Semantic Markup for Semantic Web Tools: A DAML-S description for Sesame

Marta Sabou[1] and Debbie Richards[2]

Dept. of Artificial Intelligence, Vrije Universiteit, Amsterdam, The Netherlands, email:marta@cs.vu.nl and Div. of I.C.S., Macquarie University, Sydney, Australia, email:richards@ics.mq.edu.au

Abstract. Easy integration of Semantic Web tools is a key issue for the development of the field. 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. As such 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

[SW]Semantic Web (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 to its full potential by software agents.

[Tools in SW] To implement its vision the SW community is developing its own tools. A common trend is to merge these tools in Tool Suites: i.e. a set of tools with well-defined functionality which can be combined in an arbitrary way to solve a certain task. Examples are: KAON [**?**], Sesame [**?**], RDFSuite [], Jena []. Also, many of them are available online as web-services (Sesame at least).

[Integrating SW Tools] The ease of combining these tools is essential for the development of the SW. The WonderWeb project recognized this need and it's objective is to create a infrastructure that allows for easy discovery and integration of various SW tools, provided by *different* vendors. The KAON server is developed for this task. In its initial architecture, KAON identifies 3 modules that would benefit from semantic markup of ontology tools. They also offer ideas about an ontology to describe the tools. [I wonder if there are any other efforts like this].

[WS] The need for easy discovery and integration of web-accessible services 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 [ref??], a DAML+OIL ontology which allows describing web-services in a machine interpretable way. The envisioned scenario is that service providers will advertise themselves through a middle agent and service requestors will be able to query the middle agent for a service that meets their needs. It is claimed that DAML-S descriptions will automate the discovery, invocation, composition, interoperation and monitoring of web services [ref?].

[Types of described WS-not really about components] A number of efforts have been reported which involve the usage of DAML-S. We identified four types of usage scenarios. First, within the DAML-S coalition two complete, fictitious examples (on the DAML-S site) and two concrete applications[**?**] are provided. Second, several projects use only certain parts of the DAML-S ontology, eg. matchmaking research tends to focus on the Profile ontology [**?**,**?**]. Third, we found work which extends parts of DAML-S: [**?**] enriches the Process ontology, [**?**] extends the Profile ontology with bio-informatics related properties, [**?**] extends the specification of conditions. Finally, some papers mention use of complete DAML-S as is, but the purpose of the papers was to describe other research work. The services marked up tended to be (fictitious) e-commerce services. [Check up the papers on the DAML-S site/ also the description of an RDF tool in Matchmaker].

[SW Tools are different] SW Tools (Suites) differ from traditional Web services in some critical points. First, they have a modular architecture composed of independent components which can be used in arbitrary combinations to fulfill certain tasks. In contrast, Web services typically described to-date, even if modular, have a well-defined work-flow model and their components are not usable in arbitrary combinations. Second, SW Suites exchange complex data types requiring a shift from the current practice of perceiving inputs and outputs as atomic values. Third, there are several constraints between their IO parameters [??]. [identify all these features in Sesame] To allow automatic discovery and configuration of the service provided by SW tools the description language must support these distinguishing features. Further, we think that semantic descriptions can ease the integration problem between SW Tools. To this end we chose to describe a typical SW Tool, Sesame [], using DAML-S. Since Sesame itself is a Semantic Web tool, our findings should be of interest to the wider Semantic Web community, that is, beyond those groups specifically interested in Web services. The problems that we encountered are likely to be faced by other users of Sesame who may in the future want to dynamically and automatically work with Sesame. Thus we provide our solutions and recommendations.

This paper begins with an introduction to Sesame and the various technologies involved in its semantic description. In section 3 we consider the key issues that arose in modelling Sesame including: how best to split up the multiple functionalities provided by Sesame; the role of domain knowledge; the links between the various DAML-S models; mapping between DAML-S and WSDL; the handling of conditional inputs; and the definition of complex output types. In the final section we provide our conclusions and future work.

2 Services and technology

2.1 Sesame

Sesame is an RDF(S) repository and query engine [**?**]. As such it can provide valuable support for ontology-based applications. It can be used either as a part of the application or it can be addressed via a web-interface. Currently both HTTP and SOAP communication protocols are available.

Sesame's HTTP interface makes available six different functionalities at six different URLs[1]. We shortly present these functionalities including the URL extension where they are published [2]

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 specified at a 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.

We have marked up five of the six functionalities (except `removeStatements`) available via the HTTP interface. Given the fact that many Semantic Web applications use Sesame [any references?], we envision that augmenting the web interface of Sesame with a semantic markup is of high interest for future applications which will build on the idea of semantic web services. [do we need this last paragraph it repeats our claim in the introduction?]

2.2 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.

Fig. 1. The DAML-S Service Ontology.

The Profile[3] 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 that specify characteristics of the service and a functionality description by specifying the inputs, outputs, preconditions and effects of the service (IOPE's).

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

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

[3] http://www.daml.org/services/daml-s/0.7/Profile.daml

The Process[4] 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 it 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. Note that both parts of the description augment the described elements with domain level concepts pre-defined in an external ontology. Taken together they represent the conceptual level description of the service.

The Grounding[5] specifies the operational level details of the service by linking the conceptual level descriptions to the WSDL [ref?? and perhaps definition] 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.

2.3 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 Modelling Sesame

3.1 Splitting up Sesame

Sesame offers complex functionality comprised of six simple functionalities. [is "functional modules" a better term, but then a number of other sentences also need changing]

[4] http://www.daml.org/services/daml-s/0.7/Process.daml
[5] http://www.daml.org/services/daml-s/0.7/Grounding.daml

These functionalities can be used in arbitrary ways to solve tasks: there is no predefined workflow model for their use. From a service point of view, we consider that Sesame is a composed service which integrates six atomic services. Due to their self-contained nature, we see these functionalities as independent web-services. Indeed, it is possible to use them alone, without the 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. These interdependencies are called algebraic properties [Frank?]. For example it is important to state that the extractData service and the importData service act upon the same repository.

The DAML-S solution to modelling composed services is to transform each component service in a process of the composed service and then use the Process Model to describe the workflow between these processes. We tried this solution and discovered several limitations. First, in our case, there is no fixed workflow between the components of Sesame, therefore the workflow model that was closest to our needs was `process:choice`. This expresses very little about the relationships between the components and thus the semantic value is very weak. Second, several of the atomic services offered different 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.

Due to these concerns we decided to model each functionality as a service and we wished to declare Sesame as a collection of these services. Also, we aimed at describing the relationships between these atomic services. However, because Service is the highest level modelling primitive in DAML-S there are no built-in possibilities to express any kind of relationships within services.

Our solution was to build a rich domain ontology in which we enrich some of the DAML-S concepts. The next section discusses our domain ontology, concentrating on the major modelling choices we made. It also presents how the new model allowed us to model Sesame in the desired way.

3.2 Use of Domain Knowledge

Externally defined knowledge plays a major role in each DAML-S description. Basically DAML-S offers a generic framework to describe a service, but to make it truly useful, domain knowledge is required. DAML-S supports the inclusion of external knowledge in the service descriptions.

Using domain knowledge ensures a detailed description of the service which can be used for matchmaking. If multiple services are described using the same ontology, matchmaking and integration will be based on this ontology. The ontology will provide a common vocabulary to describe the set of services. To facilitate matchmaking and integration it is important that (1) concepts accepted by a large community are used and (2) the ontology offers generic structures that can be extended for describing novel tools. These principles guided us in defining our domain ontology.

The Domain Ontology The biggest challenge for building the domain ontology[6] was to solve the issue of expressing relations between services.

Fig. 2 The Domain Ontology.

The first aspect of this issue is expressing that a service is composed of other services. We propose to do this at the level of service profiles. We decided to make a conceptual difference between tool suites and their functionalities and to define suites in terms of the functionalities they provide. To implement this we did two things. First, we specialized the DAML-S profile:Profile into RDFTool and RDFToolFunctionality. The latter is the superclass for all functionalities. We have declared the functionality types relevant for Sesame and grouped them under more generic concepts. For example, we consider ExtractRDF and PoseQuery 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.

```
<daml:ObjectProperty rdf:ID="hasFunctionality">
    <rdfs:domain rdf:resource="#RDFTool"/>
    <rdfs:range rdf:resource="#RDFTool_Functionality"/>
</daml:ObjectProperty>
```

[the next three paragraphs do not flow or read well - ”the next paragraph” does not say how Sesame defines a complex service - do you mean section? - goes backwards and forwards and I don’t know where to look] This construction allowed us to define two RDFTools by imposing constraints on the type of functionalities they have. An RDFStorage must have at least one functionality of type StoreData and one of type RetrieveData while an RDFQueryEngine is any RDFTool providing a PoseQuery functionality. The next paragraph will demonstrate how we used this to define Sesame as a complex service.

Expressing the dependencies between these different Profiles is still an open issue. However, we think that it could be done similarly to how we solved defining more complex services.

Further, the ontology contains a set of terms needed to describe Sesame, such as Repository, User, Password etc.

Note that in our modelling we followed the general principles stated above. First, we described domain knowledge to complement the DAML-S description. Second, we did this in such a way that the resulting model can be easily extended with concepts needed by other RDFTools. Therefore this ontology can be the basis for defining knowledge about a suite of RDF tools in an application. Third, we extended concepts from standard ontologies (the DAML-S `profile:Profile`), ensuring interpretability of our knowledge.

Using Domain Knowledge in DAML-S descriptions Domain concepts can be used in the three major parts of a DAML-S description.

[6] http://www.cs.vu.nl/˜marta/services/sesame/DomainOntology.daml
[6] Image created with OilEd3.5, http://oiled.man.ac.uk/

In the Profile definition one can use domain knowledge in two situations. First, the Profile of the service can be of a type defined in an external ontology. We model Sesame as an abstract service, i.e. a service which only has a Profile. Sesame has an RDFStorage Profile, and therefore, we can use the `hasFunctionality` property defined in the domain ontology to specify the composing services. The markup below declares a Sesame instance that offers three functionalities.

```
<service:Service rdf:ID="Sesame">
    <service:presents rdf:resource="#Sesame_Prof"/>
</service:Service>

<do:RDFStorage rdf:ID="Sesame_Prof">
    <service:presentedBy rdf:resource="#Sesame"/>
    <do:hasFunctionality rdf:resource="do:RetriveRepositoryList"/>
    <do:hasFunctionality rdf:resource="do:StoreData"/>
    <do:hasFunctionality rdf:resource="do:ExtractRDF"/>
</do:RDFStorage>
```

In fact, it is very easy to define different Sesame instances with different combinations of these functionalities.

Each atomic service implements one of these functionalities. Therefore, at matchmaking time, if someone wants to use Sesame he gets a list of all functionalities and then he can search for services which implement these functionalities. Note that there can be multiple services that implement the same functionality. The decision of choosing the one that is the best for a certain task should rely on facts/constraints declared in the domain ontology.

We will further illustrate our discussion with the example of the listRepository functionality. This service accepts a user name and a password and returns the repositories that are accessible with that login information. We defined our service profile as being an instance of the ListRepositoryList type, declared in the domain ontology.

```
<do:RetriveRepositoryList rdf:ID="RepositoryRequest_Prof">
```

The second situation where domain knowledge is used in a Profile is for the functional description of the service. The Profile ontology models a service in terms of a set of parameters which can be Inputs, Outputs, Preconditions and Effects (IOPEs). Each parameter 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 our example we have defined the inputs as being of types User (see markup below) and Password. The output is of type Repository.

```
<profile:input>
  <profile:ParameterDescription  rdf:ID="User_prof">
    <profile:restrictedTo rdf:resource="&do;#User"/>
    <profile:refersTo rdf:resource="&sesame_process;#User1_proc"/>
  </profile:ParameterDescription>
</profile:input>
```

Including domain knowledge in the Process model is not so obvious. The inputs and outputs of a Process are modelled as subProperties of the `process:parameter` property, which ranges over `daml:Thing`. In our modelling, we have used domain concepts to specify the range of inputs and outputs. As the following markup shows, the `User1_Proc` input of the process `RepositoryRequest_Proc` ranges over the class User.

```
<daml:Property rdf:ID="User1_Proc">
   <daml:subPropertyOf rdf:resource="&process;#input"/>
   <daml:domain rdf:resource="#RepositoryRequest_Proc"/>
   <daml:range rdf:resource="&do;#User"/>
 </daml:Property>
```

Finally, in section 3.6 we propose a solution by which complex data types in the WSDL description can be augmented with domain knowledge.

3.3 Linking between parts of the description

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

A Profile description contains links to the Process model. Firstly, it states the Process it describes through the unique property `has_process`. Secondly, the IOPE's in the Profile correspond to the IOPE's in the Process Model. Understanding this correspondence is not so trivial given the fact that the IOPE's play different roles in the Profile and in the Process models. In the Profile ontology they are treated equally as parameters of the Profile (they are subproperties of the `profile:parameter` property). In the Process ontology only IO's are regarded as subproperties of the `process:parameter` property. The effects and preconditions 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.

Even more, this leads to an inconsistency in the model. In the DAML-S Profile ontology each ParameterDescription `refersTo` exactly one element of type `process:paramater`.

```
<daml:ObjectProperty rdf:ID="refersTo">
  <daml:domain rdf:resource="#ParameterDescription"/>
  <daml:range  rdf:resource="&process;#parameter"/>
</daml:ObjectProperty>

 <daml:Class rdf:about="#ParameterDescription">
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#refersTo"/>
    </daml:Restriction>
  </rdfs:subClassOf>
 </daml:Class>
```

At this point, two important issues emerge.

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`). The DAML-S coalition states that they do allow inconsistencies between Profile and Process and that identification of inconsistencies will occur when something breaks[**?**]. Since matchmaking is based on the

Profile description, the break may occur during (attempted) usage of the service. The rationale for this design decision is not clear.

Second, because the Process ontology does not model Preconditions and Effects as subproperties of `process:paramater`, it is inconsistent to use entities of this type as values for `profile:refersTo`. Therefore Preconditions and Effects (PE) defined in the Profile should not refer to corresponding PE's in the Process model.

We conclude that this link between the IOPE's at two distinct levels of specification should be corrected and made more explicit.

Further, the ServiceGrounding builds a bridge between the Process Model and the WSDL description of the service. This is influenced by the DAML-S/WSDL mapping which we discuss in the next section.

3.4 DAML-S/WSDL mapping

The mapping between the elements of the DAML-S semantic description and those of the WSDL file is bi-directional: the WSDL file specifies the link back to the Process elements, while the WsdlGrounding grounding builds a bridge between the elements of the Process and the WSDL interface. The specification syntax is rather heavy and the multiple interdependencies make the grounding process complicated.

There are three rules which define the mapping between the DAML-S Process Model and the WSDL description [**?**]. First, each DAML-S Atomic Process corresponds to a WSDL operation. Second, following from the previous rule, each input of the DAML-S Atomic Process 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 Atomic process 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).

We encountered several limitations of these rules when modelling the addData service. Table 1 shows how to send an HTTP request to the addData service. We present our modelling experiences bringing together some of the discussion from previous sections and motivating the issues discussed in the next two subsections.

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 issues of defining constraints on inputs in the next subsection. In this section we will consider how to handle modelling the service in general, and the mapping of the DAML-S models to the WSDL grounding in particular. The Sesame addData service allows RDF data to be uploaded to Sesame. This service can acquire the RDF source in two ways: (1) by uploading the data itself (2) by uploading a URL referring to the data. The parameters specified will determine which method to use.

Intuitively, this situation is similar to the well-known parametric polymorphism mechanism: 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. Because 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), our first intuition was to model addData as a single service

Table 1. Sesame addData Service

Parameter	Required	Description
repository (rep)	yes	Specifies which repository should be added to. This parameter should contain the id of the repository
data	only for method 1	The data that should be added to the repository
dataURL (URL)	only for method 2	The URL of the data that should be added to the repository
baseURL (base)	only for method 1	Specifies the base URL for the data. Needed for method 1 as Sesame does not know where the data came from. For method 2 the base URL, defaults to dataURL
user	no	The user's login name. This parameter is needed to gain access to non-public repositories.
password (pass)	no	The user's password. The 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

and to make the differentiation between the different ways of acquiring RDF source at the grounding level.

Conceptually, we view the service as providing a means of adding some data in RDF format to a specified Sesame repository. We provide a schematic representation of our service and its main components below. To model that view at the Profile (Pr1) / Process (P1) level we described the service as accepting the data from a dataSource [this needs to be added to the ontology as with other concepts described in this section] and adding that data to the specified repository (rep). The user, password and format are specified as optional parameters. This service outputs a status report of the transaction (status).

WSDL does not support specifying parametric polymorphism, i.e. cases when an operation has the same name but a different signature. This follows from the restriction that the operation names are unique within a certain PortType. One can overcome this, by placing the operations with the same name in different ports or naming them differently. We chose the second approach because the first one does not hold in the context of DAML-S matching. Consequently, our WSDL representation of the service consists of a PortType (addData) with two operations (method1, method2) which differ through their input messages. In addition to the repository name, user login name, password and output format needed in all cases, the first operation expects the actual data and a base URL, while the second receives the URL of the actual data.

```
*Service addData1:
 *Profile:Pr1 (hasProc=P1)(I(rep), I(dataSource),I(user),
                           I(pass), I(format), O(status))
 *ProcessModel:
    AtomicProcess:P1(I(rep),I(dataSource),O(status))
 *WSDLGrounding: Grounding1
    WsdlAtomicProcessGrounding: Gr (P1->method1)  not possible!
 *WSDLGrounding: Grounding2
    WsdlAtomicProcessGrounding: Gr (P1->method2)
 *WSDL:
```

```
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 for the second method was easy. However for the first grounding (Grounding1) we realized that it was impossible to build a one-to-one mapping between the five inputs of process P1 and the six input message parts of the method1 WSDL operations. This assumption is imposed by the DAML-S/WSDL mapping approach which states that "each atomic process input and output corresponds to a WSDL message part" [**?**]. While, the DAML-S coalition acknowledge that this could be a possible limitation, they were unable to give a concrete example of a problematic scenario. Our example shows that the assumption prohibits modelling of parametric polymorphism. Therefore we encourage its revision.

[**?**] 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, i.e. type polymorphism. The solution offered is to define a higher level concept that covers all possible alternative data types. We adopted such an approach by defining a "dataSource" to cover both data and dataURL input. However, this approach does not address the additional problem that we face with different numbers of arguments.

We encountered another limitation of the DAML-S/WSDL mapping when modelling a complex internal structure for the service. At a closer look we observe that, conceptually, the addData service is a combination of three processes: a login process and two different data acquisition processes. In this situation we model two separate Profiles (P1, P2). At process level, one can chose between two composite processes (AddData1 or AddData2), each being a sequence of the LogIn process and a different DataReader (DR1 or DR2). Conceptually each composite process (AddData1, AdData2) corresponds to a WSDL operation: the inputs of the LogIn and DataReader processes combined are the same as the inputs of the WSDL operation, and the output of the DataReader corresponds to the output of the WSDL operation.

```
Service addData:
 *Profile:Pr1 (hasProc=AddData1)(I(rep),I(data), I(base), I(user),I(pass),
              I(format),O(status))
 *Profile:Pr2 (hasProc=AddData2)(I(rep),I(URL),
              I(user),I(pass),I(format),O(status))
 *ProcessModel:
    CompositeProcess: CP:Choice
    {
    CompositeProcess: AddData1:Sequence{
        AtomicProcess:LogIn(I(rep),I(user),I(pass))
        AtomicProcess:DR1(I(data),I(base),I(format),O(status))
        }
    CompositeProcess: AddData2:Sequence{
        AtomicProcess:LogIn(I(rep),I(user),I(pass))
        AtomicProcess:DR2(I(URL),I(format),O(status))
        }
    }
 *WSDLGrounding:
    WsdlAtomicProcessGrounding: Gr1 (AddData1->method1)   not possible!
    WsdlAtomicProcessGrounding: Gr2 (AddData2->method2)   not possible!
*WSDL: Same as before
```

However the first assumption of the DAML-S/WSDL mapping states that "a single atomic process corresponds to a single WSDL operation" [**?**]. Therefore we cannot map the composite AddData processes to the WSDL operations, which is logical from a conceptual point of view. The DAML-S coalition acknowledge that this rule of the mapping could be a possible limitation, but they are skeptical about "the importance of relaxing this assumption" [?]. This example shows that the assumption prohibits modelling of a complex internal structure if it is not directly reflected in the web interface of the service. We see this as a serious limitation to conceptual modelling.

```
Service addData:
 *Profile:Pr1 (hasProc=P1)(I(rep),I(data),I(base),I(user),I(pass),
               I(format),O(status))
 *Profile:Pr2 (hasProc=P2)(I(rep),I(URL),I(user),I(pass),
               I(format),O(status))
 *ProcessModel:
    CompositeProcess: CP:Choice
    {
    AtomicProcess:P1(I(rep),I(data),I(user),I(pass),
                   I(format),O(status))}
    AtomicProcess:P2(I(rep),I(URL),I(user),
                   I(pass),I(format),O(status))}
    }
 *WSDLGrounding:
    WsdlAtomicProcessGrounding: Gr1 (P1->method1)
    WsdlAtomicProcessGrounding: Gr2 (P2->method2)
*WSDL: Same as before
```

In the end, we came up with the model above. We are not completely satisfied with this model as we are unable to model the P1 and P2 processes as composite processes which we feel better reflects the structure of the service. Also, we could not express parametric polymorphism in an elegant way. Our greatest concern with this final compromise is that we are unsure whether this model would be consistent with the model developed by another team to represent this same service. We feel that if semantics are to be added in a meaningful and useful way, greater direction and precision should be provided by the modelling language so that this uncertainty is minimal.

3.5 Conditional inputs

During modelling Sesame we encountered situations where the existence of an input depended on the truth value of a condition. We identified several types of conditions. The following list is not complete, however we think these cases are likely to appear in other tools (and services) as well.

– (A) conditions which contain arbitrary logical expressions.
– (B) conditions between inputs: often the existence of an input is conditioned by the existence of another input, or generally a condition that involves other inputs. For example, in case of the addData service if the actual data is supplied, rather than a URL, then a baseURL must be specified. Another example is the password parameter which must be included only if the user parameter is specified.
– (C) conditions which are always True. Required inputs can be seen as inputs which are conditioned by a condition which is always "True".

Concerning Required/Optional inputs (C type conditions), currently DAML-S relies on the DAML+OIL cardinality restrictions to enforce that a certain input is required. Continuing the example taken from the listRepository service in section 3.2, if we want to enforce that `User1_Proc` is required we must set an exact cardinality restriction with the value 1 on the corresponding process. Similarly, optional inputs can occur maximally once (maxcardinality=1).

```
<daml:Class rdf:ID="RepositoryRequest_Proc">
  <rdfs:subClassOf rdf:resource="&process;#AtomicProcess"/>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#User1_Proc"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

We have several concerns related to this solution. First, it does not handle situations A and B, where an actual condition has to be specified. The second concern is one of usability. This mechanism assumes a good understanding of DAML+OIL and its expressivity. If DAML-S will become a standard to be used by software engineers this assumption might not hold anymore.

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. 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).

```
<daml:Class rdf:ID="ConditionalInput"/>

<daml:Property rdf:ID="ciCondition">
 <rdfs:comment> The condition of a conditional input. </rdfs:comment>
  <rdfs:domain rdf:resource="#ConditionalInput"/>
  <rdfs:range rdf:resource="#Condition"/>
</daml:Property>

<daml:Property rdf:ID="ciInput">
  <rdfs:domain rdf:resource="#ConditionalInput"/>
  <rdfs:range rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
</daml:Property>
```

A conditional input is an input that only occurs when a condition is true.

To address the usability of the language, we think it would be worthwhile to model concepts that are often used. The Required/Optional inputs are concepts that would be easily understandable by software engineers. Our proposed extension allows us to express these two concepts. A RequiredInput is a ConditionalInput where the value of the condition is always true. An OptionalInput is an input on which a condition is always imposed. The following markup models these concepts.

```
<daml:Class rdf:ID="RequiredInput">
    <rdfs:subClassOf rdf:resource="#ConditionalInput"/>
    <rdfs:subClassOf>
      <daml:Restriction>
        <daml:onProperty rdf:resource="#ciCondition"/>
        <daml:hasValue rdf:resource="#True"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>

<daml:Class rdf:ID="OptionalInput">
    <rdfs:subClassOf rdf:resource="#ConditionalInput"/>
    <rdfs:subClassOf>
      <daml:Restriction>
        <daml:onProperty rdf:resource="#ciCondition"/>
        <daml:cardinality> 1 </daml:cardinality>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

While our solution extends the DAML-S capabilities to handle inputs depending on certain conditions and allows for defining some commonly used concepts it [what happened here ??]

3.6 Complex (output) types: xsd or DAML+OIL

In the case of Sesame, four functionalities provide their outputs in an xml document that conforms to a specified DTD. We expect that this scenario will be generally valid for complex tools. To support automatic discovery and operational composition of such tools, their markup should provide information about the (1) syntax as well as (2) the semantics of their output.

Any WSDL description contains a section for complex type definition. While this section can be filled in with any XML based language (XML Schema, RDF(S), DAML), the most common practice in industry is to use XML Schema for defining complex types.

For producing WSDL descriptions that support DAML-S grounding, the DAML-S coalition suggests replacing XML Schema based type definitions with DAML+OIL based definitions. The claim is that this would use DAML+OIL's rich data typing feature [ref iswc02]. Technically, one can define these output types in an external ontology and refer to them when constructing messages, or, alternatively include their description in WSDL's type section. The available DAML-S examples demonstrate this, however, only for simple data types.

We felt 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 then its ancestor. Even so we did not achieve our goal. Second, we think that a WSDL document using purely DAML+OIL data types would be useless for those users of the service which are not DAML-OIL speakers. Ideally, a traditional WSDL description should undergo minor changes when enabled for use with semantic technology. This would ensure its backward compatibility with traditional applications. A third, more conceptual consideration, is that the approach proposed by the DAML-S coalition mixes conceptual and implementation level details. In our thinking there should be a clear separation between them [argument why].

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.

As a concrete example, consider the simplest functionality of Sesame: requestRepository. It's output conforms to the DTD specified in below.

```
<!ELEMENT repositorylist (repository)*>
<!ELEMENT repository(title)>
<!ELEMENT title          (#PCDATA)>

<!ATTLIST repository
        id       ID           #REQUIRED
        readable (true|false) #REQUIRED
        writable (true|false) #REQUIRED>
```

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. Also, in the domain ontology we do not need to define implementation level details, and therefore we can reuse it for other groundings as well.

4 Conclusions and Future work

Our global conclusion is that DAML-S offers a useful set of terms for describing tools however, as any generic ontology they only allow for building a framework of the description which has to be filled in with domain knowledge. Therefore we encountered several major issues which were not covered by DAML-S.

First we had to build a domain ontology that would describe our tool in more detail. Such domain ontologies will be used for matchmaking between many components, therefore we tried to design it to be as extendible as possible and to define our concepts in terms of standard ontologies. Currently the only ontology we know about is DAML-S. [yes but do others exist - this sounds like the problem is with our knowledge] We envision that in the future more standard ontologies will emerge and that they will be used 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 is highly relevant future work.

Also, in our domain ontology we tried to address some of the limitations of DAML-S. Some of our solutions are generic and could be considered as future extensions to DAML-S. In a future perspective, we think that applying DAML-S to different fields could lead to the emergence of more generally interesting concepts. In fact, there are already several works which propose certain extensions to DAML-S. [which ones ??]

A final note regarding domain knowledge is that when a service is used in different usage scenarios its domain knowledge can undergo modifications. In fact we have used DAML-S for describing Sesame in the context of two different projects. In both cases different views of the service were required.

Second, we have struggled with the definition of conditions. DAML-S creates a frame for specifying conditions, a Condition concept. Specifying the content of such conditions is left to the user. Unfortunately there is no standard way of encoding conditions in current semantic languages. While we could express some constraints with the built-in restriction mechanisms of DAML+OIL, defining generic rules remains a future work. We expect that expressing rules will enable us to enrich our domain knowledge.

We also have some conclusions about DAML-S in general. One of the difficulties was the unclear conceptual model of a service. In fact different parts of the descriptions regard a service according to different metaphors. The links between these models can cause inconsistencies (both in the DAML-S ontology and in the actual service descriptions). Another observation is that the current DAML-S/WSDL mapping is rather limiting. This hampered declaring parametric polymorphism as well as describing complex conceptual structures which have no direct correspondent at implementation level. Finally, DAML-S was generally difficult to learn. For a more detailed argumentation of these observations, see [wwwpaper], where we report on the modelling of two simple web-services.

During our modelling we uploaded the descriptions in Sesame and used the query engine to extract information. Our results are encouraging and we think that Sesame can be a good basis for a matchmaker. Extending Sesame's reasoning capabilities and writing a DAML-S interface are two possible future developments.