Ad-hoc Invocation of Semantic Web Services

Andreas Eberhart
AIFB, University of Karlsruhe
eberhart@aifb.uni-karlsruhe.de
http://www.aifb.uni-karlsruhe.de/WBS

***Abstract*— We present the Web Service Description Framework (WSDF), which provides both a representation mechanism and a runtime system architecture for semantically enriched Web Services. We analyze existing languages such as BPEL4WS and OWL-S before addressing their deficiencies in our proposal. Our approach allows a client to invoke a service based solely on a shared ontology, i.e. without prior knowledge on the API, providing an important building block towards a global, flexible information infrastructure. Another main point is that WSDF can be applied to clients and services written in a conventional object oriented programming language. This is achieved by lifting datastructures to an ontology level in which rich logical statements about services can be formalized. We also present a detailed system architecture that covers planning, invocation, and the automatic processing of the service results, which is accomplished using the observer design pattern and by asserting the result in the respective model. Furthermore, the required annotations can be specified conveniently by placing comments in the source code.**

I. Introduction

Web Services are quickly gaining momentum in the IT industry. A large array of tools is already available and with the major players such as Microsoft and IBM making peace in the DCOM vs. CORBA middleware war, problems such as tools interoperability and competing standards are solved. However, Web Services are currently used as a mere replacement for older remote procedure call mechanisms. This means, that they are applied for building distributed systems within the organizational control of a team of developers appointed by the business partners. Companies are very rarely using existing third party services in their applications. The very poor quality of services offered in the global UDDI registry support this observation. One of the main reasons for the, in this respect, very slow adoption of Web Services is definitely the lack of a clear business model and the lack of an accepted payment infrastructure for third party services.

While these are issues concerning the provider, the requestor also faces some problems when trying to consume a service. Currently, there are two ways in which services are being integrated. First, one can use services, which support an existing B2B standard such as RosettaNet. In this case, the standards body determines the meaning of the respective application programming interface (API) and the messages being exchanged. Client and server then develop their systems accordingly. Unfortunately, not too many standards have been successful since such undertakings very often suffer from the dilemma of having to be concise and easy to implement as well as being broad and general in order to appeal to a large community. In the second scenario, a human searches for a suitable service and incorporates it into the system by hand. This is obviously an expensive and very inflexible solution.

Obviously, the current language stack of SOAP, WSDL, and UDDI is not the final picture and new languages such as BPEL4WS, WSCI, and OWL-S are being developed. These approaches are a step in the right direction, but we feel that they do not solve the issue raised above. In this paper we propose the Web Service Description Framework (WSDF), which addresses this problem by allowing a formal specification of the service semantics that seamlessly integrates with the datastructures used. This allows several tasks such as the decision whether to call a service, the actual generation of the required parameters, as well as the proper assertion of the result into the client's data model to be carried out automatically. WSDF requires client and server to provide a mapping from the local structures to a common domain ontology at design-time. This process is carried out in a declarative fashion and does not require any modifications in existing code.

The rest of the paper is organized as follows. The next section summarizes the benefits and deficiencies of some advanced Web Service languages. Section III introduces the ideas underlying our approach as well as the actual language constructs and section IV explains how ontologies are used to mediate between different datastructures. The system architecture of WSDF is shown in section V before we outline our future research and conclude the paper.

II. Current Languages

Before presenting our approach, we first take a look at existing work. Currently, several Web Service-related languages are being discussed. They range from supporting sessions and transactions, quality of service issues, to publishing and subscribing to events. Considering the issue of actually invoking a Web Service in a more flexible way, two approaches are closely related, which we discuss in the following two subsections.

A. BPEL4WS and WSCI

The Business Process Execution Language for Web Services (BPEL4WS) was motivated by the need to have engineers and business people cooperate more efficiently. The idea is to declaratively specify a workflow or business process using a graphical editor in very much the same way processes are drawn on paper by domain experts. These processes are then connected to the technical plumbing, i.e. a messaging system and services to be invoked on the enterprise information system. The graphical representation together with the links to the technical world is then serialized in BPEL4WS and can be executed by various process engines. BPEL4WS now supersedes earlier languages like the Web Service Flow Language (WSFL) [1] or Microsoft’s XLANG [2], which was part of the BizTalk tools suite.

The main part of BPEL4WS is a process definition language with terms like sequence, choices, etc. An important aspect for the invocation is that BPEL4WS has the notion of variables and assignments in order to ”...extract and combine data in interesting ways to control the behaviour of a process” [3]. Consequently, one can think of BPEL4WS to be the glue that combines a set of services on the top level. Additionally, powerful mechanisms for exception handling, compensation, and transaction support are defined.

The Web Service Choreography Interface (WSCI) [4] has a quite similar scope compared to BPEL4WS. Politically, it is expected that the two approaches join forces with BPEL4WS taking the lead in the joint effort.

B. Critique of BPEL4WS and WSCI

BPEL4WS and WSCI are valuable tools compared to having to hand-write this glue in the traditional imperative way. However, they still require the process engineer to understand the service interfaces before manually combining them and thus they do not solve the problem described earlier. BPEL4WS and WSCI make the life of the process engineer a lot easier, but clearly, a deeper semantic service annotation would be desirable.

C. OWL-S

The Semantic Web community aims at providing machine understandable meaning to resources on the Web [5]. Consequently, the lack of semantics in solutions like BPEL4WS shows that one should also consider the respective language out of the Semantic Web area. The Web Ontology Languages for Services (OWL-S) is the language of choice if the resources to be described are services [6]. OWL-S supercedes and extends the DAML-S proposal [7].

The language defines a top-level ontology for services. This ontology aims at formalizing commonly used service concepts such as ”effect”, ”input”, or ”process”. The three main components are the services profile, i.e. what the service does, the services model, i.e. how it works, and the grounding describing how to access the service. We pick up the stock quote example presented in [8]. The parameter and return values are described using the terminology ”ticker” and ”quote” from a financial ontology. In addition to this, a precondition and effect are specified in that the user’s account must be valid and will be charged by the operation.

The main operation supported by this kind of tagging is the so-called semantic matching which allows a more specific concept to be associated with a more general one via the subsumption hierarchy. This is used to locate a service by matching the actual and requested service types. Semantic matching is also applied when comparing the service’s capabilities. A desired output, for instance a ”quote”, is satisfied by a service yielding a ”detailed:quote”, if a quote subsumes the detailed quote concept.

D. Critique of OWL-S

OWL-S is definitely striving for a completely automated framework in which services can be composed and invoked by intelligent software agents on the fly. However, we believe there are some shortcomings. We believe that semantic matching along class hierarchies is not nearly enough. The key issues described in the following points need to addressed. This list will also serve as a set of requirements for our solution presented in the next section.

1) *Datastructure Mediation*
Clearly, the datastructures of client and server need

to be aligned. This is a classical data integration problem in which domain ontologies are an essential tool. OWL-S is currently lacking this aspect.

2) *Rich Annotation*
If a domain ontology is used to mediate between client and server, such an ontology should also be used for describing parameters, return types, preconditions, and the effects of a service. This needs to be done, not just by referencing a certain concept, but also via some rule mechanism, which allows much more flexibility.
3) *Result Processing*
An important point missing is what to do with the service result. BPEL4WS allows defining the dataflow from one service to the next. Consequently, an engine processing such a workflow knows in which way an intermediate result is supposed to be used, i.e. as a parameter for the next service call. In a more automated environment, an agent would need some knowledge, not only about the ontological concept the result refers to, but what to actually do with it.
4) *Parameter Relationships*
Currently, OWL-S associates the parameters and return types of a service with concepts of an ontology. However, no knowledge about their relationships are provided. Consider the service `float getCurrentTemp (String zip)`. OWL-S would tag the parameter zip as a zip code and the return type as a temperature value. This makes sense and holds much more knowledge than the primitive data types alone. However, the information that the temperature refers to the region specified by the zip code is not represented. This is quite clear to a human programmer, though not necessarily to a software agent.
5) *Hidden Assumptions*
Besides the relation of result and parameter, quite often there are more hidden assumptions which are encoded in the method name. Consider the example above. A human will know, that the method yields the *current* temperature. Again, this needs to be made explicit for a software agent.

Summarizing the points above, OWL-S provides value in that it supplies a standard top-level service ontology and that various tasks such as service locations and service capability matching are supported by class subsumption axioms. However, we believe that this is not enough. A semantic service description should be allowed to make more detailed logical statements about the service, its parameters, its results, and its side effects. The terminology to be used must definitely be a domain ontology, which is linked to local data structures.

E. *WSMF and WSMO*

Recent initiatives are the Web Service Modeling Framework (WSMF) [9] and its successor the Web Service Modeling Ontology (WSMO) [10]. WSMF and WSMO base on four main components, namely ontologies, goal repositories, web services, and mediators. The proposals differ from OWL-S in that architectural aspects play a more prominent role. For instance, it is argued, that client and server might expose a different invocation pattern such as bulk vs. cursor oriented data retrieval. WSMO also places a much bigger emphasis on using f-logic to represent richer axioms of the ontology [11]. This viewpoint is very much in line with ours. These frameworks are currently the basis for discussion is the architectural committee of the Semantic Web Services Initiative (SWSI). WSMO and WSFL are very promising approaches but currently lack the required level of detail.

III. Web Service Description Framework

After reviewing the current alternatives, evaluating their strengths and weaknesses, and coming up with some requirements, we will now present the Web Service Description Framework (WSDF) with its underlying design principles. While working though a case study, we re-identify the requirements listed above, and show how they are addressed by WSDF.

A. *Services as State Transitions*

We start with a simple example of a pure informational weather service:

```
public float
    getCurrentTemperature(String zip)
```

This service has two preconditions: there must be an object X, which is an instance of a concept Region. The object also needs an associated ZIP code. We formalize these statements as follows:

```
Region(X) and
hasZIP(X, Z)
```

If the client's fact base allows unifying instances to the variables X and Z, then, in principle, the service can be invoked. Section IV will show, how we use ontologies in order to access an object-oriented model as if it was a fact base in a classical logic sense. Once specific values are bound to the variables, the following statement allows the actual invocation of the service. The number one denotes the fact that Z is the first (and in this case only) parameter.

```
parameter(1, Z)
```

After the client calls the service, its state changes in the sense that the client will be aware of the temperature information at location X. This is formalized as the following set of effects. Note that the variable R is implicitly assigned to hold the invocation's return value:

```
assert TemperatureInfo(T)
assert hasTemperature(X, T)
assert hasTime(T, now())
assert hasValue(T, R)
```

Thus, a successful invocation causes the client to instantiate a new temperature info object T and set the three relations specified. We can see that the relationship between the parameter Z and the result R of the service are made explicit, which covers requirement four. The explicit assertions to be performed allow the client to automatically process the result, addressing requirement three. Note that this formalism explicitly states the knowledge that the service yields the current temperature. On a syntactic level, this is only exposed by the method name. As long as the ontology provides suitable definitions, effects can also include side effects such as a change in ownership caused by an e-commerce service. This addresses requirement five, making hidden assumptions explicit.

Let us switch to a more complicated example. Consider the typical e-commerce application, where a registered user first browses products, then selects some of them in the shopping cart, before proceeding to the checkout. The API might look like this:

```
ItemList keywordSearch(String keyword)
SID createSession()
void addToCart(SID s, Item i)
bool authenticate(SID s, Passport p)
void checkout(SID s)
```

This example obviously requires sessions. We assume these sessions to be explicitly managed via session ID parameters passed along as the first argument. In principle, HTTP cookies or other session mechanisms do the same, only in a way that is transparent to the programmer. We are currently working on a built-in mechanism for tagging methods that implicitly propagate session IDs. The method addToCart does not return a result but obviously causes a side effect on the server side in that the item is now associated with the shopping cart. The respective precondition asks for a session with the e-commerce site to be present:

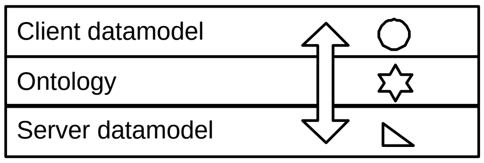

Fig. 1. The service specification is done entirely using terms from the ontology.

```
Session(S) and
Cart(C) and
withServer(S, "my e-shopping")
has(S, C)
```

The effect of the service is that the item will be in the cart. This effect tells the client how to update its model of the world after the service is called. Similar statements are possible about one's bank account being charged and a shipment being sent upon checkout.

```
assert cartElement(C, I)
```

B. Datastructure Mediation

Finally, we want to elaborate on the mediation of datastructures. The temperature example only dealt with primitive types. The second scenario uses the notion of an item. Note that item is neither a structures of the client nor the server. It is merely a concept specified in the ontology. All statements made about preconditions, parameter bindings, and effects are *always* specified in ontological terms. Figure 1 illustrates this approach. The next section explains how this seamless mapping works in detail.

IV. Objects, Classes and Ontologies

Ontologies are a formal, shared representation of domain concepts. Consequently, they have been a hot topic in the area of Enterprise Application Integration (EAI), since they allow to bridge between various schemata used by business partners [12]. Therefore, Ontologies can also be used to mediate between the datastructures used by the service provider and the client.

At this point we are facing a key question. On the one hand, the previous section showed how logical statements using terms of the ontology can help to better describe a service. On the other hand, systems are usually not implemented using a knowledge base that conforms to the domain ontology. The question is how the world of logical expressions can be married with the actual implementation platform on both the client and the server side.

A. RDF for Data Integration

For this purpose, we are using the Resource Description Framework (RDF), the most basic language in the Semantic Web language stack. Note that WSDF is also a mixture of WS and (R)DF. The following properties make RDF an ideal solution in this setting:

- RDF uses a directed labeled graph as its data model. A graph is the most general datastructure possible and consequently, just about anything can be represented as a graph.
- A major problem in any kind of data integration project is the fact that globally unique primary keys are usually not used. The primary key of a local database is completely meaningless and ambiguous outside of the respective peer. RDF uses URIs for addressing the nodes and therefore is well suited in that respect as well.
- Another frequent data integration problem is that attribute or column names need to be matched up. RDF also uses URIs here. A property labeled ”http://www.sap.com/hr/employee/name” is definitely easier to identify than a database column called ”emp_name”.
- Since the labeled arcs of the RDF graph can be interpreted as binary relationships, RDF lends itself for being used with rules and logical expressions.
- RDF graphs can be supported by an RDF Schema or better an OWL ontology which basically provides a type system and axioms for the graph.

B. An RDF Model for Objects

An RDF graph is made up of a set of so-called triples or statements. These statements consist of the source or subject, the predicate or label, and the object or destination. This set of statements is also referred to as the model. Any kind of RDF API like Jena2[1] or KAON[2] uses this notion and offers a respective interface for it. Two of the main methods are listed below:

```
interface Model {
    void add (Statement s)
    Statement[] query(Selector s)
}
```

[1]http://jena.sourceforge.net
[2]http://kaon.semanticweb.org

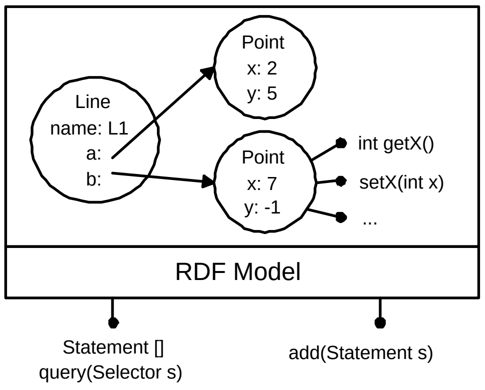

Fig. 2. The RDF model interface is provided on top of regular objects.

The method ”add” asserts a new statement on the model, whereas ”query” allows the caller to retrieve statements. For instance, one might ask for all information known about Microsoft:

```
res = query(new Selector(
  new URI("http://www.microsoft.com"),
  null, null))
```

To assert the fact that Microsoft has its headquarters in Redmond, one would write:

```
add(new Statement(
  new URI("http://www.microsoft.com"),
  new URI("http://comp.com/hasHQ"),
  "Redmond"))
```

Note that the statement’s object can be another resource identified by a URI or a literal value, like ”Redmond” in this case.

The RDF model interface can be implemented in various ways. Jena for instance, allows storing the triples in main memory or a relational database. An implementation, however, does not necessarily have to contain triples. Figure 2 shows that we access regular objects of an object oriented programming language via an RDF view implemented as an RDF model. For this view to be operational, some associations need to be defined a priori. We outline the algorithm for mapping between objects and RDF triples and explain what role the information provided by the user plays in the process.

1) Object Identity: Object instances obviously correspond to resources in the RDF graph. Using the rdf:type property, resources can be made instances of a certain class[3]. Object identity is an important issue here. Locally, objects are identified by their address in memory, whereas resources are identified by their URI. We need to distinguish two different situations in which a) the object has a (possibly composite) primary key variable, or b) the object needs to use an artificial identifier.

In the first case, the user simply specifies a prefix such as http://comp.com/emp/. Assume the object contains a variable ”emp_id” which is the primary key of the, then employee 7 gets the URI http://comp.com/emp/7. Besides the prefix, the user needs to specify the variable, which is uniquely identifying the object. In the second case, the primary key remains unspecified and defaults to the object’s memory address.

2) Class Instances: While the object identity can be quite tricky, the association of classes to concepts in the ontology is very straightforward. Every class is simply tagged with the respective concept URI. This yields the following mapping functions:

```
URI object2uri(Object o)
Object uri2object(URI u)
URI class2concept(Class c)
Class concept2class(URI u)
```

If a statement (URI_a, rdf:type, URI_b) was to be added to the model, then the RDF view would first check if uri2object(URI_a) already exists in the model. If not, a new instance of concept2class(URI_b) is be added. At this point we must assume, that a no argument constructor is available. The other way around, if the user would query for the rdf:type of http://www.microsoft.com, the answer would be class2concept(uri2object(”http://www.microsoft.com”).getClass())

3) Statements: From an RDF point of view, being an instance of a class is just another statement. From a object oriented point of view, those statements have special semantics. Consequently, the normal statements need to be associated with respective actions, i.e. methods. We simply associate an RDF property to a class variable and get two more conversion functions:

```
URI variable2property(Variable v)
Variable property2variable (URI u)
```

[3]Note that in the Semantic Web context, multiple inheritance is allowed. Resources can also be instances of several classes at the same time. See [13] for a discussion of this in an object oriented context. We can safely omit these cases if RDF is merely used as an exchange format between peers using structured datatypes.

If a statement (URI_a, URI_b, c) was to be added to the model, then we would perform uri2object(URI_a).uri2variable(URI_b) = literal2java(c, URI_b). In this case, the RDF object is a literal. Consider figure 2. The statement (urn:point_a, urn:x, ”0”) would then cause the assignment p.x = 0.

The last two mapping functions convert RDF literals to the respective programming language types[4]. We assume RDF literals to be represented as strings. Consequently, literal2java needs to be provided with the information which data type is the desired result of the conversion. This information can be derived from the RDF predicate, which can be associated with the target variable. Note that we require the target variable to be an object type. This is necessary, since ”do not know” information can only be modeled via objects, since in this case the reference to the value can be null. Primitive data types do not offer this feature[5].

```
Object literal2java(String lit, URI u)
String java2literal(Object o)
```

In case URI_c is given, instead of c, the we perform uri2object(URI_a).uri2variable(URI_b) = uri2object(URI_c). An example from figure 2 would be (urn:line_L1, urn:b, urn:point_a) would then cause l.b = p, with l being the line object and p being point a. Queries work in an analogous way.

4) Transformation: Transformation is be a process as shown in figure 1, where an object is converted from schema A to schema B. This process can easily be mapped to the operations defined earlier. Given an object, we simply query all its properties recursively, until the respective sub graph is extracted. This set of statements is then asserted into an empty model yielding the transformed object.

5) Reflection: Note that modern programming languages such as Java and C# are interpreted. This allows the RDF view to be implemented in a completely generic way via the so-called reflection mechanism. Reflection enables to dynamically determine the method to call at runtime. It is very important to note that the objects can still be accessed via their normal methods such as ”setX” in the example. The data is only stored in one place, thus there is no need to synchronize between the RDF view

[4]According to the revised RDF specifi cation, RDF literals can now also be complex XML fragments. We do not yet consider this case, since it would require us to also incorporate XML schema type information in order to process a nested XML structure.

[5]Starting from JDK 1.5, Java will support auto boxing of primitive types in objects. This feature is already available in C#.

and the normal object view.

C. *Observer*

”Observer” is a well known design pattern, which allows several listeners to be informed about state changes in a central data model via a publish subscribe mechanism [14]. For instance, the observer pattern is applied in the model view controller paradigm, which allows several views on a graphical user interface to be based on the same data source. A big advantage of the observer pattern is the clean separation of concerns, for instance between the data model and the front end.

This design pattern fits very nicely to WSDF, since it identifies a central place in which information is stored. Separate issues that need to be taken into account when updating the data are implemented in the listeners. This very decoupled architecture is very suitable to the logic-based approach described in the previous section, since new information can be asserted in the central model. Furthermore, our RDF view solution shown in figure 2 can be very nicely applied to a system which is written using the observer pattern. The idea is that the various subjects being observed are collected in the object pool underlying the RDF view.

```
interface RDFView extends Model {
    void register(Object o);
    void unregister(Object o);
}
```

This interface provides to methods to register and unregister observer subjects or other objects that do not actively notify listeners. Once an object is registered, it can be queried and modified via the RDF model interface.

V. Architecture

This section presents the technical architecture underlying WSDF. Figure 3 shows the architecture of our framework and serves as a reference for the following detailed descriptions. The architecture is currently under development using Java.

A. *RDF Model*

Both the client’s and the server’s datastructures need to be tagged property, in order to allow the data to be exposed as RDF. This tagging is performed in the JavaDoc style, which allows to parse and process the annotations using the popular XDoclet tool[6]. Table I shows the XDoclet vocabulary concerned with the data mediation.

```
/**
 * @wsdf:concept    concept-uri
 * @wsdf:uriprefix  prefix
 */
class Coord {
    /**
     * @wsdf:property  concept-uri
     * @wsdf:primarykey
     */
    Integer id;
}
```

B. *java2wsdf*

Besides the datastructure tagging, the only add-on on the server side is the java2wsdf tool, works very much the same way. The service implementation is tagged with the precondition, the parameter assignment, and the effects. Table II shows the XDoclet vocabulary concerned with the semantic service description.

```
/**
 * @wsdf:onto   ontology-url
 */
class Service {
    /**
     * @wsdf:precondition  expr
     * @wsdf:parameter     expr
     * @wsdf:effect        expr
     */
    void addToCart(SID s, Item i)
}
```

After the logical expressions are extracted, we run our prolog2ruleml[7] converter, since RuleML [15] being a web based markup for rules, lends itself for this purpose. Furthermore, RuleML is currently being combined with OWL ontologies in the SWRL proposal [16]. Obviously the actual WSDF language document is quite verbose and we restrict the discussion in this paper to the concise representation used in the Xdoclet comments. The WSDF document contains all of the logical expressions, which use ”urified” concept and property names as well as the data type mappings and a link to the underlying ontology.

C. *Client Runtime*

As mentioned before, we assume a client to be programmed using the observer design pattern. The RDF

[6]http://xdoclet.sourceforge.net/

[7]http://www.aifb.uni-karlsruhe.de/WBS/aeb/prolog2ruleml/

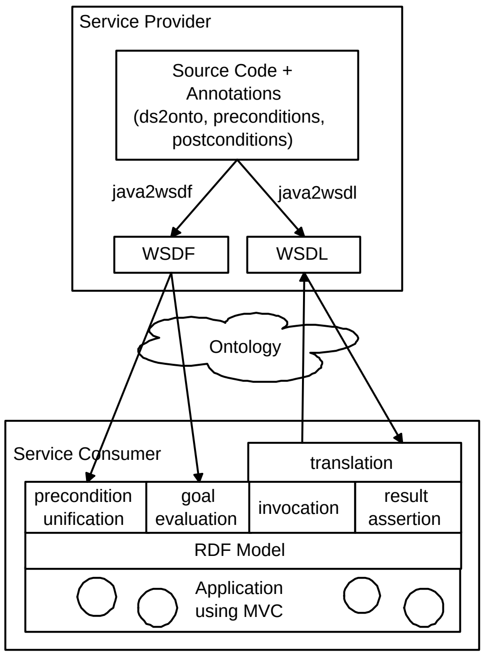

Fig. 3. System architecture

Name	*Scope*	*Description*
uriprefix	Class	prefix to be attached when object ID is translated to URI
primarykey	Variable	marks the variable as (part of the composite) primary key
concept	Class	associates a class with a concept of the ontology
property	Variable	associates a variable with a property of the ontology

TABLE I
XDOCLET VOCABULARY FOR DATA MEDIATION

Name	*Description*
precondition	conditions to be true before the invocation
parameter	parameter assignments based on precondition variable unification
effect	service effects based on precondition variable unification
onto	physical URL where the ontology can be accessed

TABLE II
XDOCLET VOCABULARY FOR THE SEMANTIC SERVICE ANNOTATION

model interface described in the previous section lies on top of the client's core datastructures. Initially, the client downloads the WSDF description along with the WSDL file that is necessary to generate the stub.

There are four major components in the client runtime environment. First, the service's preconditions are unified against the local RDF model. The goal evaluation component determines if the effects contains the desired information, e.g. the temperature at location X. This component is still work in progress. We plan to use a breadth-first search algorithm during the planning phase [17] in which the algorithm simulates possible state transitions from service invocations. Finally, the method is invoked using, in our case, Java reflection along with the information which unified data serves as which parameter. Finally, the result is transformed to RDF as specified in the effects, and asserted via the RDF to object mapping. Note that at this point the transformation explained in section IV-B.4 is applied in order to "cast" an ontological concept back to the expected parameter structure, and on the way back to mediate between return value and ontology again.

VI. FUTURE WORK

Our future work can be mainly divided in three aspects. The first one is concerned with the relationship of WSDF to other languages. Obviously WSDF can nicely complement a process description language, since the focus is clearly on a single invocation. The planning component described in the architecture section might be augmented with the information of what the process and the invocation sequence usually look like. Also, our rules syntax and semantics needs to be kept in line with the emerging work on rules and ontologies in the form of the Semantic Web Rule Language (SWRL) [16].

The second issue addresses issues such as the representation and handling of sessions, transactions, exceptions, and lists. We are in the process of defining the necessary extensions here.

Finally, we are investigating the use of transaction logic [18] in order to describe the state changes more cleanly. Currently, the transactional semantics are more or less hard-coded in the planning algorithm. We are also working on incorporating more results from the planning community, which will definitely provide a much better solution than our current approach using a simple breadth-first search algorithm.

VII. CONCLUSION

We presented WSDF both on a conceptual and an implementation level. WSDF provides semantic annotations to Web Services allowing the ad-hoc invocation of a service without prior knowledge of the API; a feature that is missing in current approaches such as OWL-S or BPEL4WS. WSDF is layered nicely on top of the well-established WSDL standard and, in contrast to our previous version [19], has been extended to fit clients programmed in an object-oriented language. The only prerequisites are that the observer design pattern is used in the client and that the object properties can be set in a uniform manner, as is the case with Java Beans for instance.

REFERENCES

[1] F. Leymann, "Web Services Flow Language (WSFL 1.0)," http://www.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, May 2001.

[2] S. Thatte, "XLANG - web services for business process design," http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, "Business process execution language for web services version 1.1," May 2003.

[4] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimek, "Web service choreography interface (wsci) 1.0," http://www.w3.org/TR/wsci/, August 2002.

[5] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, pp. 28–37, May 2001.

[6] T. O. S. Coalition, "Owl-s: Semantic markup for web services," http://www.daml.org/services/owl-s/1.0/, November 2003.

[7] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara, "DAML-S: Web Service description for the Semantic Web," in *Proceedings of the First International Semantic Web Conference (ISWC 2002)*, I. Horrocks and J. Hendler, Eds. Chia, Sardinia, Italy: Springer, June 2002, pp. 348–363.

[8] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan, "Automated discovery, interaction and composition of semantic web services," *Journal of Web Semantics*, vol. 1, no. 1, December 2003.

[9] D. Fensel and C. Bussler, "The web service modeling framework wsmf," *Electronic Commerce: Research and Applications*, no. 1, pp. 113–137, 2002.

[10] U. Keller, H. Lausen, D. Roman, J. Gomez, R. Lara, A. Polleres, C. Bussler, and D. Fensel, "Web service modeling ontology, working draft," http://nextwebgeneration.com/projects/wsmo/2004/d2/, February 2004.

[11] M. Kifer, G. Lausen, and J. Wu, "Logical foundations of object oriented and frame-based languages," *Journal of the ACM*, vol. 42, pp. 741–843, 1995.

[12] J. Pollock, "The Web Services scandal," *EAI Journal*, August 2002. [Online]. Available: http://www.eaijournal.com/PDF/AugustCoverStory.pdf

[13] A. Eberhart, "Automatic generation of Java/SQL based inference engines from RDF Schema and RuleML," in *Proceedings of the First International Semantic Web Conference (ISWC 2002)*, I. Horrocks and J. Hendler, Eds. Chia, Sardinia, Italy: Springer, June 2002, pp. 102–116.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Redwood City, CA, USA: Addison-Wesley, 1995.

[15] H. Boley, S. Tabet, and G. Wagner, "Design rationale of RuleML: A markup language for Semantic Web rules," in *Semantic Web Working Symposium*, Stanford University, CA, USA, July 2001, http://www.semanticweb.org/SWWS/program/full/paper20.pdf.

[16] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "Swrl: A semantic web rule language combining owl and ruleml," http://www.daml.org/2003/11/swrl/, November 2003.

[17] A. Eberhart and S. Agarwal, "Smartapi - associating ontologies and apis for rapid application development," submitted to the workshop: Ontologien in der und fr die Softwaretechnik at Modellierung 2004, March 25th 2004 Marburg, Germany.

[18] A. J. Bonner and M. Kifer, "Transaction logic programming," in *International Conference on Logic Programming*, 1993, pp. 257–279. [Online]. Available: citeseer.nj.nec.com/bonner93transaction.html

[19] A. Eberhart, "Towards universal Web Service clients," in *Proceedings of the Euroweb 2002: The Web and the GRID: from e-science to e-business*, B. Hopgood, B. Matthews, and M. Wilson, Eds., Oxford, UK, December 2002, http://www1.bcs.org.uk/DocsRepository/03700/3780/eberhart.htm.