmaking time, if someone wants to use Sesame he gets a list of all needed service profiles and can discover the individual services associated to those profiles.

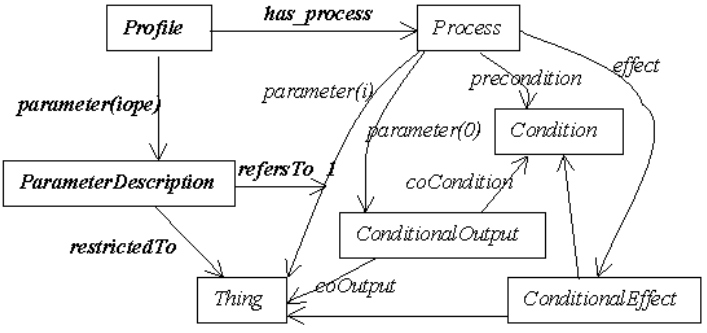

Fig. 5. Profile to Process bridge

5 Linking between Parts of the Description

The *ServiceProfile, ServiceModel* and *ServiceGrounding* are three different but linked descriptions of the same service. Identifying these links and specifying them correctly ensures the consistency of the description.

5.1 Profile to Process Bridge

A *Profile* contains several links to the *Process*. Figure 5 shows these links, where terms in bold-face belong to the *Profile* ontology and the rest to the *Process* ontology. Firstly, a *Profile* states the *Process* it describes through the unique property `has_process`. Secondly, the IOPE's of the *Profile* correspond to the IOPE's of the *Process*. Understanding this correspondence is not so trivial given the fact that the IOPE's play different roles for the *Profile* and for the *Process*. In the *Profile* ontology they are treated equally as parameters of the *Profile* (they are subproperties of the *profile:parameter* property - suggested with IOPE's between brackets in the figure). In the *Process* ontology only IO's are regarded as subproperties of the *process:parameter* property. The PE's are just simple properties of the *Process*. While technically the IOPE's are properties both for *Profile* and *Process*, the fact that they are treated differently at a conceptual level is misleading. In the DAML-S *Profile* ontology each *ParameterDescription refersTo* exactly one element of type *process:paramater*.

Several aspects of this mapping can be critiqued. First, one would expect that *Profile* parameters of a certain type can only refer to *Process* parameters of the same type. However, this is not enforced. With the current specification one can easily make a link between parameters of different types (eg. *profile:input* and *process:output*). Second, because the *Process* ontology does not model PE's as subproperties of *process:paramater*, it is inconsistent to use entities of this type as values for *profile:refersTo*. Therefore PE's defined for the *Profile* cannot refer to the corresponding PE's of the *Process*.

The DAML-S coalition acknowledges the possibility of inconsistencies between *Profile* and *Process* and that they will be discovered at some point [7]. Since matchmaking is based on the *Profile* description, the break may occur during (attempted) usage of the

Table 1. Sesame addData Service - the HTTP Interface

Parameter	Required	Description
repository (rep)	yes	Specifies the id of the repository to which data is added
data	for method1	The data that should be added to the repository
dataURL (URL)	for method2	The URL of the data that should be added to the repository
baseURL (base)	for method 1	Specifies the base URL for the data. Needed for method1 as Sesame does not know where the data came from. For method2 the baseURL defaults to dataURL
user	no	The user's login name. Needed to gain access to non-public repositories.
password (pass)	no	The user's password is only required in the case the user parameter is supplied
format	no	Determines the response format. Legal values are xml (the default) and html

service. The rationale for this design decision is not clear. We conclude that this link between the IOPE's at two distinct levels of specification should be corrected and made more explicit.

As an addition to the previous section on the use of domain knowledge, each parameter of the *Profile* has a *ParameterDescription* which mandates describing the parameter through the *profile:restrictedTo* property. This property has an unspecified range, therefore it can refer both to a DAML+OIL concept or to a data type. We recommend specifying a domain level concept: this gives the "semantics" of the parameter. For the *Process*, we have used domain concepts to specify the range of inputs and outputs in order to indicate their semantics.

5.2 DAML-S/WSDL Mapping

In this section we present our modelling experiences when writing the *Grounding* for the *addData* service and motivate the issues discussed in section 6.

The *ServiceGrounding* builds a bridge between the *ProcessModel* and the WSDL description of the service according to three rules[7]. First, each DAML-S *AtomicProcess* corresponds to a WSDL operation. Second, following from the previous rule, each input of the DAML-S *AtomicProcess* is mapped to a corresponding message-part in the input message of the WSDL operation. Similarly for DAML-S outputs, each output of the DAML-S *AtomicProcess* is mapped to a corresponding message-part in the output message of the WSDL operation. Third, the type of each WSDL message part can be specified in terms of a DAML-S parameter (an XML Schema data type or a DAML+OIL concept).

The Sesame *addData* service allows RDF(S) data to be uploaded to Sesame through an HTTP interface described in Table 1. This service can acquire the RDF(S) source either (1) by uploading the data itself (*method1*) or (2) by uploading a URL referring to the data (*method2*). The parameters specified will determine which method to use. Intuitively, this situation is similar to a specific kind of ad-hoc polymorphism, that of *overloading*: a certain method allows different signatures, but essentially it executes the same function.

The available documentation about DAML-S provided little guidance on how to model this situation. Some coalition members have considered the problem of supporting multiple interfaces and offered a solution for the situation where the number of arguments are the same and in the same order but where the data types may differ [2]. The solution

offered is to define a higher level concept that covers all possible alternative data types. This solution guided us in our own modelling.

The effect of the service is not altered by the input parameters, that is, in all but error situations (such as an invalid password or a URL site that cannot be found). Therefore we consider that, conceptually, we have a single service which provides a means of adding some RDF(S) data to a Sesame repository, even if two slightly different implementations exist for the same service. In modelling terms we define a single service with two groundings, one for each implementation.

In modelling the conceptual part we took into consideration the solution presented in [2] by defining a *DataSource* concept to generally define the source of the RDF(S) data. This concept covers the two different data sources: (1) the data supplied as an URL (dataURL) and (2) the combination of a data stream (data) and its associated URL (baseURL). Below we show the inputs, expressed in terms of domain ontology concepts, expected by the *AtomicProcess P1* of the ProcessModel:

```
*ProcessModel:
   AtomicProcess:P1(I(Repository), I(DataSource),I(User),
                    I(Password), I(Format))
```

Our WSDL representation of the service consists of a *PortType (addData)* with two *operations (method1, method2)* which differ through their input messages: the first operation expects the actual data *(data)* and a base URL *(base)*, while the second receives the URL of the actual data *(URL)*.

```
*Service(
   PortType:addData(
       method1 (IMsg(rep, data, base, user, pass, format), OMsg(stream))
       method2 (IMsg(rep, URL, user, pass, format), OMsg(stream)))
```

Specifying the grounding to *method2* was easy. For grounding *method1* we need to express a one-to-many mapping between the *DataSourse* input of the *AtomicProcess P1* and two input message parts of this method: *data* and *base*. However, this is not possible since the DAML-S/WSDL mapping imposes that "each atomic process input and output corresponds to a (single) WSDL message part" [7]. While, the DAML-S coalition acknowledge that this could be a possible limitation, they do not give a concrete example of a problematic scenario. Our example shows that the assumption prohibits modelling of ad-hoc polymorphism (overloading). Therefore we encourage its revision.

The new version of DAML-S (v.0.9)[13] is less restrictive, but nevertheless does not solve this issue. There is a possibility of defining XSLT transformations between the *Process* parameters and the WSDL message parts. In case of inputs multiple input parameters can be mapped to a single WSDL message part. Still, we cannot solve our particular problem where we need to map one input parameter *(DataSource)* to multiple message parts (*data* and *baseURL*).

As can be seen in table 1, the *addData* service has several optional input parameters, three of which become mandatory depending on certain conditions. We will take up the issue of defining constraints on inputs in section 6.1.

[13] Version 0.9 was released after we submitted our paper thus our findings refer to v.0.7.

6 IO Specification

We have previously been concerned about the service as a whole and the links between the layers of its specification. We continue with details about how the parameters of the service can be specified. The first part of this section proposes a more flexible modelling of input parameters so that inputs depending on a condition can be specified as well. In the second we propose an alternative to complex data type specification.

6.1 Conditional Inputs

DAML-S does not cover cases when inputs depend on a certain condition. Currently one can only indicate whether an input is mandatory or optional by relying on the DAML+OIL cardinality restriction mechanism . For each mandatory input a cardinality restriction must enforce its existence. However, we have encountered situations when an input is only required in combination with another input. For example, when specifying the log-in information, a *password* is only mandatory when the *user* parameter is supplied. Further, in case of the *addData* service if the actual data is supplied, rather than a *URL*, then a *baseURL* must be specified.

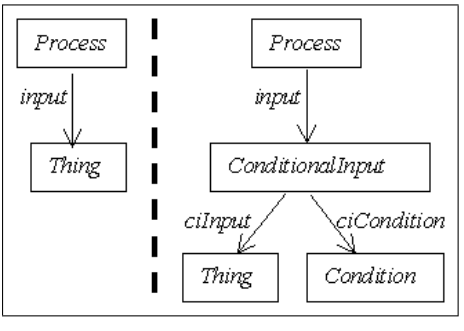

Fig. 6. Current and proposed modelling of inputs

A solution that appears to be the most consistent with the current version of DAML-S is to extend the *Process* ontology to support conditional inputs in the same way that outputs and effects may be conditional (see fig 6). Following the general template of defining conditional elements, we propose defining the *ConditionalInput* class which simply bundles a *Condition* and an input, using two properties: the condition of the conditional input (*ciCondition*) and the input of the conditional input (*ciInput*). A conditional input is an input that only occurs when a condition is true.

This modelling allows specifying a wider range of inputs than currently possible. First, inputs depending on factors external to the process can be described. For example, customers from outside the USA do not need to provide the zipcode input. In this case the customers geographic location conditions whether the zipcode input is needed or not. Second, we can capture situations exemplified before where an input is conditioned by the existence of another input, or generally a condition that involves other inputs. Third, we propose an alternative way to specify mandatory inputs by conditional inputs with a condition that is always “True". Finally, optional inputs can be modelled as UnconditionalInputs, i.e. inputs that do not depend on any condition. This list is not complete, however we think these cases are likely to appear in other tools (and services) as well. Note however that we did not deal with how conditions should be specified. In fact this is a thorny issue for the whole community.

6.2 Complex Data Types: XML Schema or DAML+OIL?

To support automatic discovery and operational integration of web services, their markup should provide information about (1) the semantics of their methods and parameters as well as (2) their syntactic signature (ex. method names, data format of parameters). The issue of how one can best specify complex data types (both at a syntactic and semantic level) has a major importance for Sesame where four methods provide complex output data types. As a concrete example, consider the simplest functionality of Sesame: *requestRepository*. It's output conforms to the DTD specified below.

```
<!ELEMENT repositorylist (repository)*>
<!ELEMENT repository(title)>
<!ELEMENT title          (#PCDATA)>
<!ATTLIST repository
          id       ID           #REQUIRED
          readable (true|false) #REQUIRED
          writable (true|false) #REQUIRED>
```

Such complex data formats are defined in the "types" section of a WSDL document. For maximum platform independence XML Schema(XSD) is used to specify the format of complex data types. The syntactic information provided by XSD allows other tools to easily parse any output that conforms to this DTD, therefore allowing an easy integration between web services. However this information has no semantics.

The DAML-S solution to this is to replace the syntactic XSD definitions with semantic definitions written in DAML+OIL, since WSDL allows using any XML-based type definition language in its "types" section. The claim is that this would use DAML+OIL's rich data typing feature [1].

We felt that this recommendation had limitations when applied to the complex types in Sesame. First, we found it difficult to express the complex syntax of the data type solely with DAML+OIL elements. We even considered using an OWL-based type definition as it provides more advanced ways for data typing than its ancestor. Even so we did not achieve our goal. This is understandable since DAML+OIL (and OWL) is an ontology language and as such it delegates the data type representation to XSD. Users are encouraged to refer to XSD complex types in their models rather then defining them using DAML+OIL.

Second, a WSDL document using purely DAML+OIL based data types would be useless for those users of the service which do not understand DAML+OIL. Ideally, an existing WSDL description should undergo minor changes when enabled for use with semantic technology to ensure its backward compatibility with traditional applications. We expect that many tools would use the XSD definitions of Sesame's WSDL description and only very few would understand DAML+OIL. Therefore we want to extend XSD types rather than to replace them.

Third, for some applications, not all parts of a complex data type were interesting semantically. For example, the syntax of the previous output is very complex, but semantically we are not interested in all its parts. We want to specify that objects of type *Repository* are returned but we do not want to specify further details.

With these considerations in mind, we used an alternative for complex type definition. We used XML Schema to specify the syntax of the output, just like for a traditional

WSDL description. To add semantics to this type we have augmented its components with references to corresponding DAML+OIL concepts. The *xsd:annotation* tag has exactly this function. We wrote the following XSD definition and augmented it with concepts defined in the domain ontology.

```
<xsd:element name="repositorylist">
 <xsd:complexType>
  <xsd:element name="repository">
   <xsd:annotation>
     <xsd:documentation>do:Repository</xsd:documentation>
   </xsd:annotation>
   <xsd:complexType>
     <xsd:element name="title" type="xsd:string"/>
     <xsd:attribute name="id" type="xsd:string" use="required"/>
     <xsd:attribute name="readable" type="xsd:boolean" use="required"/>
     <xsd:attribute name="writeable" type="xsd:boolean" use="required"/>
   </xsd:complexType>
  </xsd:element>
 </xsd:complexType>
</xsd:element>
```

This method satisfies all of our requirements. Using XML Schema to define complex types is straightforward since the language was designed for this purpose. We can easily add semantics to any parts of the description in such a way that it remains usable for non-semantic based applications.

7 Conclusions and Future Work

The goal of this paper is to assess the expressivity of DAML-S for certain service characteristics which were not discussed previously in the literature and are likely to be exhibited by Semantic Web Tools. Our global conclusion is that DAML-S offers a useful set of terms for describing semantic web tools however, we needed to make some extensions in order to achieve more expressivity. Indeed, the given properties of Sesame triggered many of these extensions. We think that research on how certain facets of web-services require certain type of modelling can be a valuable future work.

One observation is that Sesame combines a set of self-contained services which can be used in any combination in contrast to other services with a well-defined workflow of processes which cannot be used outside the composite service. While traditional DAML-S modelling favors the latter case, we were able to provide a work-around by allowing a composition mechanism between profiles of tools and functionalities within our domain ontology in section 4. The proposed construct may be appropriate beyond Sesame and a useful extension for DAML-S.

Major issues in this area still require some future work. First, the mentioned revision to DAML-S can be considered. Second, the whole conceptual view on this problem could be made clearer by aligning DAML-S with an upper level ontology such as DOLCE [8]. Third, we envision that in the future more standard domain ontologies will emerge and

be adopted by large communities. This will make the task of defining ones own domain ontology much easier. Also, it would ensure inter-operability between the different descriptions and ensure effective matchmaking. Work on such ontologies (and in particular on an ontology of semantic web tools) is highly relevant future work. Fourth, we consider that the specification of algebraic properties between services would add much more semantics to the descriptions. However, work in this area is linked to developments in rule specification.

Specifying inputs and outputs is at least as important as specifying the semantics offered by the service. Sesame introduces the issue of expressing conditional inputs and complex data types which we treated in section 6. We proposed a novel modelling to allow more flexible specifications of different types of inputs and suggested an alternative specification of complex data types that would ensure minimal changes to existing WSDL documents. Specifying the content of such conditions is left to the user. While we can express some constraints with the built-in restriction mechanisms of DAML+OIL, defining generic rules remains as future work.

Finally, we discussed in section 5 some limitations/inconsistencies of DAML-S which are generally valid but hindered us in carrying out our particular descriptions. For a more detailed argumentation of these observations, see [16], where we report on the modelling of two simple web-services.

The automatic discovery, composition and execution of web-services by software agents potentially involves the use of web accessible SW tools which themselves must be semantically described. By offering our experiences with marking up Sesame we hope to assist others to markup their SW tools and to assist the DAML-S coalition in the development of their standard.

Acknowledgements. The authors would like to thank Annette ten Teije and Frank van Harmelen for their helpful comments on an earlier draft of this paper.

References

1. A. Ankolekar. DAML-S: Web Service Description for the Semantic Web. In I. Horrocks and J. A. Hendler, editors, *The Semantic Web – ISWC 2002, First International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, Sardinia, Italy, 2002.
2. A. Ankolenkar, F. Hutch, and K. Sycara. Concurrent Execution Semantics for DAML-S with Subtypes. In I. Horrocks and J. A. Hendler, editors, *The Semantic Web – ISWC 2002, First International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 348–363, Sardinia, Italy, 2002.
3. W. Barbera-Medina, J. Padget, and M. Aird. Brokerage for Mathematical Services in MONET. In *AAMAS Workshop on Web Services and Agent-Based Engineering*, Melbourne, Australia, July 2003.
4. J. Brison, D. Martin, S.I. McIlraith, and L. A. Stein. *Agent-Based Composite Services in DAML-S: The Behavior-Oriented Design of an Intelligent Semantic Web*. Springer-Verlag, Berlin, 2002.
5. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In I. Horrocks and J. A. Hendler, editors, *The Semantic Web – ISWC 2002, First International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 348–363, Sardinia, Italy, 2002.

6. J. Cardoso and A. Sheth. Semantic e-Workflow Composition. Technical report, LSDIS Lab, Computer Science, University of Georgia, July 2002.
7. DAML Services Coalition. DAML-S: Semantic Markup for Web Services. DAML-S v. 0.7 White Paper, October 2002.
8. A. Gangemi, N. Guarino, C. Masolo, A. Oltramari, and L. Schneider. Sweetening Ontologies with DOLCE. In *Proceedings of EKAW*, Siguenza, Spain, 2002.
9. N. Gibbins, S. Harris, and N. Shadbolt. Agent-based Semantic Web Services. In *The Twelfth International World Wide Web Conference*, Budapest, Hungary, 2003.
10. A. Gomez Perez. A survey on ontology tools. OntoWeb Delieverable 1.3, May 2002.
11. M. Laukkanen and H. Helin. Composing Workflows of Semantic Web Services. In *AAMAS Workshop on Web Services and Agent-Based Engineering*, Melbourne, Australia, July 2003.
12. L. Lei and I. Horrocks. A Software Framework For Matchmaking Based on Semantic Web Technology. In *The Twelfth International World Wide Web Conference*, Budapest, Hungary, 2003.
13. A. Lopes, S. Gaio, and L.M. Botelho. From DAML-S to Executable Code. In *Proc. of the Workshop Challenges in Open Agent Systems AAMAS 2002*, 2002.
14. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In I. Horrocks and J. A. Hendler, editors, *The Semantic Web – ISWC 2002, First International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, Sardinia, Italy, 2002.
15. D. Richards, S. van Splunter, F. Brazier, and M. Sabou. Composing Web Services using an Agent Factory. In *AAMAS Workshop on Web Services and Agent-Based Engineering*, Melbourne, Australia, July 2003.
16. M. Sabou, D. Richards, and S. van Splunter. An Experience Report on Using DAML-S. In *Workshop on E-Services and the Semantic Web*, Budapest, Hungary, May 2003.
17. M. Sheshagiri, M. desJardins, and T. Finin. A Planner for Composing Services Described in DAML-S. In *AAMAS Workshop on Web Services and Agent-Based Engineering*, Melbourne, Australia, July 2003.
18. C. Wroe, R. Stevens, C. Goble, A. Roberts, and M. Greenwood. A Suite of DAML+OIL Ontologies to Describe Bioinformatics Web Services and Data . *Journal of Cooperative Information Science*, 2003.