Journal of Big Data

RESEARCH Open Access

The LRA Workbench: an IDE for efficient REST API composition through linked metadata

Diego Serrano* and Eleni Stroulia

*Correspondence:
serranos@ualberta.ca
Department of Computing Science, University of Alberta, 2-32 Athabasca Hall, Edmonton, Canada

Abstract

The number of Web APIs for accessing information and services is continuously increasing, and yet, no tools exist to automate the time-consuming and error-prone process of invoking those APIs and composing their responses. The recent emergence of widely-adopted, standardized, Web-API description formats and the development of Linked Data technologies for data integration have motivated our work on the LRA (Linked REST APIs) methodology [1, 2]. LRA relies on RDF service specifications to automate the development process around the usage of Web APIs. This automation represents a great opportunity to systematize and improve the quality of service-oriented application development. However, LRA's reliance on SPARQL as the user-interaction model may hinder its adoption, because it requires developers to learn the intricacies of the unconventional graph data model and its associated datasets. In this paper we have developed the LRA Workbench (LRA_{Wbench}), which takes advantage of the emergent schema of Web-API specifications, in order to simplify the formulation of LRA-compliant SPARQL queries. Our empirical evaluation of the LRA_{Wbench} usability demonstrates that our tool significantly improves the performance of developers formulating SPARQL queries for LRA. A subsequent study on the effectiveness of the LRA_{Wbench} demonstrated that developers using LRA tend to produce code with considerable better structural complexity, in less time, than developers manually composing APIs.

Keywords: Intelligent web services and Semantic Web, Services integration framework, Service-oriented architecture

Introduction

The evolution of the Web has led to an increased adoption of Web APIs[1], revolutionizing the way end users and software systems access information and services. Web APIs have enabled companies to increase their revenues, by discovering and joining forces with complementors. For example, Salesforce generates 50% of its revenue through APIs, Expedia generates 90%, and eBay, 60% [3]. It is abundantly clear that Web APIs represent a huge potential for creating added value through service orchestration across providers. Nevertheless, the industry still relies on manual service-composition approaches, requiring a considerable development investment. Application developers need to examine large amounts of natural-language documentation in order to understand how

[1] Web API is a common way to refer to a programmatic interface, exposed via the web, and typically using a request-response messaging system.

to write client code for specific APIs, compose compatible APIs, and manage authentication credentials. And they have to do it all over again, when any of these APIs change. As a result, even though data can, in principle, be easily accessed through Web APIs, in many cases it still remains captive in isolated silos that do not interoperate with other resources and services on the Web.

Recently, the evolution of REST services[2] has led to widely adopted description formats, such as OpenAPI[3] and RAML[4], that specify relevant implementation details, including resources, status codes and input arguments, but completely ignore the underlying service semantics. In parallel, the development of semantic-representation languages has driven the creation of promising knowledge-description formalisms, such as the Linking Open Data (LOD) project [4], and *schema.org* [5], which demonstrate the potential of Linked Data approaches for data integration.

The synergy of these two popular technologies, namely standardized specifications for REST APIs and Linked Data semantics, has motivated the emergence of approaches for semantic annotation of Web APIs, such as JWASA [6], AutomAPIc [7], and our own work on Linked REST APIs (LRA) [1, 2], which attempt to simplify, and partially automate, the way developers interact with data provided from Web APIs.

The Linked REST API approach starts by describing a general conceptual framework for REST-service specification based on Linked Data models, and a corresponding software architecture for automatically composing API calls to respond to data queries, considering quality and access-control constraints. This work was subsequently formalized in [2], in which we defined the formal semantics of the LRA API specification, and the discovery and composition algorithms grounded on this formalism. We validated the usefulness and accuracy of the LRA composition algorithm in the context of several realistic use-case scenarios around a collection of publicly accessible Web APIs.

The LRA methodology entails the use of SPARQL: developers no longer browse Web API documentation but they declaratively specify the desired functionality in terms of a SPARQL query, and the LRA composition-and-enactment algorithm retrieves appropriate APIs from multiple providers and automatically composes them in a data production-consumption graphs. SPARQL has been shown to be a powerful tool in the hands of experienced users, but is considered as difficult to understand for average application developers. In fact, many authors acknowledge that the Semantic-Web technologies are only accessible to highly-trained experts, offering benefits often considered unworthy of the additional effort investment [8–11]. Thus, in spite of the realism of our evaluation methodology and its promising results, we recognize that LRA's reliance on SPARQL may hinder its real-world adoption. This is why, in this paper, we implement the LRA_{Wbench}, initially introduced in [12], that develops the LRA algorithms and provides a set of tools to support software development using LRA-specified APIs.

The LRA_{Wbench} is an integrated-development environment that streamlines the use of LRA, enabling developers to express their information needs without extensive

[2] REST service is an API that conforms to the design principles of the REST (Representational state transfer architectural style), based on HTTP methods to access resources and the use of JSON or XML to transmit data.

[3] https://www.openapis.org/.

[4] https://raml.org/.

knowledge of the LRA encoding formalisms, delegating the query formulation and interpretation process to a set of tools, that abstract SPARQL and RDF from the user. The LRA_{Wbench} builds upon previous research on visual-query formalisms for Linked Data systems, and guides the creation of query structures through the inference of a global schema of the data that can be provided by the Web APIs.

The creation of the LRA_{Wbench} led us to formulate our first research question: *Does the* LRA_{Wbench} *help developers to formulate LRA-compliant SPARQL queries?* To answer this question, we conducted an empirical study to assess the accuracy, efficiency, and perceived usability when asked developers to produce queries using the LRA_{Wbench}. Furthermore, we compared the performance of developers using the LRA_{Wbench} against that of developers using YASGUI, a popular SPARQL interface, used as front-end by a large number of Linked Data publishers. The results demonstrate that developers perceive the LRA_{Wbench} as considerably more usable than YASGUI.

Having confirmed the effectiveness of the LRA_{Wbench} in making SPARQL more easy to use, we proceeded to examine its impact in the software-development process, which led us to our second research question: *Does the use of the* LRA_{Wbench} *result in improved developer efficiency and code quality?* To answer this question, we conducted an empirical study, using the traditional hand-crafted endpoint request workflows as the industry baseline. In this study we measure the quality of the produced software artifacts, and the accuracy and efficiency of developers when asked to develop a *mashup* web application using LRA_{Wbench}. We compared the quality of their code and their performance with that of developers using the traditional approach where the developer manually designs a workflow that submit requests directly to the service endpoints. We found that, when implementing service compositions, LRA developers produce significantly less complex code, when compared to developers using classic development environments. Additionally, the results show that LRA developers spend less time to code the solutions than developers using the traditional approach. For simple composition workflows, namely service discovery, we identified that the performance and code complexity of developers using LRA is comparable to that of developers using the traditional approach. Our study also revealed that the overhead imposed by the matchmaking at runtime of LRA is negligible.

The remainder of this paper is organized as follows. "The linked REST APIs approach" section introduces the methodology used by LRA to automatically compose Web APIs. "Review of Linked-Data query systems" section presents a systematic survey on Linked-Data query systems, used to identify the desired features of our development environment. "The $\mathrm{LRA}_{\mathrm{Wbench}}$" section introduces a detailed description of the proposed query environment, and "Usability evaluation of the $\mathrm{LRA}_{\mathrm{Wbench}}$" section discusses the protocol of our usability study of the LRA_{Wbench}. "Effectiveness evaluation of the $\mathrm{LRA}_{\mathrm{Wbench}}$" presents the empirical evaluation of the LRA environment, shows the results of the study, and discusses our findings. Finally, "Conclusions" section summarizes the contributions of our approach.

The linked REST APIs approach

This section briefly describes the LRA approach for Web API semantic description and automated discovery and composition. The Linked REST APIs define a data model and a composition methodology that delineate the requirements and constraints implemented in the development environment in "Review of Linked-Data query systems" section. Interested readers should look at [1, 2] for a detailed overview of the LRA approach.

In our previous work [1], we introduced the concept Linked REST APIs, a middleware for REST-service integration based on Linked Data models, that brings together a long history of web-service description formalisms into a concise Web API description model. In LRA, the data exposed by REST services is mapped to Linked Data schemas, and based on those descriptions, a middleware can automatically compose API calls to respond to a family of SPARQL queries, called *well-designed graph patterns* [13]. In addition, the LRA model can describe the access-control protocols of the said APIs and the quality of the data they expose, enabling the creation of "legal" compositions with desired levels of quality. As a consequence, LRA can potentially change the interaction paradigm from the manual formulation of procedures, to the automatic definition of compositions based on graph matching between queries and available service operations.

Conceptually, the LRA model considers the data provided by Web APIs as RDF graphs under the control of external service providers. LRAs are described by the schema of the data they serve, expressed as SPARQL graph patterns. A Linked REST API is specified as a set of operations, each defined by a 6-tuple $S = (E, P, In, Out, A, T)$ as follows: endpoint information, a graph pattern, inputs, outputs, authentications mechanism, and quality attributes, respectively.

In order to leverage the LRA descriptions, SPARQL is used to mediate the integration of heterogeneous sources. Therefore, a query graph pattern[5] is used to find service operations that can provide all or part of the answer, by matching subgraphs of the query graph against subgraphs of the graph patterns defined in the LRA operations.

Once the query has been defined, the LRA composition problem is formulated as that of finding a sequence of API operations $S = \langle S_1, \cdots, S_n \rangle$, such that: (a) each service operation $S_i \in S$ has a non-empty mapping with the query, (b) each required query triple is mapped by at least one of the operations (c) the variables projected in the SELECT clause are included as the output in one of the service operations, and (d) the service operations, when executed in sequence, form compatible subgraph matches, where a match is considered compatible if the service operation can be executed with the inputs available in the query. A detailed description of the data model and the composition algorithm can be found in [2].

Although the preliminary results for performance and accuracy of LRA are encouraging, the early adopters of the approach reported difficulties formulating SPARQL queries, despite they are well-versed in similar technologies, such as SQL. Unfortunately, while powerful and expressive, structured query models like SPARQL are fundamentally difficult for users to adopt because they require users to learn a new data model and to

[5] LRA queries are restricted to well-designed graph patterns, as defined by Perez et al. [13]

comprehend the structure of the dataset, in order to express queries in terms of that particular structure. We argue that, while proficiency in SPARQL is crucial for an optimal interaction with LRAs, a higher-level query system is needed to ease the learning curve and allow developers to formulate queries in a natural way.

Review of Linked-Data query systems

Supporting users with the task of formulating SPARQL queries has long been recognized as an important research objective, and a number of methods have been proposed to address this problem. Hence, in this section we analyze the previous work on Linked-Data Query Systems, in relation with the requirements specified by the LRA middleware, and then, based on the most auspicious characteristics, we design a development environment that abstracts the complexity of querying Linked REST APIs. The development environment is evaluated through an empirical study that reveals the effectiveness of developers using our IDE in the formulation of LRA-complaint queries.

In order to guide the design of a development environment that enhances accessibility to LRA, we conducted a survey of the literature and comparatively analyzed the most prominent works on Linked-Data query systems, with a particular focus on the more recent contributions. We performed a systematic search of "Linked Data query systems" in the ACM Digital Library, IEEE Xplore, Google Scholar, Google Search, and Springer Link. We further retrieved relevant references cited by the works found through the first search. We found 17 highly relevant publications, summarized in Table 1. In this section, we organize our review of the main features of these systems around five key design dimensions: (a) user interface, (b) query interpretation, (c) usability, (d) structural-discovery support, and (e) results rendering. To the best of our knowledge, this is the most comprehensive study of Linked Data query systems.

User interface

Query interfaces typically rely on the capacity of the human perceptual system to draw inferences based on symbols and their properties, enabling the interpretation of concisely encoded, large quantities of information. The user interfaces of the existing systems are organized into four categories.

Text editors are a natural choice for software developers accustomed to writing code in programming-language IDEs and relational-database clients. These editors are typically used by large dataset providers, such as DBpedia[6] and Bio2RDF[7], and can be employed to compose queries using SPARQL directly. Some of them, such as OpenLink Virtuoso [14], provide only a simple text box for the user to type a complete SPARQL query. Others, such as YASGUI [15], offer syntax-directed editing features, including autocompletion, syntax highlighting, and error checking. Although this type of interfaces have been adopted as the de-facto standard interaction model in triple stores, *several authors recognize that textual interfaces are generally inaccessible to typical software developers* [16, 17], because in order to formulate semantically meaningful queries, users need to know, not only the structure of the data, but also the syntax of the query language.

[6] Available at http://dbpedia.org/.

[7] Available at http://bio2rdf.org/.

Table 1 Linked Data query systems description matrix

	User interface				Interpretation			Structure			Usability		Rendering			Available	Active
	Text	Form	Visual	NLI	Navigation	Composition	Approx.	Browser	Step	Path	Casual +	Expert *	List	Table	Diagram		
Virtuoso (2006)	×					×								×		×	×
Querix (2006)				×			×				1		×				
iSPARQL (2007)			×			×								×		×	
NLP-Reduce (2007)				×			×				1	3		×		×	
gFacet (2008)		×	×			×			×					×		×	
K-Search (2008)		×			×			×	×		2	2		×	×		
MashQL (2008)		×	×			×		×	×					×		×	
Nitelight (2008)			×			×		×						×			
Ginseng (2010)				×			×		×		1	3	×				
Semantic Crystal (2010)			×			×		×	×		3	1		×			
SPARQL2NL (2013)				×			×						×			×	
SemFacet (2014)		×			×				×				×	×		×	×
SPARQLBuilder (2014)		×	×			×				×				×		×	
YASGUI (2014)	×					×			×					×	×	×	×
AffectiveGraphs (2015)			×			×			×		3	1		×			
QueryVOWL (2015)			×			×			×					×		×	
PepeSearch (2016)		×			×			×	×		2			×		×	×

+Numbers reflect the general rank, being NLIs the preferred approaches, followed by form-based and visual approaches

* Numbers reflect the general rank, being graph-based the preferred approaches, followed by form-based and NLIs

Form-based interfaces capitalize on common user-interface design patterns, adopted by most of modern computer systems, thus increasing the initial familiarity with users. In those approaches, the queries are specified through the use of text boxes, drop-down menus, check boxes, and other well known form elements. The systems typically follow a faceted search interface, enabling users to explore a data set by applying multiple filters. K-Search [18], SemFacet [19], and PepeSearch [20] are some of the systems adopting such a faceted-search interface. Some of them combine elements from other interaction models, such as SPARQL Builder [16] and K-Search that provide a graph visualization to illustrate the relationships among classes.

Although casual users prefer this type of interface over other visual approaches, building queries by exploring the search space through facets, as in form-based approaches, can be very time consuming, especially as the ontology gets larger and the query gets more complex [21].

Natural-Language Interfaces (NLIs) enable users to formulate factoid queries with wh-terms (which, what, who, when, where), or commands (give, list, show). Some representative examples of these interfaces are SPARQL2NL [17] and NLP-Reduce [22]. Due to linguistic ambiguities of natural language, some NLIs, such as Ginseng [23] and Querix [24] restrict the user input to a controlled/structured natural language, constrained in its grammar and lexicon. As a result, although such "almost natural" language may be perceived as more intuitive than other approaches, *the restriction on the acceptable language constructs limits expressiveness and imposes cognitive challenges on the users.*

Visual approaches adopt a diagrammatic language, consisting of geometric shapes, textures, and connections between them, to convey the query semantics. The systems in this category vary depending on their adopted graph data model. For example, iSPARQL [25], Nitelight [26], Affective Graphs [27], Semantic Crystal [23] and QueryVOWL [28] represent RDF triples using nodes for subjects and objects, and links for the predicates that connect them. gFacet [29] uses a UML-like representation of classes with their properties inside a box. MashQL [30] uses visual pipes to represent the flow of data, where nodes define filter functions and edges depict the flow of data. Some of the node representations take concepts from form-based interfaces, allowing users to define properties through embedded form elements, such as check boxes and drop-down lists, as in the case of gFacet and MashQL.

Past usability studies have shown that *expert users prefer graph-based interfaces, due to their reduced complexity and high expressiveness* [21]. Accordingly, LRA's Visual Query Assistant (VQA) adopts a visual-query interface based on a graph model, where the direct mapping between the visual syntax and the underlying query language may facilitate an eventual transition to text-based interfaces.

Query interpretation

Query interpretation is the process of translating the expression constructed through the system's interface, to a request into the formalism used by the data store. The interpretation approaches are organized into three categories.

Navigation approaches present concepts with property values of focal concepts, logically organized in groups. By iteratively selecting subsets of these values, a query is created, requesting the refinement of the data of interest. Form-based approaches, such as K-Search, SemFacet, and PepeSearch, fall under this category.

In this category, the query-language expressiveness is limited to the direct properties of the concepts in focus. *In order to formulate complex queries, users have to navigate through several concepts and properties, which has been identified as more laborious to use than other approaches* [21].

Composition approaches present an unambiguous mapping from the configuration of elements in the user's input to the underlying formal query language. In the most trivial case, a query composed in a text editor is submitted directly to the data store, resulting the highest possible expressiveness for the query interface. Graph-based approaches typically translate nodes and edges into RDF resources and predicates respectively. However, representing certain constructs (e.g. filters, functions, or disjunctions) may not be supported, or may lead to the inclusion of graphic elements that are not intuitive for the user. *Despite this potential loss of expressiveness, previous studies remark on the good capabilities of visual languages to formulate complex queries* [21]; *for this reason VQA adopts a composition approach for its query-interpretation step*.

Approximate mapping is based on the idea of inferring the semantic structure of the query, by applying NLP techniques such as named-entity recognition. These approaches have varying levels of complexity, ranging from naïve techniques that use minimal natural-language processing (e.g. NLP-Reduce) to more sophisticated approaches that can generate triple structures involving disjunctions and aggregations (e.g. SPARQL2NL). *Although these approaches lead the user to believe that the system fully understands the input language, only a subset of the language is correctly identified by the system, which is commonly referred in the literature as the 'habitability problem'* [31]. This results in low response accuracy, or forces these systems to limit the expressiveness of the queries or to rely on large amounts of background domain knowledge.

Usability

Although the primary objective of the aforementioned tools is to support users to formulate SPARQL queries, only a few have addressed the perceived ease-of-use and effectiveness. The results of their usability studies can be analyzed taking into consideration two different types of users: *casual* and *expert* users.

Casual users are characterized by their lack training in formal query languages, and unfamiliarity with the details of the internal organization of the query system. Previous research has investigated the effectiveness of natural-language approaches (NLP-Reduce, Querix, Ginseng), and more formal approaches, such as form-based (PepeSearch) and visual interfaces (OptiqueVQS, Semantic Crystal, Affective Graphs) with casual users [23, 32, 33]. These studies indicate that casual users favor form-based approaches over graph-based visualization approaches, spending less time with the tool and finding a satisfactory trade-off between the complexity of the query and difficulty of the interaction. Despite their preference for form-based approaches over graph-based approaches, casual users seemed to find NLIs most intuitive, even when the language is restricted. However, the *habitability problem* of these interfaces causes dissatisfaction with the answers they return.

Expert users are the target users of LRA. They possess a broad set of skills on programming languages and database-management systems that are indirectly associated with Linked-Data query systems.

Elbedweihy et al. [21] conducted a study where they distinguished between casual and expert users when comparing five different approaches (NLP-Reduce, Ginseng, K-Search, Semantic-Crystal and Affective Graphs). In that study, the expert users preferred graph-based approaches, followed closely by form-based approaches, arguing that these approaches allowed them to formulate more complex queries than NLIs. Unlike casual users, the restrictions imposed by the restricted NLI approach were regarded as annoying, and were seen as an impediment to constructing expressive queries. Similar to casual users, experts found free-form NLIs, such as NLP-Reduce, more intuitive, but found the results unacceptable.

Structural-discovery support

Considering the difficulties involved in composing complex queries, some approaches provide assistance by generating suggestions, based on the structure of the data graph. The support offered by the systems to discover structural relationships are grouped into three categories.

Ontology browsers facilitate the query-formulation process by providing users with a starting point for their query, presenting the classes and relationships that constitute the 'schema' of the stored data. Approaches, such as PepeSearch and MashQL present simply a list of the top classes available in the dataset. Yet other approaches, such as K-Search, Nitelight, and Semantic Crystal, employ more sophisticated techniques and communicate information about the data ontology through tree-views and ontology graphs. Ontology browsers are quite useful for simple ontologies, but their visualizations do not scale as the ontologies become more complex [34].

Step-recommendation methods explore the neighboring classes and predicates that may be reached from a focus object, and suggest a list of plausible query elements. For example, YASGUI provides an autocomplete feature that predicts the rest of a predicate or class a user is typing, based on the prefixes added to the query. Visual approaches, such as QueryVOWL, also use autocomplete to search for available instances and accessible predicates, which subsequently insert nodes and edges in the graph visualization. The approaches that follow a faceted-search interface use the step-recommendation feature to create navigation links and produce filters. *VQA adopts a step-recommendation support mechanism, by showing the immediate predicates available from and to a focus entity, facilitating the exploration, which has been detected as one of the strengths of form-based approaches.*

Path discovery helps to find connections between two resources, where an immediate relationship may not exist. This strategy attempts to find paths between resources, by exploring the data graph and obtaining chains of properties that connect the two resources. From the approaches reviewed, only SPARQLBuilder proposes a path discovery methodology, by preprocessing the datasets and extracting the necessary metadata to construct a class graph, which is then used to find all the possible paths between two classes[8]. *VQA improves upon previously proposed approaches by incorporating semantic knowledge into the emergent schema and ranking the paths based on proximity.*

Results rendering

The presentation of the information returned is also an important component of the query system. Elbedweihy et al.[21] found that organizing answers in a table or having a visually-appealing display has a direct impact on results readability and clarity and, in turn, user satisfaction. Accordingly, query systems usually present the results by means of a table.

However, taking into account the context of the data, an appropriate diagram of the query results, such as a map or a timeline, would allow users to better capture the relationships amongst the output data. In the reviewed systems, results from K-Search can be visualized as 2-D charts, and, YASGUI incorporates YASR [15], which supports visualization via Google Charts. *VQA uses a combination of graphic and tabular representations, which has been found to yield a better 'view' of the data and therefore produce more accurate comprehension of it* [35].

The LRA$_{\text{Wbench}}$

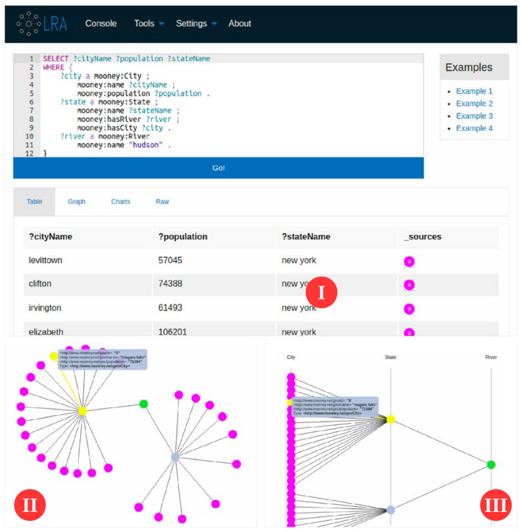
This section presents the development environment and the components that facilitate the automation of discovery, composition and enactments of external web services, provided by LRA.

The Query Console

For experienced users, direct manipulation of SPARQL queries is provided through a Query Console (Fig. 1a), similar to other SPARQL query editors. The Query Console provides the following three alternative representations of the responses: (a) a table renderer for the projected variables in the SELECT clause, along with a column that specifies the data sources of the row (Fig. 1a-I); (b) the raw JSON-LD response as it would be consumed by third-party applications using LRA as a service; and (c) graph visualizations that represent entities as nodes, and their relationships as edges (Fig. 1a-II and a-III).

The graph visualizations assist users in their exploration of the returned result. The first visualization offers a *force directed layout* (Fig. 1a-II) to produce visual densities that facilitate the detection of clusters. This representation can be considered as a direct analogy to the underlying data model, which may help developers to get familiar with the graph data model. The second visualization represents a *parallel-coordinates layout* (Fig. 1-III), which is a visualization technique suitable for exploration and analysis, where each dimension represents a class of entities, and the nodes in the result are assigned to one of the classes. Parallel coordinates have been found to be effective in exploratory data analysis, like in the diagnosis of marginal densities, correlations among attributes, and clustering [36].

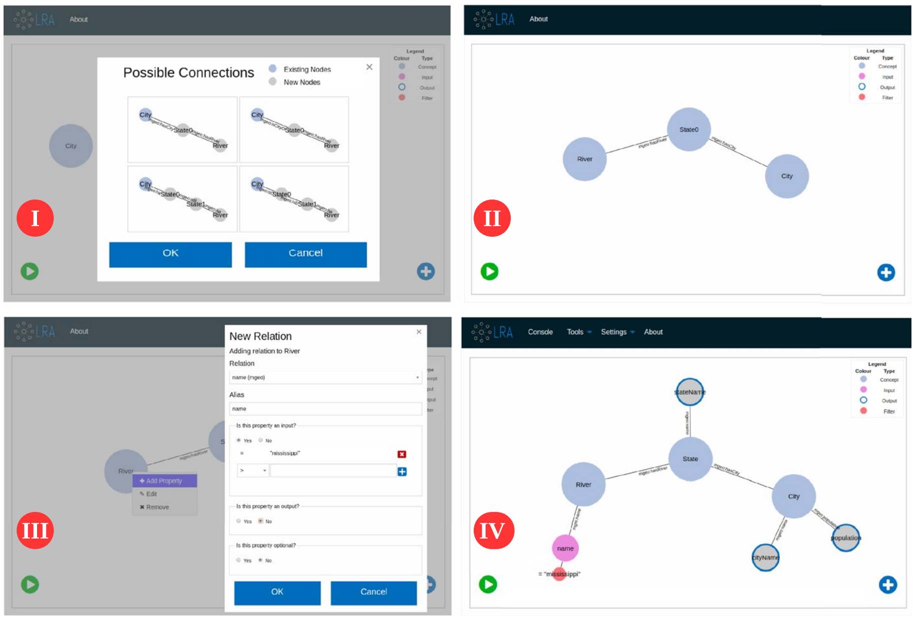
[8] It is worth noting that SPARQL Builder does not provide a reference implementation or a demo of their path-discovery interface.



```
SELECT ?cityName ?population ?stateName
WHERE {
    ?city a mooney:City ;
        mooney:name ?cityName ;
        mooney:population ?population .
    ?state a mooney:State ;
        mooney:name ?stateName ;
        mooney:hasRiver ?river ;
        mooney:hasCity ?city .
    ?river a mooney:River
        mooney:name "hudson" .
}
```

?cityName	?population	?stateName	_sources
levittown	57045	new york	0
clifton	74388	new york	0
irvington	61493	new york	0
elizabeth	106201	new york	0

(a) LRA Query Console.



(b) Visual Query Assistant.

Fig. 1 LRA development environment

The Visual-Query Assistant (VQA)

In this section, we describe the visual model used by VQA in order to provide the semantics to support an effective semantic matching between developers' information needs and the LRA model.

Graphical notation

VQA's notation consists of symbols and rules for connecting them, following the triple-based structure of RDF models, representing entities and literal values as nodes, and predicates as links, thus establishing a direct correspondence between visual query expressions and the underlying SPARQL language. We hypothesize that this mapping between the query-editing graphical notation and the SPARQL syntax may support learning which, may eventually enable the users' transition to using the textual SPARQL editor.

The design of the Visual Query Assistant is guided by the three key laws of Gestalt Theory in Visual Screen Design [37]: simplicity, similarity, and proximity. The *simplicity* law in VQA is represented by its visual-language constructs, composed by circles representing nodes, and lines representing links among such nodes, each of them with a label that differentiates its role in the query. This prevents the information overload of other approaches, such as AffectiveGraphs, which has a large number of combinations of distinctive features. The *similarity* law in VQA is depicted by the color and size of the circles, which distinguish class instances, properties and filters, as shown in Fig. 1b-IV. More specifically, class instances are represented with large blue circles, since they determine the main structure of the query; properties are represented as small circles, whose fill and stroke color change to indicate whether they are used as input (pink fill) or output (blue stroke) properties; and filters are smaller red circles attached to the filtered property, emphasizing with the color their importance for the result of the query. Approaches such as iSPARQL and Nitelight make all elements look alike, and fail to capitalize on the opportunity to visually distinguish them. Finally, the *proximity* law in VQA is represented by the link distances, which are strategically designed to group class instances and their properties into a congruent cluster, and keep other class instances and properties at a distance that enables users to perceive inter-class relationships, without interfering with the visual clusters. In contrast, most of the visual approaches, including iSPARQL and Nitelight, do not take into account the distance among elements as an important part of their graphic representations.

Emergent schema for structure discovery

VQA relies on the underlying structure of the data provided by the LRA services, which we call *emergent schema*, in order to build a model to assist users with no knowledge of SPARQL and the specific dataset structure. The *emergent schema* is a specialized graph, based on other approaches [16, 38], whose nodes and edges correspond to classes (or data types) and predicates, respectively.

Given a set of triples from LRA service descriptions, denoted as R, we define the *emergent schema* as a directed labeled graph $G = (V, E, \ell)$ such that, the set of vertices V represent classes and value types; the set of directed edges E of the form $l(u, v)$, where $u \in V, v \in V$ and $l \in \ell$, denote predicates between resources; and the labels ℓ are predicate names or labels. An edge of the form $l(u, v)$ represents the RDF triple (u, l, v). In order to extract the set of initial class nodes of the emergent schema, VQA adds all the classes c, such that c appears in triples of R of the form $(s, \texttt{rdf:type}, c)$. To obtain the set of initial edges, VQA adds all the edges $p(c_1, c_2)$ such that there exists three triples in R of the form $(s_1, \texttt{rdf:type}, c_1)$, $(s_2, \texttt{rdf:type}, c_2)$, and (s_1, p, s_2).

In addition, we noted that semantic properties, such as subsumption, may provide more efficient ways to represent the relationship between classes. For example, a developer querying scholarly APIs, instead of referring to the individual types of publications, like *blogs*, *papers*, or *patents*, may aggregate all the publications by referring to *documents*. Nevertheless, including all the superclasses of V would lead to superfluous classes and a cluttered user interface. VQA only includes the superclasses that cover more than one class from the initial set of classes. Then, for each pair of class nodes u and v, it adds their lowest common ancestor in the class hierarchy to the set of classes V. Accordingly, the added superclasses will take part in the edges where their subclasses have relationships. More formally, VQA adds an edge $l(c'_1, c'_2)$ to the set E, if there exists an edge $l(c_1, c_2)$ such that $c_1 \subseteq c'_1$ and $c_2 \subseteq c'_2$, where $c \subseteq c'$ indicates that class c is a subclass or the same as c'.

Once the emergent schema has been constructed, when a user adds a new class instance in the VQA query graph, the system computes the possible relationships between the new class and the existing classes in the query, by exploring the emergent schema. The relationship options are ranked giving priority to short path lengths, which intuitively emphasizes more direct relationships (Fig. 1b-I). This improves the interaction presented in other graph-based approaches, where the user required explicit knowledge of the relationships between classes. The selection of one of the options results in the creation of graph elements in the query (Fig. 1b-II), which then can be used to add properties to the elements (Fig. 1b-III and b-IV). Once a query graph has been created, an expression generator component traverses the graph and gradually constructs a SPARQL query that comprises all the restrictions defined in the graph.

Usability evaluation of the $\mathrm{LRA}_{\mathrm{Wbench}}$

We conducted an empirical study that seeks to address our first research question: *Does the* LRA_{Wbench} *improve the effectiveness of developers formulating LRA-complaint queries?* We decided to compare our tool against YASGUI, because, besides providing structural discovery support, syntactic assistance, and being actively maintained, it is one of the most widely used SPARQL query interfaces, being incorporated into RDF frameworks and projects, such as Apache Jena-Fuseki, HealthData.gov, and the Smithsonian American Art museum.

Study protocol

For the study, we recruited 20 software developers and computer-science students, with more than 1 year of experience in software development and SQL, and no previous experience in SPARQL. The participants were divided randomly in two groups of equal size: one group was assigned to use VQA, and the other was given YASGUI. The participants were trained through a series of interactive video lessons, based on material from [39].

In order to design a representative set of query-formulation tasks, we considered three factors: complexity of real-world queries, existing SPARQL benchmarks, and related usability studies. For the *complexity of real-world queries*, we examined the characterization extracted by Gallego et al. [40] from logs of the DBPedia and Semantic Web Dog Food public endpoints. For our study, we considered the query complexity indicators found by the authors: number of joins, type of operators, patterns of triples, and topology of the pattern. In addition, we considered *SPARQL performance benchmarks*, such as SP^{2}Bench [41], but their queries were discarded from our study, since they were not representative of the real-world complexity indicators mentioned above, and their domains may be too intricate to understand for our test subjects. We also reviewed *comparable usability studies*, such as [21, 23], and found they were based on a small dataset containing geographical information about the US[9], which also contains predefined English language questions. For our study, besides using the same dataset, we considered the questions that these studies selected and their SPARQL translation. The above considerations resulted in the following query-specification tasks.

1. *Which are capitals (name and population) of the USA?*
2. *What are the cities (names) in states through which the Mississippi river runs?*
3. *Which are the lakes (names and area, if available) in states bordering Minnesota?*
4. *Which rivers (name and length) run the states that border the state with the capital Atlanta?*
5. *Which states (names) have a city named Springfield with a city population over 100,000?*

Questions 1, 2, and 5 correspond to questions found in [21]. Questions 3 and 4 were formulated to make use of optional operators and complex relationships, representing the higher complexity observed in real-world queries, which have not been exercised in other usability studies.

Data collection

The queries were presented to the participants one at a time in the same order, and both tools allowed the execution of the queries, and presentation of results against the geographic dataset. Subjects were asked to formulate each of the five queries in turn, using the assigned tool's interface.

In order to compare the usability of the tools, we collected quantitative and qualitative data. The set of quantitative data we collected is based on information-retrieval and human-computer interaction literature, and includes: (a) the time required to formulate the query, (b) the number of attempts, and (c) the accuracy of the answer, represented by the F-measure that combines precision and recall. After the tasks were completed, we asked the participants to reflect on their experience and provide feedback using a free form text area for comments, and the System Usability Scale (SUS) questionnaire [42], a standardized usability test that has proven to be very useful when investigating interface usability [43].

The working sessions were instrumented with LimeSurvey[10], a popular open-source tool for online surveys. The survey presents one task at the time until all tasks in the questionnaire are answered. The application was configured to disallow participants to return to a task once it has been completed. The survey application measures the total time spent by the participant in each task. The total time is calculated as the time between a task is presented to the participant, and the time a final answer is submitted.

[9] The Mooney Natural Language Learning Data, available at http://www.cs.utexas.edu/users/ml/nldata/geoquery.html.

[10] https://www.limesurvey.org/.

Table 2 Results of the VQA usability evaluation

Criterion	Question	VQA	YASGUI	p-value
Time (s)	1	156	729	0.0017
	2	204	1077	0.0162
	3	334	842	0.0250
	4	193	518	0.0282
	5	119	534	0.0083
	Average	201	740	0.0048
F-measure	1	1.000	0.857	0.3559
	2	1.000	0.833	0.1723
	3	0.750	0.555	0.2274
	4	0.888	0.422	0.2033
	5	1.000	0.666	0.0781
	Average	0.927	0.695	0.0978
Num. Attempts	1	1.000	4.571	0.0323
	2	1.000	11.857	0.0111
	3	1.125	9.285	0.0151
	4	1.625	3.857	0.2033
	5	1.125	8.000	0.0690
	Average	1.175	7.257	0.0048
SUS	Average	79.062	55.357	0.0015

Each tool was instrumented to log the queries submitted to the system, which is then used to extract the number of attempts per task. The accuracy of the query was evaluated using the precision and recall of the returned records against the expected record set, which is summarized in the F1-measure.

Results and discussion

The results for both tools, VQA and YASGUI, are presented in Table 2. The first two columns present the evaluation criterion and the name of the query, the following two columns report the average results for VQA and YASGUI respectively, and the last column shows the probability values after applying a *t*-test, in order to evaluate if the difference between the two samples is statistically significant.

The results show that VQA users took significantly less time overall and required fewer attempts to formulate the queries than YASGUI users (95% confidence). In addition, queries formulated through VQA were more accurate than queries created with YASGUI, although the difference was not statistically significant. Furthermore, VQA presents higher usability ratings than YASGUI.

Our results are not directly comparable to the results of other graph-based approaches using the same dataset [21, 23], due to differences in experiment design and background of test subjects. Nevertheless, we found similarities in the required number of attempts and an increase in accuracy, perceived usability, and average time with VQA. Compared to these studies, however, there is a seemingly unfavorable increase in the average time that VQA users required relative to AffectiveGraphs users. We hypothesize that this may be due to the *learning effect* [44] phenomenon, which may have affected the aforementioned studies; in the above studies, the experimental protocol had participants answering the same question in several interfaces, significantly increasing the potential to become familiar with the ontology and the questions from one interface to the next, thus improving their performance over time, and reducing the overall average.

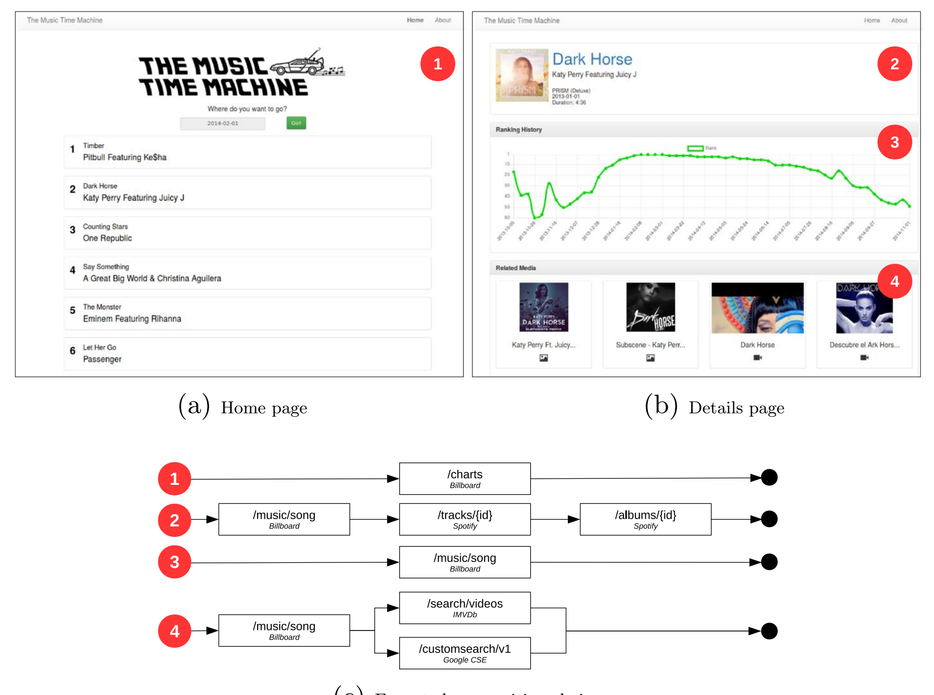
We also noted that the visual representation of the query facilitated the inspection of the semantic correctness of the query, which is one of the reason why VQA users required fewer attempts to formulate the query. On the other hand, many of the YASGUI users submitted erroneous queries (e.g., typos in variables, non-existing predicates), or created the query by incrementally executing partial query patterns. Most of the erroneous queries supplied by YASGUI users were due to inconsistent variable naming and absence of links between classes, which are two of the main assistive features in VQA.

The difficulty to formulate queries resulted in an increase in time for YASGUI users, when compared to VQA users. In the experiment, despite providing several examples and covering the necessary query constructs, at some point, some YASGUI users declared they felt frustrated devising syntactically and semantically correct SPARQL queries, especially when the query demands joining chain patterns. This also resonated in the accuracy of the queries, where VQA users achieved 92.7% and YASGUI users only 69.5%. Particularly, we noted that YASGUI users have a relatively good accuracy for simple queries (above 66% in queries 1, 2, and 5), but for the complex queries, the accuracy drops considerably below 55%, while for VQA users it remains above 75%.

In summary, VQA provides a favorable environment to introduce LRA developers into SPARQL, since it offers an interface that reduces the time to formulate a query, while maintaining high accuracy. Additionally, the interface restricts the compositions to valid SPARQL queries, that can be visually inspected for semantic errors, which increases the perceived usability of the tool. However, it is important to note that YASGUI provides a valuable user interface, that allows more flexibility and expressiveness, but requires significant experience in SPARQL.

Effectiveness evaluation of the $\mathrm{LRA}_{\mathrm{Wbench}}$

After evaluating the user interface of the development workbench, we turned our attention to assessing the usability of LRA as a framework that automates the process of reusing services in software applications and reduces the manual work required by software developers. Particularly, we conducted an empirical study to answer our second research question: *Does the use of the LRA environment result in improvements of developers' efficiency and code quality?* In this study, we hypothesize that enabling developers to retrieve data from multiple Web APIs by using a declarative language can significantly improve not only their performance, but also the quality of code artifacts. In order to determine the differences in performance and quality, we designed an evaluation study around a realistic use case of software-engineering teams building applications that integrate multiple web services, using the approach followed by Brueckmann et al. [45]. In this use case, a developer, who has previously identified a set of relevant service providers, formulates a service workflow that produces functionality needed in the application under development.


The Music Time Machine Home About

THE MUSIC TIME MACHINE

Where do you want to go?

2014-02-01 Go!

1	Timber Pitbull Featuring Ke$ha
2	Dark Horse Katy Perry Featuring Juicy J
3	Counting Stars One Republic
4	Say Something A Great Big World & Christina Aguilera
5	The Monster Eminem Featuring Rihanna
6	Let Her Go Passenger

The Music Time Machine Home About

Dark Horse
Katy Perry Featuring Juicy J
PRISM (Deluxe)
2013-01-01
Duration: 4:36

Ranking History

Related Media

Katy Perry Ft. Juicy... Subscene - Katy Perr... Dark Horse Descubre el Ark Hors...

(a) Home page (b) Details page

(c) Expected composition chains

Fig. 2 The Music Time Machine (MTM) test application

The use case involves the implementation of a test web application, called *The Music Time Machine* (MTM). In essence, MTM shows detailed information about songs that were popular at an specific date, provided by the end-user. The application is similar to other applications, such as The Billboard Hot 100[11], My Birthday Hits[12], or Playback.fm's Birthday Song[13].

The MTM application comprises 4 software components. The first component (T_1) contains a date picker and a button, which loads the top-10 songs at the specified date (Fig. 2a-1). The second component (T_2) shows information related to the song, such as the title, artist, release date and more (Fig. 2b-2). The third component (T_3) presents a line graph showing the ranks of the song over time (Fig. 2b-3). And the fourth component (T_4) displays links to images and videos related to the song or the artist (Fig. 2b-4).

The data for the components is provided by 4 data sources. *Billboard*[14] provides historical information about song ranks, *Spotify*[15] contains detailed information of songs, albums, and artists, *IMVDb*[16] contains data about music videos, and *Google CSE*[17] was configured to provide images related to music. Additionally, the provided LRA semantic

[11] http://www.billboard.com/charts/hot-100.
[12] https://www.mybirthdayhits.com/.
[13] http://playback.fm/birthday-song.
[14] http://billboard.modulo.site/.
[15] https://developer.spotify.com/web-api/.
[16] https://imvdb.com/developers/api.
[17] https://developers.google.com/custom-search/.

descriptions of the aforementioned data sources use *schema.org*, which has been widely adopted to annotate the semantics of music-related data for search engines.

In order to test the performance and quality of the application, we used the previously mentioned components, and the characterization of composition patterns proposed by Jaeger et al. [46], to create a set of tasks that aim at understanding how developers produce service compositions using different tool treatments. These tasks evaluate three prevalent composition structural patterns: discovery, sequential composition, and parallel composition. *Discovery* refers to binding a consumer to a single service that can provide all the required data. This pattern is used to retrieve data for T_1 and T_3. *Sequential* specifies a composition where the execution of services follows a logical order, and subsequent services depend on the results extracted from previous service invocations. Hence, the discovery pattern is also a special case of sequential pattern, whose length is limited to one service. The sequential structural pattern corresponds to the workflow in T_2. Finally, *Parallel* refers to compositions where the result of one or more independent composition chains are eventually merged. More specifically, this pattern corresponds to a parallel flow with AND split and join [46]. This pattern is used for T_4. Figure 2c shows diagrammatically the workflows for each component.

Study protocol

The study involved 32 students of Computing Science at the University of Alberta. All participants were enrolled in the fourth-year course "Software Process and Product Management", which includes lessons on service systems. This course also includes lab sessions, where the students apply the concepts learned in class by incrementally developing software applications. The lab sessions account for 5% of the final course grade, thus participation in the sessions is mandatory; however, the students were asked for their consent prior to the data collection for our study.

All of the participants reported having used the programming language (Javascript). The reason to use a single programming language is to reduce the variability in structural complexity that may be attributed to the inherent properties of the language model. Furthermore, none of the participants had experience with the selected Web APIs prior to this course, which does not give an unfair advantage to some of the participants.

The protocol of the study was divided in three stages:

Stage 1 (Preparation sessions) involved two 2-h leveling sessions covering topics related to version control systems, front-end programming, and service-oriented architectures. The students worked in pairs, which is a regular practice in the course, and has demonstrated higher effectiveness in industrial and academic scenarios [47, 48]. At the end of this stage, all the students had a working version of MTM that uses a local Web API as the data provider, in such a way that subsequent stages can be focused on data management and integration, instead of the presentational aspects of the application.

Stage 2 (Training session) introduced the LRA approach, including a description of the semantics in the API specifications. This session included a laboratory workshop that provided hands-on experience with the LRA environment, where T_1 is used to illustrate the differences between the traditional approach and LRA. Then, the students were asked for their voluntary participation, emphasizing that their consent to participate has no repercussions on the course grades. At the end of this session, the participant pairs were divided randomly in two groups of equal size, one of them assigned to use LRA, and the other to traditional HTTP requests to individual service providers.

Stage 3 (Working session) involved a session with a maximum time restriction of 2.5 h. This session was instrumented by a survey that presents one task at a time, in the same order, until all tasks in the questionnaire are answered. The linear questionnaire strategy followed in our study has been used in similar state-of-the art software development and comprehension studies, such as in [49, 50]. In the survey, the students were asked to submit the commit hash, in order to have a history that allows to audit the times and code changes. After the tasks were completed, the software repositories were collected, and the participants were asked to reflect on their experience using the SUS questionnaire, like in our previous study.

Data collection

In order to asses the performance of developers and the quality of the code, we collected the following quantitative data: (a) the time required to develop the task, (b) the execution time to render the task component, (c) the size of the response, (d) the structural complexity metrics derived from the code, and (e) the accuracy of the answer.

The *development time* for each task is calculated as the time between a task is presented to the participant, and the time an answer is submitted. The *size* of the response is the total amount of bytes transferred to the client application from direct service calls. The *execution time* for each task is calculated as the page load time with only the task component, where each measurement corresponds to the average time of five load times, and the first two load times were discarded. For the collection of execution times, the LRA framework was configured to perform exploration of maximum length of $l = 3$, and caching of query plans and service responses were disabled.

Since the variations of partial implementations is considerable, rather than try to enumerate them all, we designed a 3-level scoring mechanism to evaluate the accuracy. Each task assigns a score of 1 if all the entities and their attributes are included in the answer; 0.5 if some, but not all the entities and their attributes are included in the answer; and 0 if the task was not implemented or does not return the expected answer. This approach provides ease of explanation of the grading rubric, while allowing for variability in participants' solutions. It is important to note that participants are capable of identifying goal states correctly, since the only difference with the application created in Stage 1 is the data source.

Structural complexity metrics are based on the program's intrinsic attributes, such as the syntactic structure and program size. In this category, lines of code, Halstead's software science [51] and McCabe's complexity [52] are the most popular complexity metrics. Although some authors have questioned the consistency of the aforementioned metrics [53], many researchers accept these attributes as significant indicators of effort, maintainability, and defectibility [54–58], and they are still widely used in industrial scenarios, and have been incorporated in popular IDEs, like Visual Studio and Eclipse. We excluded from this study the object-oriented complexity measures, such as depth of inheritance tree, since the tasks were planned in a way that answers do not require

Table 3 Results of the effectiveness evaluation of LRA

	LRA			Traditional			p-value		
Criteria	T_2	T_3	T_4	T_2	T_3	T_4	T_2	T_3	T_4
LOC	13.25	16.60	16.80	30.00	15.67	31.80	1.0E−6	1.1E−1	2.4E−3
Vocabulary	45.25	45.20	44.20	51.88	36.50	55.00	4.7E−5	5.7E−4	4.9E−3
Length	107.00	100.40	106.60	193.00	88.67	205.00	4.0E−6	9.5E−3	7.0E−3
Volume	588.59	542.24	582.94	1100.18	460.34	1188.63	5.0E−6	5.7E−3	7.6E−3
Difficulty	13.79	14.00	14.19	16.64	13.22	16.71	3.5E−4	5.0E−2	8.1E−2
Effort	8117.51	7599.75	8269.45	18415.75	6090.06	20369.58	3.5E−5	6.1E−3	1.9E−2
Est. Bugs	0.20	0.18	0.19	0.37	0.15	0.40	5.0E−6	5.7E−3	7.6E−3
Cyclomatic	2.00	2.00	2.00	3.75	2.00	3.60	1.3E−5	-	2.8E−3
Maintainability	55.87	53.99	53.67	46.05	55.03	45.40	2.0E−8	3.6E−2	7.9E−4
Accuracy	1.00	0.63	0.56	0.81	0.75	0.50	1.9E−1	6.1E−1	7.9e−1
Size (B)	3389.88	3220.20	6654.00	3273.63	692.00	12440.80	9.2E−1	1.0E−7	9.3E−2
Exec. Time (s)	1.63	1.61	1.82	1.58	1.56	1.79	4.9E−4	1.0E−1	3.7E−1
Dev. Time (min)	40.25	13.40	8.00	66.00	7.00	55.00	5.9E−3	9.9E−2	4.2E−3
SUS	42.68			59.56			0.020860		

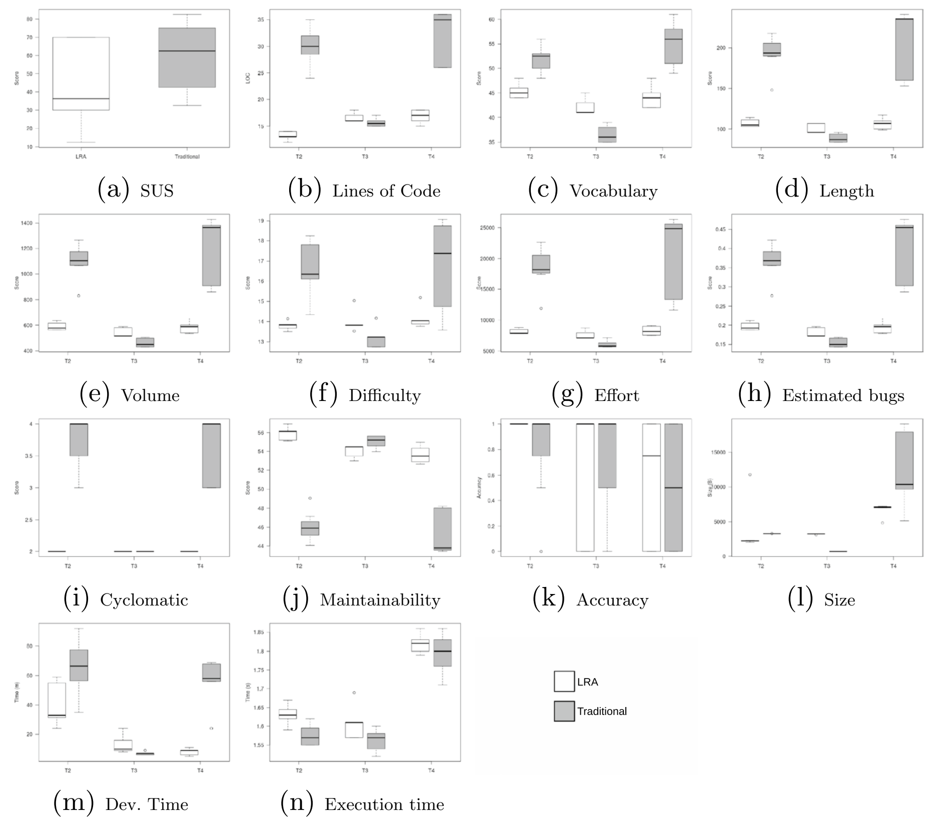


Fig. 3 Boxplots representing the data analyzed for the effectiveness metrics

significant considerations regarding the object-oriented design. In addition, we also excluded extrinsic metrics, such as readability, since they depend on individual style and preference, and in most of the cases can be avoided. In summary, the complexity metrics considered in this study are: effective lines of code, Halstead's software science [51] (vocabulary, length, volume, difficulty, effort, and estimated bugs), cyclomatic complexity [52], and maintainability index [59].

Results

In Table 3, we present the results of the performance and complexity metrics for each task. The first column presents the evaluation criteria, and the following two sets of columns show the average of each task using the two approaches. The last set of columns present the probability values after applying a *t*-test, in order to evaluate if the difference between the two approaches is statistically significant. Furthermore, Fig. 3 presents fourteen box and whisker diagrams that portray the distribution of the recorded measurements for each task in both approaches under consideration. Since T_1 was used to illustrate the differences between LRA and the traditional approach, the data collected for this task was not used in the analysis.

The task T_2 seeks to explore the differences of the approaches under a sequential service composition pattern. The second component ideally combines one operation from *Billboard*, and two from *Spotify*. The results show that all the structural complexity measures favor the LRA approach with statistical significance. For example, the lines of code used with the traditional approach averaged 30 lines, while LRA only required an average of 13.25 lines. The time to develop the functionality for the second task was significantly lower for LRA developers, showing an average of 40.25 min, in contrast with the 60 min spent in average by developers using the traditional approach. The execution time and response size give an advantage to the traditional approach, but the difference is not considerable.

The goal of task T_3 is to investigate the differences of the traditional and LRA approach, in service discovery, which is the basic operation of service-oriented systems. In principle, this operation selects and invokes one single service from *Billboard*. When performing this task, although there were no considerable differences regarding structural complexity, the traditional approach showed slightly less complexity. For example, the average estimated bugs in the traditional approach is 0.15, while in LRA is 0.18. In terms of performance, only the difference in size of the returned response was statistically significant, with an average of 3220 bytes for LRA, and 692 bytes for the traditional approach. This difference is caused by the overhead of LRA, which includes information about the request, such as status codes, query URLs, and detailed description of the services invoked, while the service response from Billboard is concise in its format.

The task T_4 investigates the differences of the approaches under a parallel service composition pattern. This composition is designed to request information from *Billboard*, and then split its execution flow into requests to *IMVDb* and *Google CSE*, which is subsequently merged. The results for this task are similar to the results for the task T_2. The structural complexity measures significantly benefit software artifacts created using LRA, and although there are differences in execution time and response size, the difference does not seem considerable. Likewise, the development time was significantly lower for LRA developers, with an average of 8 min, as opposed to the 55 min in average spent by developers using the traditional approach.

It is important to note that there were no statistically significant differences in accuracy. However, there is a marginally higher accuracy in favor of LRA developers, with an average of 0.73, when compared with developers using the traditional approach, who averaged 0.69. When only correct implementations are considered, LRA developers properly implemented 17 (out of 24) components, while traditional developers implemented 15 components.

In the subjective evaluation, instrumented with the SUS questionnaire [42], the results show that the traditional approach outperformed the LRA approach significantly. The average usability score of the traditional approach was 59.56, while for the LRA approach it was 42.68, which are are considered as *OK* and *Poor*, respectively, according to the adjective ratings introduced by [60].

Discussion

The results of this study show that complex workflows, involving service compositions, benefit from the structural complexity abstraction provided by LRA, since software applications only need to invoke a single endpoint, which in turn, automates the dynamic binding of the consumer application to multiple service endpoints. Despite the overhead incurred due to the service composition at runtime of LRA, it does not seem to affect the performance considerably, in terms of execution time, when compared to the traditional approach.

Nevertheless, for simple workflows, involving only service discovery, the results indicate that the structural complexity and performance of the two approaches is comparable. This result can be explained by the similarity of the two approaches, when the task is simple, since both are reduced to a single request to an endpoint. In addition, we hypothesize that for our study, given the limited number of relevant Web APIs and semantic concepts, the task of browsing Web API documentations and finding a single service can be regarded as equivalent as scanning through the ontology documentation and formulating queries in a domain-specific (visual) language.

Likewise, LRA developers spent less time implementing the composition workflows, when compared to developers using the traditional approach, and comparable times for the discovery workflow. However, in this aspect, it can be seen that LRA developers take a considerable amount of time in the first task T_2, which is then amortized significantly for subsequent tasks, showing clear signs of the effect of learning to use the approach. While LRA developers spent in average 40.25 min in the second task, they spent only 13.4 and 8 min in the following tasks.

Nevertheless, the perceived usability of the traditional approach was considered higher than LRA. This phenomenon, where the benefits seem attractive, but end-users express opposition to new technologies, is known as innovation resistance, and has been studied extensively in the literature [61, 62]. Previous research found that resistance to technological changes by end users is to be expected, and that one of the causes may be the perceived lack of ability or skill to successfully perform a given task. In particular, the LRA developers stated they perceived the tasks as more difficult, because they had to learn about the use of new tools, such as VQA and *schema.org*, while the traditional developers did not have to use the new knowledge to solve their tasks. Hence, a successful implementation of LRA should consider appropriate strategies for dealing with different forms of endogenous and exogenous factors of resistance.

Limitations

The limited number of participants and their limited expertise on web-based technologies is a concern for the external validity of the study. However, this study was conducted with a pool of participants with significant programming experience, and an intensive training in service-oriented architectures. In addition, the relative inexperience of the participants with the particular technologies used in the study guarantee that preconceived bias towards the traditional approach do not result in biased-blocking effects on the technologies under evaluation [63].

We are aware that the learning curve of the approaches under consideration may impact the developers performance. In order to minimize the impact of this threat to validity, we included stages for preparation and training in our protocol, which provided hands on experience with the development environment.

Participants were not allowed to return to a task once it was completed, which might fail to account for the exploratory and non-linearity nature of software development. Limiting our survey application to a strictly linear answering mechanism was motivated by our desire to precisely measure the time spent by developers implementing individual tasks. We minimize this threat to validity by thoroughly exploring the case studies during the preparation and training stages, and by designing the development tasks in such a way that coupling is minimal. Although our participants did not report the need to reconsider previous answers, more sophisticated mechanisms are desirable to allow a more flexible answering strategy, and increasing the generality of our results. The linear questionnaire strategy followed in our study has been used in similar state-of-the art software development and comprehension studies, such as in [49, 50]

The study was divided in three stages in a span of 3 weeks in order to minimize the fatigue of developers. Additionally, our protocol was designed to gradually increase the familiarity with the system, allowing participants to review the training material in between sessions. Finally, LRA's user interface had been tested previously, as described in "Usability evaluation of the $\text{LRA}_{\text{Wbench}}$" section, which allowed us to improve accessibility and intuitiveness of the users' interactions with LRA queries.

The test application used in this study was desgined and developed for research purposes. Although we identified several real-world applications that provide similar functionality, we can not claim that the results in this study can be generalized to industrial ecosystems. Considering the scope of our research, we believe that the results of this study provide substantial insights on how developers produce applications that integrate data from multiple service providers, via Web APIs.

Even though the number of tasks considered in the study are limited, they investigated the performance of developers dealing with the most common structural patterns. Indeed, the results of this study can be generalized to the performance of developers dealing with simple service discovery, as well as in non-trivial service compositions.

Conclusions

This paper builds on the previously proposed conceptual framework of Linked REST APIs, a methodology for semantically specifying and automatically composing Web APIs [1]. The LRA methodology aims to move the process of web-service composition from the current state, where interpretation of natural-language service documentation and manual programming of client code for service invocation are the norm, to a new model-driven engineering process, relying on declarative RDF-annotated, REST-service specifications. This methodological shift necessitates the use of SPARQL to reason about these RDF-annotated, REST-service specifications, and, in spite of the process-efficiency and system-quality improvements that it promises, the use of SPARQL as the query language creates barriers to the adoption of the LRA approach for traditional software developers.

This need, to mitigate the challenges that software developers face when they have to use SPARQL, has motivated the development of the LRA_{Wbench}. This tool exploits the underlying graph model implied by RDF and infers an *emergent schema* of the dataset to present to developers the data elements and relations, relevant to their current focal element. Accordingly, developers create SPARQL queries, corresponding to their desired composition, through the exploration of possible connections between query entities, in the emergent schema.

In this paper, we have described the $LRA_{Wbench}$and we have reported on two studies designed to evaluate (a) its usability and (b) its effectiveness in supporting software development. We first designed a collection of tasks that aim at evaluating the performance of developers when producing LRA-compliant queries. We evaluated the developers' performance in terms of three objective measures, namely (a) accuracy, (b) the number of attempts, and (c) the overall time needed to produce a query. We also collected feedback from our participants on their perceptions of the system's ease of use. In all these measures VQA, the visual SPARQL-query composition tool of the LRA_{Wbench}, was found to be superior to YASGUI, and, in most of them, significantly so. Next, we examined the developers' performance and quality of their code when asked to create applications that retrieve and integrate data from multiple Web APIs. In this study, we examined the following five indicators: (a) the time required to develop the task, (b) the execution time, (c) the size of the response, (d) the structural complexity of the code, and (e) the accuracy of the answer. We observed that developers using LRA for complex compositions produce code with better structural complexity, in less time, than developers using classic development approaches. Furthermore, the study revealed that, for service discovery, the performance and code quality of developers is comparable. The study also shows that the overhead incurred by the LRA middleware is not considerable.

In our future research agenda, we plan to investigate the relations found in textual descriptions of web services, in order to automatically map input and output relationships to ontology elements, which would simplify the the development life cycle with semantically-enhanced web services.

Acknowledgements
Not applicable.

Authors' contributions
All authors contributed equally to this manuscript. All authors read and approved the final manuscript.

Funding
Not applicable.

Availability of data and materials
The usability evaluation of the workbench used a small dataset containing geographical information about the US, which also contains predefined English language questions. The dataset is known as The Mooney Natural Language Learning Data, and is available at http://www.cs.utexas.edu/users/ml/nldata/geoquery.html.

Declarations

Ethics approval and consent to participate
This paper is an original work by Diego Serrano and Eleni Stroulia. The work includes two empirical studies, which received research ethics approval from the University of Alberta Research Ethics Board: 1) LRA Query Editor Usability Evaluation No. Pro00075495, approved on August 15, 2017; and 2) LRA Study No. Pro00078367, approved on January 15, 2018.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 24 May 2021 Accepted: 20 August 2021
Published online: 14 September 2021

References

1. Serrano D, Stroulia E, Lau D, Ng T. Linked REST APIs: A middleware for semantic REST API integration. In: IEEE International Conference on Web Services 2017 (ICWS 2017) 2017.
2. Serrano D, Stroulia E. Semantics-based api discovery, matching and composition with linked metadata. Serv Orient Comput Appl. 2020;14(4):283–96.
3. Iyer B, Subramaniam M. The strategic value of APIs. Harv Bus Rev. 2015;1:2015.
4. Schmachtenberg M, Bizer C, Paulheim H. State of the LOD cloud 2014. Mannheim: Data and Web Science Group: University of Mannheim; 2014.
5. Guha RV, Brickley D, Macbeth S. Schema.org: Evolution of structured data on the web. Communications of the ACM 2016;59.
6. Wang X, Sun Q, Liang J. Json-ld based web api semantic annotation considering distributed knowledge. IEEE Access. 2020;8:197203–21.
7. Cremaschi M, Paoli FD. A practical approach to services composition through light semantic descriptions. In: European Conference on Service-Oriented and Cloud Computing ESOCC 2018. Lecture Notes in Computer Science, 2018;11116, pp. 130–145. Springer.
8. Pedrinaci C, Domingue J. Toward the next wave of services: linked services for the web of data. J Univ Comput Sci. 2010;16(13):1694–719.
9. Dadzie A-S, Rowe M, Petrelli D. Hide the stack: toward usable linked data. In: Extended Semantic Web Conference. Springer; 2011, 93–107
10. Davies S, Donaher C, Hatfield J, Zeitz J. Making the semantic web usable: interface principles to empower the layperson. J Digit Inf. 2011;12(1).
11. Schraefel M, Karger D. The pathetic fallacy of rdf. In: International Workshop on the Semantic Web and User Interaction (SWUI), 2006; 2006.
12. Serrano D, Stroulia E. The lra workbench for composing linked rest apis. In: 2018 IEEE World Congress on Services (SERVICES), 2018; p. 29–30.
13. Pérez J, Arenas M, Gutierrez C. Semantics and complexity of SPARQL. ACM Trans Database Syst. 2009;34:1–45.
14. OpenLink Software: Virtuoso
15. Rietveld L, Hoekstra R. The YASGUI family of SPARQL clients. Semant Web. 2017;8:373–83.
16. Yamaguchi A, Kozaki K, Lenz K, Wu H, Kobayashi N. An intelligent SPARQL query builder for exploration of various life-science databases. In: Proceedings of the 3rd International Conference on Intelligent Exploration of Semantic Data; 2014. CEUR-WS.org.
17. Ngonga Ngomo A-C, Bühmann L, Unger C, Lehmann J, Gerber D. Sorry, i don't speak SPARQL: translating SPARQL queries into natural language. In: Proceedings of the 22nd International Conference on World Wide Web; 2013. ACM.
18. Bhagdev R, Chapman S, Ciravegna F, Lanfranchi V, Petrelli D. Hybrid search: Effectively combining keywords and semantic searches. In: European Semantic Web Conference ;2008. Springer.
19. Arenas M, Cuenca Grau B, Kharlamov E, Marciuska S, Zheleznyakov D, Jimenez-Ruiz E. SemFacet: semantic faceted search over YAGO. In: Proceedings of the 23rd International Conference on World Wide Web; 2014. ACM.
20. Heggestøyl S, Klüwer JW, Waaler A. PepeSearch: Easy to use and easy to install semantic data search. The Semantic Web: ESWC 2016 Satellite Events; 2016.
21. Elbedweihy K, Wrigley SN, Ciravegna F. Evaluating semantic search query approaches with expert and casual users. In: International Semantic Web Conference; 2012. Springer.
22. Kaufmann E, Bernstein A, Fischer L. NLP-Reduce: A naive but domain-independent natural language interface for querying ontologies. In: 4th European Semantic Web Conference ESWC; 2007.
23. Kaufmann E, Bernstein A. Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases. Web Semant. 2010;8:377–93.
24. Kaufmann E, Bernstein A, Zumstein R. Querix: A natural language interface to query ontologies based on clarification dialogs. In: 5th International Semantic Web Conference; 2006. Springer.
25. OpenLink Software: iSPARQL
26. Russell A. Nitelight: a graphical editor for SPARQL queries. In: Proceedings of the 2007 International Conference on Posters and Demonstrations; 2008. CEUR-WS.org.
27. Mazumdar S, Petrelli D, Elbedweihy K, Lanfranchi V, Ciravegna F. Affective graphs: the visual appeal of linked data. Semant Web. 2015;6:277–312.

28. Haag F, Lohmann S, Siek S, Ertl T. QueryVOWL: Visual composition of SPARQL queries. In: Revised Selected Papers of the ESWC 2015 Satellite Events; 2015. Springer.
29. Heim P, Ziegler J, Lohmann S. gFacet: A browser for the web of data. In: Proceedings of the International Workshop on Interacting with Multimedia Content in the Social Semantic Web, 2008;417.
30. Jarrar M, Dikaiakos MD. MashQL: a query-by-diagram topping SPARQL. In: Proceedings of the 2nd International Workshop on Ontologies and Information Systems for the Semantic Web; 2008. ACM.
31. Thompson CW, Pazandak P, Tennant HR. Talk to your semantic web. IEEE Internet Comput. 2005;9:75–8.
32. Vega-Gorgojo G, Slaughter L, Giese M, Heggestøyl S, Soylu A, Waaler A. Visual query interfaces for semantic datasets: an evaluation study. Web Semant. 2016;39:81–96.
33. Elbedweihy K, Wrigley SN, Ciravegna F, Reinhard D, Bernstein A. Evaluating semantic search systems to identify future directions of research. In: Extended Semantic Web Conference; 2012. Springer.
34. Frasincar F, Telea A, Houben G-J. Adapting graph visualization techniques for the visualization of RDF data. Visualizing the semantic web; 2006.
35. Powers M, Lashley C, Sanchez P, Shneiderman B. An experimental comparison of tabular and graphic data presentation. Int J Man Mach Stud. 1984;20:545–66.
36. Wegman EJ, Luo Q. High dimensional clustering using parallel coordinates and the grand tour. Classification and Knowledge Organization; 1997.
37. Chang D, Dooley L, Tuovinen JE. Gestalt theory in visual screen design: a new look at an old subject. In: Proceedings of the 7th World Conference on Computers in Education ;2002. Australian Computer Society, Inc.
38. Pham M-D, Passing L, Erling O, Boncz P. Deriving an emergent relational schema from RDF data. In: Proceedings of the 24th International Conference on World Wide Web; 2015. ACM.
39. DuCharme B. Learning SPARQL: Querying and Updating with SPARQL 1.1. O'Reilly Media, Inc.; 2013.
40. Gallego MA, Fernández JD, Martínez-Prieto MA, de la Fuente P. An empirical study of real-world SPARQL queries. In: USE-WOD Workshop; 2011.
41. Schmidt M, Hornung T, Lausen G, Pinkel C. SP2Bench: a SPARQL performance benchmark. In: Proceedings of the 25th International Conference on Data Engineering ;2009. IEEE.
42. Brooke J, et al. SUS-a quick and dirty usability scale. Usability Eval Ind. 1996;189(194):4–7.
43. Bangor A, Kortum PT, Miller JT. An empirical evaluation of the system usability scale. Int J Hum Comp Interact. 2008;24:574–94.
44. Rosenthal R, Rosnow RL. Essentials of behavioral research: methods and data analysis. New York: McGraw-Hill; 2008.
45. Brüeckmann T, Gruhn V, Koop W, Ollesch J, Pradel L, Wessling F, Benner M. Codeless engineering of service mashups—an experience report. In: Proceedings of the Fourteenth IEEE International Conference on Services Computing 2017 (SCC 2017); 2017.
46. Jaeger MC, Rojec-Goldmann G, Muhl G. QoS aggregation for web service composition using workflow patterns. In: Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International, pp. 149–159 ;2004. IEEE.
47. Cockburn A, Williams L. Extreme programming examined, pp. 223–243. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA ;2001. Chap. The Costs and Benefits of Pair Programming. http://dl.acm.org/citation.cfm?id=377517.377531.
48. McDowell C, Werner L, Bullock HE, Fernald J. Pair programming improves student retention, confidence, and program quality. Commun ACM. 2006;49(8):90–5.
49. Hermans F, Aivaloglou E. Do code smells hamper novice programming? a controlled experiment on scratch programs. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), 2016;1–10 .
50. Guana V, Stroulia E. End-to-end model-transformation comprehension through fine-grained traceability information. Softw Syst Model. 2017;18(2):1305–44.
51. Halstead MH. Elements of software science. Amsterdam: Elsevier; 1977.
52. McCabe TJ. A complexity measure. IEEE Trans Softw Eng. 1976;4:308–20.
53. Sjøberg DI, Anda B, Mockus A. Questioning software maintenance metrics: a comparative case study. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, 2012;107–110 ACM.
54. Shen VY, Yu T-J, Thebaut SM, Paulsen LR. Identifying error-prone software-an empirical study. IEEE Trans Softw Engi. 1985;4:317–24.
55. Zhang H, Zhang X, Gu M. Predicting defective software components from code complexity measures. In: Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium On, 2007;93–96. IEEE.
56. Menzies T, Greenwald J, Frank A. Data mining static code attributes to learn defect predictors. IEEE Trans Softw Eng. 2007;33(1):2–13.
57. Turhan B, Menzies T, Bener AB, Di Stefano J. On the relative value of cross-company and within-company data for defect prediction. Empir Softw Eng. 2009;14(5):540–78.
58. Zhang H. An investigation of the relationships between lines of code and defects. In: Software Maintenance, 2009. ICSM 2009. IEEE International Conference On, 2009;274–283. IEEE.
59. Welker KD. The software maintainability index revisited. CrossTalk. 2001;14:18–21.
60. Bangor A, Kortum P, Miller J. Determining what individual SUS scores mean: adding an adjective rating scale. J Usability Stud. 2009;4(3):114–23.
61. Ellen PS, Bearden WO, Sharma S. Resistance to technological innovations: an examination of the role of self-efficacy and performance satisfaction. J Acad Market Sci. 1991;19(4):297–307.
62. Kuisma T, Laukkanen T, Hiltunen M. Mapping the reasons for resistance to internet banking: a means-end approach. Int J Inf Manag. 2007;27(2):75–85.
63. Luse A, Townsend AM, Mennecke BE. The blocking effect of preconceived bias. Decis Support Syst. 2018;108:25–33.