Semantic Web Services on the .NET Platform

Andre Loker
a.loker@student.utwente.nl

ABSTRACT
Semantic Web Services have attracted a lot of interest. The prospect of automatic discovery and processing of Web Services promises new possibilities in fields such as business-to-business communication and automatic data integration. However, despite the demand from the industry little practical support for developers on popular software platforms such as Microsoft .NET is available. By implementing a prototype based on OWL-S we found that especially support for logic reasoning and a convenient way to represent ontologies is currently lacking on the .NET platform.

Keywords
Semantic Web Services, OWL-S, .NET

1. INTRODUCTION
In the last years Web Services [1] have attracted a lot of attention. Today thousands of public and non-public Web Services are available (for example through [2]). This development can be explained by several reasons. First of all, software systems have become much more interconnected, often resulting in complex heterogeneous systems. Using Web Services and the related protocols can solve the problem of heterogeneity to a large extent [3], primarily due to the vast acceptance of HTTP and XML. Second, Web Services open numerous possibilities for business-to-business and business-to-customer relations. For example a travel agency might consume services provided by hotels and airlines to find and book flights and accommodations. The use of open or semi-open Web Services in business-to-business relationships is considered to become even more important in the future [4]. Finally, on a more technical side, the application of Service Oriented Architectures tends to improve scalability and maintainability of large software systems [3].

1.1 Web Services
With Web Services a standardised technical platform for distributed systems is available. Web Services leverage well known protocols and infrastructure (such as XML and HTTP) to let different systems communicate with each other. Web Services are mainly based on three standards: First, WSDL [5] (Web Service Description Language), which provides metadata to define the type of data used as input and output by a certain Web Service. It also syntactically describes the process model provided by the Web Service. Second, SOAP [6] (originally: Simple Object Access Protocol), which is an actual communication protocol between Web Service clients and servers. And finally, UDDI [7] (Universal Description, Discovery and Integration), which describes a repository for Web Services.

1.2 Semantic Web Service
While Web Services provide the technical infrastructure for distributed systems they do not include semantic information [8], that is, none of the three standards above describes what the web service does in terms of the respective domain and how each of the web service’s methods contributes to a complex business process. As a consequence users of conventional Web Services generally need to select appropriate Web Services by hand and a certain amount of manual intervention is needed to define the interaction between multiple services (orchestration, [9]).

Semantic Web Services (SWS) [10] try to fill this gap by adding semantic features to Web Services. The semantics of an SWS are expressed in a domain specific ontology which formally describes the given domain by defining concepts, instances and relations of those concepts and axioms within the domain. Ontology languages such as OWL [11] are machine-readable.

The semantic dimension can be used to formally describe inputs and outputs of an SWS method and the meaning of a method invocation in terms of preconditions (conditions that must be met before a web service method can be invoked) and postconditions or effects (conditions that hold after a successful invocation) [12]. The four-tuple of inputs, outputs, preconditions and effects are often abbreviated as the service’s IOPE.

Adding semantics to web services has the major advantage that it makes automatic discovery, invocation and composition possible. A client capable of SWS asks an SWS repository for a web service by providing the requirements in terms of the desired effects or outputs using the ontology. The repository, using logic deduction and the knowledge expressed in the ontology, can provide the client with a set of services that suits its needs [13]. The selected SWS then provides semantic metadata about the business process that it represents and the valid order and combination of service method calls that are required to execute those processes. Because the requirements and effects of each method invocation are expressed in terms of a domain ontology the client can execute complex business processes without user intervention as described in [10].

Several standards for SWS have been developed and proposed, two important of which are OWL-S [14, 15] and WSMO[16, 17].

1.3 Microsoft .NET
Microsoft .NET [18] is primarily a software development platform. It consists of a runtime environment, an extensive class library and support for different kind of services, such as web services. .NET is one of the most widely used enterprise platforms today [19]. Although .NET Web Services are not strictly tied to a specific server infrastructure, Microsoft’s web server IIS (Internet Information Services) is the preferred server to run .NET Web Services on. According to [20] the current market share of IIS servers on the Internet is around 35%.
2. PROBLEM STATEMENT

Despite the great potential that Semantic Web Services offer and the demand from the industry [21] SWSs are not widely used in practice yet. The lack of available software tools and libraries for SWS consumption and engineering is considered an important reason for this [22]. As a consequence institutions such as the S3 Contest [22] or the SWS Challenge [23] have been established, asking for practical realisations of SWS systems.

By providing and improving support for SWSs on a major enterprise software platform such as .NET we therefore suspect that the rollout of SWSs can be notably accelerated. Trying to improve the support for SWS on .NET initially leads to three major questions:

1. What are the general functional requirements to provide and consume Semantic Web Services?
2. What could a possible software architecture that meets those requirements look like?
3. Which of those requirements are currently met by available software running on the .NET Framework and which parts are currently missing?

The first two questions will be answered in section 5, whereas the third question is answered in section 6. Answering those questions will help us to answer the research question:

What has to be done to provide sufficiently complete support for SWSs on the .NET platform?

3. APPROACH

The research has been done in three main phases:

1. Identification of requirements for SWS processing
2. Extraction of a suitable architecture
3. Identification of missing support on .NET, verified using a prototype

3.1 Requirement identification

To answer the first question we compared several promising SWS standards (OWL-S and WSMO) with respect to the functionality they provide. The selection was motivated based on two main criteria:

- Availability of documentation, i.e. how well is the standard documented. This was necessary to ensure full understanding of the provided functionality.
- Presence in literature as a measurement of importance. We consider standards that are often referenced in contemporary literature to be of high interest.

The careful analysis of the selected SWS standards led to a set of functionality those standards provide. By comparing those sets we were able to identify the general functional requirements for an SWS implementation, on the client side as well as the server side.

3.2 Architecture extraction

Based on the list of functional requirements we derived a software architecture that we consider suitable of providing the required functionality. This architecture is independent of any programming language or library and should be applicable to any software environments such as Java or .NET. This phase led to the answer of the second question.

3.3 Identification of support and verification

The final phase of research was the identification of available and respectively missing support on the .NET platform to put the proposed architecture into practice. In this context, “support” has to be understood as the availability of software libraries and application programming interfaces (APIs) providing functionality that is required to implement a working SWS client and server. We searched for suitable facilities available

- in the framework class library (FCL) that comes with the current .NET version 3.5
- as third party libraries, provided that they are freely available

To verify the validity of the proposed software architecture a prototype implementation has been developed on the .NET software development platform. The prototype demonstrates the following functionality:

1. Server side: a small set of semantically similar SWS which can be “discovered” by using semantic queries (SWS discovery)
2. Client side: a simple SWS client which generates queries against the server and executes “discovered” SWS using the process model as described in those services (SWS execution)

Section 6.2 describes the test cases in more detail. We chose for .NET as the exemplary platform because it is widely used in the enterprise sector and already has built-in support for conventional, non-semantic web services.

Functionality that can not be provided by available .NET APIs (neither by the FCL nor as third party libraries) has been added in one of two ways:

1. Where it can be done with reasonable effort, the missing functionality has been implemented
2. Otherwise, the expected behaviour has been implemented as stubs, that is, code that will work as expected for the test cases but not necessarily in other contexts.

4. SWS FRAMEWORKS

Due to the increased interest in semantics extensions to the Internet in general and specifically Web Services a number of different SWS standards has been developed which differ in the amount of functionality and flexibility. Especially two SWS standards are often mentioned in literature.

- OWL-S [14, 15], which is an extension to the Web Ontology Language (OWL [11]) created to describe semantic web services in terms of a service profile, a process model and the service grounding
- WSMO [16, 17], a conceptual model of semantic web services which is manifested in the Web Service Modelling Language (WSML [24]).

Initially we considered WSDL-S [25] as well. While this standard is certainly interesting because of it being an extension to the well known WSDL, we still excluded it from the research because it does not support complex processes as much as OWL-S and WSMO. Furthermore it is not referenced in literature that much.

In the next two subsections we will briefly introduce the concepts of OWL-S and WSMO without going into detail too much.
4.1 OWL-S

OWL-S is a Semantic Web Service specification developed by the World Wide Web Consortium (W3C, [26]). It builds upon the well established Web Ontology Language (OWL), which stores semantic knowledge in the form of triplets.

A Semantic Web Service based on the OWL-S specification consists of three parts: profiles, process models and groundings. The profiles of an OWL-S Web Service act as a description of the service. They can be used during service discovery to search for Web Service with specific capabilities. A process model describes the way that the Web Service is fulfilling those capabilities. It is built as a hierarchy of processes. So called atomic processes form the leaves of those hierarchies, whereas complex processes allow for more complicated process models, such as sequences of processes or conditional processes. The third part of the OWL-S specification is the service grounding, which describes how the process model should be executed physically. Typically, the atomic processes of the process model are mapped to invocations of conventional, non-semantic web services using protocols such as HTTP, XLM and SOAP.

4.2 WSMO

The Web Service Modeling Ontology is a specification for semantic web service systems that aims for high modularity and interoperability. WSMO is based on four top level concepts:

1. Like OWL-S, WSMO leverages the possibilities to express facts in terms of the business domain by using the concept of Ontologies.

2. Web Services perform the tasks to reach certain goals. Web Services are defined in terms of capabilities (what they are able to do) and interfaces (how they can be accessed and combined).

3. The concept of Goals describes what the user of a WSMO based service wants to achieve and how the user wants to communicate with the service. It is noteworthy that goals and web services are two distinct concepts that can exist independently. The connection between goals and web services are established by mediators.

4. Finally, Mediators are used to handle heterogeneity and to link concepts together. Different kinds of mediators are defined by WSMO, for example to mediate between different ontologies or goals or to provide the link between goals and web services.

4.3 Selection

OWL-S as well as WSMO only describe a model of semantic web services without being explicit with respect to an implementation. In essence both provide comparable functionality which can be categorized into three main aspects:

1. Description of the service’s capabilities
2. Description of the service’s process model
3. Description of the service grounding

The main advantage of WSMO over OWL-S is the clear modularity and separation of concerns. Mediators are used to link heterogeneous processes and ontologies, which allows a more modular development of partial solutions and more flexibility in heterogeneous environments.

OWL-S on the other hand uses a hierarchical approach that combines atomic processes to complex processes. The main advantage of OWL-S is the fact that it is based on W3C standards (such as OWL, RDF [27]). In the past standards backed by the W3C have shown to be adopted rather well in practice. Popular examples are HTTP, HTML and CSS. Consequently a large amount of information regarding OWL-S is available in form of literature and documentation. Furthermore, we found the basic concept of OWL-S to be less complex compared to WSMO, making it more suitable for a simple prototype.

We therefore chose to base the rest of the research – especially the prototype – on OWL-S. However, the research results are not limited to OWL-S, especially when the functional similarity of OWL-S to standards such as WSMO is considered.

5. AN SWS ARCHITECTURE

In this chapter we will present the functional requirements that we could extract from the examination of the different SWS standards. We will then provide a conceptual model of an architecture that is capable of handling those requirements.

5.1 Functional Requirements

To identify missing functionality regarding SWS on .NET it is essential to first find out what is actually required by an SWS system. In this section we will describe the functional requirements that we consider most important. We limit ourselves to functional requirements. Non-functional requirements (so-called quality requirements) are deliberately ignored.

The functionality that should be provided by a SWS system can be broken down into three major groups:

1. Advertising: This is the server side capability to tell the environment which semantic web services are available and what they provide.
2. Discovery: The client side ability to find a set of web services and to pick one that matches the client’s needs.
3. Invocation: The client side ability to execute a service model as described by the service meta data.

5.1.1 Advertising

A server that provides semantic web services needs to advertise this capability to the environment. Conventional web services use WSDL for this purpose. Semantic web services on the other hand will provide semantically enhanced information. OWL-S for example uses the ServiceProfile for this, while WSMO advertises service by means of its Goals and Capabilities.

Furthermore the server needs to provide a process model which describes precisely how the client is supposed to use the services to achieve a specific effect and result.

The advertising functionality results in the following requirements:

R1: The server must be able to advertise the semantic web services it provides in terms of a specific ontology.

R2: The server must be able to provide a process model of the services it offers.

5.1.2 Discovery

The client needs to find a web service that is capable of achieving the desired results. The process of finding and selecting such a service is called discovery. Although automatic discovery is one of the major advantages of semantic web services with respect to conventional web services, the discovery process may be skipped if the client already knows which semantic web service to use. Automatic discovery is therefore desirable but not necessarily crucial. If automatic discovery is not provided a suitable semantic web service has to
be selected manually by the client. From that point on, however, execution can still happen automatically. This leads to the following requirements:

R3: The system should provide a repository that acts as a collection of all known semantic web services.

R4: The repository should be able to select an SWS from the set of known services given the desired result and/or effect.

5.1.3 Invocation

After the client has selected an SWS (either manually or by automatic discovery) that service needs to be executed. This execution has to be done according to the process model as advertised by the SWS server. Depending on the complexity of the process model the client needs to communicate with one or more remote servers once or multiple times. In the OWL-S process model each atomic process invocation corresponds to one communication roundtrip to a server. We will use the term “atomic invocation” to denote those individual roundtrips. Regarding process invocation the following requirements have been identified:

R5: The system must be able to perform atomic invocations from the client to the server.

R6: The system must be able to execute a given process model by performing the atomic invocations as required.

5.2 Components

After we have identified the functional requirements of an SWS system we will now introduce a software architecture that we consider capable of handling those requirements.

An SWS system can be broken into five core parts:

1. Common functionality used throughout the system
2. Server side specific functionality
3. Client side specific functionality
4. Registry side specific functionality
5. Functionality required for client-server communication

The diagram in Appendix A depicts a conceptual model of the proposed architecture that shows the different components and the data flow between them. Furthermore, Appendix B contains an activity diagram that shows the different stages that are executed when a Semantic Web Services is discovered and executed.

Common functionality:

- **Ontology abstraction.** Ontologies are a core component of SWS, therefore the system will need to deal with ontologies in some way or another. Different abstraction methods are possible, ranging from a full domain model to a simple in-memory storage (triplets in the case of RDF).

- **Ontology reasoner.** While this is closely related to the ontology abstraction it can be clearly distinguished. The ability to reason about an ontology is independent of its representation. The ontology reasoner allows the system to make use of the knowledge represented by an ontology, for example the fact that an entity “Ferrari” is a subclass of “Car”.

- **Logic reasoner.** Again, this is closely related to the ontology reasoner. The logic reasoner is used wherever preconditions or effects are described. Also, clients may want to express the desired effect and result using some logic language (such as SWRL [28]) when searching for a specific SWS. Because the language used to express logic can be completely separate from the representation of the ontology we distinguish this part from the ontology reasoner.

- **Data serialization.** Because client and server are not expected to reside in the same machine process or even the same machine data that needs to be transferred between the two peers needs to be converted into a format suitable for the transportation medium, for example from and to SOAP [6].

Server side:

- **SWS implementation.** Obviously the semantic web service needs to be implemented in one way or another on the server.

- **Metadata generation.** The server needs to somehow generate metadata for the SWS it provides. The metadata should provide information about the SWS’s purpose, its process model and the way it needs to be accessed (grounding). The metadata is generally presented in a format specific to the SWS standard, such as OWL-S.

- **Atomic process handler.** When requested by the client, the server needs to execute an atomic process as provided by the SWS implementation. In general this functionality is not specific for a certain SWS but can be implemented rather generically, for example by using the possible ability to let the system invoke a method with a given name.

Client side:

- **Semantic query.** The client executes an SWS in order to achieve specific goals, like “having a trip booked”. To find an SWS that can achieve those goals the client queries the SWS repository using a semantic query.

- **Execution engine.** Once an SWS has been found the execution engine is responsible for the proper execution of that service’s process model. Specifically this includes the proper flow of parameters between atomic process invocations.

- **Execution state.** For complex processes the execution engine needs to store some intermediate state.

- **Atomic process invocation.** Just as the server needs to handle atomic process request, the client needs to generate those requests. More specifically, the execution will convert each required atomic process invocation into a request according to the selected SWS standard.

Registry side

We separated the “registry side” from the client and server side because it can reside on either of the two sides or even independently. The registry acts as a repository of available SWS that can be queried to discover an SWS suitable for a given task.

- **Repository.** The repository acts as a dictionary of available semantic web services.

- **Matchmaker.** The matchmaker accepts a semantic query that describes the desired results and/or effects of an SWS and returns the description of services in the Repository that possibly lead to those effects and results. It uses logic reasoning to fulfill this task.

Client-server communication

This last section essentially provides the infrastructure for client-server communication. For Web Services this generally
consists of an HTTP server on the server side and some functionality to send HTTP requests on the client side.

5.3 Requirements coverage
Table 1 gives an overview about which components described in section 5.2 are used to fulfill certain requirements. As can be seen from the table, all requirements are covered. In the next section we will illustrate to what extent the components have been implemented in our .NET based prototype.

Table 1: Requirements covered by architecture. An X in a cell means that the component (row) supports a given requirement (column).

<table>
<thead>
<tr>
<th>Requirement</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ontology abstraction</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Ontology reasoner</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Logic reasoner</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Data serialization</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWS implementation</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metadata generation</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic proc. handler</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semantic query</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution engine</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution state</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic proc. invocation</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Repository</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matchmaker</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6. AVAILABILITY IN .NET

6.1 Prototype
To verify the validity of the architecture proposed in 5.2 we built a small prototype that is supposed to execute the test cases described in the next section. The prototype consists of a server side which provides several semantic web services according to the OWL-S standard and a client application that is supposed to discover and invoke those web services. The software has been written using C# 3.0 and the current version 3.5 of the .NET framework. When implementing the components described in 5.2 we applied the following priority:

1. If the feature can be implemented using the framework class library, do so.
2. Else try to find a third party library that can help to implement the feature.
3. If no third party library is available, try to implement a simple working version
4. If the feature is too complex write a stub that will let the test cases pass without being necessarily applicable in general cases.

The interested reader is pointed to http://andreloker.de/sws where a detailed description of the prototype as well as a running example and the source can be found.

6.2 Test cases
To test the suitability of the proposed architecture and to find lacking support on .NET we considered two test cases. Although they are not very sophisticated they still cover all components of the proposed architecture. Especially we were interested in the matchmaking process and the execution engine.

Test case 1 – “Online book store”: In the first test case the client wants to find an online book shop that sells print media (books, magazines etc.) about the topic “Programming in C#”. At a central service repository several stores are registered, each selling different kinds of goods. Among those shops some are selling print media, some of which are about “Programming in C#”. The client wants the repository to return those shops that sells print media and that support a web service to query for books by topic. The client then invokes the semantic web service to receive a list of books on the desired topic. All those steps should be executed automatically by the system without any intermediate interaction by the user.

This scenario has been chosen to test the process of service discovery and the invocation of atomic processes.

Test case 2 – “Travel agency”: In this second test case, the execution of more complex process models is tested. At a service repository several online travel agencies are registered. They all provide the same functionality: booking a vacation trip to a specific country, including a hotel, a flight and a rental car. However, they implement this using different process models. The client tries to book trips at each of those agency, based on the user’s wishes (destination country, hotel class, car category) and executes the process models properly using the execution engine. If the process models are executed correctly, all travel agencies will respond with a similar answer, that is, a booking number and information on the flight number, hotel name and car number.

This test case is interesting because it tries to show another major advantage of SWS: the integration of services that fulfil a similar task using different process models.

6.3 Available support for SWS in .NET
The attempt to build a prototype that could run the two test cases has pinpointed the parts where support on .NET is missing. The amount of support for SWS related features on the .NET platform differed from component to component, as was expected. In this section we will briefly summarize to what extent components could have been implement using the FCL, third party libraries or our own implementation.

6.3.1 Common components
Ontology abstraction, ontology reasoner and logic reasoner:
The FCL does not provide specific support for ontologies at all. One of the few publicly available libraries that can handle RDF – which is the recommended representation of OWL – is SemWeb [29]. While SemWeb provides good support for processing RDF documents it is not aware of OWL specific extensions. The available reasoners (RDfs and Euler) can be used to reason about the RDF data itself but not about expressions used in pre-conditions and effects. The prototype therefore currently only uses expected outputs for SWS matchmaking.

Data serialization
As expected, the .NET framework class library provides rather good support for data serialization in formats required for the SWS prototype. SoapFormatter and XmlSerializer can (de)serialize whole object graphs from and to SOAP or more generally XML. This feature is also used by the built-in (conventional) Web Service processing facilities.
6.3.2 Server side facilities
SWS implementation and atomic process handler
Conventional web services are implemented as classes with special attributes. Each method that has the [WebMethod] attribute applied can be invoked as a Web Service method, each of which can be considered an atomic process. Therefore no additional implementation is required for the atomic process handler. By using the built-in Web Service capabilities the server keeps backwards compatibility to conventional SOAP based services.

Metadata generation
Metadata for conventional Web Services, i.e. the WSDL document, is generated automatically by .NET using reflection (the ability of .NET to access metadata of compiled code). There is no built-in support to generate OWL-S related metadata, such as the service profile, process model or grounding information. Neither did we find a library that could support this task. Writing the XML/RDF documents manually however is not very convenient. Just as .NET supports the automatic generation of a WSDL document from source code there should also be convenient means to generate the OWL-S metadata document. Different possibilities can be thought of, for example
1. Use of metadata (attributes) and reflection in source code
2. Use of a document object model with a streamlined API, that is, a programming interface that is more specialised towards the use in semantic web services (this approach is used by the prototype)
3. Introduction of a Domain Specific Language (DSL) to describe semantic web services

6.3.3 Client side facilities
Semantic query
A framework that allows the expression of semantic query is currently missing. Only desired outputs (instead of effects) can be easily defined in the prototype as of now. This relates to the lack of support for logic reasoners.

Execution engine and state
While there is no ready-made .NET based SWS execution engine as of now, its implementation is very well possible, as the prototype shows. Special attention has to be given to data flow and concurrency.

Atomic process invocation
Most often clients of conventional web services will let the .NET framework create a proxy class based on an imported WSDL resource, for example by using wsd.exe (part of the framework). Invoking a web service method then boils down to a call to one of the methods generated for the proxy class. We can leverage this facility for the invocation of atomic processes as well. The class ServiceDescriptionImporter in the System.Web.Services.Description namespace can generate the proxy classes on demand during runtime given a specific WSDL document. Because the OWL-S services can be grounded to WSDL documents the prototype is able to invoke atomic processes without much extra effort.

6.3.4 Registry side facilities
Repository and matchmaker
The quality of the search results returned by the matchmaker highly depends on the availability of a well working logic reasoner. As such a reasoner is currently lacking the prototype provides only limited support for matchmaking. Only the type of the result can be queried, not for example the effect in terms of the ontology.

OWL-S does not define a standard for SWS repositories and matchmaking. However, with Matchmaker [30] such a system is available for Java as an open source project. A port to .NET should be possible.

7. CONCLUSION
Our prototype has shown that the proposed architecture for SWS systems is suitable in the case of OWL-S. Because of the similarities between OWL-S and WSMO we expect it to be applicable for WSMO systems and maybe other SWS related standards as well.

By developing a (mostly) working prototype we were able to find out to what extent an SWS system can be implemented using .NET technology in practice. It was not surprising that the .NET framework class library provides extensive support for everything related to the consumption and advertising of conventional web services, because web services have been an area of application from the first version of .NET on. Luckily our prototype could benefit from that support because parts of the OWL-S implementation, namely the service grounding, makes use of existing facilities such as WSDL.

On the other hand there are some obvious gaps in available functionality and/or convenience.
1. Foremost the lack of extensive support for logic reasoning makes automatic discovery difficult. The ability to discover one or more services based on expected effects and results is one of the major benefits of Semantic Web Services. It simplifies the integration of similar but heterogeneous services.
2. We would like to see a simplified way to handle ontologies in code. While SemWeb provides an in memory view of an ontology, a type safe representation of the ontology as classes would be beneficial. A promising approach to bridge between the triple representation of RDF and the object oriented world is the ROWLEX.NET library [31].
3. While our prototype has implemented OWL-S as an example other SWS standards should be supported as well. We encourage the development of an abstraction layer that makes the differences between different SWS standards transparent to the user. By providing an abstraction layer users could switch SWS implementations without knowing specific details. Likewise, a service could be provided.

7.1 Discussion
The prototype that has been implemented in the course of the research does not cover all possibilities of the OWL-S standard. Therefore it is possible that some caveats in the proposed architecture were not discovered and that there is more functionality missing.

Furthermore it is arguable whether all SWS standards can be covered by the proposed architecture. First of all, only a subset of SWS standards has been examined. Although these standards appeared to be very similar in the functionality they provide there might be differences that cannot be bridged by the provided architecture.

Because the test cases are rather simple, we recommend the development of more sophisticated test cases to do further research for missing SWS related functionality on .NET.
One obvious problem that arises when looking for SWS related functionality not available for .NET is the fact that it is inherently difficult to verify the absence of this functionality. If we had limited ourselves to functionality provided by the FCL it would have been much easier: either something is available in the FCL or it is not. However, the answer would not have been very practical. Most often software will use the FCL as well as some third party libraries. Therefore we see the necessity to look for available third party libraries as well. We present the research results therefore to the best of our knowledge without the assurance of completeness and actuality.

ACKNOWLEDGMENTS
We would like to thank Mr. Maarten Fokkinga for providing valuable support.

REFERENCES
APPENDIX A: CONCEPTUAL MODEL OF THE PROPOSED ARCHITECTURE

Figure 1 – Conceptual model of the proposed architecture. Rectangles represent components, arrows represent data flow between those components. The components in the “Common” group are used by the client as well as the server. In this version the registry side with the repository and the matchmaker is integrated into the SWS client. It could as well exist at a different location.
APPENDIX B: ACTIVITY DIAGRAM

Figure 2 – Activity diagram of the discovery and invocation of a semantic web service. Arrows represent control flow between the different states.