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

G. Alary, N. Hernandez, J.-P. Arcangeli, S. Trouilhet, J.-M. Bruel

Institut de Recherche en Informatique de Toulouse
University of Toulouse, France

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.

State of the art shows that considering ontologies when describing services improves their discoverability and invocability. However, existing solutions are limited to the description and composition of provided services.

In this paper, we consider ontologies to describe component-based services implemented by component assemblies. First, existing service ontologies are studied and compared to the requirements that an ontology for component-based services must satisfy. Then follows a presentation of an extension of OWL-S called Comp-O to describe component-based services. Finally, through a proof-of-concept, we demonstrate how the developer is assisted when assembling components to build a composite service and how composite service descriptions are automatically generated by combining component unit 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 [1,2]. In order to make a component fully operational, i.e., actually provide its services, each of its required services must be realized by (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 which behavior depends on the components that are involved in the assembly.

Building services by means of software components promotes flexibility: the behavior of a service can be modified by replacing a component by another one in the assembly.

To improve discoverability by third parties, *component-based services* 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. Therefore, it must be synthesized from the semantics of the components that compose the assembly.

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 component-based services 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.

State of the art shows that considering ontologies when describing services improves their discoverability [3] and their composition [3][4]. Several ontologies and approaches exploiting them have thus been proposed. However, existing solutions mainly consider Web services and are not suited for component-based services 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 component-based services and we demonstrate how the added semantics can be handled to automatically build composite service descriptions.

The remainder of this paper is organized as follows. Section 2 briefly introduces software components, component-based development, and component-based services, then the characteristics of component-based services to describe are exposed, and the related work is analyzed. 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 component-based services

Component-based software engineering is a paradigm that emphasizes composability and reuse of software components. *Software components* [1,2] are loosely

coupled self-contained runtime units that *provide* services specified by interfaces. To operate, i.e., actually provide their services, they may *require* external services. Fig. 1 shows the UML representation of a software component called VoiceToTextConverter, where the provided services (here, the VoiceProcess service) are pictured by a bullet and the required services (here, the TextProcess service) 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.

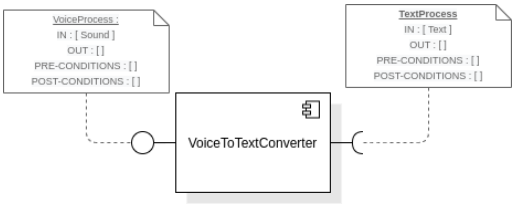

Fig. 1: UML representation of the VoiceToTextConverter component

Flexibility is one of the main advantages of component-based development. Components are versatile and reusable in different contexts. In an assembly, a component is substitutable: it 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* a service that is provided by a software component. If the latter requires external services, the component-based service is implemented by an assembly, and its actual semantics depends on the components that are involved in the assembly.

2.2 Illustrative examples

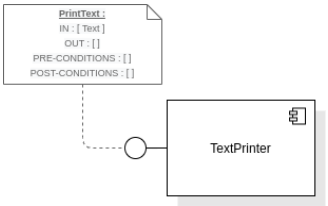

Fig. 2: The PrintText component-based service

Fig. 2 represents the TextPrinter component that provides the component-based service called PrintText. PrintText takes a Text as the only input: when invoked, the input text is printed by a printer and there is no result in return. Like a Web service, PrintText is ready to use since TextPrinter does not need to be bound to any service as it 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 component provided service that takes a text as input, e.g., PrintText of TextPrinter (assuming that PrintText matches TextProcess). Fig. 3 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.

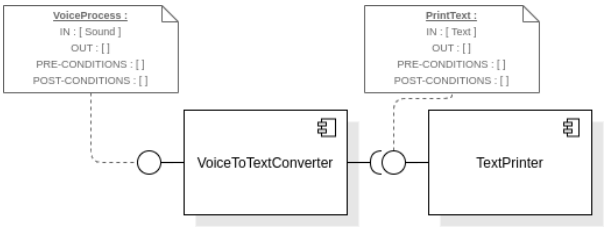

Fig. 3: Implementation of the VoiceProcess composite service

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. 4: when invoked, the TextTransformer component demands the translation in French of the input text then requires the PrintText service.

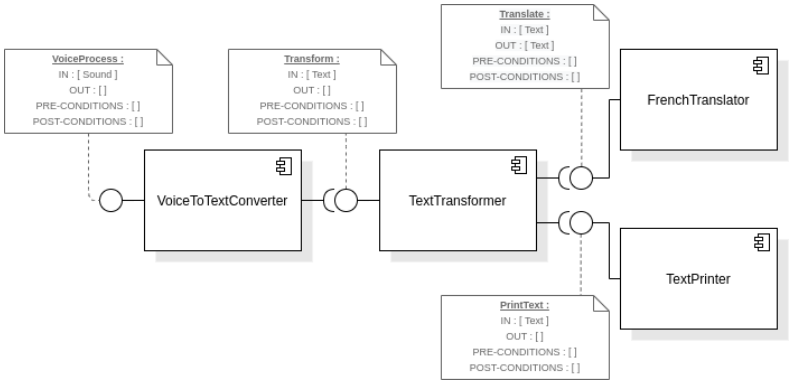

Fig. 4: Another implementation of the VoiceProcess composite service

2.3 Issues

In the previous section, two implementations of the VoiceProcess composite service have been presented in Fig. 3 and Fig. 4: 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 component-based service depends on the components that compose the implementing assembly and what these components actually do. Thus, to determine this nature, it is necessary to introspect 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 component-based services with interfaces only, i.e., as black boxes, is not enough. Interfaces support matching but do not explicit the behavior. 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, component-based services must be described semantically. The problem is to build the semantic description of a component-based service 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. The next section presents existing works related to these issues.

2.4 Related work

Descriptions including a semantic level help considering all potential characteristics of a service and to state them in a way that improves interoperability as the added semantic is meant to be understandable by humans and machines.

Semantic Web services are Web services whose functional and non-functional semantics are described with ontologies [5]. Functional semantics of the service is dedicated to describing its input and output parameters, as well as logic-based description of preconditions and effects. Specific parameters can also be added such as service provenance and quality of service. The description may also include the process model expliciting how the service internally works in terms of the relationship between data and control flow for its subservice interactions.

To represent such descriptions, several description languages differing in their level of formalization have been proposed. OWL-S (Web Ontology Language for Web Services, the successor of DAML-S) [6], WSML (Web Service Modeling Language) [7], USDL (Unified Service Description Language) [8], Linked USDL [9], as well as the microformat hRESTS [10] are languages that can be used to define Web services descriptions and include in these descriptions entities of OWL ontologies. For our purposes, we are mainly concerned about general descriptions of services and thus leave apart trading aspects covered by vocabularies, such as USDL and Linked USDL. More recently ontologies such as SAREF [11],

SOSA/SSN [12] covering services aspects have been proposed in the IoT domain. In the same vein, MSM [13] is a common model proposed to support the publication and discovery over Web APIs, Web services, and sensors.

Based on these different languages, several service composition approaches have been proposed. They rely on automated logic-based composition planning or adaptive and hybrid semantic selection mechanisms exploiting strict and approximated relations [4].

However, all the previously cited vocabularies and composition approaches are dedicated to Web services and do not consider the required component-based service interface.

For component-based service, the composition requires finding and binding the required and provided interfaces. Predefined assembly templates are usually used for this matching. For example, [14] worked on a three-layer architecture for automatic and dynamic software component composition. It is the second layer, called "Change Management", that uses pre-calculated plans to schedule and automate a composition. In [15], the composition of software components is made by an assembling engine that accepts four inputs: a set of components, an application description, an assembling policy and an execution context. This solution requires a specific manually generated description of the desired architecture for the "application description" input. The application is statically defined at design time. In [16], a constraint solving problem is created and resolved to determine the adapted composition. The task of the developer is to define functional objectives on component ports in order to later identify constraints. In [17], the authors propose a semantic enhancement of software components with their properties and functionality to support matching.

[18] aim to build descriptions of emergent applications dynamically. Our objective is not only to semantically extend the description possibilities but also to be able to automatically generate these descriptions in order to reuse them.

In this paper, we propose to take advantage of existing works in the field of semantic Web services in order to facilitate the description and composition of component-based services.

3 Requirements and comparison with existing ontologies

The development of the requirements of our ontology is compliant with the NeOn methodology [19]. Before developing it, 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. We then have used the collection of competency questions to compare and evaluate existing ontologies.

3.1 Purpose and scope

The motivation and final goal of this ontology is to offer a way to describe component-based services, 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 component-based services, 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 component-based services or composite services that will be invokable and bindable by clients. 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 [2,20] 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 represents 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 S1 ?	(Boolean, Int); (String);
CQ3	What are the types of the outputs of S1 ?	(ON/OFF Command, Int); (ON/OFF State); ()
CQ4	What are the preconditions of S1 ?	(cond1; cond2); (cond1); ()
CQ5	What are the post-condtions of 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 **CQ6**, the notion of service binding is only relevant for component-based services as it means that the service S1 invokes another service through one of its required interfaces.

CQ9 is capital 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 in 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 developed in 3.2 to compare and analyze which ontologies satisfy which part of the requirements. We compared six ontologies relevant to our requirements: SAREF [11], SOSA/SSN [12], MSM [13], OWL-S [6] (formerly DAML-S), WSML [7] and HRests [10]. Comp-O is also added to show that it covers all the competency questions.

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	SAREF	SSN/SOSA	MSM	OWL-S	WSML	HRests	Comp-O
CQ1	**	**	**	**	**	**	**
CQ2	**	*	**	**	**	**	**
CQ3	**	*	**	**	**	**	**
CQ4	-	-	-	**	**	-	**
CQ5	-	-	-	**	**	-	**
CQ6	-	-	-	**	**	-	**
CQ7	-	*	-	**	-	-	**
CQ8	-	*	-	**	-	-	**
CQ9	-	-	-	**	-	-	**
CQ10	-	-	-	-	-	-	**
CQ11	-	-	-	-	-	-	**
CQ12	-	-	-	-	-	-	**
CQ13	-	-	-	-	-	-	**
CQ14	-	-	-	-	-	-	**
CQ15	-	-	-	-	-	-	**
CQ16	-	*	-	*	-	-	**

Table 2: Comparison between the competency questions and the ontologies

All the studied ontologies totally 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.

OWL-S totally satisfies the questions **CQ4**, **CQ5**, **CQ7**, **CQ8** and **CQ9**. The preconditions and post-conditions can be described in the profile of the services. The invocations of other services and the internal orchestrations are described in the services' process. Finally, the service offered can be semantically described using the pre-conditions and post-conditions. However, WSML only covers the questions **CQ4**, **CQ5** and **CQ7** as services described with WSML are described as black boxes, without information about their internal workings. Though, 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* property. 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.

Finally, SOSA/SSN and OWL-S partially cover the question **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 semantics offered by SOSA/SSN and OWL-S are slightly similar to the one required by **CQ16** but the binding of interfaces is a mechanism specific

to component-based services where SOSA/SSN and OWL-S are used to describe Web services.

Based on the comparison with the competency questions, we conclude that OWL-S is the ontology that covers the best our requirements. This observation motivated us to develop Comp-O as an extension for OWL-S in order to reuse most of the ontology and its semantics.

4 Comp-O, an OWL-S extension for component-based services

Comp-O, the ontology we propose, is a minimal ontology extending OWL-S that helps to efficiently describe component-based services. The ontology is available at `https://gregoryalary.github.io/comp-o`. All the namespaces used in this paper are given in Table 3. In this section, we present the key concepts of OWL-S, an overview of Comp-O, its concepts and predicates and an example of a component-based service 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://gregoryalary.github.io/comp-o##`

Table 3: Namespace prefixes used in this paper

4.1 Key concepts of OWL-S

As explained in [6], 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 component-based services, theirs internal orchestrations and their interfaces.

A *service:ServiceProfile* presents the parameter of the service (the *process:Input*s and the *process:Output*s), the *process:Precondition* and the *process:Result*s (the outputs and the 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 and 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 [6], *"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 component-based services. This is detailed in the next sections.

4.2 Comp-O

An overview of Comp-O is presented in Fig. 5. The ontology defines three new concepts, and one object property.

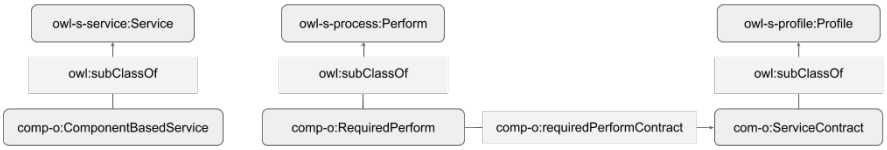

Fig. 5: Architecture of Comp-O

Basically, this ontology defines a new class specialisating *process:Perform*, *RequiredPerform* used to describe the required interfaces of a component-based service. This control construct, instead of referencing another process like the OWL-S *process:Perform* control construct, references a service contract represented by a service profile that does not reference any process (a black-box).

4.3 Concept and predicates

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

$$\text{comp-o:ComponentService} \sqsubseteq \text{service:Service} \tag{1}$$

Required perform A *Required perform* is a sub concept of the Perform control construct. A required perform describes a required interface and hence cannot reference a service through the *process:process* predicate but has to reference a service interface through a the *requiredPerformContract* predicate.

$$\text{comp-o:RequiredPerform} \sqsubseteq \text{process:Perform} \tag{2}$$

Service contract The last defined concept in this ontology is the concept of *Service contract*. A *ServiceContract* 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, it describes a service contract.

$$\text{comp-o:ServiceContract} \sqsubseteq \text{profile:Profile} \tag{3}$$

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

$$\geqslant 1\text{comp-o:requiredPerformContract} \sqsubseteq \text{comp-o:RequiredPerform} \quad (4)$$

$$\top \sqsubseteq \forall\text{comp-o:requiredPerformContract} \cdot \text{comp-o:ServiceContract} \quad (5)$$

4.4 Use case and instantiation

In order to further describe the semantic of Comp-O, this section contains the descriptions of the component-based services presented in 2.2. We focus on the most original and key services that highlight the different uses of Comp-O.

A component-based service without any required interface can be described as a Web service. Therefore, to describe the *TextPrinter* service, one does not need to use Comp-O at all but can rely solely on OWL-S. Obviously, component-based services describes with Comp-O can still be bound with traditional OWL-S services like *TextPrinter*.

Describing the required interface of a component-based service As explained in the section 4.3, to describe the required interface of a component-based service with the Comp-O ontology, one must use the *compo-owl-s:RequiredPerform* concept. The *compo-owl-s:RequiredPerform* concept is a special *Perform* that does not reference a concrete service but a service contract specifying the type of the inputs and outputs and the preconditions and 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 *compo-owl-s: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: Description of the VoiceToTextConverter required interface with Comp-O

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 *compo-owl-s:RequiredPerform*.

5 Using Comp-O to build composite services and automatically generate theirs descriptions

This section focuses on how, using Comp-O, the developer can be assisted when assembling components to build a composite service and how service descriptions can be automatically generated by combining component unit descriptions. Our solution has been implemented and tested against several key use cases, ranging from very simple to complex assemblies.

5.1 Assisted building of composite services

To assist the developer in the building of Comp-O composite services, we propose a multi-step approach synthesized in Fig. 6.

In a first step, a list of the available component-based services 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. As explained in Section 4.3, a component has a required interface if one of the control construct of the process of the service it provides is a *comp-o:RequiredPerform*, thus this property can be comprehensively checked considering all the control constructs defined in OWL-S 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 to another service to run

If the component providing the chosen service does not require any service, it can be described as an OWL-S service, which 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 component-based services.

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 propositions have been made (see e.g., [3], [21] and [22]).

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 the description of the composite service can be generated.

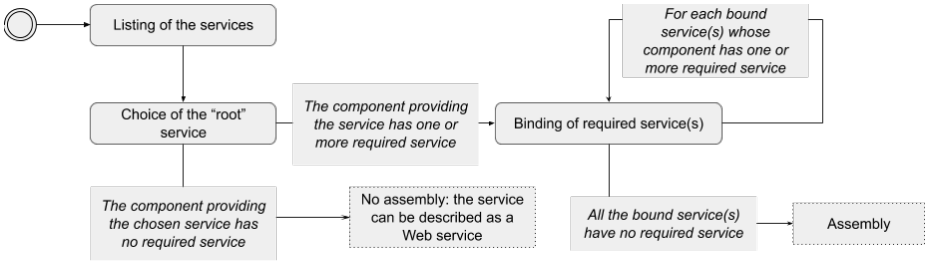

Fig. 6: Building of a Comp-O assembly

5.2 Automatic generation of Comp-O composite service descriptions

As explained in Section 1, semantic description of services eases discovery and composability. An important feature of Comp-O is its use for the automatic generation of 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 component-based service can reference 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*.

These two steps being completed, all the component-based services are now described as services with OWL-S given that their required interfaces are bound with other services.

5.3 Proof of concept

To ensure and show that the solution for building composite services and generating their descriptions based on Comp-O works, we have developed a proof of concept (POC) that implements it. It is available online at `https://github.com/gregoryalary/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. This POC has been used to test the proposed approach against twelve key component-based services that have been 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 component-based services, 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 of component-based services: 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 that are 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 automatically generated description.

7 Acknowledgment

This work is part of the AILP (Assistance InteLligente et proactive en environnement Professionnel) project, which is supported by the French region Occitanie and the operational program FEDER-FSE Midi-Pyrénées et Garonne.

References

1. OMG. *Unified Modeling Language (OMG UML) Version 2.5.1*, chapter 11.6.3.1 Components semantics. 2017.
2. I. Sommerville. Component-based software engineering. In *Software Engineering*, chapter 16, pages 464–489. Pearson Education, 10^{th} edition, 2016.
3. 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.
4. K. Kurniawan, F.J. Ekaputra, and P.R. Aryan. Semantic Service Description and Compositions: A Systematic Literature Review. In *2nd Int. Conf. on Informatics and Computational Sciences (ICICoS)*, pages 1–6, 2018.

5. M. Klusch. Semantic Web Service Description. In *CASCOM: Intelligent service coordination in the semantic Web*, pages 31–57. Springer, 2008.
6. OWL-S: Semantic Markup for Web Services, 2004. https://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/.
7. J. De Bruijn, H. Lausen, A. Polleres, and D. Fensel. The web service modeling language wsml: an overview. In *Europ. Semantic Web Conf.*, pages 590–604. Springer, 2006.
8. Srividya Kona, Ajay Bansal, Luke Simon, Ajay Mallya, and Gopal Gupta. Usdl: a service-semantics description language for automatic service discovery and composition. *International Journal of Web Services Research (IJWSR)*, 6(1):20–48, 2009.
9. Carlos Pedrinaci, Jorge Cardoso, and Torsten Leidig. Linked usdl: a vocabulary for web-scale service trading. In *European Semantic Web Conference*, pages 68–82. Springer, 2014.
10. Jacek Kopeckỳ, Karthik Gomadam, and Tomas Vitvar. hrests: An html microformat for describing restful web services. In *2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, volume 1, pages 619–625. IEEE, 2008.
11. Jasper Roes Laura Daniele, Frank den Hartog. Created in Close Interaction with the Industry: The Smart Appliances REFerence (SAREF) Ontology. 2015.
12. Simon J D Cox Maxime Lefrançois Kerry Taylor Danh Le Phuoc Joshua Lieberman Raúl García-Castro Rob Atkinson Armin Haller, Krzysztof Janowicz and Claus Stadler. The SOSA/SSN Ontology: A Joint W3C and OGC Standard Specifying the Semantics of Sensors, Observations, Actuation, and Sampling. 2018.
13. MSM: Minimal Service Model (LOV), 2017.
14. Daniel Sykes, Jeff Magee, and Jeff Kramer. FlashMob: Distributed Adaptive Self-Assembly. In *Proc. of the 6th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*, pages 100–109, 2011.
15. Guillaume Grondin, Noury Bouraqadi, and Laurent Vercouter. MaDcAr: An Abstract Model for Dynamic and Automatic (Re-)Assembling of Component-Based Applications. In *Component-Based Software Engineering*, number 4063 in LNCS, pages 360–367. Springer-Verlag, 2006.
16. Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Automated and Unanticipated Flexible Component Substitution. In *Proc. of 10th Int. Symp. on Component-Based Software Engineering*, 2007.
17. J. M. Gomez, S. Han, I. Toma, B. Sapkota, and A. Garcia-Crespo. A Semantically-enhanced Component-based Architecture for Software Composition. In *Int. Multi-Conf. on Computing in the Global Information Technology (ICCGI'06)*, pages 43–47, Aug 2006.
18. Maroun Koussaïfi, Sylvie Trouilhet, Jean-Paul Arcangeli, and Jean-Michel Bruel. Ambient intelligence users in the loop: Towards a model-driven approach. In *Software Technologies: Applications and Foundations*, pages 558–572. Springer, 2018.
19. María del Carmen Suárez de Figueroa Baonza. *NeOn methodology for building ontology networks: specification, sheduling and reuse.* PhD thesis, 2010.
20. 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.
21. Matthias Klusch, Benedikt Friesb, and Katia Sycara. Owls-mx: A hybrid semantic web service matchmaker for owl-s services. 2009.
22. G. Fenza, V. Loia, and S. Senatore. A hybrid approach to semantic web services matchmaking. *Int. J. Approx. Reason.*, 48:808–828, 2008.