Semantic Web Services are described using semantically enriched markup languages. This semantic description lets external agents understand Semantic Web Services’ functionality and internal structure so the agents can discover, compose, and invoke such services. DAML+OIL or OWL could be the markup language, but we’d have to combine them with standard Web Services languages to use the current Web Services infrastructure. Following this approach, researchers have proposed two specifications that describe services semantically: OWL-S (formerly DAML-S) and the Web Service Modeling Ontology. OWL-S uses OWL in combination with WSDL and SOAP. WSMO uses F-Logic to perform inferences with services, and its XML-based format gives external agents access to the service features.

However, before specifying Semantic Web Services in a Semantic Web-oriented language, programmers should design the service at a knowledge or conceptual level to avoid inconsistencies or errors among the services constituting the Semantic Web Service. In this context, designing Semantic Web Services consists of specifying a service’s descriptive, functional, and structural features. Semantic Web Services composition, therefore, implies combining different services to obtain a new service. So Semantic Web Services design and composition are similar, but composition operates on services already created and is a semiautomatic process. During design, users manually create services using a graphical interface, although some support for composition helps.

Here, we present a framework for designing and composing Semantic Web Services at the knowledge level in a language-independent manner. We base our framework on a stack of ontologies that explicitly describes different Semantic Web Services features, and on the assumption that the Semantic Web Services are modeled as problem-solving methods (PSMs) that describe the service’s internal structure.

**Description ontologies for designing Semantic Web Services**

The conceptual architectures of both Semantic Web Services and Web Services schematize the service design as the specification of a set of layers that would cover all service features. Four types of features enable programs or external agents to discover, invoke, and compose new services.

- **Access (or communication)** features describe the communication protocol (such as SOAP or HTTP) required to invoke the service execution.
- **Descriptive** features detail Semantic Web Services e-commerce properties such as the geographical location, commerce classification (for example, the United Nations Standard Products and Services Code), or provider. These features generally define the domain (such as minerals in the UNSPSC) in which the service operation is carried out. They can also guide the service discovery by rejecting the services that operate on a different domain (for example, the medical domain).
Functional features specify Semantic Web Services capabilities (their input and output data), effects, and pre- and postconditions of execution. Once the service domain has been established, these features let an external agent determine whether the service execution can obtain the requested results. Furthermore, invoking a service requires specifying the input and output data.

Structural features describe a composite service’s internal structure—that is, its structural components (subservices) and how they combine to execute the service. Typically, agents use these features for service composition as they determine if interactions exist between the subservices and the other services used to compose a new service.

These features represent different but complementary views of a service. The feature set used to describe a Semantic Web Service depends on the operation (invocation, discovery/publishing, or composition) that the agent requiring the service performs. For example, to invoke a given Semantic Web Service, the agent must specify both its access and functional features (input and output parameters), although it needn’t know the service’s internal structure (how it executes). The goal of designing Semantic Web Services is to make explicit the four features just discussed. In particular, exploiting each feature’s structural components guarantees the proposed design’s correctness and avoids the inconsistencies among the subservices that are combined manually or automatically to achieve the service’s requirements. For example, to operate in the same domain, subservices should have the same commercial classification (such as medical software in the UNSPSC).

Because we aim to design and compose Semantic Web Services semiautomatically, we need to perform inferences about the service features to determine whether the proposed design is correct. This means we should describe the service features (and the service itself) explicitly and semantically. Using ontologies for this description seems the most appropriate solution. Others have followed this approach using a semantic-enriched markup language to create an ontology (OWL-S) that describes the service features. Our proposal differs in that it aims to develop an ontology set that describes Semantic Web Services at the conceptual (or knowledge) level and that is independent of the language used to specify the service. However, once we create the Semantic Web Services model, external agents must be able to access it. So, we can then translate the Semantic Web Services into a Semantic Web Services-oriented language such as OWL-S.

Figure 1 shows the stack of ontologies that describes Semantic Web Service’s features (and the service itself) using well-known specifications or de facto standards. This favors the framework’s interoperability with applications or solutions that follow one of those specifications. The stack includes four ontologies:

- The problem-solving-method ontology describes the PSMs used to represent a Semantic Web Service’s internal structure and functional features.
- The Semantic Web Services ontology describes the upper-level concepts that define a Semantic Web Service’s features.
- The knowledge representation ontology defines the KR entities that model a Semantic Web Service and a domain ontology at the knowledge level.
- The data types ontology describes the data types that the domain ontology uses.

**Problem-solving-method ontology**

Researchers have traditionally modeled the internal structure of both Semantic Web Services and Web Services as a (business) process,7,8 carrying out a set of activities or actions to execute the process. This approach breaks the process (or service) into activities whose interactions can be modeled as workflow patterns9 that basically describe the coordination of those activities during process execution. Because of this, some researchers have proposed process-based languages such as BPEL4WS (Business Process Execution Language for Web Services) and OWL-S 25 (for Semantic Web Services). These languages describe a service’s internal structure using a predefined set of workflow-like patterns (sequence, choice, parallel split, and so forth).

From a modeling viewpoint, this approach’s main drawback is its lack of an explicit, declarative decoupling between a process’s functional features (what) and its structural description (how). This means that the functional features link directly to the parameters used in the process’s internal structure. So, the process is specifically designed to carry out a particular operation (for example, to book) in a particular domain (such as flight booking). With this approach, reusing processes among domains becomes difficult, and service composition, where software agents use services to obtain a new service, must be programmatically solved. For example, a service that deals with theatre booking shares some operations with a flight booking service (select seat, check credit card, confirm booking, and so forth). Processes that execute such operations should be (quasi) reusable among both services, but this requires differentiating between the description of those operations and how they’re solved.

**PSM-based approach.** To decouple a service’s functional features from its internal specification, we propose applying PSMs10 to model Semantic Web Services, because they claim to be knowledge components that are reusable among different domains and tasks. The Unified Problem-Solving Method...
Language (UPML)\textsuperscript{11} is a de facto standard that describes a PSM’s components: its task, method, and adapter (see Figure 2).

The task describes the operation to be solved in the executing of a method, specifying the required input and output parameters and pre- and postconditions. This description is independent of the method used to solve the task.

The method details the control of the reasoning process to achieve a task. It also describes both the decomposition of the general tasks into subtasks and the coordination of those subtasks to achieve the required result (control flow). The UPML, however, doesn’t define a set of program elements to specify a method’s control flow.

The adapter specifies mappings among a PSM’s knowledge components, adapting a task to a method and refining tasks and methods to generate more specific components.\textsuperscript{12} So, adapters can achieve reusability at the knowledge level because they bridge the gap between a PSM’s general description and the particular domain in which it’s applied.

Researchers developed UPML in the context of the IBROW project\textsuperscript{13} with the aim of enabling the semiautomatic reuse and composition of PSMs distributed throughout the Web. This objective seems similar to that of service composition, and the IBROW project highlights the close relationship of PSMs with Semantic Web Services.\textsuperscript{14}

The WSMO project\textsuperscript{4} has also used the PSM-based approach. The project defines an ontology to describe the components of an infrastructure that facilitates Semantic Web Services discovery and composition. Our approach differs in that it uses a different approach to describe the Semantic Web Service’s internal structure and defines a different (and OWL-S-based) communication model to interact with external agents.

**PSM description ontology.** On the basis of the UPML specification, we created a PSM ontology that enhances the description of the UPML elements. It does this by adding new relationships to the elements, concepts, and axioms that describe more accurately the main PSM components. These add-ons (the dashed lines in Figure 2) are explicit relations, programming primitives, and axioms.

We define explicit relationships between tasks and methods as follows: a method, which can be primitive or composite, solves a task; composite methods decompose a task into subtasks. Composite methods also have an operational description composed of a sequence of program elements that specify the control flow required to solve the general task. Moreover, both tasks and methods have effects associated with changes in the environment that aren’t related to input and output data. The designer should model these effects as an effect ontology’s concepts to favor interoperability among services.

A minimal set of programming primitives describes a composite method’s operational description. These primitives will operate with concept instances (parameters) and tasks (usedTasks), and can express instance assignments, conditions applied to instances and tasks, loops (both conditional and iterative), and parallel execution of other program elements or tasks. Although we select these primitives from programming languages, combining them derives several basic workflow-like patterns such as sequence or exclusive and multiple choice.

A set of axioms describe interactions among the PSM ontology’s instances that should be avoided—instantiations of method and task concepts inconsistent with the PSM theory. For example, a method can’t decompose any of the subtasks of which it’s composed (subsumption problems). Additionally, a (general) task’s inputs should be included in the collection of inputs of its subtasks because the composite method that describes interactions among those subtasks must have the same inputs as the general task. Therefore, we can use these axioms to decide whether a composite method has correctly decomposed a task.

The PSM ontology doesn’t incorporate details about how to make a method network-accessible, because it assumes that the communication protocol description is directly associated with Semantic Web Services.

![Figure 2. The problem-solving-method ontology is based on the Unified Problem-Solving Method Language specification and a programming ontology that describes the different elements that compose a method’s operational description. (Dashed lines represent add-ons.)](image_url)
**Semantic Web Services ontology**

As Figure 3 shows, the Semantic Web Services ontology replicates the OWL-S ontology’s upper-level nodes (or concepts), which describe a service in terms of its descriptive and functional features (profile), access protocol (grounding), and internal structure (model). The ontology incorporates all OWL-S concepts and attributes of the service grounding and profile (except the profile’s functional features, which are replaced by task descriptions). However, the ontology substitutes the concepts associated with the (process-based) service model with method descriptions. It uses the OWL-S specification as a skeleton to define a service, but because the concepts that define the service features are PSM-based, the ontology complies with PSM-based approaches such as WSMO. Considering this, the Semantic Web Services ontology defines the following relationships with the PSM ontology (see Figure 3).

The *profile* concept establishes a relationship (hasTask) with the PSM ontology’s task concept. This is consistent with the notion of tasks and functional features: they specify an interface to describe a service’s capabilities in terms of its input and output data and the pre- and postconditions of task (or service) execution. So, a service profile will have only a task that defines the service’s functional features.

The *model* concept defines a relationship (hasMethod) with the PSM ontology’s method concept. This means that a service will be executed by a method that solves or decomposes the task associated with the service’s profile. Moreover, consistencies in the relationships between tasks, methods, and services are guaranteed: if a task functionally describes a service, and a method solves that task, then the service will be associated with the method. Although several methods can solve a task, the execution of a service must be associated with only one of those methods. For example, a *theatre booking* service could have a method in which the subtask *user register* is mandatory or a different method without such a task.

The *grounding* concept establishes relationships with an ontology based on the WSDL/SOAP specification (such as in OWL-S). This ontology defines relationships with the inputs and outputs of the task related to the service profile. The OWL-S grounding could be substituted for the WSMO choreography, which specifies a more flexible message interaction among the service and requesters.

According to the Semantic Web Services ontology, a composite method describes a service’s internal structure (a composite service) and decomposes a task into subtasks. Each subtask specifies a service profile’s functional features (subservices), whose internal structure is described by a method that solves or decomposes such a subtask. Therefore, unlike process-based approaches (such as OWL-S), the description of subservices in the Semantic Web Services ontology follows the same structure as that of composite services. This means that subservices are services per se—that is, they’re described independently of any service.

This approach facilitates Semantic Web Services design and composition because analyzing the service features doesn’t depend on the services in which the features are integrated. For example, if *check credit card* is a sub-service of *theatre booking*, then deciding whether to include this subservice in a *car booking* service isn’t required to analyze the *theatre booking* features. Moreover, this approach specifies service (or PSM) adapters based on the description of each service’s features.

**Knowledge representation ontology**

This ontology describes the KR model’s primitives. The KR model represents the domain ontology, which contains descriptions about the knowledge and data that the Semantic Web Service uses. This explicit representation of the domain ontology is necessary to perform inferences about the input and output data that the Semantic Web Service uses in its operations. For example, in service composition, we must determine the primitive type of the service’s inputs (for example, a concept) to look for services that use inputs with such a primitive.

Considering these facts, we selected the WebODE knowledge models as the KR ontology. WebODE is frame-based and incorporates formulas (first-order logic) to represent ontology axioms. In addition, it’s a workbench for ontological engineering that offers reasoning capabilities (using ontology axioms) and facilities to export and import ontologies to and from Semantic Web-oriented languages.

**Data types ontology**

As Figure 1 shows, the KR ontology is constructed on top of an ontology that describes types of concepts and attributes. This ontology will be based on the XML Schema Datatypes, a standard specification formally incorporated into the Semantic Web...
Services-oriented (and Web Services-oriented) languages such as OWL-S (and WSDL). Using this ontology, we’ll facilitate translating the Semantic Web Services conceptual model into the Semantic Web Services languages used in service specification.

**The framework**

Our proposed framework details how to create a service with the capabilities that an external agent or user requires (see Figure 4).

We start with an *instance model*. Designing and composing Semantic Web Services involves instantiating all the ontologies that define a service. The domain ontology that the service uses is instantiated in the data types and KR ontologies, and the service features are instances of the PSM and Semantic Web Services ontologies. All the instances constitute a model that specifies the Semantic Web Service at the knowledge level. The service designer could carry out this specification using a graphical interface, introducing the service features and generating the instances through wrappers from the graphical representations.

We then use a *checking model*, because once we’ve created the instance model, we need to guarantee that the ontology instances don’t include inconsistencies. For example, a composite method’s inputs and outputs must be included in the collection of all the inputs and outputs of that method’s subtasks, subservices might need to have the same commerce classification, and so on. So, we need design rules to check that neither inconsistencies nor errors appear when the service designer automatically creates (through the graphical interface) the ontology instances. These rules are codified as axioms of each of the ontologies that constrain how the instances are related. As Figure 4 shows, if an instance or set of instances violate an axiom, the designer should replace them with correct instances. (Our framework assumes that the domain ontology was created earlier and corrected through a platform for ontology development [such as WebODE]; thus, we don’t need to check instances of the data types and KR ontologies.)

Next, we use a *translate model*. Although we model the service at the knowledge level, we must specify it in a Semantic Web Services-oriented language to let programs and external agents access its capabilities. So, we need to translate Semantic Web Services ontology instances into languages such as WSMO or OWL-S. Wrappers usually carry out these translations, which we can assume will be accurate.

Our framework enables the semiautomatic composition of Semantic Web Services using PSM refiners and bridges to adapt the PSM ontology instances to the new service’s required capabilities. It uses design rules to reject both the PSM and Semantic Web Services ontology instances that present errors or inconsistencies among them. It also uses design rules to reduce the service candidates combined to obtain the new service. For example, to check whether the required inputs of the service to be composed are included in the collection of the candidate subservice’s inputs, it’s necessary to adapt, when possible, those subservice inputs to the ontology that the composite service uses. Then we can perform axiom checking after carrying out this adaptation (probably by setting ontology mappings).

We implemented a prototype of an environment for developing Semantic Web Services, called ODE SWS (see http://kw.dia.fi.upm.es/odesws). The prototype uses WebODE functionalities to access the ontologies that define the services’ input and output parameters. In this development environment, users graphically define all Semantic Web Services features. Once the user (or service designer) has specified the service, ODE SWS automatically creates the Semantic Web Services ontology instances and translates the service into the OWL-S or WSMO specifications.

**A framework operation example**

Let’s suppose that, through a graphical interface of an environment for developing Semantic Web Services at a conceptual level (such as ODE SWS), a user designs a service called *theatre booking*. The goal is to book a theatre ticket for a particular film (film) in a given city (city). Figure 5 shows this service’s design using the ODE SWS development environment. The user specifies graphically the input and output interactions among the subservices that compose the *theatre booking* service (see the left side). As we can see, this service comprises three subservices: *select theatre*, *select timetable*, and *buy ticket*. Information about credit cards is included in the user data (user data), and *buy service* validates it.

Once the service has been graphically designed, the ODE SWS environment’s graphical interface creates instances of the framework ontologies. Figure 6 shows the instantiation of the main concepts of both the PSM and Semantic Web Services ontologies for *theatre booking*. The method *theatreBookingMethod* decomposes the tasks associated with this service (theatreBookingTask) into subtasks related to the subservices.
After instantiating the ontology instantiation, we must check whether the service is correctly designed. From the service’s input and output data (see Figure 6), the violation of an axiom is detected: input $\text{cityC}$ of the theatre booking service doesn’t appear as an input of any subservice. This means that $\text{cityC}$ isn’t used in service execution, and the method that decomposes the service into subservices ($\text{theatreBookingMethod}$) doesn’t provide the capabilities required by the task associated with the theatre booking service. So, we should revise this service’s design to include a method that solves correctly the task related to the service.

Finally, once the design has been checked, wrappers perform the translations from the instances of the framework ontologies into the OWL-S or WSMO specifications.

Proposals already exist that support the semiautomatic composition of both Semantic Web Services and Web Services (see the “Related Work” sidebar). However, our approach differs in that it complies with other PSM-oriented approaches such as WSMO: the Semantic Web Services’ internal structure and functional features are based on methods and tasks, respectively. Our future work will explore the interoperability between OWL-S and the ODE SWS framework. The ODE SWS guarantees that services based on the ODE SWS model could be translated into OWL-S. However,

---

**Related Work**

Evren Sirin, Jim Hendler, and Bijan Parsia describe a service in OWL-S and use OWL reasoning capabilities to determine whether the features of the services to be composed match the functionality the new service requires.¹

Srini Narayanan and Sheila McIlraith² translate the semantics behind the OWL-S specification into a first-order logic language, obtaining a set of axioms that describe the service features. On the basis of these axioms, they use the Petri net formalism to represent a process-based service model for reasoning about the interaction among the processes composing a service’s structure. This approach seems similar to our framework, because it uses an ontology that describes what a service is (OWL-S) and then translates this ontology into a formalism for reasoning about the service’s structure and features.

However, the main difference between these two proposals and our framework lies in the way the service is modeled. We consider a PSM-based approach where the tasks specify a service’s functional features, the methods describe the interactions among a service’s components, and the adapters describe declaratively how the reuse is achieved among the services to be composed.

**References**


because of the OWL-S specification’s flexibility, services currently expressed in OWL-S aren’t imported into the ODE SWS environment (there’s not a straight translation from OWL-S to the ODE SWS framework).

Furthermore, we’ll extend the ODE SWS environment’s functionalities to include straight access to distributed libraries of (Semantic) Web Services. Specifically, we plan to provide access to UDDI, OWL-S, and WSMO registers to enhance the number of services available in the ODE SWS environment for the designer. Working with these libraries, on one hand, the service designers could explore which are the more appropriate Semantic Web Services to achieve the required functionality of the new service. On the other hand, a semiautomated algorithm for service composition (based on the service description framework) should be included in the ODE SWS environment to support the development of Semantic Web Services.

Acknowledgments

We thank the Xunta de Galicia and the Esperonto project (IST-2001-34373) for their financial support in carrying out this work.

References


For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.