Comp-O: an OWL-S Extension for Composite Service Description

Grégory Alary, Nathalie Jane Hernandez, Jean-Paul Arcangeli, Sylvie Trouilhet, Jean-Michel Bruel

Comp-O: an OWL-S Extension for Composite Service Description

Gregory Alary, Nathalie Hernandez[0000−0003−3845−3243], Jean-Paul Arcangeli[0000−0002−0521−9082], Sylvie Trouilhet[0000−0003−4330−5034], Jean-Michel Bruel[0000−0002−3653−0148]

Institut de Recherche en Informatique de Toulouse, University of Toulouse, France
{firstname.lastname}@irit.fr

Abstract Component-based software engineering is a paradigm that fosters software flexibility and emphasizes composability and reuse of software components. These are runtime units that provide services and, in turn, may require other services to operate. Assembling components consists in binding components' required services to provided ones to deliver composite services with added value. Building a composite service is a challenging task as it requires identifying components and services that are compatible, binding them to implement the service, and describe it for discovery. For that, the vocabulary used to describe component-based services (i.e., services offered by components or assemblies) must support the description of required services, and descriptions must be combinable in order to automatically generate composite service descriptions. However, existing solutions are limited to the description and composition of provided (and not required) services. In this paper, we consider ontologies to describe component-based services implemented by component assemblies. After comparing existing service ontologies, we present an extension of OWL-S called Comp-O. Through a proof-of-concept, we demonstrate how the added semantics can be handled to automatically build composite service descriptions.

1 Introduction

Component-based software engineering consists in designing software as assemblies of reusable and versatile *software components*. Software components are building blocks that implement and provide services. As they exhibit the services they require at the same level as the services they provide, components are easily composable [1]. In order to make a component fully operational, i.e., actually provide its services, each of its required services must be bound to a service that is provided by another component. Composing components, that is to say building assemblies of components, means binding services based on their abstract specifications (e.g., signatures, pre- and post-conditions). Composition leads to complex *composite services* with added value whose behavior depends on the components that are involved in the assembly.

To improve discoverability by third parties, *component-based services*(CBSs) must be semantically described. When they result from composition, their semantics depend on the ones of the components. The semantics of the services

provided by a component depends on the semantics of the services required by this component. Since these required services are abstracted, the actual semantics depend on the semantics of the provided services they are bound to. In a way, the semantics of a composite service is distributed among the components.

The problem is to describe the services provided by components that have required services, both to enable assistance to the service developer when she/he assembles components and to combine such descriptions to automatically generate composite service descriptions. We propose to describe CBSs with ontologies in order to leverage the semantics of such knowledge representations regarding two issues : (i) support a detailed description of composite services; (ii) support the composition of services and produce a description of a composite service depending on the components participating to the assembly.

Considering ontologies in the description of services improves their discoverability [2] and their composition [2][3]. Several ontologies and approaches exploiting them have thus been proposed. However, existing solutions mainly consider Web services and are not suited for CBSs requiring specific services.

In this paper we propose Comp-O, an extension of the well-known OWL-S ontology in order to consider specific characteristics of CBSs and we demonstrate how the added semantics can be handled to automatically build composite service descriptions. The paper is organized as follows. Section 2 briefly introduces software components, component-based development, and CBSs, then the characteristics of CBSs are exposed. In Section 3, the requirements for a component-based service ontology are presented and tested against several existing ontologies. Comp-O, an extension of OWL-S complying with the requirements, is then presented and instantiated in Section 4. Section 5 proposes an approach to assist the developer in the building of Comp-O composite services and to generate their descriptions automatically. Last, Section 6 summarizes the contribution and discusses some future works.

2 Component-based services

2.1 Components and CBSs

Component-based software engineering is a paradigm that emphasizes composability and reuse of software components. *Software components* are loosely coupled self-contained runtime units that *provide* services specified by interfaces. To provide their services, they may *require* external services. Fig. 1 shows the UML representation of the VoiceToTextConverter component, where the provided services (VoiceProcess) are pictured by a bullet and the required services (TextProcess) by a socket. Unlike objects, software components bring the required services at the same level as the provided ones. As a result, components are building blocks that can be *assembled* by *binding* required to provided services if their interfaces match, to deliver a composite service with added value.

Flexibility is one of the main advantages of component-based development. Components are versatile and reusable in different contexts. In an assembly, a

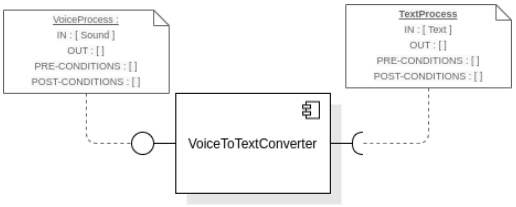

Fig. 1: UML representation of the VoiceToTextConverter component

component can be replaced at design or execution time by another component that offers an "equivalent" functionality, this equivalence being based on compatibility of the interfaces. Interfaces specify a contract of use containing the type of the inputs and outputs, pre-conditions to satisfy when invoking the service and guaranteed post-conditions. We call *component-based service* (CBS) a service that is provided by a software component. If the latter requires external services, the CBS is implemented by an assembly, and its actual semantics depends on the components that are involved in the assembly.

2.2 Illustrative examples

The right side of Fig. 2 represents the TextPrinter component that provides the CBS called PrintText. PrintText takes a Text as the only input: when invoked, the text is printed and there is no result in return. Like a Web service, PrintText is ready to use since TextPrinter has no required interface.

The VoiceProcess service provided by the VoiceToTextConverter component is however not ready to use. To make it work, the TextProcess required service must be bound to a CBS that takes a text as input, e.g., PrintText of TextPrinter (assuming that PrintText matches TextProcess). Fig. 2 represents an assembly that implements a component-based composite service that takes a voice record as input, converts it to a text, and prints it.

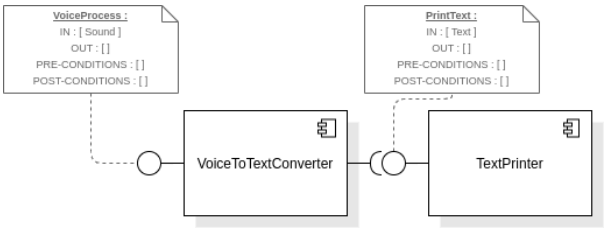

Fig. 2: Implementation of the VoiceProcess composite service

As components are replaceable, a TextTransformer component can be inserted between VoiceToTextConverter and TextPrinter (assuming the services

match), to translate the text before printing it. The result is shown in Fig. 3: when invoked, the TextTransformer component demands the translation in French of the input text then requires the PrintText service.

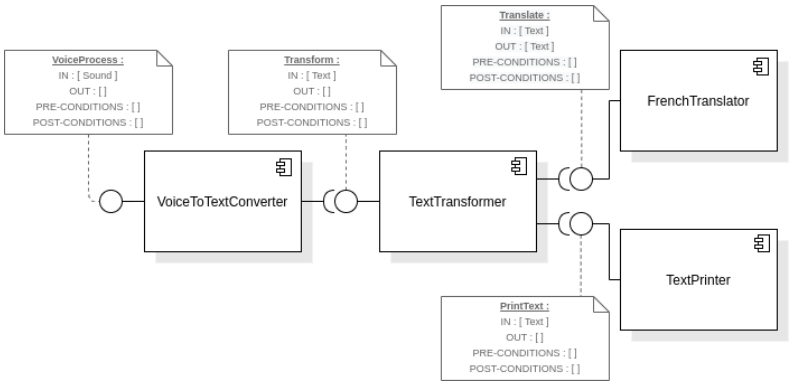

Fig. 3: Another implementation of the VoiceProcess composite service

2.3 Issues

In the previous section, two implementations of the VoiceProcess composite service have been presented in Fig. 2 and Fig. 3: although its interface does not change, the semantics vary from one implementation to another (print a speech, print a speech after its translation into French, ...). So, the true nature of a CBS depends on the components that compose the implementing assembly and what these components actually do. Thus, to determine this nature, it is necessary to inspect the different components. In a component, how a service is delivered depends on the services the component requires, the ordering of the requests and the internal operating process. Therefore, describing CBSs with interfaces only, i.e., as black boxes, is not enough. Interfaces support matching but do not make the behavior explicit. For example, they do not specify that the PrintText service of the TextPrinter component prints the text on paper or elsewhere. Maybe a human could guess this information from the service name, but a machine certainly could not.

Thus, to support efficient service discovery and composition, CBSs must be described semantically. The problem is to build the semantic description of a CBS by a combination of the ones of the components' services. Indeed, describing services provided by a component with one or more required services is fundamental for our work. In addition, these unit descriptions must be combinable.

3 Requirements and comparison with existing ontologies

The development of the requirements of our ontology is compliant with the NeOn methodology [4]. We have specified the purposes and the scope of the ontology,

the uses and the final users, and the competency questions the ontology should satisfy. Competency questions are used to evaluate existing ontologies.

3.1 Purposes and scope

The motivation and final goal of this ontology is to offer a way to describe CBSs, in particular the service offered and the required interfaces that must be bound to make a service operational. Concomitantly, during the development of a new service built with CBSs, the description of each service can be used to automatically generate the description of the composite service.

We have identified two types of users : the service publishers and the service developers. A **service publisher** is an agent wishing to publish the description of CBSs or composite services that will be invokable and bindable. A **service developer** is an agent wishing to bind one or more published services to build a more complex application. In both cases, the services must be described as unambiguously as possible in order to automatize the tasks.

3.2 Competency questions

These competency questions come from an analysis of the component-based software engineering domain and several use cases [1,5] similar to the one presented in 2.2. The use cases are not seen as an end *per se*, but as an instantiation of the general domain of component-based software engineering. Therefore, the competency questions presented in Table 1 represent the knowledge required for a reusable ontology, with no regard for the application domain. The answers are simplified for the sake of readability but should be represented thanks to corresponding resources.

ID	Competency questions	Answers
CQ1	What are all the available services?	(S1, S2, S3); ()
CQ2	What are the types of the inputs of the service *S1*?	(Boolean, Int); (String);
CQ3	What are the types of the outputs of the service *S1*?	(ON/OFF Command, Int); (ON/OFF State); ()
CQ4	What are the preconditions of the service *S1*?	(cond1; cond2); (cond1); ()
CQ5	What are the post-conditions of the service *S1*?	(cond1; cond2); (cond1); ()
CQ6	What is the service offered by the service *S3*?	Square root
CQ7	Does the service *S1* invoke any services?	Yes; No
CQ8	What services are invoked by the service *S1*?	(S2; S3); (S4); ()
CQ9	What is the internal orchestration of the service *S1*?	(invokeS2, invokeS3); (invokeS2); ()
CQ10	Is the service *S1* a component-based service?	Yes; No
CQ11	What are the required interfaces of the service *S2*?	(perform1, perform2); ()
CQ12	What are the types of the inputs of the service required by the required interface *perform1*?	(ON/OFF Command); (Int); ()
CQ13	What are the types of the outputs of the service required by the required interface *perform1*?	(Boolean); (String);
CQ14	What are the post-conditions of the service required by the required interface *perform1*?	(cond1, cond2); (cond1); ()
CQ15	What are the preconditions of the service required by the required interface *perform1*?	(cond2); ()
CQ16	Is the service *S1* already bound with any other services?	Yes; No

Table 1: Competency questions

In **CQ8** , the notion of service binding is only relevant for CBSs as it means that the service S1 invokes another service through one of its required interfaces. **CQ9** is crucial for the generation of composite service descriptions as the behavior of the internal orchestration will help deducing the operational aspects

of the service. The expected answer is an ordered list of operations executed by the service such as invocations, variables operations and returns.

3.3 Comparison with existing ontologies

As recommended by NeOn, reusable ontologies that are compliant with parts of the requirements have been integrated in our design process. Therefore, we have used the competency questions to analyze which ontologies satisfy which part of the requirements. We compared six ontologies: SAREF [6], SOSA/SSN [7], MSM [8], OWL-S [9] (formerly DAML-S), WSML [10] and HRests [11]. For each competency question, the absence of star means that the corresponding ontology does not satisfy at all the question, one star means that the question is partially covered and two stars that the question is totally satisfied.

Competency Question	CQ1	CQ2	CQ3	CQ4	CQ5	cQ6	CQ7	CQ8	CQ9	CQ10	CQ11	CQ12	CQ13	CQ14	CQ15	CQ16
SAREF	**	**	**	-	-	-	-	-	-	-	-	-	-	-	-	-
SOSA/SSN	**	*	*	-	-	-	*	*	-	-	-	-	-	-	-	*
MSM	**	**	**	-	-	-	-	-	-	-	-	-	-	-	-	-
OWL-S	**	**	**	**	**	**	**	**	**	-	-	-	-	-	-	*
WSML	**	**	**	**	**	**	**	-	-	-	-	-	-	-	-	-
HRests	**	**	**	-	-	-	-	-	-	-	-	-	-	-	-	-
Comp-O	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**	**

Table 2: Comparison between the competency questions and the ontologies

All the studied ontologies cover the questions **CQ1**, **CQ2** and **CQ3** as they all provide a way to type a resource as a service and to define the types of its inputs and outputs. WSML also covers **CQ4**, **CQ5** and **CQ6** but not the others as WSML describes a service as black-box, without information about its internal working. SOSA/SSN only partially satisfies the questions **CQ7** and **CQ8** as the invocation of a service by another should be described by using the *observes* and *detects* properties. However, by using these predicates, the semantics are different since the service S2 is not invoked by S1 *per se* but is self invoked when a new observation is detected, as SOSA/SSN is used to describe sensors and observations and is not dedicated to services. OWL-S totally satisfies **CQ4**, **CQ5**, **CQ6**, **CQ7**, **CQ8** and **CQ9**. Preconditions and post-conditions can be described in the profile of the service. Invocations of other services and internal orchestration are described in the service's process. Moreover, the service offered can be semantically described using the preconditions and post-conditions. SOSA/SSN and OWL-S partially cover **CQ16** as they both provide a mechanism to describe the invocation or actuation of a service by another but is not specific enough to describe the binding of an interface with a service. The binding of interfaces is a mechanism specific to CBSs where SOSA/SSN and OWL-S are used to describe Web services. No studied ontology satisfies questions **CQ10** to **CQ15**.

Based on the comparison, we conclude that OWL-S is the ontology that covers the best our requirements. We develop Comp-O, an extension for OWL-S that covers all the competency questions.

4 Comp-O, an OWL-S extension for CBSs

Comp-O is a minimal ontology extending OWL-S that helps to efficiently describe CBSs. The ontology is available at `https://github.com/comp-o`. All the namespaces used in this paper are given in Table 3. This section presents the key concepts of OWL-S, an overview of Comp-O and an example of a CBS description using this ontology.

Prefix	Namespace
service	`http://www.daml.org/services/owl-s/1.2/Service.owl#`
profile	`http://www.daml.org/services/owl-s/1.2/Profile.owl#`
process	`http://www.daml.org/services/owl-s/1.2/Process.owl#`
comp-o	`https://comp-o.github.io/comp-o#`

Table 3: Namespace prefixes used in this paper

4.1 Key concepts of OWL-S

As explained in [9], the description of a service with OWL-S is split in three parts, the service profile presents what the service does, the service grounding how to access it and the service model how to use it. We focus on the service profile as the purpose of our work is on the behavior of CBSs, their internal orchestrations and their interfaces.

A *service:ServiceProfile* presents the service's parameters (*process:Input*s and *process:Output*s), the *process:Precondition* and the *process:Result*s (outputs and effects). The profile describes the service as a black-box as the description is dedicated to its contract and not to its behavior with the clients nor its orchestration. To describe the internal orchestration of a service, a *process:CompositeProcess* can be linked to the process resource which is defined in a service profile. A composite process is used to describe the choreography of messages between the client and the service but also to the invocation of others services. As explained in [9], *"any composite process can be considered a tree whose nonterminal nodes are labeled with control constructs, each of which has children specified using components. The leaves of the tree are invocations of other processes, indicated as instances of class process:Perform (an invocation of another service)"*. Based on this definition, we defined in Comp-O a new control construct used to describe the required interfaces of CBSs.

4.2 Comp-O: concepts and properties

An overview of Comp-O is presented in Fig. 4. The ontology defines three new concepts, and one object property.

ComponentBasedService is the first and main concept: it is a service that can have no or several *RequiredPerform* (required interface) in its process, and that is not operational until all its *RequiredPerform* are replaced with an actual perform referencing another process.

A *Required perform* is a sub concept of the Perform control construct: it describes a required interface. It references a service interface through the *requiredPerformContract* predicate.

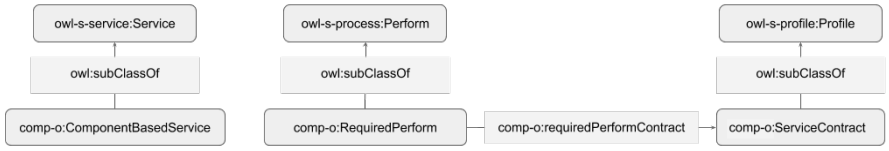

Fig. 4: Architecture of Comp-O

The third concept is the *Service contract*: it is a *ServiceProfile* that does not specify an implementation through the *has_process* predicate. Practically, this concept is used to define the types of the inputs and outputs and the pre/post-conditions specified by a required interface.

Finally, the *requiredPerformContract* predicate is used to link a *RequiredPerform* with the *ServiceContract* it requires.

4.3 Use case and instantiation

This section contains the descriptions of the CBSs presented in 2.2. We focus on the most original and key services that highlight the different uses of Comp-O.

A CBS with no required interface can be described as a Web service. Therefore, the *TextPrinter* service description does not need to use Comp-O at all but can rely on OWL-S only. Obviously, CBSs described with Comp-O can still be bound with traditional OWL-S services like *TextPrinter*.

As explained in 4.2, a required interface of a CBS is described with the *comp-o:RequiredPerform* concept. This concept is a special *Perform* that does not reference a concrete service but a service contract specifying the type of the inputs and outputs, the preconditions and the post-conditions. Therefore, to describe the *VoiceToTextConverter*, instead of referencing another service with a *Perform* as presented in Listing 1.1, we can now use the *comp-o:RequiredPerform* as shown in Listing 1.2.

```
:voice-to-text-converter-perform
    rdf:type        process:Perform ;
    process:process :the-other-process;
```

Listing 1.1: Invocation of another process with OWL-S

```
:voice-to-text-converter-req-interface
    rdf:type                        comp-o:RequiredPerform ;
    comp-o:requiredPerformContract :text-input-contract ;
    process:hasDataFrom      # ...

:text-input-contract
    rdf:type comp-o:ServiceContract ;
    profile:hasInput [
        rdf:type              process:Input ;
        process:parameterType "[...]#Text" .
    ] ;
```

Listing 1.2: Comp-O description of the VoiceToTextConverter required interface

Also, to describe a service with several required interfaces, the mechanism is the same as the one used to describe VoiceToTextConverter. A process can contain an unlimited number of *comp-o:RequiredPerform*.

5 Using Comp-O

5.1 Assisted building of composite services

To assist the developer, we propose a multi-step approach synthesized in Fig. 5.

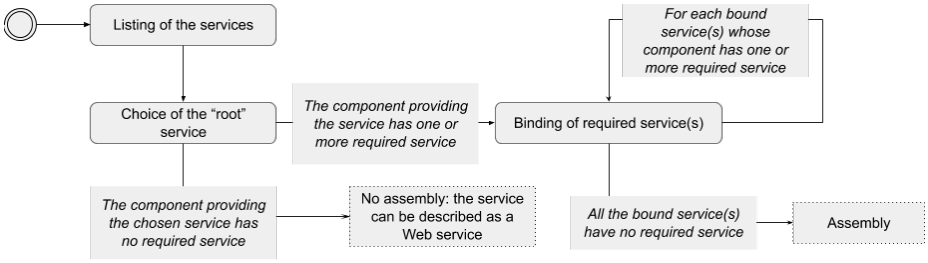

Fig. 5: Building of a Comp-O assembly

In a first step, a list of the available CBSs is presented. To do so, all that is needed is to retrieve the set of resources typed by the *service:Service* class. Then, the service developer must choose the "root" service, i.e., the service to implement. Comp-O helps to determine whether the component that provides the chosen service has any required interface. A component has a required interface if one of the control constructs of the process of the service it provides is a *comp-o:RequiredPerform*. This property can be comprehensively checked considering the OWL-S control constructs using the SPARQL request of Listing 1.3.

```
ASK {
    <service> service:presents/profile:has_process/process:composedOf/(
        process:then|process:else|process:whileProcess|process:untilProcess|
        process:components)*/(owl-list:rest*)/owl-list:first+ ?instruction .

    ?instruction a comp-o:RequiredPerform
}
```

Listing 1.3: SPARQL request to determine whether a service requires to be bound

If the component providing the chosen service does not require any service, it can be described as an OWL-S service, whose description is available and publishable as it is. Contrariwise, if the component has one or more required service, the latter must be bound to external CBSs.

If so, to ease the binding decisions, it is possible to determine if a provided service is compatible with a required one, i.e., if the two services match. This requires to check if the types of the inputs and outputs, the preconditions and the post-conditions match. The strategy used to determine whether there is a match depends on the application domain; it is not specified in our solution but several proposals have been made (see e.g., [2], [12] and [13]).

Finally, when a required service is bound and if the provider also has one or more required services, this step must be repeated for these services until the assembly is closed, i.e., all the required interfaces in the assembly are bound. At this point, an assembly is available and its description can be generated.

5.2 Automatic generation of Comp-O composite service descriptions

We propose an algorithm that implements the generation of a composite service description from an assembly.

The first step consists in replacing every *comp-o:RequiredPerform* by a *process:Perform* referencing the process associated in the assembly, using the *process:process* predicate instead of referencing a *comp-o:ServiceContract* via the *comp-o:requiredPerformContract* predicate. The process of a CBS references as variables the inputs and outputs of a *comp-o:ServiceContract* it requires. For each service, the second step is therefore to replace the references to these variables by references to the equivalent variable of the associated service. This step can be easily accomplished by processing all the *process:fromProcess* predicates having as object a resource of the type *comp-o:ServiceContract*. After these steps, all the CBSs are now described as services with OWL-S since their required interfaces are bound with other services.

5.3 Proof of concept

To ensure and show that the solution works, we have developed a proof of concept (POC) that implements it. It is available online at `https://github.com/comp-o/comp-o-poc`. It proposes a command line interface that helps the user to build the composition plan and outputs the OWL-S description of the assembly. The POC has been used to test the approach against twelve key CBSs chosen for their representativity of the recurrent topologies encountered in component-based software engineering. The description of these services also are available online and are not described in this paper due to space limitation.

6 Conclusion and perspectives

This paper has introduced Comp-O, an extension of OWL-S for CBSs, which are services provided by software components. Comp-O has been developed following the principles of the Neon methodology. One of them is the reuse of ontologies that partially meet the requirements.

As OWL-S is the most compliant with our requirements, we have proposed to extend it: Comp-O supports the description of required services and a combination of descriptions in order to automatically generate the description of composite services. Beyond the semantic description of services for publication purposes and to facilitate their discovery, Comp-O helps the developer: at design time, based on Comp-O, the matching between required and provided interfaces can be controlled and the services (so, the components) that are available for

the composition may be proposed. In addition, supplying the description of the composite services under construction gives the engineer useful feedback. Using a proof-of-concept prototype, we have demonstrated the ability to assist the service developer and to automatically generate composite descriptions from component unit descriptions that have required services.

Now, we plan to use Comp-O in an ongoing project carried out in our team, which aims to make user-oriented services emerge at runtime in ambient environments. There, an intelligent engine builds on the fly composite services from software components present at the time in the environment, without having been required by the user. As a consequence, composite services that emerge must be described to inform the user who can accept, modify or reject them. Then, a user-intelligible description is required for a sound understanding of the service, that could be computed from the Comp-O generated description.

References

1. I. Sommerville. Component-based software engineering. In *Software Engineering*, chapter 16, pages 464–489. Pearson Education, 10^{th} edition, 2016.
2. M. Klusch, P. Kapahnke, S. Schulte, F. Lecue, and A. Bernstein. Semantic Web Service Search: a Brief Survey. *KI-Künstliche Intelligenz*, 30(2):139–147, 2016.
3. K. Kurniawan, F.J. Ekaputra, and P.R. Aryan. Semantic Service Description and Compositions: A Systematic Literature Review. In *ICICoS*, pages 1–6, 2018.
4. M. C. Suárez-Figueroa, A. Gómez-Pérez, and M. Fernández-López. *The NeOn Methodology for Ontology Engineering*, pages 9–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
5. M. Koussaifi. User-oriented Description of Emerging Services in Ambient Systems. In *Int. Conf. on Service-Oriented Computing, PhD Symposium (ICSOC 2019)*, number 12019 in LNCS. Springer, 2019.
6. L. Daniele, F. den Hartog, and J. Roes. Created in close interaction with the industry: The smart appliances reference (saref) ontology. In Roberta Cuel and Robert Young, editors, *Formal Ontologies Meet Industry*, 2015.
7. A. Haller, K. Janowicz, S. Cox, M. Lefrançois, K. Taylor, D. Phuoc, J. Lieberman, Raúl García C., R. Atkinson, and C. Stadler. The SOSA/SSN Ontology: A Joint W3C and OGC Standard Specifying the Semantics of Sensors, Observations, Actuation, and Sampling. *Semantic Web*, 2018.
8. MSM: Minimal Service Model (LOV), 2017. https://lov.linkeddata.es/dataset/lov/vocabs/msm.
9. OWL-S: Semantic Markup for Web Services, 2004. https://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/.
10. J. de Bruijn, H. Lausen, A. Polleres, and D. Fensel. The web service modeling language wsml: An overview. In York Sure and John Domingue, editors, *The Semantic Web: Research and Applications*, pages 590–604, 2006.
11. J. Kopeckỳ, K. Gomadam, and T. Vitvar. hrests: An html microformat for describing restful web services. In *2008 IEEE/WIC/ACM*, volume 1, pages 619–625. IEEE, 2008.
12. M. Klusch, B. Fries, and K. Sycara. Owls-mx: A hybrid semantic web service matchmaker for owl-s services. *Journal of Web Semantics*, 7(2):121 – 133, 2009.
13. G. Fenza, V. Loia, and S. Senatore. A hybrid approach to semantic web services matchmaking. *Int. J. Approx. Reason.*, 48:808–828, 2008.