Implementation-independent Function Reuse

Ben De Meester[a], Tom Seymoens[b], Anastasia Dimou[a], Ruben Verborgh[a]

[a] *Ghent University – imec – IDLab, Department of Electronics and Information Systems*
AA Tower, Technologiepark-Zwijnaarde 122, 9052 Ghent, Belgium
[b] *imec – smit, Vrije Universiteit Brussel*
Pleinlaan 9, 1050 Etterbeek, Belgium

Abstract

Functions are essential building blocks of information retrieval and information management. However, efforts implementing these functions are fragmented: one function has multiple implementations, within specific development contexts. This inhibits reuse: metadata of functions and associated implementations need to be found across various search interfaces, and implementation integration requires human interpretation and manual adjustments. An approach is needed, independent of development context and enabling description and exploration of functions and (automatic) instantiation of associated implementations. In this paper, after collecting scenarios and deriving corresponding requirements, we (i) propose an approach that facilitates functions' description, publication, and exploration by modeling and publishing abstract function descriptions and their links to concrete implementations; and (ii) enable implementations' automatic instantiation by exploiting those published descriptions. This way, we can link to existing implementations, and provide a uniform detailed search interface across development contexts. The proposed model (the Function Ontology) and the publication method following the Linked Data principles using standards, are deemed sufficient for this task, and are extensible to new development contexts. The proposed set of tools (the Function Hub and

*Corresponding authors
Email addresses: ben.demeester@ugent.be (Ben De Meester), anastasia.dimou@ugent.be (Anastasia Dimou), ruben.verborgh@ugent.be (Ruben Verborgh)

Function Handler) are shown to fulfill the collected requirements, and the user evaluation proves them being perceived as a valuable asset during software retrieval. Our work thus improves developer experience for function exploration and implementation instantiation.

Keywords: Function, Linked Data, Reuse

1. Introduction

Functions are processes that perform a specific task by associating one or more inputs to an output. They are essential building blocks of information retrieval and information management [1], and of computer science in general. For example, during extraction of birthdates from a semi-structured dataset, normalization of these dates improves further analysis.

However, the development, maintenance, and support efforts of *implementing* these functions are fragmented [2]. Efforts are fragmented across different *development contexts* (i.e., a combination of, among others, programming language, programming paradigm, and architecture), and it is not feasible to consolidate these efforts by limiting all developers to the same development context [3]. On the one hand because implementations are tuned to meet different requirements, on the other hand due to prior investment [2]. Thus, the same function can have multiple implementations, each within a specific development context. For example, a function to normalize dates may have implementations available as a piece of JavaScript source code, as part of a JAVA software package, and within a RESTful Web service.

The *exploration* of functions is fragmented across programming documentation and different search engines (e.g., general search engines vs. online code repositories vs. package managers), inhibiting their discovery and reuse. Search engines are tailored to different contexts, and require a laborious manual search effort to find relevant functions using fuzzy search terms across broadly varying access methods. Users thus need many different search engines returning different result sets in different formats when looking for a relevant function, in-

creasing the chance of using a suboptimal result. A global overview of functions and their descriptions across development contexts is missing.

The *instantiation* of these implementations, i.e., automatic execution or example source code generation, pose a challenge as well. Integration of an implementation requires interpretation of the programming documentation and manual adjustments to adhere to a common predefined interface [3], burdening the developer. No uniform and unambiguous implementation description format is available.

Therefore, an important research challenge is to describe functions and their associated resources (i.e., input, output, mapping to concrete implementation, and instantiation) independent of the development context, and to facilitate exploitation, i.e., enabling exploration of functions and (automatic) instantiation of implementations or generation of source code. Users get an overview of existing functions without being limited to the development context, and integration requires less manual effort.

In this paper, we tackle this challenge by addressing two main problems: (i) facilitating *functions' description, publication, and exploration* by modeling and publishing descriptions of functions and associated resources; and (ii) enabling *implementations' automatic instantiation* by exploiting that published data. Our approach makes the following contributions:

- **A collection of requirements and scenarios** for function exploration and implementation instantiation, based on the state of the art. We use this collection to assess the features and limitations of our approach.
- **An implementation-independent descriptive model** for functions, named the Function Ontology (FnO).
- **A method for publishing function descriptions** as an open semantic knowledge base on the Web. Functions are published using the aforementioned model and made available – together with their associated resources – as Web resources using Linked Data [4] and standardized Web

technologies. This method provides a uniform interface for function descriptions across development contexts.

- **An evaluated set of tools for exploiting function descriptions**, helping users to find and access functions, and integrate and reuse (existing) implementations. This set of tools comprises the Function Hub and the Function Handler, which enable searching, retrieving, and automatically instantiating functions and their associated implementations.

Reuse of metadata increases by adhering to the FAIR principles [5]. These FAIR principles have been devised for scholarly data, with its benefits being shown across domains such as medicine and biology [6]. As such, we can consider applying the FAIR principles to the descriptions of functions and associated implementations to improve reuse.

The remaining paper is organized as follows. Section 2 presents related work. Section 3 describes the main scenarios we want to address, and the requirements derived from these scenarios and the state of the art. Section 4 addresses the first main problem, i.e., it introduces our approach for publishing function descriptions, conform to the requirements. Section 5 addresses the second main problem, i.e., it introduces a set of tools for exploiting function descriptions. Section 6 discusses the conformance of our framework against the requirements, and presents use cases that benefit from this approach, specifically: (i) a generic software package of OpenRefine[1] functions, (ii) an approach for exploiting Docker[2] image metadata, and (iii) inclusion in the DBpedia Extraction Framework [7]. Section 7 describes our user evaluation of these exploitation tools. Finally, we present conclusions and future work in Section 8.

[1] `http://openrefine.org/`

[2] `https://www.docker.com/`

2. Related work

Related work is grouped into the two main problems we aim to address: enabling exploration of functions (Section 2.1), and enabling automatic instantiation of implementations (Section 2.2).

2.1. Function exploration

Function exploration works are grouped with respect to how specific and detailed the available metadata is, related to functions (generic, high-level, or specific). The specificity influences the exploration operators: generic metadata allows search based on fuzzy text search, whereas specific metadata also allows using unambiguous descriptions, finding related implementations, and detailed filtering on, e.g., input data types.

Generic metadata. Generic metadata of functions or implementations is usually automatically generated and indexed by search engines, based on analyzing and indexing source code or textual descriptions of that source code. It is mostly used to provide (fuzzy) text-based search. General search engines such as Google, Bing, and DuckDuckGo[3] can index published code, but are not tailored to functions and implementations. More specific search engines include version control systems such as GitHub[4], or language-specific search engines. The latter includes search engines that analyze the source such as Sourcerer [8], and package managers such as Maven or NPM[5]. Analysis of the source code can provide additional statistics, and analysis of the package manager metadata provides, among others, the dependency graph. However, the metadata is limited, does not provide search engines with detailed filtering operators, and targets a single development context.

[3] `https://www.google.com/`, `https://www.bing.com/`, and `https://duckduckgo.com/`.

[4] `https://github.com/`

[5] `https://maven.apache.org/`, `https://www.npmjs.com/`.

High-level metadata. High-level metadata of functions or implementations includes descriptions of software packages via the Description of a Project (DOAP) format [9], and provenance information about any type of actions using the PROV Ontology (PROV-O) [10]. The P-Plan Ontology [11] describes workflows, i.e., a series of actions, based on the PROV Ontology, and the work of Garijo et al. [12] and others show how having metadata tailored to workflows allows for advanced exploitation of these annotated workflows. Functions and implementations that are semantically related to each other can be linked together using the Simple Knowledge Organization System (SKOS) [13], providing a more connected knowledge graph. High-level metadata allows identifying and relating larger systems, but is not detailed enough to describe individual functions and link them to implementations, and thus to provide means to automatically instantiate implementations.

Specific metadata. Specific metadata about functions are usually coupled to the development context. Programming language-specific descriptions of functions include the Java Modeling Language (JML) [14]; the works of Hoogle [15] and García-Contreras et al. [16], which describe analysis methods to automatically generate detailed descriptions of functional programming languages; and the CodeOntology, which analyzes and describes source code, applied to JAVA libraries [17]. Other related works include OWL-S [18], which allows to describe Web services; and Linked Software Dependencies (LSD) [19], which allows to describe software configurations and dependencies. These works can be used to have more advanced searching capabilities such as filtering on input type, or combining multiple functions automatically. However, they are coupled to a specific development context, limiting their scope: they cannot be used to, e.g., replace a local JAVA library with a remote Web service.

2.2. Implementation instantiation

Implementation instantiation works are grouped with respect to how closely they resemble or depend on procedural programming, from close to distant:

(i) an abstract programming language is used, (ii) the source code is embedded in the description, or (iii) an abstract function description is used to instantiate actual code. These works allow for automation instantion, – i.e., transparent execution or code generation – based on a description. The closer the resemblance with procedural programming, the more effort is needed to reuse existing implementations.

Abstract programming language. Works that provide an abstract programming language, just like procedural programming, allow a detailed specification of the function's task: the steps are defined in lines of source code. Current works have in common that they rely on an existing programming language to compile to. Examples include PL/SQL[6] as procedural language for SQL, LDScript [20] as procedural language for SPARQL (a query language and protocol for querying Linked Data [21]), the work of Reiss et al. [22] that use JML to execute JAVA code, and VOLT: a SPARQL endpoint proxy where functions are described using a custom syntax, and compiled into JavaScript code [23].

Embedded source code. Related works that embed source code in the function's description link abstract descriptions with implementations, but are highly coupled with a programming language. On the one hand, implementations are hardwired in the source code, as is done for extending querying endpoints with additional functions [24][7]. On the other hand, code is embedded as a string in the description. Examples of the latter include R2RML-F that embeds JavaScript code in a description for generating Linked Data data from relational data [25], and procedures to include additional functions in a dataset validation language, either by embedding complex queries [26] or JavaScript code [27].

Abstract functions. Related works that provide fully abstract descriptions of functions do not explicitly link to implementations. Instead, custom software is

[6] `https://www.oracle.com/technetwork/database/features/plsql/index.html`

[7] `https://jena.apache.org/documentation/query/writing_functions.html`

needed to understand and instantiate these function descriptions. The descriptions themselves are no longer coupled with a specific development context, but the software executing them is. Related works are mostly targeted to Web services, as HTTP request use a restricted API (i.e., a limited set of HTTP verbs), and implementation of custom software is limited. REST APIs have been de facto standardized using OpenAPI [28]. Other works include Hydra for semantic descriptions of REST APIs [29], and GraphQL as a means to describe how to query Web APIs[8]. Components.js uses abstract function descriptions of local libraries instead of Web services, written in JavaScript [19].

3. Requirements

In this section we review, summarize, and expand on requirements defined in the literature and relevant use cases. We first present our scenarios in Section 3.1, after which we present our summarized requirements in Section 3.2.

3.1. Scenarios

Our work tackles functions' description, exploration, and (automatic) implementation instantiation. We illustrate in the scenarios below. Identified requirements are mentioned by their identifier as used in Table 1. Furthermore, we align the scenarios's implied publishing requirements to the FAIR principles, strengthening our claim to enable reuse [5].

Scenario 1: Declare. For our first scenario, we look into the DBpedia Extraction Framework [7]: the framework generating the DBpedia knowledge graph largely based on the Wikipedia info boxes' data. Many data transformations are needed to create a knowledge graph of sufficient quality as Wikipedia data is maintained by a large heterogeneous community (e.g., all dates need to be normalized into a standard format). This set of data transformations was hard-coded in the extraction framework, and thus hard to maintain [30]. A fully

[8] `https://graphql.org/`

declarative description of the generation process would improve the generation process' quality, coverage, and sustainability [30]. This requires declarative and machine-processable function descriptions, independent of the use-case and of the implementation [31]. We specify the following user story:

> Alice wants to describe a date normalization function, independent of an implementation (R1,20). Such a date normalization function transforms a string into a valid date (R2,3,17). She can also describe the execution of a date normalization function (R16,19): when using a string such as "23rd of April, 2016" as input, she expects "2016-04-23" as output (R12).

This scenario shows the need for publishing *findable* and *interoperable* functions: describe data processing independent of the implementation, allowing for more sustainable processes.

Scenario 2: Explore. Our second scenario is based on the *software (component) retrieval problem* [32]. Handling such problem is done by combining a *problem space*, and a *solution space*. In the *problem space*, the to be solved task, as understood by a developer, is translated into a query. In the *solution space*, a set of function implementations' descriptions – grouped per implementation – is indexed into a queryable endpoint. When trying to retrieve a relevant piece of software, the developer's query is matched against the descriptions [32]. As a result, the developer finds a relevant software component and knows how to integrate it. However, the query and matching complexity can vary highly: when using a set of keywords, the retrieval process can consist of browsing vendor literature or examining programmer documentation; when using a detailed specification of the needed software component, the query becomes more complex, but retrieval is more robust [33]. We distinguish between these two user stories – one with a simple and one with a complex query – as certain requirements are specific to one user story, e.g., filtering on semantic constraints (R17,18) is specific for a complex query user story.

For *simple queries*, we can look at general function descriptions, such as questions asked on the popular developer website, StackOverflow: many most-voted questions are of the form "How to [description of a function] in [description of a development context]", for example "How to check whether a string contains a substring in JavaScript?"[9]. Such queries contain a function description, a problem description, and a link to a development context description. When we review programmer documentation, we can also remark that functions are categorized, e.g., all "string"-related functions are collected in one category. We specify the following user story:

> Bob wants to explore and evaluate the different existing functions for a given problem domain. Some general keywords are used (R4). Given the results, he estimates which functions solving which sub-problems are prevalent (R2,20), and which development context provides most functions (R7–9), and thus is most mature for the given problem domain.

For *complex queries*, related work shows that filtering on input and output types is important to validate that the found functions are compatible with the existing code base [12, 34]. We specify the following user story:

> Bob's use case requires specific functions, each of which he can accurately describe (R4,5). He knows the requirements concerning the input and output types of the needed functions (R2,3,9,10,17,18). Bob estimates that for a specific function, it will be beneficial to use a Web service, but he would also like to discover existing local libraries in different programming languages (R7,8,11,13).

This scenario shows the need for *findable* and *accessible* functions and associated implementations: understand and evaluate different functions, and collect different implementations of the same function using a standardized format.

[9] `https://stackoverflow.com/q/1789945`, retrieved from `https://stackoverflow.com/questions?tab=Votes`, accessed September 20$^{\text{th}}$, 2019.

Scenario 3: Instantiate. Our third scenario addresses the *reproducibility problem*: without a clear description of how a function was executed, reproducibility is hampered [35]. Related works address this by providing unambiguous descriptions of execution environments [19], as this allows adding of additional knowledge, easily share and refer to instructions, and machine-understandability [36]. However, existing works are implementation-dependent. Similarly for the DBpedia Extraction Framework, unambiguous descriptions of how a function was executed by which implementation allows for a more sustainable and reproducible generation process [30]. We specify following user story:

> Claire tries to find the best implementation of a given function (R3,7–10,18). She is not tied to a specific development context, instead, she has specific metrics (e.g., accuracy and performance), and wants to compare implementations (R12). For any existing implementation, Claire wants to measure and compare the metrics of the execution results (R17,19), without needing to worry about how to retrieve and execute the different implementations (R13–15).

Scenario 4: Update. Finally, to provide for a sustainable solution, it should be feasible to create new and update existing functions, and add implementations [12]. We specify following user story:

> David tries to re-run an unmaintained piece of software. One of the functions implemented in the software depends on a software package that is no longer available. David searches the function dataset for alternative implementations, but no other implementation is available which fits in the current development context. David decides to re-implement the function for his development context. Due to the accurate function descriptions, David provides an implementation that adheres to the existing interface (R13) and re-runs the piece of software (R1,2,4,5). By publishing the mapping between the new implementation and the existing function (R6–11), anyone can be made aware that this new implementation is available.

This scenario shows the need for *interoperable* and *reusable* functions and mapped implementations: having a standardized way to understand the same function across development contexts, and (automatically) instantiate a function's implementation without requiring implicit knowledge.

3.2. Requirements summary

The scenarios show the need to FAIRly describe and explore functions independent of the development context, and automatically instantiate them. With that regard, the requirements of the previous scenarios are grouped into three main categories, and expanded with one overarching category. The results are summarized in Table 1, where the requirements are listed per category, and linked to the scenarios using "✓". The requirement categories are:

- **Function description**: represent and update the descriptions of functions, inputs, outputs, and problems, the links between them, and any restriction that needs to be imposed. This category also includes the metadata of the function, necessary for tracking authors, contributors, version, created data, etc.

- **Implementation mapping**: represent and update relationships among functions and implementations, and map the abstract inputs and outputs to the implementation-specific parameters.

- **Execution description**: represent the description of a function execution, (i.e., its inputs, outputs, and implementation mapping), and the metadata associated with the execution (i.e., its provenance). This additional information allows to connect argument data to a function, and automatically invoke an implementation mapped to a function.

- **Semantics**: include the types, links, and semantic constraints of the handled function, input, output, problem, and implementation descriptions. Explicit semantics make the descriptions unambiguous and machine-understandable, thus allowing for automatic implementation instantiation. Explicit semantics also have following advantages: (i) exploration improves as implicit

links between, e.g., specific implementations and broader problem domains can be inferred; (ii) uniquely identifiable resources are universally discoverable and linkable; and (iii) storing additional execution metrics and comparing different implementations is eased, as adding metadata to semantic descriptions is natively supported. Finally, explicit semantics make resource more interoperable and reusable, thus aid in adhering to the FAIR principles.

To adhere to all requirements, on the one hand, we need an approach to provide function descriptions, on the other hand, we need ways to consume these descriptions. The description and publication of functions and their mapped implementations is further detailed in Section 4, and the consumption of these descriptions is further detailed in Section 5.

4. Description and publication

The requirements show the need to distinguish between the (abstract) function and the (concrete) implementation, similarly observed by Garijo et al. for workflows [37]. We describe functions and implementations separately, and further model the mapping between abstract functions and (parts of) implemented packages or services. This allows us to make use of existing implementations. If we would specify a new abstract programming language, existing implementations cannot be reused, and functions would need to be re-coded in this new language. One function can be linked to multiple implementations, for example: a date normalization function is linked to a maven package, a JavaScript snippet, and a RESTful Web service. We retain a result's provenance information by separately describing the function's execution. The model is further detailed in Section 4.1. Using these function descriptions and associations to (existing) implementations requires a publication approach, further detailed in Section 4.2.

Table 1: Summarized and generalized requirements, according to four main categories and the scenarios – Describe (1), Explore (2), Instantiate (3), and Update (4) – they belong to

N°	Category	Requirement	1	2	3	4
R1	Function description	Identifyable [functions, inputs, outputs, problems]	✓			✓
R2		[inputs, outputs, problems] belonging to a function	✓	✓		✓
R3		Functions associated with specific input parameters	✓	✓	✓	
R4		[label, description, keywords] of a [function, input, output, problem]		✓		✓
R5		[version, author, contributor, creation date, modification date] of the function		✓		✓
R6	Implementation mapping	Identifyable implementations				✓
R7		[source, repository] of the implementation		✓	✓	✓
R8		[package, Web service] implementation descriptions		✓	✓	✓
R9		Functions associated with specific implementations		✓	✓	✓
R10		Explicit mapping of inputs to parameters of implementation		✓	✓	✓
R11		[label, description, keywords] of an implementation		✓		✓
R12	Execution description	[input data, output data, function] involved in the execution	✓		✓	
R13		Download location for software packages		✓	✓	✓
R14		Instantiating/executing implementations			✓	
R15		Instantiating/executing implementations of different development contexts			✓	
R16		[status, timestamp, metrics] of an execution	✓		✓	
R17	Semantics	Functions that have an [input, output] with specific semantic constraints	✓	✓		
R18		implementations with specific semantic constraints		✓	✓	
R19		comparing (metrics of) executions	✓		✓	
R20		Functions that link to [broader, narrower] [functions, problems]	✓	✓		
	FAIR	Findable	✓	✓		
		Accessible		✓	✓	
		Interoperable	✓			✓
		Reusable			✓	✓

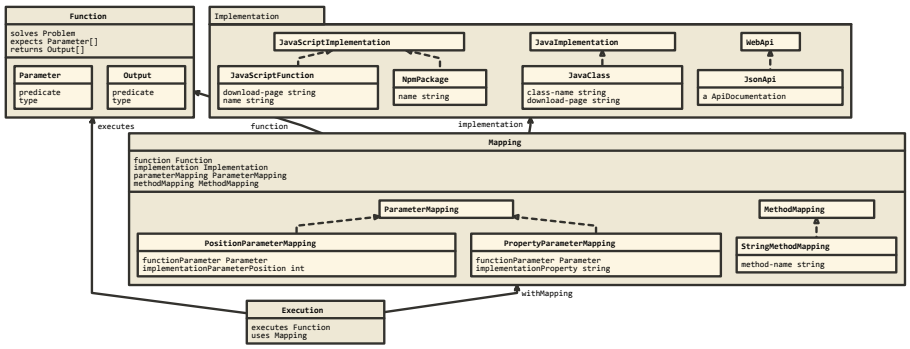

Figure 1: The FnO model

4.1. Modeling function descriptions

Functions are represented as a transformation of input data into output data. The main components are modeled in the Function Ontology (FnO) (Figure 1) [38], a revised and extended version of our original proposal [39]: the *Function*, the *Execution* of a Function, and a *Mapping* that maps (a part of) an *Implementation* to a function[10]. FnO is a reusable ontology of base concepts, that needs to be specialized for real world use cases, following the Content Ontology Design Pattern[11]. The specification is available at `https://w3id.org/function/spec`, a normative SHACL shape is available at `https://w3id.org/function/shape`. The latter allows users to validate their own FnO descriptions to be conformant to the FnO model.

Function. A *Function* expects a list of *Parameters* and returns a list of *Outputs* (Figure 1, top-left). The parameters are ordered in a list, define the relationship that is used for the execution (*predicate*) and can have a specific type or other metadata (e.g., required or not, having a default value or not). A function's output is also a list, as multiple values can be returned (e.g., a Web API can return a body and a status code, a local implementation can return a value or

[10] *Mapping* and *Implementation* are novel contributions, enabling automatic implementation instantiation.

[11] `http://ontologydesignpatterns.org/wiki/Category:ContentOP`

throw an error). Functions can be linked to *Problems*, which are more general descriptions than functions, e.g., the "Euclidean distance"-function is related to the "Distance"-problem. To create a more specific organization, problems can be further interlinked with each other using the SKOS standard.

Execution. An *Execution* links the input data and resulting output data with the function's parameters and outputs (Figure 1, bottom). This allows to describe how specific input data gets transformed in output data, independent of the implementation. On the one hand, this allows a declarative approach for specifying the input data of a concrete execution of an implementation. On the other hand, this allows attaching additional provenance after an execution, useful for publishing reproducible results. The function and execution model are aligned [40] with the W3C recommended provencance ontology, PROV-O [10].

Implementation. An *Implementation* is a set of function units (Figure 1, top-right): a Web API can consist of multiple endpoints, and a software package can implement multiple functions. The description of the implementation itself is decoupled from FnO. By allowing any development context to be specified, our model is not limited to a specific set of supported development contexts. Existing specifications can be reused: DOAP is reused to further describe a piece of JavaScript code (*JavaScriptFunction*), and Hydra is reused to describe a Web API (*ApiDocumentation*). The types of development contexts are thus addressed in a generic way. In order to support emerging needs, it allows extensions to existing and new development contexts.

Mapping. A *Mapping* connects an abstract function with a specific part of a concrete implementation (Figure 1, middle). This connection is twofold: link between the function and the implemented method, and link between the function's inputs and output and the method's parameters and return values. For example, a method name can be mapped to a function, and the first argument of that method can be mapped to a parameter. This way, two implementations that implement the same function, but where the order of parameters is

different, can be mapped to the same function. Moreover, existing software component descriptions can be used as implementation metadata, and specific methods within those components can be linked to abstract functions using a mapping description. Multiple parameter mapping types can exist: Web APIs are not executed using an ordered list of parameters, but using properties in the body of the Web API request. Currently, two parameter mappings are modeled: mappings for software packages (*PositionParameterMapping*) and for Web APIs (*PropertyParameterMapping*). As with the Implementation model, the Mapping model allows extensions.

4.2. Publishing and linking functions as data on the Web

As evidenced [5], following the Linked Data principles [4] allows us to publish resources on the Web FAIRly. These principles state: (i) use URIs as names (i.e., identifiers) for things, (ii) use HTTP URIs so that people can look up those names (i.e., making those URIs dereferenceable and available in any browser), (iii) provide useful information when someone looks up a URI (by showing the resources that are related to that URI), and (iv) include links to other URIs, so anyone can discover additional information.

Publishing functions and their resources as Linked Data has the following advantages: (i) linking resources is a native feature of Linked Data; (ii) interoperability is achieved by design, using standardized HTTP operations and formats; and (iii) already available (Web) resources can be linked to, e.g., by referring to existing implementations in package managers using their published URI.

We use the established publishing method used for government data [41] and other domains such as energy consumption [42]. This method consists of five main steps: *specification*, *modeling*, *generation*, *publication*, and *exploitation*.

Specification. The URI naming convention for the created resources is designed and the resulting dataset's license is defined. All generated URIs follow the W3C recommendation for cool URIs [43]: they are unique, permanent, and produced

under a domain of our control. Each URI identifies a resource that can be individually accessed and dereferenced with content negotiation [44]: the same URI can handle requests from humans (returning HTML) and machines (returning, e.g., JSON). The generated URIs adopt the following naming scheme:
Base URI = `https://w3id.org/function/`
Ontology URI = `https://w3id.org/function/ontology#`
Assertion URI = `https://w3id.org/function/hub/data/resource/[Type]/[ID]`.
We use the CC0 license[12], putting all published data in the public domain and allowing everyone to copy, modify, and distribute them as they desire.

Modeling. Which vocabularies to use are decided according to the aforementioned requirements and scenarios. We use the presented Function Ontology (FnO) together with well established standardized vocabularies such as SKOS [13] and PROV-O [10, 40].

Generation. The data is serialized in a structured format, using the Resource Description Framework (RDF) [45], and possibly cleaned and linked with other sources. We rely on both manual and automatic generation. On the one hand, we provide an editing environment for manual generation of function descriptions, where the identifiers and resulting descriptions are generated automatically according to best practices. On the other hand, we provide the necessary APIs for programmatic access, e.g., for adding automatically generated function descriptions based on analysis of a package manager's dataset. To this end, existing efforts such as the LSD framework [19] and CodeOntology [17] can be re-used.

Publication. The RDF data produced by the generation step is made available online. We load the data into a triple store and make it available through a public endpoint (i.e., an access point for both users and machines). We have selected GraphDB[13] as triple store, but any triple store that provides a com-

[12] `https://creativecommons.org/publicdomain/zero/1.0/`
[13] `https://www.ontotext.com/products/graphdb/`

pliant SPARQL API can be used. The public SPARQL endpoint is available online at `https://w3id.org/function/hub/data/sparql`, along with a set of sample queries to retrieve basic data from functions and demonstrate its main functionality[14]. For non-Semantic Web experts, a GraphQL endpoint is also available at `https://w3id.org/function/hub/api/graphql`, using GraphQL-LD to directly query the SPARQL endpoint via GraphQL [46].

After these four steps, we have a published dataset of function descriptions and associated resources such as inputs, outputs and problems, and mappings to (existing) implementations that reside elsewhere on the Web. Querying the published descriptions allows a user to find relevant functions and implementations. Once a suitable implementation is found, the user can choose to write the integration code to use the implementation into his/her program, or take advantage of the semantic descriptions to either automatically execute the implementation or use automatically generated source code. This and other advantages of this published dataset – i.e., the *exploitation* step – are further detailed in Section 5.

5. Consumption

Once function descriptions are available as Web resources, each can be accessed individually. To facilitate different use cases, more complex relationships can be retrieved by querying the public endpoints. The data is also machine-readable and thus automatically processable, as all resources are dereferenceable and offered in multiple serialization formats. However, the users are anyone who want to explore functions and instantiate implementations. Usually, they are not Semantic Web experts, are not familiar with query languages such as SPARQL and GraphQL, and do not know the models or schemas of the function descriptions.

In this section, we present a suite of tools for exploiting the published function descriptions in an end-to-end scenario. We first give an overview (Sec-

[14] `https://w3id.org/function/hub/sparql`

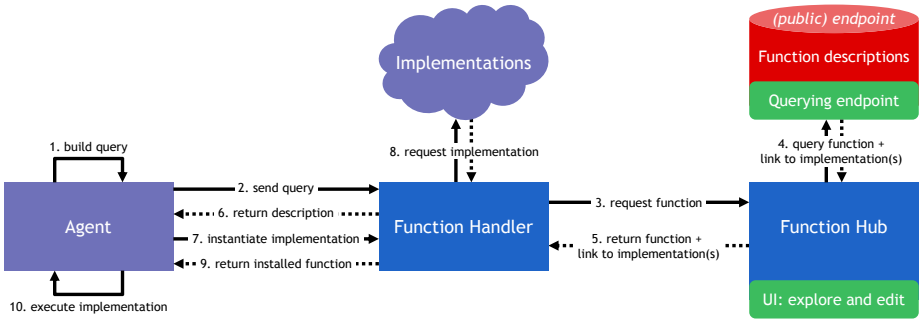

Figure 2: Overview of the overall architecture, showing the different steps to automatically find, retrieve, instantiate, and execute a function. The components in cloud icons can be remote, the components in rectangles are local.

tion 5.1), after which we detail the two main components: the Function Hub (Section 5.2) and the Function Handler (Section 5.3).

5.1. Overview

Our architecture follows the client-server model (Figure 2).

Server-side, the *Function Hub* is responsible for managing and publishing the function descriptions and their implementation mappings. The Function Hub connects to the (public) querying endpoint of the data store (Figure 2, top-right), and provides a set of developer-friendly APIs[15] and a user interface[16] (Figure 2, bottom-right).

Client-side, the *Function Handler* is responsible for retrieving function descriptions and instantiating implementations. The Function Handler acts as a proxy between the user and the function descriptions residing in the Function Hub. The Function Hub can act independent of the development context, whereas specific versions of the Function Handler are needed per development context that requires automatic instantiation of an implementation.

Next, we present the steps that are followed – common for any development context – when an *Agent* requires to automatically retrieve and execute an im-

[15] `https://w3id.org/function/hub/api`
[16] `https://w3id.org/function/hub`

plementation (also depicted in Figure 2). The term agent denotes both humans and machines. Examples of agents include (i) a user that wants to execute a function on the commandline, (ii) a piece of software that retrieves relevant implementations at runtime, and (iii) a user that wants to find and manually integrate an implementation for a required function in his or her application.

1. An agent builds a query, e.g., a function that returns a float value, and accepts two integer values as input. Keywords might be included to further narrow down the available options.
2. The query is sent to the Function Handler...
3. ...which results in an API call to the Function Hub.
4. The Function Hub interacts with the querying endpoint to receive the query's results, i.e., function descriptions and implementation mappings.
5. The result is returned to the Function Handler as response to the request of step 3...
6. ...and returned to the agent.
7. The agent chooses (manually or automatically) which implementation to instantiate.
8. The Function Handler uses the implementation mapping. In the case of a local library, the Function Handler requests the implementation code from the Web – possibly caches the result – and instantiates the implementation. In the case of a Web API, the Function Handler creates a proxy function that accesses the Web API when the function is invoked. Alternatively, the Function handler generates the example code to retrieve the implementation using, e.g., a package manager, and instantiate the specific method.
9. An instantiated implementation is returned to the agent.
10. The agent executes the implementation.

5.2. Function Hub: exploring and updating function descriptions

We help users explore (i.e., *browse* and *search*), and *update* the function descriptions, via the user interface of the Function Hub. This allows them to

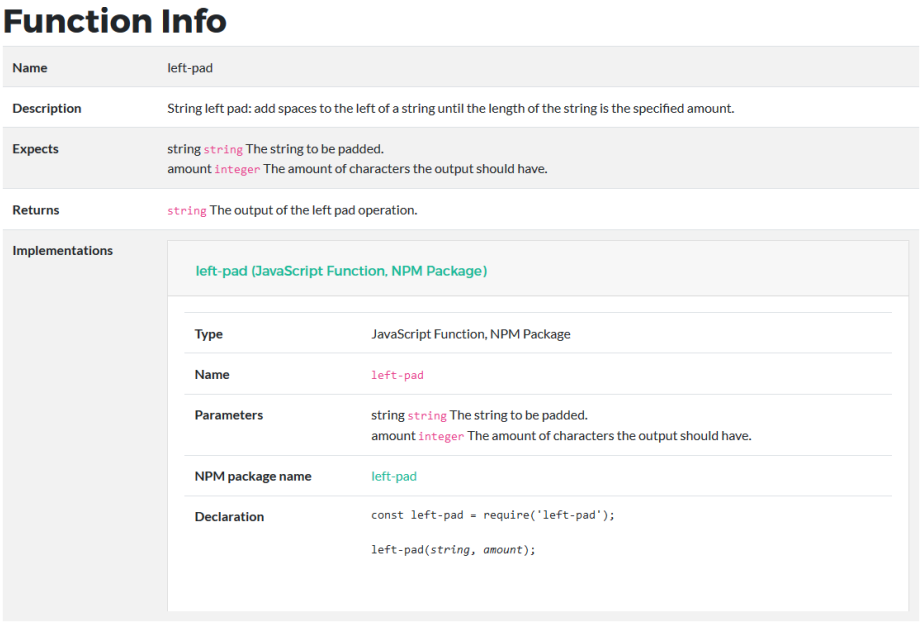

Figure 3: Snapshot of the Function Hub showing the information of a function

interact with the dataset without needing to execute any queries via the public querying endpoint.

Browse. In the Function Hub, users can navigate the different function descriptions, and their implementation mappings. The entry point for the application is a function name. Whenever a user selects a function, information appears on the screen as shown in Figure 3. For every function, the Function Hub shows the name, description, parameters, outputs, and a list of implementations. Each resource is a link that can be resolved in the browser for more information. The Function Hub's browse functionality, available as a Web application, makes it easy to discover the available implementations of a function, what type of parameters are used, and code is automatically generated to instantiate the implementation.

Search. Two interfaces are available for searching in the Function Hub: a simple search which performs a keyword search over the functions' names and descrip-

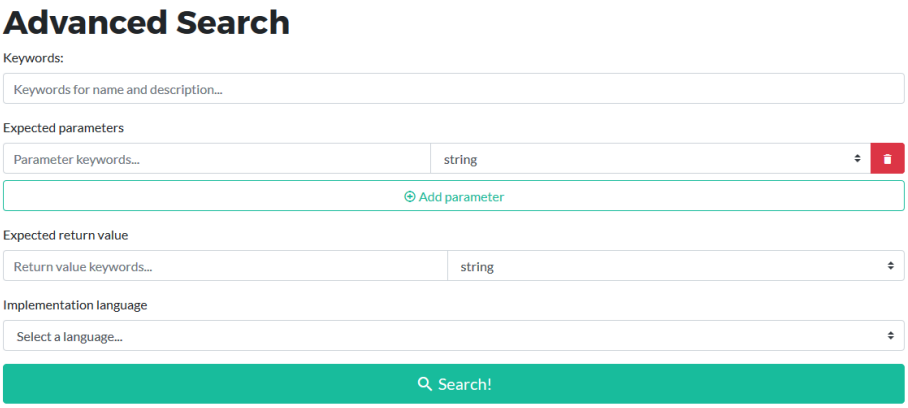

Figure 4: Snapshot of the Function Hub's search functionality

tions, and an advanced search which includes filtering functionality, e.g., by parameter or return type (Figure 4). All search operators are applied directly on the SPARQL endpoint: keyword search is applied by filtering on linked string literals. The usage of semantics allow to also search within linked resources, for example, broad problem descriptions can be taken into account when searching for specific functions and vice versa. The results are displayed as a list of functions, which can be further inspected via the browse functionality.

Update. The editing environment supports creating new and editing existing function descriptions (Figure 5). More specifically, all metadata of the function and its associated resources (inputs, outputs, problems) can be edited, and mappings with implementations can be added. Updates are directly reflected in the function dataset.

5.3. Function Handler

A Function Handler is a type of client application that interacts with the Function Hub and published implementations. When automatically integrating data from the Function Hub with a software project in a specific development context, a specific Function Handler for that development context is needed. A Function Handler receives the implementation code and performs the nec-

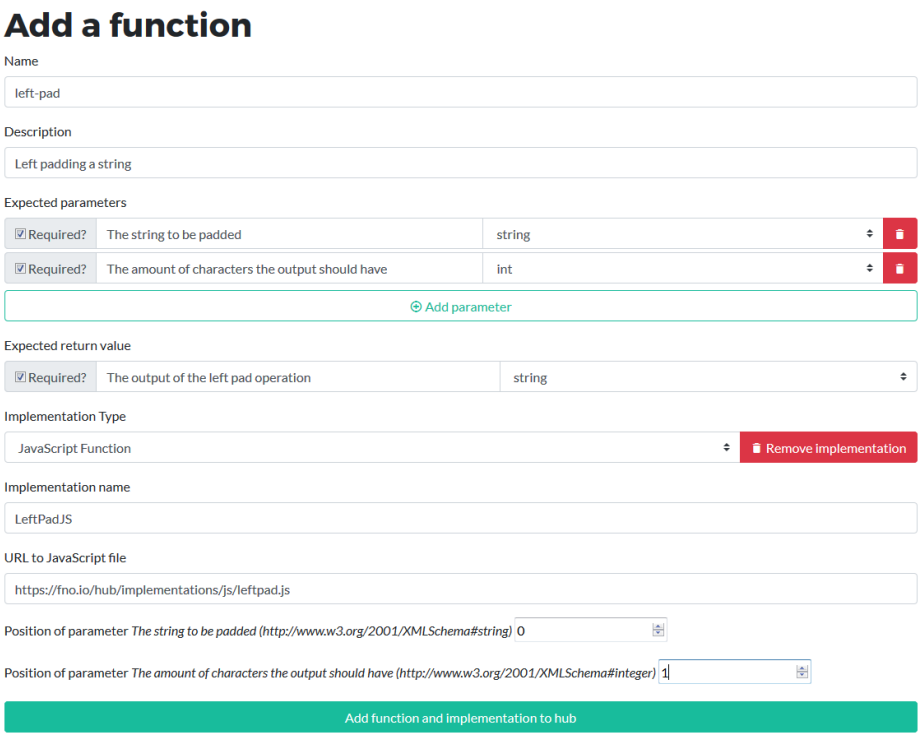

Figure 5: User interface to add a function to the Function Hub

essary tasks to instantiate a subroutine that can be executed in the application. As such, it allows non-Semantic Web experts to interact with the Funtion Hub. Multiple implementations of the Function Handler are currently available, specifically, a JavaScript[17] and a JAVA[18] Function Handler.

Listing 1 shows how we can use the JavaScript Function Handler. A query is built (lines 5–9), and the Function Hub is queried for functions with mapped implementations (line 12). In the example, the first returned function is selected (line 15). The Function Handler automatically filters for implementations compatible with the current development context. In the example, the first implementation is used (line 18). That implementation is invoked automatically, returning the required result (line 21). Note how any other (automatic) selection process can be used, both for function and implementation results.

```
1  // Import the package and specify the Function Hub location
2  const fnHub = require('functionHub')('https://fno.io/hub/api');
3
4  // Construct a query that provides us with a function to indent a
       string
5  const indentQuery = {
6    expects: [{ type: 'string' }, { type: 'integer' }],
7    returns: { type: 'string' },
8    keywords: ['indent']
9  };
10
11 // Query the FunctionHub server with the constructed query
12 const queryResult = await fnHub.doQuery(indentQuery);
13
14 // Use the first function that has been returned
15 const func = queryResult[0];
16
17 // Use the first implementation of this function
18 var indentImplementation = (await fnHub.
       getImplementationsFromFunction(func))[0];
```

[17] https://github.com/FnOio/function-handler-js
[18] https://github.com/FnOio/function-handler-java

```
19
20 // Execute this implementation with the parameters 'Hello' and '8'
21 console.log(indentImplementation('Hello', 8));
22 // Output: "        Hello"
```

Listing 1: Using the JavaScript Function Handler to retrieve and invoke an indenting function.

The automatic invocation of an implementation allows to easily substitute an implementation, or combine multiple implementations. For example, a mobile application that uses image recognition could, by default, make use of a remote Web service to execute the image recognition. However, when online connectivity is not stable, this Web service can be substituted with a local implementation using the Function Handler, using the same interface.

Moreover, a Function Handler can also automatically generate the code needed to integrate a function in an existing codebase due to the detailed semantic descriptions residing in the Function Hub. For example, a command-line interface can generate the code to require a specific NPM module and invoke a function within that module. This way, when preferred, existing package managers such as NPM and Maven are fully leveraged and no overhead is necessarily introduced to process a function description. For example, Figure 3 shows how a Function Handler is integrated in the Function Hub: exemplary code of how the "left-pad" function can be directly instantiated using NPM is shown, automatically generated based on the metadata of the function, implementation, and mapping.

6. Results and discussion

In this section, we validate the feasibility of our publishing method (Section 6.1), present function descriptions' usage (Section 6.2), assess whether our suite of tools fully addresses the requirements identified in Section 3 (Section 6.3), and evaluate adherence to the FAIR principles (Section 6.4).

6.1. Publishing feasibility

Our model is validated by answering each of the requirements listed in Table 1 with the FnO model and examples from the Function Hub. For each requirement, we assigned a SPARQL query to address it. The table from this effort is available online at `https://w3id.org/function/requirements/conformance`.

The publication of the resources is validated according to the Linked Data principles [4], the Best Practice Recipes for Publishing RDF Vocabularies [47], and the cool URIs specification [43] using the Vapour system [48][19]. Vapour launches a set of tests retrieving the exposed data in different formats typically consumed by humans (e.g., HTML) and machines (e.g., RDF/XML). The test results are available online at `https://w3id.org/function/hub/vapour`.

Every requirement can be (partially) answered successfully, and the Vapour test results are positive. We thus conclude that FnO is sufficient to represent functions and link them to implementations and executions, and that our approach is sufficient to publish FnO resources as data on the Web.

6.2. Function description usage

To showcase that our approach is use case independent, we present different use cases and development contexts where function descriptions were applied.

General data transformations. We produced function descriptions of an existing software package of commonly used functions. Specifically, we provided descriptions for the GREL functions[20], a set of functions and implemented software package defined for the OpenRefine framework[21]. These descriptions are available at `http://semweb.mmlab.be/ns/grel/`.

[19] `http://linkeddata.uriburner.com:8000/vapour`
[20] `https://github.com/OpenRefine/OpenRefine/wiki/GREL-Functions`
[21] `http://openrefine.org/`

Docker files. We provided the description of Dockerfiles using a list of function descriptions. The individual statements within a Dockerfile are described using FnO [36]. This allows (i) adding additional knowledge to the instructions, (ii) sharing (references to) instructions outside the context of a Dockerfile, and (iii) using these semantic annotations for self-descriptive instructions that are understandable outside of the Docker ecosystem.

Linked Data generation. We combined function descriptions with a declarative knowledge graph generation language. Specifically, we combined FnO with the RDF Mapping Language (RML) [49]. This allowed for a fully descriptive process, applied to the DBpedia Extraction Framework [31], responsible for generating the DBpedia dataset [7]. The implementation-independent function descriptions allowed us to extract the implementation of transformation functions from the DBpedia Extraction Framework as is, and publish it as an independent software package[22]. We validated that this approach can generate an equivalent dataset in a sustainable way [30]. As opposed to the original framework, the created system is use case independent, and the data transformations are reusable across systems and data sources.

6.3. Function exploitation: conformance to the requirements

In this section, we analyze whether the requirements defined in Section 3 are addressed by the tools proposed to facilitate the exploitation of function data.

Function descriptions. Thanks to the browse functionality of the Function Hub, function metadata like inputs and outputs (R2) and documentation (R4) are shown to the user when accessing a function by its identifier (R1) or name (R4). The search functionality allows to find functions associated with specific input parameters (R3). No data currently exists concerning the version information of the function data itself (R5). However, version information of linked implementations is available in package managers and version control systems.

[22] `https://github.com/FnOio/dbpedia-parsing-functions-scala`

Implementation link. When browsing functions in the Function Hub, implementations are identified (R6) and listed (R9) by name (R11), each linked to their original publication location (R7), with explicit links to the software package or Web service (R8) and their inputs and outputs (R10). The Function Hub itself does not host any code, thus, code descriptions or insights thereof are available on the relevant repositories instead (R11).

Execution. The execution requirements are addressed by the Function Handler. Provenance traces can be generated automatically based on the execution data [40]. These contain the used function, input and output data (R12), and additional metadata (R16). In the case of using a software package, the Function Handler can automatically retrieve the code via a download location (R13). It provides a uniform API for automatically instantiating and executing implementations (R14) of different development contexts (R15). Furthermore, the Function Hub contains Function Handler functionality: each linked implementation on the Function Hub is also visualized using automatically generated code snippets, based on the detailed mapping metadata.

Semantics. The semantics requirements are handled by both the Function Hub and the Function Handler. The advanced search functionality allows filtering on specific semantic constraints for parameters and outputs (R17), and implementations (R18). Search takes implicit links, such as related broader functions or problems (R20) into account. The execution data is implementation independent and can thus be compared across development contexts (R19). However, no dedicated software is currently available.

In summary, 17 out of 20 requirements are fully supported by our suite of tools. For the remaining requirements, 2 are supported by source code repositories, and the final requirement can be handled using dedicated visualization tools for test result metrics.

6.4. Function exploitation: adherence to the FAIR principles

We review adherence to the FAIR principles [5][23] of our approach and suite of tools. Each (sub-)principle is denoted between brackets (e.g., "(F1)").

Findable. As we make use of the Linked Data principles, we assign globally unique and persistent identifiers (F1). Our data model is shown to be sufficient (F2), and functions can be linked to existing implementations (F4). All data is indexed in the Function Hub (F3).

Accessible. Standard HTTP is used to retrieve all resources (A1), with standardized APIs such as REST and SPARQL (A1.1). Authentication and authorization is complementary, and available by default, for example via HTTP authorization (A1.2). The function descriptions are complementary to the implementations. Even when an implementation is no longer available, the function metadata remains accessible (A2).

Interoperable. The function descriptions are serialized in RDF [45], a standardized representation format (I1). By adhering to the Linked Data principles, the used vocabularies follow the FAIR principles by design (I2). References to complementary data is available, e.g., via the use of keywords (I3).

Reusable. The Function Ontology model has been tested to be conform to the requirements, so we can state the functions are accurately described (R1). All metadata is published with a CC0 license (R1.1), and with detailed generation provenance (R1.2). We evaluated that our publishing approach meets the community standards, by testing our published data against a third-party system, Vapour (R1.3).

We conclude that our proposed systems allows for FAIR publishing and sharing of function descriptions.

[23] `https://www.force11.org/group/fairgroup/fairprinciples`

7. Evaluation

In this section, we evaluate our presented exploitation tools. Specifically, we evaluate the Function Hub in which a Function Handler is integrated that automatically generates code snippets. We evaluate how, for different development contexts, the exploitation tools affect developers' *effectiveness* during software retrieval – i.e., when discovering functions and retrieving their linked implementations – compared to traditional methods such as investigating the programmer documentation or querying general-purpose search engines. Next, we describe the evaluation's research questions (Section 7.1), scope (Section 7.2), methodology (Section 7.3), set-up (Section 7.4), results (Section 7.5), and threats to validity (Section 7.6).

7.1. Research Questions

Effectiveness of using the Function Hub for a software retrieval task can be measured (i) quantitatively as efficiency, namely, correctness of task result and time to complete, and (ii) qualitatively as perceived usability. We pose the following Research Questions (RQ), with the following hypotheses (H):

RQ1 To what degree does using the Function Hub affect the efficiency during software retrieval?

H1 Developers can immediately use the Function Hub without any statistically significant difference in correctness of task result or time to complete of executing a given software retrieval task, compared to using traditional methods.

RQ2 To what degree does using the Function Hub affect the perceived usability during software retrieval?

H2.1 Developers perceive using the Function Hub as a more appropriate method for software retrieval tasks compared to traditional methods, and this difference is statistically significant.

H2.2 The system usability score (SUS) [51] of the Function Hub is above average, i.e., above 68, based on the findings of Bangor et al. [50].

7.2. Scope

Development contexts. We use development contexts that use a declarative programming language. This way, results of the evaluation focus on the effect of using semantically enhanced versus plain-text descriptions, as the amount of used programming syntax is limited. The domain-specific languages SQL, XPath, and SPARQL are used as they are W3C Recommendations – thus well-documented and accepted by a wide community – and parts of their documentation explicitly link each other. We added a less-known language to validate that our evaluation's findings are not influenced by pre-cognition effects of the participants. Specifically, we include a related general-purpose language used in OpenRefine: the General Refine Expression Language (GREL).

Data. We use the programmer documentation's data to provide descriptions within the Function Hub, making sure the amount and detail of metadata is the same across methods.

Tasks. The tasks are devised from a set of functions commonly defined across all development contexts (e.g., extracting the year from a date object, or retrieving a substring), and their phrasing is based on highly-voted StackOverflow questions. This way, we make sure the amount of needed metadata is the same across development contexts and the tasks are representative for real-life problems.

7.3. Methodology

Our user evaluation follows Armaly and McMillan's methodology [52], which evaluates an implementation-dependent software retrieval solution. The methodology consists of three steps. First step: collect the **participants' profile** using an initial survey inquiring about general demographics and development expertise based on surveys of related works [52, 53]. Second step: present **software**

retrieval tasks to be solved by the participants. Third step: take an **exit survey** which asks each participant to quantitatively rate and qualitatively compare both methods they had employed during the study, and which includes the SUS questionnaire. Ratings require multiple-choice Likert-Scale answers that range from 1 (strongly disagree) to 5 (strongly agree).

The second step consists of two rounds. For the first (scoped) round, participants are only allowed to use either the programmer documentation (made available by directly accessing their online version via a browser), or a Function Hub instance containing the function libraries' metadata. For the second (real-world) round, participants of either groups could additionally use any means they deemed necessary (e.g., Web search), to measure the effect of using the Function Hub in a real-world setting.

Each round consists of two randomly picked tasks. In the first round each task is solvable using one function, in the second round each task is solvable using a combination of at least two functions, thus further increasing the real-world effect. The order was alternated for each participant per round: for every round, half of the participants performed the first task using the Function Hub and the second task without, and the other half vice versa. By alternating, we ensure against possible bias of one method falsely appearing more effective because the participant was already introduced to the task in the preceding task. Each task needs to be solved twice for two randomly picked development contexts, i.e., each task contains two sub-tasks. For each sub-task, we measured correctness and completion time, and an intermediate survey was conducted after every sub-task's completion.

As such, both functionality of the Function Hub (i.e., exploring functions and linked implementations) and the Function Handler (i.e., automatic execution or code generation of implementations) are evaluated quantitatively and qualitatively, both in a scoped and a real-world scenario.

7.4. Set-up

Tests. We use the Wilcoxon signed-rank test [54] with Pratt modification [55] to determine whether differences between participants' paired answers are statistically significant. We use the Wilcoxon signed-rank test for this analysis because (i) it is nonparametric, applicable to our data as we cannot test that our data is normally distributed due to the small sample size; and (ii) it is paired, which is conform with the design of this test: each participants performs similar tasks both with and without the Function Hub. We use the Pratt modification to deal with the limitation of the original Wilcoxon signed-rank test, namely, that it is unable to handle equal results. For example, in the case of a 5-point Likert scale, equal results are likely, hence the Pratt modification. The time duration to complete each sub-task was limited to at most 15 minutes after internal validation, as the Wilcoxon signed-rank test requires that data is measured on at least an interval scale [54].

Participants. We recruited 11 developers to participate in our study. These participants have an average programming experience of 10 years, and their experience with the given domain-specific languages is distributed: the average experience for the given languages, scored 1 (inexperienced) to 5 (experienced), is GREL 1.6, XPath 2.7, SPARQL 3.6, and SQL 4. Our assumption that GREL is a less known language is thus confirmed. The participants were further inquired about their experience with handling programmer documentation and using the Function Hub, to validate that the Function Hub is less known. There was a statistical significant difference between their experience with handling programmer documentation compared to using the Function Hub (p-value 0.0067), i.e., they are more experienced using programmer documentation, confirming our assumption.

Metrics. We answered RQ1 using the results of the tasks with or without using the Function Hub: on the one hand comparing the number of correctly found methods (to test correctness of discovery), and on the other hand comparing the number of correctly used methods (to test correctness of instantiation).

Furthermore, we compared the total amount of time spent for each task. We answered RQ2 by evaluating the qualitative answers, and by calculating the average SUS score.

7.5. Evaluation Results

We evaluated both quantitatively as qualitatively the Function Hub, as discussed below. All relevant evaluation surveys, tasks, and (anonymized) evaluation results are available as a dump at `https://doi.org/10.6084/m9.figshare.9879890.v1`.

Function Hub Effectiveness. Statistical testing showed no statistical significance between correctness of functions found, functions applied, and overall solution, nor for time spent for the given tasks. Such statistical significance was neither found when filtering the results for either the first or second round. However, the participants were not given any introduction of how to use the Function Hub, and were less experienced in using the Function Hub compared to using programming documentation. We thus conclude that using the Function Hub functionally provides a complementary solution to the software retrieval problem, without affecting effectiveness.

Perceived Ease of Use. After each task, and in general after the evaluation, participants were inquired about the perceived ease of use. For each task and per used method, they were given the questions "I found the task very difficult", "I found the method I used to be appropriate for this task", and "I can see why someone would use this method for this task", taken from [52]. For all three questions, there is a strong statistically significant difference between the answers that compare using the Function Hub or programming documentations. Overall, participants found (i) **tasks easier using the Function Hub**, (ii) **using the Function Hub more appropriate** for software retrieval tasks than using the programmer documentation, and (iii) **prefer using the Function Hub** for the given tasks (p-values respectively 0.0307, 0.0007, and 0.0020).

After the evaluation, we also inquired to general feedback and compared the results to the answers of "I found the Function Hub very easy to use" versus "I found examining the programmer documentation very easy to use". Participants found **the Function Hub statistically significantly easier to use** than examining the programmer documentation (p-value 0.0046). This is supported by the average SUS score of 83.4. This puts the usability score of the Function Hub in the top quartile compared to other SUS evaluations [50].

Finally, we summarize the participants' feedback and link them to the requirements of Section 3.

- The Function Hub's **common layout across implementations** was well-received (signifying the importance of a **common metadata framework**, i.e., R2,3,4,9,10,11).
- The **importance of search functionality** was apparent both when using the Function Hub as when examining programmer documentation (signifying the importance of the explore scenario requirements). For the latter case, the browser search functionality was commonly used as a substitute.
- The fact that within the Function Hub a function descriptions with linked implementation contains an **auto-generated code snippet** was well-received (signifying the importance of detailed metadata mappings functions and implementations, i.e., R12,14,15).
- Getting presented with many search results was not well-received. In this case, **advanced filtering** was sporadically used (signifying the importance of detailed metadata, i.e., R2,3,9,10,17,18,20).
- The need for the Function Hub to **contain more data** was expressed commonly (signifying the importance of the update scenario requirements).
- Function descriptions linked to multiple implementations, on the one hand, **provide more context** to understand a function, but, on the other hand, **cause confusion** in certain cases, e.g., GREL's substring function starts

counting the first character as index 0, SQL's substring function starts from index 1 (signifying the importance of the links between implementation, i.e., R2,9,10, but also the need for better descriptions for specific implementations).

- The **semantic descriptions** were well-received, (i.e., R2,3,17,18,20) specifically, the datatype descriptions, the fact that broader and related keywords also lead to the right function, and links to additional information is provided.

- Some **suggestions** were given about possible improvements to the Function Hub: autocomplete, relevance sorting of the search results, and inclusion of versioning metadata.

We can conclude that this feedback closely follows the requirements as presented in Section 3, and provides us with insightful insights for future work.

7.6. Threats to validity

As with any study, our evaluation carries threats to validity.

For any search interface, the amount of data in the index is important to make sure all relevant results are found. This is no issue during the first round, as the amount of data within the Function Hub is the same as found in the programmer documentation. For the second round, we investigated the effect of including global search engines such as Google and StackOverflow, and notice no statistical significant difference in effectiveness compared to the first round. This threat can be mitigated by the update functionality of the Function Hub to harness the power of the crowds, together with related efforts that automatically generate implementation-dependent metadata [17].

The proposed tasks are limited, and tailored to the contents available in the Function Hub. This is partially due to the mitigation of the previous threat. Indeed, a large-scale evaluation where the Function Hub contains a large body of functions is envisaged in the future. For this evaluation, we chose commonly

occurring problems, derived from and phrased like popular StackOverflow questions, to make the tasks real-world, partially mitigating this threat.

The evaluation could create experimenter bias [56]: the participant supports our approach, knowing it is what we want. We attempted to mitigate this threat through our evaluation design which rotated the order of the methods used in each round. We also recruited our participants from a diverse body of developers and confirmed our results with accepted statistical testing procedures.

8. Conclusion

This paper handles the publication of abstract function descriptions, with mappings to concrete implementations as Linked Data resources on the Web. This way, we can link to existing implementations, and we provide a uniform search interface, usable across development contexts, with more advanced search operators. Additionally, we provide tooling for users to consume these resources, and allow for automatic execution and code generation. As shown, our proposed approach and set of tools are functionally sufficient for (automatic) exploration of functions and exploitation of implementations, and the user evaluation shows it is more usable than traditional methods.

The approach is interoperable, extensible, and scalable. Standards are used where possible, whether it be for the format (RDF, JSON), the query interface (SPARQL, GraphQL), or the model (PROV-O, SKOS). This improves interoperability of the published dataset with other systems. The proposed model, FnO, consists of the base concepts with little restrictions, to be highly reusable, and the mappings model is extensible: existing and new development contexts can be facilitated. The declarative mapping has the advantage that translating or wrapping existing implementations into a new defined interface is not needed: there is no need to wrap or re-code existing implementations into a new programming interface, instead, pre-existing package and source code repositories can be used as is. This decreases the effort for creating function descriptions and increases chance of uptake by the devel-

opment community. The Function Hub provides merely links to implementations: the amount of stored data is limited to metadata, leading to better scaling behavior with respect to the amount of available descriptions. Finally, we made a long-term commitment to preserve its longevity, as presented at `https://github.com/FnOio/fnoio.github.io/wiki/Sustainability-Plan`.

The exploration of functions and exploitation of implementations are complementary. The Function Hub can be used as a context-independent search engine that returns sufficient metadata to manually integrate an implementation of a requested function. The Hub can thus already be used in existing software projects, without depending on a Function Handler. A Function Handler can be used to automatically invoke implementations or generate example code, based on a set of metadata. This metadata is self-descriptive, i.e., it does not need to originate from – or be published in – the Function Hub.

Multiple areas for future work can be identified: improved natural language search, more generic Function Handlers, integration with automatic function description approaches, and implementation-independent testing suites. The current search functionality is limited to SPARQL operators. Using engines specialized for natural language search, such as ElasticSearch, in combination with the addition of synonym lists, are topics for future work that could further improve the usability of the Function Hub for end-users. Different Function Handlers are needed for different development contexts: a Function Handler for a JavaScript development context is tailored at executing JavaScript functions, and cannot be used in a JAVA development context. The usage of code generators could partially alleviate this issue: common functionality such as querying the Function Hub can be maintained within one development context, and other code, specific for the development context at hand, can be generated on request. This, together with investigating the role of Docker containers to (partially) remove the dependency on the development context, are topics for future work.

Our current tooling relies on manual creation of the function descriptions and implementation mappings. To scale up, automatic metadata extraction ap-

proaches can be used. We can make use of existing frameworks to automatically generate meaningful function descriptions, based on data of package managers such as NPM, and web service hubs. Then, the Function Hub represents – and allows to search for – the contents of these software packages in a uniform, development-independent way.

This work further paves the way for implementation-independent testing suites. There are existing efforts for describing evaluations, such as the DAWG test case structure [57] or the Evaluation and Report Language (EARL) [58]. However, executions of these evaluations still require manual development. By aligning with FnO descriptions, and making use of the Function Hub and Function Handler, we can provide automatic executions of these evaluations, for existing and new implementations.

Acknowledgements. The described research activities were funded by Ghent University, imec, Flanders Innovation & Entrepreneurship (VLAIO), and the European Union. Ruben Verborgh is a postdoctoral fellow of the Research Foundation – Flanders (FWO).

References

[1] E. A. Rundensteiner, Letter from the Special Issue Editor, IEEE Data Engineering Bulletin 22 (1) (2000) 2.

[2] M. Atkinson, S. Gesing, J. Montagnat, I. Taylor, Scientific workflows: Past, present and future, Future Generation Computer Systems 75 (2017) 216–227.

[3] C. S. Liew, M. P. Atkinson, M. Galea, T. F. Ang, P. Martin, J. I. V. Hemert, Scientific workflows: moving across paradigms, ACM Computing Surveys 49 (4) (2017) 66:1–66:39.

[4] C. Bizer, T. Heath, T. Berners-Lee, Linked Data – The Story So Far, International Journal On Semantic Web and Information Systems 5 (3) (2011) 205–227.

[5] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, J. Bouwman, A. J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C. T. Evelo, R. Finkers, A. Gonzalez-Beltran, A. J. Gray, P. Groth, C. Goble, J. S. Grethe, J. Heringa, P. A. 't Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S. J. Lusher, M. E. Martone, A. Mons, A. L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S.-A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M. A. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, B. Mons, The FAIR guiding principles for scientific data management and stewardship, Scientific Data 3 (2016) 160018.

[6] J. Piñero, À. Bravo, N. Queralt-Rosinach, A. Gutiérrez-Sacristán, J. Deu-Pons, E. Centeno, J. García-García, F. Sanz, L. I. Furlong, DisGeNET: a comprehensive platform integrating information on human disease-associated genes and variants, Nucleic Acids Research 45 (D1) (2016) D833–D839.

[7] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, C. Bizer, DBpedia – A large-scale, multilingual knowledge base extracted from Wikipedia, Semantic Web 6 (2) (2015) 167–195.

[8] S. Bajracharya, J. Ossher, C. Lopes, Sourcerer: An infrastructure for large-scale collection and analysis of open-source code, Science of Computer Programming 79 (2014) 241–259.

[9] R. Gardler, Project Catalogues and Project Descriptors using DOAP, Briefing note, OSS Watch (Feb. 2013).
URL `http://oss-watch.ac.uk/resources/doap`

[10] T. Lebo, S. Sahoo, D. McGuinness, K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, J. Zhao, PROV-O: The PROV

Ontology, Recommendation, World Wide Web Consortium (W3C) (Apr. 2013).
URL https://www.w3.org/TR/prov-o/

[11] D. Garijo, Y. Gil, The p-plan ontology, Tech. rep., Ontology Engineering Group (Mar. 2014).
URL http://purl.org/net/p-plan#

[12] D. Garijo, Y. Gil, O. Corcho, Abstract, link, publish, exploit: An end to end framework for workflow sharing, Future Generation Computer Systems 75 (2017) 271–283.

[13] A. Miles, S. Bechhofer, SKOS Simple Knowledge Organization System Reference, Recommendation, World Wide Web Consortium (W3C) (Aug. 2009).
URL https://www.w3.org/TR/skos-reference/

[14] P. Chalin, J. R. Kiniry, G. T. Leavens, E. Poll, Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2, in: F. S. de Boer, M. M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Formal Methods for Components and Objects, Vol. 4111 of Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2006, pp. 342–363.

[15] N. Mitchell, Hoogle Overview, The Monad.Reader 12 (2008) 27–35.

[16] I. García-Contreras, J. F. Morales, M. V. Hermenegildo, Semantic code browsing, Theory and Practice of Logic Programming 16 (5-6) (2016) 721–737.

[17] M. Atzeni, M. Atzori, CodeOntology: RDF-ization of Source Code, in: The Semantic Web – ISWC 2017, Vol. 10588, Springer, Cham, 2017, pp. 20–28.

[18] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan,

K. Sycara, OWL-S: Semantic Markup for Web Services, Member submission, World Wide Web Consortium (W3C) (Nov. 2004).
URL http://www.w3.org/Submission/OWL-S/

[19] J. Van Herwegen, R. Taelman, S. Capadisli, R. Verborgh, Describing configurations of software experiments as linked data, in: D. Garijo, W. R. van Hage, T. Kauppinen, T. Kuhn, J. Zhao (Eds.), Proceedings of the First Workshop on Enabling Open Semantic Science (SemSci), no. 1931 in CEUR Workshop Proceedings, Aachen, 2017, pp. 23–30.

[20] O. Corby, C. Faron-Zucker, F. Gandon, LDScript: a Linked Data Script Language, in: The Semantic Web – ISWC 2017, Vienna, Austria, 2017, pp. 208–224.

[21] C. B. Aranda, O. Corby, S. Das, L. Feigenbaum, P. Gearon, B. Glimm, S. Harris, S. Hawke, I. Herman, N. Humfrey, N. Michaelis, C. Ogbuji, M. Perry, A. Passant, A. Polleres, E. Prud'hommeaux, A. Seaborne, G. T. Williams, SPARQL 1.1 Overview, Recommendation, World Wide Web Consortium (W3C) (Mar. 2013).
URL http://www.w3.org/TR/sparql11-overview/

[22] S. P. Reiss, Semantics-based code search, in: Proceedings of the 31st International Conference on Software Engineering (ICSE), IEEE Computer Society, Washington, DC, USA, 2009, pp. 243–253.

[23] B. Regalia, K. Janowicz, S. Gao, VOLT: A Provenance-Producing, Transparent SPARQL Proxy for the On-Demand Computation of Linked Data and its Application to Spatiotemporally Dependent Data, in: Proceedings of the 13th International Conference on The Semantic Web. Latest Advances and New Domains, Springer International Publishing, Springer, 2016, pp. 523–538.

[24] S. Harris, A. Seaborne, SPARQL 1.1 Query Language, Recommendation, World Wide Web Consortium (W3C) (Mar. 2013).
URL https://www.w3.org/TR/sparql11-query/

[25] C. Debruyne, D. O'Sullivan, R2RML-F: Towards Sharing and Executing Domain Logic in R2RML Mappings, in: Workshop on Linked Data on the Web, CEUR Workshop Proceedings, CEUR, 2016.

[26] H. Knublauch, D. Allemand, S. Steyskal, SHACL Advanced Features, Working group note, World Wide Web Consortium (W3C) (Jun. 2017).
URL https://www.w3.org/TR/shacl-af/

[27] H. Knublauch, P. Maria, SHACL JavaScript Extensions, Working group note, World Wide Web Consortium (W3C) (Jun. 2017).
URL https://www.w3.org/TR/shacl-js/

[28] OpenAPI Specification, Tech. Rep. 3.0.2, Swagger (Oct. 2018).
URL https://swagger.io/specification/

[29] M. Lanthaler, Hydra Core Vocabulary: A Vocabulary for Hypermedia-Driven Web APIs, Unofficial draft, World Wide Web Consortium (W3C) (Jan. 2019).
URL http://www.hydra-cg.com/spec/latest/core/

[30] W. Maroy, A. Dimou, D. Kontokostas, B. De Meester, R. Verborgh, J. Lehmann, E. Mannens, S. Hellmann, Sustainable linked data generation: The case of DBpedia, in: C. d'Amato, M. Fernandez, V. Tamma, F. Lecue, P. Cudré-Mauroux, J. Sequeda, C. Lange, J. Heflin (Eds.), The Semantic Web – ISWC 2017: 16$^{\text{th}}$ International Semantic Web Conference, Vienna, Austria, October 21–25, 2017, Proceedings, Part II, Vol. 10588 of Lecture Notes in Computer Science, Springer, Cham, Vienna, Austria, 2017, pp. 297–313.

[31] B. De Meester, W. Maroy, A. Dimou, R. Verborgh, E. Mannens, Declarative data transformations for Linked Data generation: the case of DBpedia, in: E. Blomqvist, D. Maynard, A. Gangemi, R. Hoekstra, P. Hitzler, O. Hartig (Eds.), Proceedings of the 14$^{\text{th}}$ ESWC, Vol. 10250 of Lecture Notes in Computer Science, Springer, Cham, 2017, pp. 33–48.

[32] H. Mili, F. Mili, A. Mili, Reusing Software: Issues and Research Directions, IEEE Transactions on Software Engineering 21 (6) (1995) 528–562.

[33] A. W. Brown, K. C. Wallnan, Engineering of component-based systems, in: Proceedings of ICECCS 96: 2nd IEEE International Conference on Engineering of Complex Computer Systems (held jointly with 6th CSESAW and 4th IEEE RTAW), IEEE Comput. Soc. Press, 1996, pp. 414–422.

[34] C. M. Keet, A. Ławrynowicz, C. d'Amato, A. Kalousis, P. Nguyen, R. Palma, R. Stevens, M. Hilario, The Data Mining OPtimization Ontology, Journal of Web Semantics 32 (2015) 43–53.

[35] J. B. Buckheit, D. L. Donoho, WaveLab and reproducible research, in: Wavelets and Statistics, Springer New York, 1995, pp. 55–81.

[36] R. Tommasini, B. De Meester, P. Heyvaert, R. Verborgh, E. Mannens, E. Della Valle, Representing Dockerfiles in RDF, in: N. Nikita, D. Song, A. Fokoue, P. Haase (Eds.), ISWC 2017 Posters & Demonstrations and Industry Tracks, Vol. 1963 of CEUR Workshop Proceedings, CEUR, Vienna, Austria, 2017.

[37] D. Garijo, Y. Gil, A New Approach for Publishing Workflows: Abstractions, Standards, and Linked Data , in: Proceedings of the 6th workshop on Workflows in support of large-scale science - WORKS '11, ACM Press, 2011, pp. 47–56.

[38] B. De Meester, A. Dimou, The Function Ontology, Unofficial Draft, Ghent University – imec – IDLab (2016).
URL `https://w3id.org/function/spec`

[39] B. De Meester, A. Dimou, R. Verborgh, E. Mannens, R. Van de Walle, An Ontology to Semantically Declare and Describe Functions, in: H. Sack, G. Rizzo, N. Steinmetx, D. Mladenić, S. Auer, C. Lange (Eds.), The Semantic Web: ESWC 2016 Satellite Events, Heraklion, Crete, Greece, May

29 – June 2, 2016, Revised Selected Papers, Vol. 9989 of Lecture Notes in Computer Science, Springer, 2016, pp. 46–49.

[40] B. De Meester, A. Dimou, R. Verborgh, E. Mannens, Detailed provenance capture of data processing, in: D. Garijo, W. R. van Hage, T. Kauppinen, T. Kuhn, J. Zhao (Eds.), Proceedings of the First Workshop on Enabling Open Semantic Science (SemSci), Vol. 1931 of CEUR Workshop Proceedings, CEUR, 2017, pp. 31–38.

[41] B. Villazón-Terrazas, L. M. Vilches-Blázquez, O. Corcho, A. Gómez-Pérez, Methodological guidelines for publishing government linked data, in: Linking Government Data, Springer New York, 2011, pp. 27–49.

[42] F. Radulovic, M. Poveda-Villalón, D. Vila-Suero, V. Rodríguez-Doncel, R. García-Castro, A. Gómez-Pérez, Guidelines for linked data generation and publication: An example in building energy consumption, Automation in Construction 57 (2015) 178–187.

[43] L. Sauermann, R. Cyganiak, Cool URIs for the Semantic Web, Interest group note, World Wide Web Consortium (W3C) (Dec. 2008).
URL `https://www.w3.org/TR/cooluris/`

[44] R. T. Fielding, J. Reschke, Hypertext Transfer Protocol (HTTP/1.1): Semantics and content – content negotiation, Tech. rep., IETF (Jun. 2014).
URL `http://tools.ietf.org/html/rfc7231#section-3.4`

[45] R. Cyganiak, D. Wood, M. Lanthaler, RDF 1.1 Concepts and Abstract Syntax, Recommendation, World Wide Web Consortium (W3C) (Feb. 2014).
URL `http://www.w3.org/TR/rdf11-concepts/`

[46] R. Taelman, M. Vander Sande, R. Verborgh, GraphQL-LD: Linked Data Querying with GraphQL, in: Proceedings of the 17$^{\text{th}}$ International Semantic Web Conference: Posters and Demos, 2018.

[47] D. Berrueta, J. Phipps, Best Practice Recipes for Publishing RDF Vocabularies, Working group note, World Wide Web Consortium (W3C) (Aug.

2008).
URL https://www.w3.org/TR/swbp-vocab-pub/

[48] D. Berrueta, S. Fernández, I. Frade, Cooking HTTP content negotiation with Vapour, in: C. Bizer, S. Auer, G. A. Grimnes, T. Heath (Eds.), Proceedings of 4th workshop on Scripting for the Semantic Web 2008 (SFSW2008). Co-located with 5th European Semantic Web Conference – June 1-5, 2008, Tenerife, Spain, Vol. 368 of CEUR Workshop Proceedings, 2008.

[49] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, R. Van de Walle, RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data, in: Proceedings of the 7th Workshop on Linked Data on the Web, Vol. 1184 of CEUR Workshop Proceedings, CEUR, 2014.

[50] A. Bangor, P. Kortum, J. Miller, Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale, Journal of Usability Studies 4 (3) (2009) 114–123.

[51] J. Brooke, SUS: a 'quick and dirty' usability scale, Usability evaluation in industry 189 (194) (1996) 1.

[52] A. Armaly, C. McMillan, Pragmatic source code reuse via execution record and replay, Journal of Software: Evolution and Process 28 (8) (2016) 642–664.

[53] P. Heyvaert, A. Dimou, B. De Meester, T. Seymoens, A.-L. Herregodts, R. Verborgh, D. Schuurman, E. Mannens, Specification and implementation of mapping rule visualization and editing: MapVOWL and the RMLEditor, Web Semantics: Science, Services and Agents on the World Wide Web 49 (2018) 31–50.

[54] M. D. Smucker, J. Allan, B. Carterette, A Comparison of Statistical Significance Tests for Information Retrieval Evaluation, in: Proceedings of the

sixteenth ACM conference on Conference on information and knowledge management - CIKM '07, ACM Press, 2007, pp. 623–632.

[55] B. Derrick, P. White, Comparing two samples from an individual Likert question, International Journal of Mathematics and Statistics 18 (3).

[56] N. Dell, V. Vaidyanathan, I. Medhi, E. Cutrell, W. Thies, Yours is better!: participant response bias in HCI, in: Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems - CHI 12, ACM, ACM Press, 2012, pp. 1321–1330.

[57] S. Harris, J. Broekstra, L. Feigenbaum, DAWG: Test case structure, Working document, World Wide Web Consortium (W3C) (2001).
URL https://www.w3.org/2001/sw/DataAccess/tests/README

[58] S. Abou-Zahra, Evaluation and Report Language (EARL) 1.0 Schema, Working group note, World Wide Web Consortium (W3C) (Feb. 2017).
URL https://www.w3.org/TR/EARL10-Schema/