An Ontology to Semantically Declare and Describe Functions

Ben De Meester(✉), Anastasia Dimou, Ruben Verborgh, and Erik Mannens

Data Science Lab – iMinds, Ghent University, Ghent, Belgium
{ben.demeester,anastasia.dimou,ruben.verborgh,erik.mannens}@ugent.be

Abstract. Applications built on top of the Semantic Web are emerging as a novel solution in different areas, such as decision making and route planning. However, to connect results of these solutions – i.e., the semantically annotated data – with real-world applications, this semantic data needs to be connected to actionable events. A lot of work has been done (both semantically as non-semantically) to describe and define Web services, but there is still a gap on a more abstract level, i.e., describing interfaces independent of the technology used. In this paper, we present a data model, specification, and ontology to semantically declare and describe functions independently of the used technology. This way, we can declare and use actionable events in semantic applications, without restricting ourselves to programming language-dependent implementations. The ontology allows for extensions, and is proposed as a possible solution for semantic applications in various domains.

Keywords: Application · Function · Ontology · Semantic web

1 Introduction

Semantic applications are emerging as a solution to data-driven problems. For instance, the ORCA project aims at automatically assigning nurses to patient calls in a hospital based on their context [1]. Linked Connections define a way to publish raw transit data, to be used for intermodal route planning [3]. In projects like the aforementioned, it is common that the Semantic Web is not solely used as a means to publish data, but also as a catalyst to execute other actions, e.g., calling a real-world nurse, or executing a route planning algorithm. Usually, this implies querying the RDF results for possible actions, parsing the data using custom parsing rules, and executing the actual function. E.g., in the ORCA project, a semantic reasoning system derives the most suitable nurse. The resulting turtle data is queried, resulting in the triple `ex:call1 ex:assign ex:nurseA`. How to convert this triple to the execution of the function `callNurse(nurseA, call1)` is not semantically defined, but is defined ad hoc per use case.

H. Sack et al. (Eds.): ESWC 2016 Satellite Events, LNCS 9989, pp. 46–49, 2016.
DOI: 10.1007/978-3-319-47602-5_10

There exist many specifications handling Web services, both non-semantically (e.g., WSDL and WADL) and semantically (e.g., OWL-S and Hydra)[1]. These specifications target different facets (e.g., HTTP-based vs SOAP-based access, defining RESTful APIs, etc.), but have in common that they *define Web services*. Thus, they clearly specify, e.g., which HTTP method to invoke with which parameter to correctly call the Web service. The big drawback of these specifications is thus that they are very coupled with the technology stack. However, not all actions can be executed using Web APIs, either because of performance or practicality reasons. For example, the nurse call system is a near real-time system, which implies that unnecessary HTTP connections should be avoided.

In this paper, we present a general vocabulary as a data model, specification, and ontology to semantically declare and describe functions. Instead of *defining* technology-specifics, i.e., hard-coding executable actions, the functions are *described* independently of the technology that implements them. Complementary to the current specifications that describe *how* to execute a certain service, this vocabulary describes *what* a functions does. By semantically defining these functions, we provide a uniform and unambiguous solution, and thus, we close the gap between semantic data and any real-world action, enabling semantic applications to be used in real-world scenarios.

2 The Function Ontology

The Function model allows users to declare that a certain function exists, and associates this function with certain problems and the algorithms it implements. Furthermore, it allows for the description of *executions* of functions. Execution descriptions of functions assign values to the input parameters, e.g., `function int sum(int a, int b)` is a description of a function, whilst `sum(2, 4)` is a description of the execution of the `sum` function. The Function model does not describe the internals of a function, as this depends on the used technology. E.g., the aforementioned `sum` function can be interpreted for implementation in JavaScript as `return a + b`, in PHP as `return $a + $b;`, using a Web service, or via another technology.

The Function Ontology[2] consists of a set of base classes. A problem, algorithm or function can be defined as an instance of those base classes. As new problems and algorithmic solution arise daily, it would not be beneficial to include all possible problems and algorithms in the Function Ontology. Problems, functions, and algorithms can be classified using the SKOS model [4]. For example, a `sumProblem` can be classified as `ex:sumProblem skos:broader ex:mathProblem`. These classifications can be reused across domains, independent of the used technologies.

[1] See https://www.w3.org/TR/wsdl20/, https://www.w3.org/Submission/wadl/, https://www.w3.org/Submission/OWL-S/, and http://hydra-cg.com/spec/latest/core/, respectively.

[2] Published on http://semweb.mmlab.be/ns/function, with accompanying specification at http://users.ugent.be/~bjdmeest/function/.

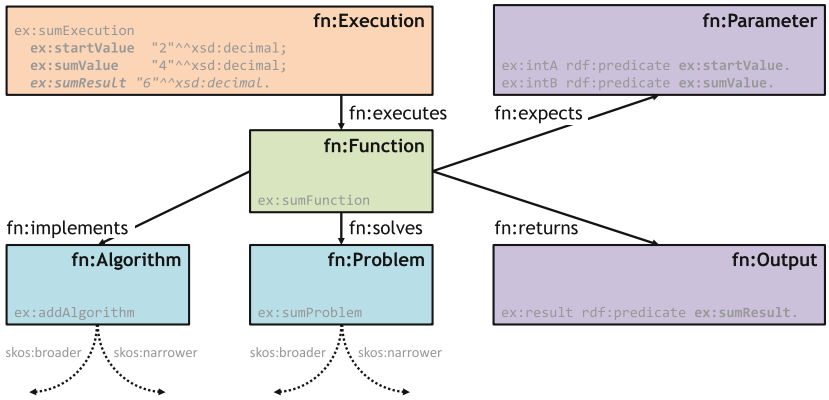

Fig. 1. The Function ontology, with an example of how reification is used to connect named parameters and output values to the function execution.

The Function Ontology (see Fig. 1) consists of the following classes:

- *Function* The declared function (e.g., `function sum`)
- *Problem* A problem that a function solves, e.g., adding two numbers.
- *Algorithm* A declaration of an algorithm. We separate the algorithm concept, as there are no one-to-one mappings between problems, functions, and algorithms [2].
- *Parameter* A parameter of a function (e.g., `function sum` has two parameters, `a` and `b`). `rdfs:range` can be used to describe the type of that parameter, and reification (as opposed to, e.g., using RDF lists) is used to create named connections between a function and its parameters (cfr. the parameter and execution declaration in Fig. 1).
- *Output* An output of a function, e.g., `function sum(a, b)` has `sumResult` as output. Typing and reification paradigms are similar as with parameters.
- *Execution* An execution assigns actual input values to function parameters, and holds the output values. E.g., `sum(2, 3)` is an execution of `int function sum(a,b)`.

3 Example and Conclusions

These base classes do not define any specific problem, function or algorithm, but *allow to declare and describe* any specific problem, function, or algorithm. For example, if John Doe needs a description of a *sum* function – given it does not already exist – he can publish his own descriptions as shown in Listing 1. Software systems supporting this description are thus able to compute the execution of that function. The deployment/implementation of the described functions is independent of the programming language and the access interface.

```
johndoe:sumProblem   a fn:Problem  ; skos:broader johndoe:mathProblem.
johndoe:sumAlgorithm a fn:Algorithm.

johndoe:sumFunction  a fn:Function;
    dcterms:title "The sum function"^^xsd:string;
    dcterms:description "This function can do the sum of two integers."^^xsd:string;
    fn:solves        johndoe:sumProblem;
    fn:implements    johndoe:sumAlgorithm;
    fn:expects  [ rdf:predicate johndoe:startValue; fn:required "true"^^xsd:boolean ];
    fn:expects  [ rdf:predicate johndoe:sumValue  ; fn:required "true"^^xsd:boolean ];
    fn:returns  [ rdf:predicate johndoe:sumResult ; fn:required "true"^^xsd:boolean ].

johndoe:startValue rdfs:range xsd:integer.
johndoe:sumValue   rdfs:range xsd:integer.
johndoe:sumResult  rdfs:range xsd:integer.

johndoe:sumExecution   a fn:Execution;
    fn:executes        ex:sumFunction;
    johndoe:startValue "2"^^xsd:integer;
    johndoe:sumValue   "4"^^xsd:integer.
```

Listing 1. Sum function declaration

This example shows how the Function Ontology allows for semantic systems to connect to non-semantic processing functions. By remaining technology-agnostic, both small and large-scale RDF processing actions can be defined, without solely depending on Web services as execution platform. As such, it will be evaluated in the frame of, among others, ORCA, Linked Connections[3], and the COMBUST project[4]. Future work includes the creation of an upper level reference function model and a specification of the technical integration process.

References

1. Arndt, D., et al.: Ontology reasoning using rules in an eHealth context. In: Bassiliades, N., Gottlob, G., Sadri, F., Paschke, A., Roman, D. (eds.) RuleML 2015. LNCS, vol. 9202, pp. 465–472. Springer, Heidelberg (2015). doi:10.1007/978-3-319-21542-6_31
2. Bratas, C., Maglavera, s.N., Quaresma, P.: A framework to describe problems and algorithms in medical informatics via ontologies. In: 4th European Symposium on Biomedical Engineering, pp. 1–4. University of Patras, Patras, Greece, June 2004. http://www.di.uevora.pt/~pq/papers/patras.pdf
3. Colpaert, P., Llaves, A., Verborgh, R., Corcho, O., Mannens, E., Van de Walle, R.: Intermodal public transit routing using Linked Connections. In: Proceedings of the 14th International Semantic Web Conference: Posters and Demos, October 2015. http://ceur-ws.org/Vol-1486/paper_28.pdf
4. Miles, A., Bechhofer, S.: SKOS simple knowledge organization system reference. In: W3C Recommendation, W3C, August 2009. https://www.w3.org/TR/skos-reference/. Accessed 7 Mar 2016

[3] http://linkedconnections.org/.
[4] http://www.iminds.be/nl/projecten/2015/03/11/combust.