Automated API Discovery, Composition, and Orchestration with Linked Metadata

by

Diego F. Serrano Suarez

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science
University of Alberta

© Diego F. Serrano Suarez, 2018

Abstract

Over the last decade, an exponentially increasing number of REST services have been providing a simple and straightforward syntax for accessing rich data resources. In today's highly distributed, multi-platform world, there is a huge potential for creating added value through service coordination. However, the data exchanged by Web APIs do not include a semantic representation to enable machines to automatically interpret and appropriately route it. Thus, developing applications that rely on Web APIs still requires a considerable effort by developers, who still have to find relevant APIs, write code for invoking individual APIs, and transform the data in conformance to their application structure. This process is difficult and error-prone, especially as the number and overlap of the underlying services increases, and the mappings become opaque, difficult to maintain and practically impossible to reuse.

Early on, the Semantic Web envisioned service providers annotating service descriptions with semantic specifications, to produce so-called Semantic Web Services (SWS). Semantic meta-data could enable automatic discovery and composition, which could, in turn, reduce the time and development effort needed to create applications. However, after not an insignificant number of years, the impact of SWS has been minimal, mainly due to the steep usability challenges that arise with the usage of semantic languages to create, annotate, and use Web APIs. More specifically, research has focused on highly expressive conceptual models that introduce more complexity than they conceal, and bring additional heterogeneity to an already overwhelming stack of specifications.

The maturity of service technologies, along with the recent advances around Linked-Data formalisms have the potential to provide data integration, regulated by Web APIs that can control access to resources. Thus, the key motivation of this work is to align Web APIs as the preferred standard for exchanging data on the Web and Linked Data vocabularies as the mechanism enabling the automated semantic integration of services.

This work presents Linked REST APIs (LRA), a simplified description model, based on SPARQL graph patterns, for capturing the semantics and relationships of Web APIs. Based on this model, we propose a methodology for a automated process that produces semantically valid composition chains, based on iterative subgraph isomorphism. The performance of the approach is evaluated using a set of publicly available Web APIs in the domain of scholarly work and data, and the usability and effectiveness is evaluated through empirical studies. Our results demonstrate the high accuracy and efficiency of our method, and that LRA developers tend to produce code with better structural complexity in less time, than developers manually composing APIs.

Preface

This thesis is an original work by Diego Serrano. The work includes two empirical studies, which received research ethics approval from the University of Alberta Research Ethics Board.

- *LRA Query Editor Usability Evaluation* No. Pro00075495, approved on August 15, 2017.
- *LRA Study* No. Pro00078367, approved on January 15, 2018.

The literature review presented in Chapter 2, and the conceptual framework presented in Chapters 3 and 4 was published in:

- **D. Serrano**, E. Stroulia, D. Lau, T. Ng. "Linked REST APIs: A Middleware for Semantic REST API Integration", *IEEE International Conference on Web Services (ICWS 2017)*, Honolulu, USA, June 2017.
- **D. Serrano**. "Context-Aware Automated Workflow Composition for Interactive Data Exploration", *International Conference on Service-Oriented Computing (ICSOC 2016)*, Banff, Canada, October 2016. [Doctoral Symposium]
- **D. Serrano**, E. Stroulia. "Semantics-based API Discovery, Matching and Composition with Linked Metadata", *Journal of Web Semantics*. [Under review]

The formalization of the model and composition procedure, along with the composition evaluation methodology presented in Chapter 4 was published in:

- **D. Serrano**, E. Stroulia. "Semantics-based API Discovery, Matching and Composition with Linked Metadata", *Journal of Web Semantics*. [Under review]

The LRA Workbench (Chapter 5) was built with the assistance of Sarah Hoven and Gillian Pierce. Sarah and Gillian contributed with the creation of the web-based interface for the Visual Query Assistant. Chapter 5 and the usability study presented in Chapter 6 have been published in:

- **D. Serrano**, E. Stroulia. "The LRA Workbench for Composing Linked REST APIs", *IEEE World Congress on Services (SERVICES 2018)*, San Francisco, USA, July 2018.
- **D. Serrano**, E. Stroulia. "The LRA Workbench: An IDE for Efficient REST API Composition through Linked Metadata", *Transactions on Services Computing (TSC)*. [Under review]

Finally, other research work completed during the development of this thesis, but not discussed in this manuscript, include:

- K. Eng, **D. Serrano**, J. Jarenko, and E. Stroulia, "(Semi)Automatic Construction of Access-Controlled Web Data Services", *International Conference on Computer Science and Software Engineering (CASCON 2018)*, Toronto, Canada, October 2018. [In press]
- D. Turner, **D. Serrano**, E. Stroulia, K. Lyons. "A T-Shaped Multidisciplinarity Measure: The GRAND Research-Network Case Study", *INFORMS Conference on Service Science 2018*, Phoenix, AR, USA, November 2018. [In press]
- **D. Serrano**, T. Baldassarre and E. Stroulia. "Real-time Traffic-based Routing, based on Open Data and Open-Source Software", *IEEE World Forum on Internet of Things*, Reston, VA, USA, December 2016.
- **D. Serrano** and E. Stroulia, "From Relations to Multi-Dimensional Maps: A SQL-to-HBase Transformation Methodology", *International Conference on Computer Science and Software Engineering (CASCON 2016)*, Toronto, Canada, October 2016.
- Y. Li, D. Turner, E. Maemura, **D. Serrano**, K. Lyons and E. Stroulia. "Comparing Measures of Research Output", *The VIVO Conference 2016*, Denver, USA, August 2016. [Poster]
- M. Fokaefs, **D. Serrano**, R. Veleda, M. Litoiu. "Locality-Enhanced Geographic Information System", International IBM Cloud Academy Conference (ICACON 2016), Edmonton, Canada, June 2016.
- **D. Serrano**, H. Zhang, E. Stroulia, "Kaleidoscope: A Cloud-Based Platform for Real-Time Video-Based Interaction" , *IEEE International Conference on Cloud Computing (SERVICES 2016)*, San Francisco, USA, June 2016.
- T. Baldassarre and **D. Serrano**, "Software Quality on the Cloud", *International IBM Cloud Academy Conference (ICACON 2016)*, Edmonton, Canada, June 2016.
- **D. Serrano**, D. Han, E. Stroulia, "From Relations to Multi-Dimensional Maps: Towards A SQL-to-HBase Transformation Methodology", *IEEE International Conference on Cloud Computing (CLOUD 2015)*, New York, USA, June 2015.

- **D. Serrano**, E. Stroulia. "Towards a Uniform Data Model in Multi-Database Environments". *Emerging Technologies Track at CASCON 2014*, Toronto, Canada, 2014

Information Boxes

We use boxes to highlight important information, including section summaries and terminology clarifications

Contents

1 Introduction **1**
1.1 Thesis Contributions . . . 3
1.2 Thesis Outline . . . 6

2 Background and Related Work **7**
2.1 Early Database-Integration Methods . . . 7
2.2 The Web of Data . . . 8
2.3 Semantic Web Services . . . 10
2.3.1 Service Specifications . . . 10
2.3.2 Adoption and Application . . . 11
2.4 Pragmatic Source Mapping . . . 13
2.5 Emerging Technologies . . . 15
2.6 APIs and Vocabularies . . . 17

3 Definitions, Notation and Semantics **20**
3.1 Preliminaries . . . 20
3.2 Motivating Example . . . 22
3.3 The LRA Data Model . . . 23
3.4 Service Discovery . . . 25

4 Service Composition and Orchestration **30**
4.1 LRA Composition Algorithm . . . 30
4.1.1 Pairwise Matching . . . 31
4.1.2 Blocking . . . 32
4.1.3 Pruning . . . 32
4.1.4 Convergence . . . 32
4.2 Refinement Heuristics . . . 33
4.3 Service Orchestration . . . 34
4.3.1 Access Control . . . 35

4.3.2 Provenance and Transaction Logging . . . 36
4.3.3 Data Extraction and Lifting . . . 36
4.3.4 Record Linkage . . . 37
4.3.5 Caching . . . 38
4.4 Performance Evaluation . . . 39
4.4.1 Semantic Web Service Benchmarks . . . 40
4.4.2 The LRA Web API Test Dataset . . . 43
4.4.3 Relevance Judgments . . . 48
4.4.4 Service and Query Formalization . . . 49
4.4.5 Results . . . 49
4.4.6 Discussion . . . 53

5 The LRA Workbench 55
5.1 Linked-Data Query Systems . . . 55
5.1.1 Interface . . . 56
5.1.2 Query Interpretation . . . 59
5.1.3 Usability . . . 61
5.1.4 Structural-Discovery Support . . . 62
5.1.5 Results rendering . . . 63
5.2 Workbench Design . . . 63
5.2.1 The Query Console . . . 64
5.2.2 The Visual-Query Assistant (VQA) . . . 64
5.3 Usability Evaluation . . . 67
5.3.1 Empirical Study Design . . . 67
5.3.2 Empirical Study Protocol . . . 69
5.3.3 Results . . . 71
5.3.4 Discussion . . . 72
5.3.5 Threats to Validity . . . 72

6 Engineering SOA Systems with LRA 74
6.1 Empirical Study Design . . . 74
6.1.1 Hypothesis . . . 74
6.1.2 Variables . . . 75
6.1.3 Tasks . . . 75
6.1.4 Participants . . . 77
6.1.5 Data Analysis . . . 78
6.2 Empirical Study Protocol . . . 78
6.2.1 Preparation . . . 78

6.2.2 Training 79
6.2.3 Working Session 79
6.2.4 Data Collection 79
6.3 Results 83
6.4 Discussion 86
6.4.1 Threats to Validity 86

7 Conclusions **89**
7.1 Contributions 90
7.2 Future Work 93

Bibliography **94**

List of Tables

2.1 Vocabularies for common Web API categories 19

3.1 Graph representations and textual graph patterns of the data in the example services . . 25

4.1 Overview of Semantic Web Services Test Collections 40
4.2 Effort in LRA formalizations 50
4.3 Runtime and retrieval performance 54

5.1 Linked Data query systems description matrix 56
5.2 Results of the VQA Usability Evaluation 71

6.1 Results of the Usability Evaluation 83

List of Figures

2.1 Evolution of web service technologies. 16

3.1 Linked REST API Model. 24

5.1 LRA Query Console. 65
5.2 Visual Query Assistant. 67

6.1 The Music Time Machine (MTM) test application 77
6.2 Metrics 84

1

Introduction

Over the past 30 years, the idea of a data-integration method that can correctly and efficiently answer queries over data stored across multiple, heterogeneous repositories, in a way that gives users a fully integrated view of the world, has been examined from several perspectives. In the early 1980s, Smith et al. [1] began designing systems for interoperability of heterogeneous databases, by providing a unified global schema and a high-level query language. Once relational databases became the standard, a number of methods were developed to establish mappings between different relational models and a global one, including the transformation of queries submitted to the overall system into queries to the original data sources [2, 3, 4].

Years later, the World Wide Web revolutionized the way people access digital data. Every day the number of data providers increases, as does the volume of accessible data, the variety of available services, and their overlap. Nowadays, a substantial amount of Web data is exchanged through Web APIs that expose data in formats such as JSON or XML. According to *ProgrammableWeb*[1], the number of Web APIs has increased from only 250 APIs in 2006, to more than 20,000 in 2018. Web APIs have become ubiquitous. From government and financial organizations, to games and sports companies, most of the major software services offer some form of Web API. Nearly all the top 100 websites from *Alexa*[2] provide their own APIs, which is an indicator of the degree to which web services have been adopted as the de-facto syntax for interacting with external resources.

The evolution of the Web has led to an increased adoption of Web APIs, transforming 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,

[1] *ProgrammableWeb* is a website that publishes a repository of Web APIs

[2] *Alexa* (`http://www.alexa.com/topsites`) provides information about web traffic data.

60% [5]. 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 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.

Early on, the Semantic Web envisioned service providers annotating service descriptions with semantic specifications, to produce so-called Semantic Web Services (SWS). Semantic meta-data could enable automatic discovery and composition, which could, in turn, reduce the time and development effort needed to create applications. However, after not an insignificant number of years, the impact of SWS has been minimal, mainly due to the steep usability challenges that arise with the usage of semantic languages to create, annotate, and use Web APIs. More specifically, research has focused on highly expressive conceptual models that introduce more complexity than they conceal, and bring additional heterogeneity to an already overwhelming stack of specifications. Consequently, SWS have been regarded as a niche technology, only accessible to highly-trained experts, offering benefits often considered unworthy of the additional effort investment [6].

More recently, the evolution of REST services 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 [7], and *schema.org* [8], which demonstrate the potential of Linked Data approaches for data integration.

The key intuition motivating this thesis is that the alignment of these two widely adopted technologies, Web APIs and Linked Data vocabularies, which can provide the level of interoperability required for supporting automated composition of Web APIs, in practice. Recently, REST APIs have emerged as the preferred de-facto standard for exchanging data and sharing functionalities on the web. On the other hand, the increasingly popular Linked-Data repositories enable the automated semantic integration of data within a specific domain, but, without APIs to regulate their access, they are bound to exclude proprietary information, which fundamentally limits their scope to open data and precludes business adoption. Bridging these technologies can address the shortcomings of both, and enable the automated semantic integration of services.

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

The objective of this work is to enable automatic API integration, through the synergy of Linked Data (RDF) specifications of APIs, and a middleware that answers SPARQL queries, by automatically discovering and composing APIs, based on their semantic descriptions. Then, this work proposes a methodology to automatically specify execution plans for queries, that link and compose responses from independently deployed services, and optimize the trade-offs of data quality and performance. Additionally, this thesis addresses service engineering approaches, supported by specific tools, to simplify the formulation of queries and interactive exploration of their data sets. Regarding the evaluation of our work, the high accuracy and performance of the composition methodology was demonstrated using a collection of publicly available Web APIs composed to represent real-world use cases. Furthermore, the usability of the development environment was evaluated in an empirical study demonstrating that our tool significantly improves the performance of developers formulating SPARQL queries. And finally, a subsequent effectiveness study demonstrated that developers using our approach tend to produce code with better structural complexity, in less time, than developers manually composing APIs.

In the following, we provide an overview on the contributions of this thesis.

1.1 Thesis Contributions

In this work, I have developed Linked REST APIs as a means to implement my thesis and prove its plausibility. Towards this goal, Linked REST APIs make 5 contributions: (C1) a semantic data model to annotate Web APIs, accompanied by (C2) a composition methodology that can automatically create Web API workflows that respond to data queries, considering quality and access-control constraints; (C3) a curated composition evaluation dataset, including 52 realistic service offerings and 8 queries formulated by domain experts; (C4) an integrated development environment for the creation of composition requests, and analysis of responses; and finally, (C5) an empirical study that investigates the impact of LRA, in terms of complexity, accuracy, and developers' performance in the software-development process

C1: Semantic Service Model Specification

The emergence of REST APIs as the de-facto standard to securely exchange data on the web, along with the increasingly popular Linked-Data repositories, enable the automated semantic integration of data within a specific domain. This thesis introduces the Linked REST API (LRA) model for annotating the functional semantics of REST APIs, using Linked-Data ontologies (Chapter 3). The semantic model is derived from relevant work on service modeling and description, mainly from the Minimal Service Model [9], SAWSDL [10], LIDS [11], Hydra [12], and OpenAPI. For automated processing, the LRA model serves as a layer of abstraction over the concrete service implementation,

that uncovers the underlying structure of the data regulated by the web service.

The design of the LRA model is guided by two principles: Proximity to underlying standards, adopting the structural organization of OpenAPI, a widely adopted REST API description format; and Lightweight minimality, adding only very few new constructs on top of the underlying technologies that are already well-known. Moreover, the model relies on SPARQL graph patterns to express API data offerings, which provides a flexible and extensible way to represent information about REST API resources.

C2: Automated Service Composition Methodology

The preceding contribution presents a model that enables semantic annotations on Web service descriptions. These semantic descriptions are intended to support tasks such as service discovery and composition, therefore one of our contributions is an end-to-end composition methodology that supports the entire SOA life cycle, including discovery, composition, and orchestration, in contrast to typical approaches, that only provide composition suggestions or require manual intervention.

Accordingly, this work presents a novel methodology for an automated specification of execution plans, based on SPARQL queries, whose graph patterns are used to find chains of service operations that can reach a structural equivalence (Chapter 4). The composition methodology is enabled by an iterative process involving pairwise matching between queries and service operations, using subgraph isomorphism, and complemented with pruning and filtering strategies to reduce the search space. The methodology not only involves the discovery of composition chains, it also considers the enactment of Web services. After a set of composition chains have been identified, they are filtered based on access-control constraints and quality trade-offs. Subsequently, the services are invoked, lifting their responses with semantics, and consolidating the data from multiple composition chains.

Using our evaluation dataset, comprising more than 50 real service offerings in the domain of scholarly content, the LRA composition methodology exhibited high accuracy and reasonable running time, improving the performance of approaches evaluated with datasets of similar size.

C3: Semantically Annotated Evaluation Dataset

The vast amount of literature on semantic web service descriptions and automated composition approaches contrasts with the scarcity of benchmarks that enable comparative evaluation of composition strategies. This resulted in a proliferation of controversial validations based on illustrative examples, or on synthetic datasets under questionable assumptions. Additionally, the unavailability of some of the synthetic datasets and the lack of publicly available semantic Web services only aggravates the problem.

In order to provide a more realistic evaluation methodology, we created a web service test

collection extracted from real service offerings (Section 4.4.2). The collection of service descriptions is complemented with a set of representative queries, at different levels of difficulty, defined by a domain expert with experience in research-evaluation metrics and research impact assessments. These queries can be answered through a number of potential providers in the dataset, thus providing valuable data for comparing the compositions generated by automated composition frameworks.

C4: Integrated Development Environment for Linked REST APIs

Despite the potential benefits, in terms of performance and code quality, of enabling developers to retrieve data from multiple Web APIs by using a declarative language, the lack of an integrated-development environment that streamlines the use of LRA may hinder the adoption of the framework. This thesis also introduces the LRA Workbench, which enables developers to express their information needs without extensive 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 (Chapter 5). The LRA Workbench 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 suitability of the development environment to abstract the inherent complexities of SPARQL is assessed through an empirical study that investigates the performance of developers when formulating LRA-compliant queries. Our results demonstrate that the LRA Workbench 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.

C5: An Empirical Study on End-to-End Automated Service Composition

Finally, this thesis contributes an empirical study that investigates the effectiveness of LRA, by comparing the quality of code and performance of developers when asked to develop a *mashup* web application using LRA, against that of developers using the traditional approach where they manually design a workflow that submit requests directly to the service endpoints (Chapter 6). We measured the structural complexity of the produced code artifacts, and the accuracy and efficiency of developers when asked to create software components that integrate data services using LRA.

We found that for complex composition flows, developers using LRA tend to produce code with lower structural complexity in less time, than developers using the traditional approach. For simple workflows, involving single-service discovery, the results indicate that the structural complexity and performance of the two approaches is comparable. Furthermore, we found that the overhead incurred in the runtime service composition of LRA has minimal impact on the performance of the system.

1.2 Thesis Outline

The content of this thesis is structured as follows. Chapter 2 gives an overview of the evolution of semantic description of web services, in parallel with the development of integration systems based on web-services. Chapter 3 presents the proposed semantic description for Linked REST APIs, discussing its design guidelines and improvements over previous approaches. Chapter 4 introduces the methodology to leverage the semantic descriptions for automatic service composition, along with the results of our evaluation on service discovery and composition. Chapter 5 analyzes previous work on Linked-Data query systems, which guides the design of the LRA Workbench for assisted query formulation. Then, the usability of the workbench is evaluated through an empirical study. Chapter 6 presents the empirical evaluation of the entire LRA environment, showing the results of the study, and discussing our findings. Finally, Chapter 7 revisits the contributions, and summarizes our future avenues of research.

2

Background and Related Work

Almost since its inception, the service-oriented architecture (SOA) paradigm envisioned general-purpose agents that would automatically discover, compose, and take action on information from various service providers. The main design consideration in building such agents is the description of the underlying service providers, and the possible connections among them, in a way machines can understand. A key element of these descriptions is the semantic mappings of the correspondence between the data elements and attributes manipulated by the various providers and a shared mediating schema.

This chapter provides essential context around semantic alignment of web services, with respect to traditional data-integration systems and the evolution of the Semantic Web. Likewise, the chapter also explores how the creation of semantic description models led to the development of tools to assist in discovery and composition of web services.

2.1 Early Database-Integration Methods

Most of the original work on data integration (between 1982 and 2002) focused on integrating disparate relational databases, with the underlying data sources specified in terms of logic-oriented languages, such as Datalog. In this phase, two fundamental mapping formalisms were proposed [13]: *local-as-view*, and *global-as-view*. These two formalisms define the relationship between a global, or mediating, schema and the local database schemas. In the *global-as-view* (GaV) paradigm, the global schema is defined as a set of views over the data sources. An example of a system that used GaV mappings is TSIMMIS [2]. On the other hand, in *local-as-view* (LaV), each local schema is treated as a view defined over the global schema. One of the most representative LaV-based systems is Information Manifold [3]. While GaV mappings are conceptually simple and provide an easy

query reformulation over the local schemas, they suffer from two key drawbacks. First, the mappings into the global schema may lose information about the attributes and relationships in the source schemas. Second, adding and removing sources involves considerable work and intimate knowledge of the sources. Conversely, LaV mappings make adding and deleting sources easier than in GaV, but, with the increased flexibility comes the need to develop more complex query reformulation algorithms. To balance this trade-off, Friedman et al. [14] proposed a combination of the two approaches, named *global-local-as-view* (GLaV). In this method, the schema mappings include a query over the data sources on the left-hand side and a corresponding query on the mediating schema on the right-hand side. GLaV mappings have been mainly investigated in data-exchange systems, with Clio [15] being one of the most representative examples. All these mapping formalisms provide the basis of recent work on data integration and data exchange.

Regardless of the particulars of the data-integration architecture, describing and, possibly, discovering mappings between sources and mediating schemas is an integral part of a heterogeneous-data-integration scenario. Early integration systems, such as [2, 3, 4, 16], specified the mediating schema in terms of a relational model. These traditional data-integration systems have met limited success in practice, mainly because they lack flexibility, robustness, and scalability. One of the major bottlenecks in setting up a data-integration application is the effort required to create the source descriptions and the semantic mappings, which requires database expertise and business knowledge [17]. Some approaches have explored semi-automated generation of schema mappings, such as [18, 19, 20], which only partially reduce the amount of manual development effort required.

2.2 The Web of Data

In the early 2000s, Berners-Lee et al. [21] envisioned a transition from an Internet of loosely interlinked text documents, designed for human consumption, to the *Semantic Web*, a thoroughly described and tightly interlinked "Web of Data", intended for automatic machine processing. During the past decade, the community has worked to realize this vision, creating standards based on the principles of knowledge-representation languages, and adapting them to the World Wide Web. Some of the relevant core developments are the Resource Description Framework (RDF) for semantic markup [22], and its associated schema languages, RDF Schema (RDFS) [23] and the Web Ontology Language (OWL) [24].

The Semantic Web has had a great impact in our everyday activities. However, the vision described by Berners-Lee and colleagues is far from real, and, in fact, many question its practical feasibility. The most pragmatic effort has focused on publishing interlinked datasets, in what is called the Linking Open Data (LOD) project. The purpose of LOD is to bootstrap the Web of Data by identifying publicly available data sets, and publishing them on the Web as RDF graphs [25]. In

2014, there were more than 1,000 such datasets, spanning a number of diverse thematic areas such as government, media, and life sciences, among others [7].

The LOD project integrates multiple databases that have been translated into RDF, using a mixture of common vocabularies and terms specific to the data sources; these datasets are connected to other datasets through the use of URIs and equivalence relations to external databases. The most prominent example of a linked dataset in the LOD project is DBpedia [26], a knowledge base extracted from Wikipedia, that serves as a connectivity hub of the LOD initiative. Currently, DBpedia is linked to more than 30 other data sets in the LOD cloud.

In the last decade, ontologies have prevailed as the preferred conceptual-modeling language for data integration. Ontologies are particularly useful because of their flexibility and ease of evolution. Additionally, the concept of ontology aligns better with integration systems, since ontologies are considered the result of a collective effort, at a high level of abstraction, while relational models are specific to particular implementations, lacking axioms that describe the reality in its representation [27]. The versatility of ontologies combined with the *global-as-view* mapping formalism laid the foundations for ontology-based data integration, which has been used in large-scale projects, such as the LOD project, and *schema.org*.

Although the LOD project proves that the Linked Data approach is, in principle, capable of integrating data from a large number of repositories [11], there is still a lot of data residing in silos, accessible only through procedural interfaces, even though they could be beneficially linked with other datasets. There are three main reasons that may prevent the publication of a data set in LOD. Some data may be proprietary and, therefore, illegal to share. Yet other data may be dynamically produced, for example through sensors; in this case, even though it would be useful to share the most recent version of the data, or some desired subset of the complete data series, it is impractical to do so due to the continuous evolution of the underlying data repository. Finally, some data sets are "big" (in the "big data" sense) and their sharing through LOD standards is impractical. Even if, however, a particular data set may be impossible to share through LOD standards, it may still be accessible through web services (that observe the relevant access-control rules, or compute the most recent values, or retrieve the relevant big-data slice) that return structured documents, primarily in XML or JSON. Those structured responses offer a common syntactic format, but they lack necessary semantics to interpret and interlink the content of the documents. In order to address this problem, the services have to include semantic annotations, as prescribed by the so-called a *Semantic Web Service* (SWS) schema.

2.3 Semantic Web Services

Web services are described in terms of their operations and their input and output types: WSDL [28] specifications describe web services exchanging data through the the SOAP protocol, and WADL [29] specifications describe RESTful web services that exchange XML or JSON data through HTTP. These specification languages were subsequently extended to include semantic information, in order to enable automation discovery and composition, that could in turn reduce the manual effort required to create applications based on web services.

2.3.1 Service Specifications

When SOA gained momentum, the Defense Advanced Research Projects Agency (DARPA) realized the potential benefits of semantic descriptions for automatic service discovery and composition, and created DAML-S (in 2003 renamed to OWL-S [30]), a semantic markup language for describing web services and related ontologies, that can be used in conjunction with WSDL, through its extensibility elements. Compared to WSDL, the semantic specification provides analogous constructs to represent inputs, outputs, preconditions, and effects. The inputs and outputs are defined in terms of ontologies, which allows for the use of concepts defined and shared as part of the Semantic Web. Several other similar approaches were proposed around the same time, such as the Web-Service Modeling Ontology (WSMO) [9], and the Semantic Web Services Framework (SWSF) [31].

A number of W3C submissions of semantic service descriptions and data-integration projects explored incremental semantic extensions of existing standards, which led to the eventual creation of a W3C working group on developing an annotation language of web-service descriptions. This group produced the SAWSDL definition, the Semantic Annotations for WSDL and XML Schema [10]. SAWSDL, a W3C recommendation, is a lightweight restricted version of a previous W3C submission, WSDL-S [32]. SAWSDL does not specify any ontological service model, but defines a *modelReference* extension attribute to associate a WSDL or XML Schema component and a concept in an ontology, and two other extension attributes, *liftingSchemaMapping* and *loweringSchemaMapping* to specify mappings between semantic data and XML. One of the main criticisms of SAWSDL is that it is only a syntactic extension of WSDL, without any formal semantics. In addition, its simple model references do not fully express the functional semantics, since, in some cases, the semantic association is ambiguous. For example, if we annotate the input of a bibliographic web service with `foaf:name`, it is not clear if it corresponds to names of publication authors, editors, or reviewers. More recently, Kopke et al. [33] proposed to extend SAWSDL to include paths in the model reference, in such a way that concept associations are preserved, for example, `dc:creator/foaf:Person/foaf:name` would unambiguously refer only to the name of the creators of a given entity.

Soon after SAWSDL was proposed, Sheth et al. [34] presented a new formalization, named SA-REST, which is a derivative of SAWSDL, adapted for RESTful APIs. In addition to annotating inputs, outputs, operations, and faults, SA-REST also supports the specification of the type of request that is needed to invoke the service, and the data format used in the API. MicroWSMO [35], also based on SAWSDL, defines another formalism for the semantic description of RESTful web services. MicroWSMO relies on hRESTS, an HTML microformat for describing RESTful web services. SAWSDL and its derivative approaches focus on defining simple extensions to link elements of syntactic service specifications with semantic descriptions. In the particular case of MicroWSMO, the microformat extends hRESTS with three additional properties, namely *model*, *lifting*, and *lowering* that have the same semantics as their SAWSDL counterparts. One of the limitations of MicroWSMO, and microformats in general, is that the HTML structure of the documentation of some API does not permit a proper nesting of the microformat-annotated HTML elements [36].

In order to support both web services based in SOAP, as well as RESTful web services, WSMO-Lite [37] was submitted to W3C in 2010. WSMO-Lite is the evolution of SAWSDL, extending the annotations with concrete semantic service descriptions. WSMO-Lite adopts the WSMO model and simplifies its semantics. This approach uses simple taxonomies to represent the high-level semantics of a web service. These taxonomic service-category hierarchies, such as eClass[1] or UNSPSC[2], provide a simple definition of service functionalities [38]. WSMO-Lite describes the following four aspects of a web service: the *Information Model*, which defines the data model for input, output, and fault messages; the *Functional Semantics*, which defines the functionality offered by the service in terms of functionality classifications and/or logical expressions representing preconditions and effects; the *Behavioral Semantics*, which defines how a client has to talk to the service, represented by annotating the service operations with functionality classifications and/or preconditions and effect; and, the *Non-functional Semantics*, which defines non-functional properties, using ontologies that capture properties such as quality of service or price. The WSMO-Lite approach is not bound to a particular service description format such as WSDL, thus it can be used to integrate approaches, such as SAWSDL and MicroWSMO.

2.3.2 Adoption and Application

The proliferation of all these semantic descriptions constitutes compelling evidence for the importance of providing unambiguous, machine-understandable representations of heterogeneous and independent data and service providers. However, despite its preponderance and extensive literature, semantic integration remains an extremely difficult problem, in part, due to the heterogeneity and scale of the Web.

[1] Available at `http://www.eclass.eu/`
[2] Available at `https://www.unspsc.org/`

The initial description approaches, including OWL-S [30] and WSMO [9], proposed heavy-weight descriptions, specifying the IOPE (inputs, outputs, preconditions, and effects) signature of services, and, sometimes, their non-functional properties. However, the difficulties to create full service profiles forced the vast majority of discovery and composition procedures to use only inputs and outputs. One of the most popular examples of frameworks using OWL-S descriptions is OWLS-MX [39] (and its variants), which uses a mixture of logical subsumption-based signature matching and text similarity for service discovery. This signature-based approaches are the most popular in the literature, including approaches like WSColab [40], IRS-III [41], and the algorithms proposed by Paolucci et al. [42], Jaeger et al. [43], and Huang et al. [44].

Some of the few full IOPE matchmakers were iSeM [45] and DIANE [46]. iSeM uses an SVM classifier to learn from different kinds of service IOPE matching results, including text similarity, matching of service signatures, and matching of service preconditions and effects. Nevertheless, iSeM was limited to service discovery, and requires a set of manually-labeled service-request pairs for training. Furthermore, the transferability of this approach to complex composition scenarios is uncertain. On the other hand, DIANE [46] defines its own description formalism, called DSD, which allows to define constraints on the values instances and properties can take. In DSD, service offers are described as the set of effects they can provide, whereas service requests are described as the set of effects that are acceptable for the requester. The DSD matchmaker uses a multi-phased approach, where offers are matched with regard to whether they are able to provide at least a subset of the requested effects. Offers are combined in a way that each combination provides each effect and each effect only once. This implies that the composition patterns are limited to parallel structures, disregarding sequential compositions.

In response to the complexity and limited use patterns of previous approaches, a few lightweight description approaches emerged, such as SAWSDL [10] and WSDL-S [32]. Some prominent frameworks that used lightweight descriptions are METEOR-S [47], that works by specifying workflow templates where concrete services are bound to each workflow node at runtime; SWSComposer [48], which relies on SAWSDL specifications to construct approximate compositions; and ComposIT [49], which proposes a graph-based service composition focused on the semantic input-output parameter matching of services' interfaces. Despite being used by prominent reference frameworks, the composition algorithms based on lightweight descriptions are susceptible to incorrect solutions, since the semantic descriptions do not capture the relations between input and output elements.

Despite their prevalence in the literature, none of the aforementioned proposals were supported by a working middleware with enough traction to drive the development of a community, which led to their abandonment. In addition to the lack of a practical integration middleware, and incentives to semantically describe services, there is no generally accepted formalism for describing semantic mappings of web services. Currently, while some of the standards for semantic web services can be

used to define mappings to a shared standard ontology, in practice, only very few services have such semantic descriptions.

One of the reasons for the lack of adoption of semantic web services is *semaphobia* [12], i.e., the fear of typical developers to use, seemingly complex, Semantic-Web technologies. Some authors acknowledged the difficulties of interpreting standards and defining semantic web services, and have created tools to support users in annotating web services, such as MWSAF [47] and ASSAM [50]. These tools offer graphical user interfaces for the semiautomatic annotation of web services, that suggests which ontology class should likely be used to annotate each element in the web service. However, these tools also suffer from a lack of broad adoption, due to the unpopularity of semantic web services, thus experiencing a classic chicken-and-egg problem.

2.4 Pragmatic Source Mapping

In spite of the complexity and the general lack of adoption of semantic approaches, some authors have proposed service-integration systems using semantic web-service descriptions. For instance, Pedrinaci and Domingue [6] used the Minimal Service Model (MSM) to capture the formal semantics of services. MSM adopts a simple ontology consisting of the concepts shared by a number of existing ontologies, including SAWSDL and WSMO-Lite. MSM describes *Services* as a composition of *Operations*, which, in turn, have input, output, and fault descriptions. The model is complemented by the WSMO-Lite vocabulary, which defines classes for describing the four core aspects described previously. Although the description formalism was not adopted in the community, the authors emphasize that one of the advantages of MSM over other semantic web-services research is its support for both SOAP-based web services, as well as the numerous, and increasingly popular, Web APIs and RESTful services. In addition, MSM offers high flexibility, allowing additional framework-specific extensions, of use to clients aware of these formalisms.

In light of the shortcomings of semantic web services, some approaches proposed domain-specific languages to define the semantics of web services, such as Swashup DSL [51] and Resource Linking Language (ReLL) [52]. Nevertheless, having other languages and formats in the services-integration system increase the complexity, and thus, its *semaphobia*. Sbodio et al. [53] proposed SPARQL, recognized as one of the key technologies of the Semantic Web, to represent preconditions and postconditions of web-service operations. In this approach, the `CONSTRUCT` clause defines the postconditions or outputs, and the `WHERE` clause defines the preconditions or inputs. However, this work was never evaluated on anything but a synthetic data set and is fundamentally limited in that the SPARQL meta-data do not express non-functional aspects, such as access-control rules or quality-of-service requirements.

Speiser and Harth [11] proposed a slightly different approach to describe web services, by using

RDF and SPARQL graph patterns, in what they call Linked Data Services (LIDS). In this approach, the authors describe the inputs and outputs through SPARQL strings that captures the relationships among the attributes. LIDS presents a compact representation, compared to other approaches, however, because the inputs and outputs graph patterns are represented as two independent strings, this approach leads to inconsistencies and relies heavily on string encoding of basic graph patterns, thus requiring a computational overhead for the parsing of these strings, and precluding the use of automated reasoners for validation and composition.

Several approaches, including RESTdesc [54], and Karma [55] have used RDF, not only to store data, but also as the language employed to describe semantic mappings between the mediating schema and the local sources. The RESTdesc [54] formalization focuses on the functional descriptions of Web APIs in N3[3]. The functional descriptions are composed of preconditions that entail certain postconditions, such as the existence of an HTTP request. Given a goal expressed as a (set of) desirable postcondition(s) and a set of preconditions, one can deduce an HTTP request that achieves the transformation. Since RESTdesc is a logic-based specification, in theory, existing semantic-web reasoners with Notation3 support can create compositions of services by chaining RESTdesc descriptions. Nevertheless, a description based in first-order logic rules, like RESTdesc, may not look familiar to developers accustomed to describing services in other formalizations, which hinders its widespread adoption. The Karma framework [55] has a flexible ontology, created by re-using other vocabularies, not only to define the input and output types, but also to model their relationships. Thus, in Karma, the concepts in the input and output of the service form a connected graph, on which more complex questions can be answered. The input and output models of the formalism are represented with the SWRL (Semantic Web Rule Language) vocabulary [56], which has a verbose syntax and is not easy to read; to mediate this issue, the framework provides a web application [57] that supports the creation of mappings between the data source and a set of ontologies. The main drawback of this approach, besides its verbosity, is that, in its actual form, it only works with RESTful Services using the GET method, and with parameters specified as key-value pairs (not as part of the path of the URL). In recent years, Karma has been focused on the transformation of RDF datasets, disregarding further improvements to web-service descriptions. One of the additional benefits of representing web services in RDF is the possibility to use an expressive declarative language, such as SPARQL, not just to submit data queries created by end users, but also to discover services and to reason about their composition.

Figure 2.1a summarizes the evolution of the semantic descriptions of web services. Every node in the graph represents a version of the formalization, nodes with dashed border are recommendations of W3C, and circles with white fill are submissions to W3C. Connections among nodes denote

[3]Notation3 or N3, is a serialization of RDF models, designed with human-readability in mind, that includes concepts such as variables

that the formalization represented by the target node uses concepts or is inspired by the source node. Figure 2.1a shows a natural evolution of semantic descriptions of web services, resembling a waterfall development, where previous approaches are used to improve successive semantic descriptions.

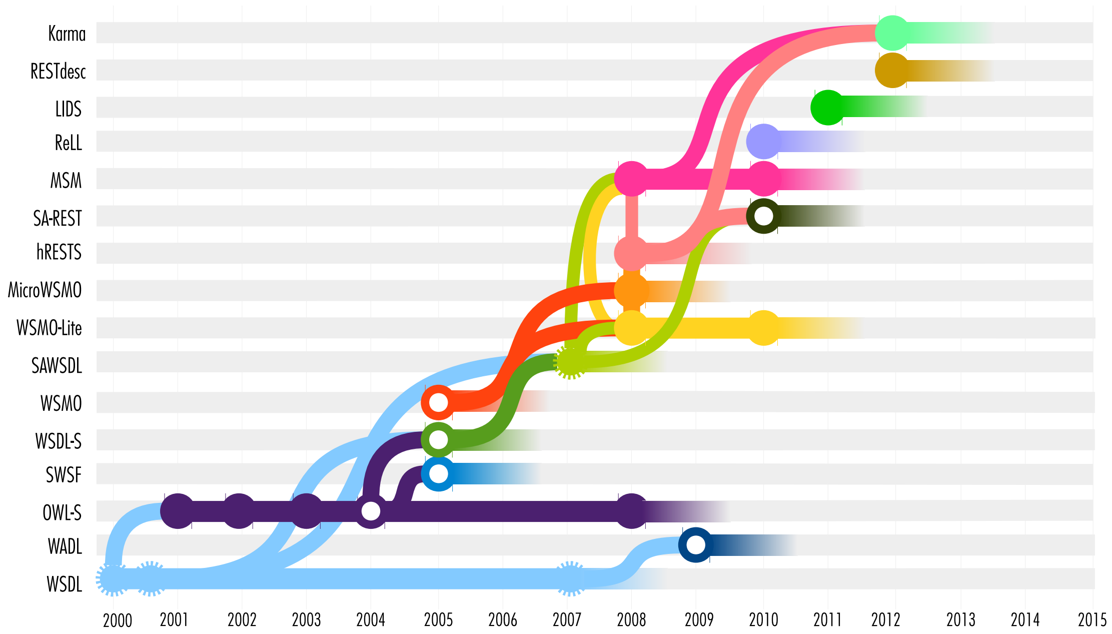
As is evident from Figure 2.1a, the community has not reached a consensus on a single way to describe the semantics of web services. Moreover, none of the standards or data-integration systems has been widely adopted, or more effective than the others, in practice. In addition, it can be seen that in the current decade, the interest in semantic descriptions of web services has decreased, which may be explained by the increasing adoption of the REST architectural style in this decade, alongside the relegation of SOAP-based services, as shown in Figure 2.1b. In semantic web services, most of the approaches depend directly or indirectly of a syntactic description, such as WSDL, however, in REST, those descriptions are no longer necessary.

2.5 Emerging Technologies

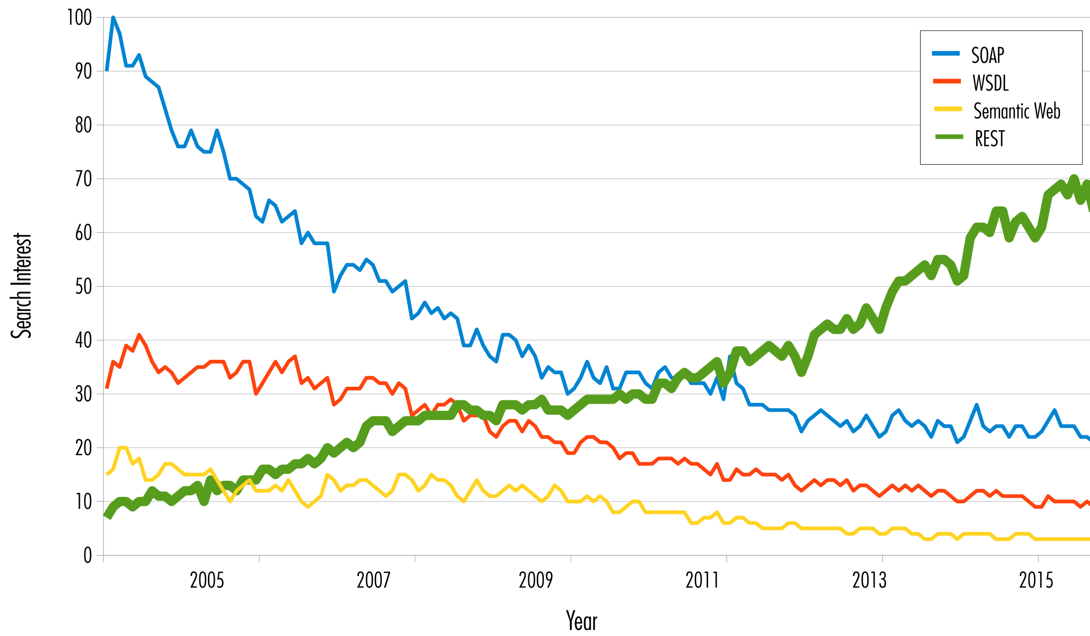
In the industrial community, approaches such as Yahoo! Pipes[4] and more recently Node-RED[5] have created visual assistants to help in the creation of service *mashups*. In addition, some automated composition approaches are starting to emerge. Facebook's GraphQL [58] is a query language for APIs, created by defining types and fields, using a domain-specific language, and then providing functions to retrieve the data for each field on each type. The execution procedure in GraphQL is defined by the recursive execution of each field, which is backed by resolve functions. However, GraphQL compositions are fairly simplistic in that it only represents data using a syntactic schema (without semantics), and the compositions are not designed to expand across multiple service providers.

REST services have now become more common in practice, but there are no universally accepted standards for describing them. In recent years, a few formats have emerged in order to provide a description that can be easily interpreted by humans and machines. OpenAPI[6], and RAML[7] are the most popular formats to describe REST APIs, that have been traditionally documented with natural-language descriptions. OpenAPI is a YAML/JSON format, that started its development in 2010, under the name of Swagger, and by 2015, it became an organization, under the sponsorship of the Linux Foundation, called the Open API Initiative, where several companies, including Google, IBM and Microsoft are founding members. Among REST API description formats, OpenAPI/Swagger

[4]Unavailable since 2015. Description available at: `https://en.wikipedia.org/wiki/Yahoo!_Pipes`
[5]Available at: `https://nodered.org/`
[6]OpenAPI, previously known as Swagger: `http://openapis.org/`
[7]RAML (RESTful API Modeling Language) documentation: `http://raml.org/`



(a) Evolution of semantic web services.



(b) Search interest of service architectural styles.

Figure 2.1: Evolution of web service technologies.

is considered to have the largest and most active developer community[8]. In turn, RAML is also a YAML-based language, first released in 2013, and backed by companies such as PayPal, Intuit, Cisco, ProgrammableWeb, and MuleSoft, the API management company.

Until now, these formats have been used, mainly, to automatically generate SDKs in multiple programming languages and to create documentation, test cases, and API consoles for testing web services from any web browser. Such formats include invocation details, like request methods, status codes, and input arguments, but completely ignore the underlying service semantics. Our work is motivated by our belief that the emergence of REST APIs, as the preferred syntax for exposing data and functionalities on the web, and RDF, as the formalism for representing data semantics and linking data, provide a unique opportunity for building a pragmatic semantic service-integration methodology.

2.6 APIs and Vocabularies

In order to become aware of the possibilities to describe Web APIs using ontologies or shared vocabularies, we conducted a study, analyzing Web APIs from the ProgrammableWeb directory. For easier search and browsing, each API in the directory is grouped into multiple categories, which are presented to the users in a sorted list, that includes the number of APIs associated to the category. In our study, we covered the categories that have at least 400 APIs per category, which corresponds to the top-75 categories. As the study explores the possibility of reusing vocabularies to represent business domains represented by Web APIs, in the analyzed Web APIs we filtered out categories about crosscutting concerns, such as *API*, *Data*, and *Real-time*. In addition, categories for similar domains were merged, for example, *Video*, *Photos*, and *Images* were merged into *Media*.

For each category, we use the Linked Open Vocabulary repository and Google Search to seek for ontologies or vocabularies that can represent the data of the APIs in the specific category. The queries submitted to the search engines contained the name of the category, and equivalent or related words (e.g.: *geographic*, to find ontologies associated to *mapping*) in order to cover the wide variety of ways to refer to the same category. The results were filtered based on the availability of the ontology or vocabulary, and the existence of supporting documentation. In addition, we also checked the vocabularies referenced from and to the selected ontology or vocabulary, which were subsequently filtered based on relevance with respect to the business domain represented by the category. Our search produced a large number of models to represent business domains, finding that more than 80% of the categories have a corresponding ontology or vocabulary. From the uncovered categories, we also found that some experimental vocabularies exist, which are reported in academic

[8] SmartBear, the company that maintained Swagger, reports more than 350,000 downloads per month of Swagger and Swagger tooling

articles. Table 2.1 shows the results of our analysis in terms of Web API categories, their prominence expressed in number of APIs in that category, some examples APIs, and some vocabularies that can be used to describe the category.

In summary, for most of the common business domains of Web APIs, there are ontologies capable of representing specific vertical domains, which companies can adopt for their own use. Nevertheless, a realistic approach will have to extend existing vocabularies for specific uses, and combine vocabularies in order to adapt for domains that expand over multiple categories, for example, *e-commerce* sites need to describe their commercial activities, also including information about *payments*, and particular features of products, for instance *food*.

Table 2.1: Vocabularies for common Web API categories

Category	**Num. APIs**	**Percentage APIs**	**Example APIs**	**Vocabularies**
Mapping	4869	30.05 %	Google Maps, GeoNames	LinkedGeoData ontology (lgdo), FAO Geopolitical Ontology (geop)
Social	3862	23.84 %	Facebook	Semantically-Interlinked Online Communities (sioc), Friend of a Friend vocabulary (foaf), Open Graph Protocol Vocabulary (og)
eCommerce	2786	17.20 %	eBay	GoodRelations Ontology for Semantic Web-based E-Commerce (gr), Schema.org (schema), Product vocabulary (provoc)
Search	2681	16.55%	Bing	Sindice search vocabulary (search)
Telephony and Messaging	2090	12.90 %	Skype, Slack	Nepomuk Messaging Ontology (nmo)
Media (Video, Photos, etc.)	2055	12.68 %	Youtube, Flickr, Pinterest	Ontology for Media Resources (ma-ont), Cinelab ontology (cl), Lightweight Image Ontology (lio)
Financial	1937	11.96 %	Kiva, PayPal	Corporate Financial Reports and Loans Ontology (cfrl), Payments ontology (pay)
Cloud	1864	11.51 %	OpenStack	-
News Services	1379	8.51 %	Guardian	Press.net ontology, News Storyline Ontology (nsl)
Travel	1302	8.03 %	Expedia	Schema.org (schema), Tickets ontology (tio)
Music	1280	7.90 %	Last.fm	The Music Ontology (mo)
Government	1176	7.26 %	GovTrack	Central Government Ontology (cgov), Government Data Vocabulary (gd), Ontologies for e-Government (oeGov)
Security	1153	7.12 %	Google OpenID	Security Ontology (security), Ontology of digital identity (identity)
Email	1049	6.47 %	MailChimp	-
Games	956	5.90 %	Yahoo Fantasy Sports	The Video Game Ontology (vgo), Ludo Game Model Ontology (ludo-gm)
Events	954	5.88 %	Eventbrite	The Event Ontology (event), Event Programme Ontology (prog)
Database	886	5.47 %	Amazon SimpleDB and S3	-
Advertising	821	5.07 %	Facebook Ads	-
Blogging	805	4.97 %	Tumblr	Semantically-Interlinked Online Communities (sioc), Creative Work Ontology (cwork), VIVO Core Ontology (vivo)
Education	795	4.91 %	Mendeley, Udacity	VIVO Core Ontology (vivo), Teaching Core Vocabulary (teach)
Transportation	767	4.73 %	Uber	Transport Administration Ontology (trao), Passim ontology (passim), schema.org (schema)
Sports	741	4.57 %	Sportradar	Sport Ontology (sport)
Storage	682	4.21 %	DropBox	Nepomuk File Ontology (nfo)
Health	523	3.23 %	FitBit, ApiMedic	Translational Medicine Ontology (tmo), Healthcare metadata - DICOM ontology (dicom), SMASH ontology (smash)
Text	489	3.02 %	AlchemyAPI	The Knowledge Diversity Ontology (kdo)
Real Estate	477	2.94 %	Zillow	OpenLink Zillow Ontology
Stocks	468	2.89 %	Nasdaq DOD	-
Weather	459	2.83 %	Weather Underground	Home Weather (hw), The Linked Earth Ontology (earth)
Food	396	2.44 %	FoodFacts, Spoonacular's Food API	Food Ontology (fo)

3

Definitions, Notation and Semantics

More often than not, the narrow capabilities offered by an individual service operation cannot satisfy complex needs, in terms of functionality and comprehensive coverage of a domain. Service composition methods aim at combining several service operations to fulfill a request. For example, to retrieve the titles of a person's publications, when there is no a single service operation that can achieve such a request on its own, a composition method may combine operations that return publication DOIs given a researcher's name and publications titles given a DOI.

This chapter starts by introducing preliminary concepts that will be used in the rest of the chapter. Then, the chapter presents the formalization of the semantic description model for Linked REST APIs, and describes the matchmaking methodology that identifies existing web services that can potentially be used in a composition.

3.1 Preliminaries

The foundation for automatic service composition is the semantic description of the underlying resources. Linked REST APIs (LRAs) make extensive use of the Resource Description Framework (RDF) [22] and SPARQL [59], as the data model for representing information about the resources, and the mechanism to provide access to such data.

RDF is a graph-based data model, generally used for conceptual description or information modeling, which facilitates integration of data among multiple sources, even if the underlying schemas differ. In this section, the basic RDF notions used throughout the chapter are introduced.

The graph data model in RDF is based on triples. Each triple, formed by a subject, a predicate, and an object (also called property), denotes a statement of a relationship between the things

represented by the nodes that it links.

Definition 3.1.1 (Triple) *A triple $t = (s, p, o)$ is a triad, such that $t \in (U \cup B) \times U \times (U \cup B \cup L)$, where U, B, L are disjoint infinite sets of URIs, blank nodes, and literals, respectively. A triple t is commonly referred with the tuple (s, p, o), due to its association of its elements, where s is called the subject, p the predicate and o the object.*

A set of such triples is called an RDF graph. This graph can be depicted by a set of nodes and directed edges, in which each triple is represented as a node-edge-node link.

Definition 3.1.2 (RDF Graph) *An RDF graph is a finite set of triples. Commonly, a set of triples is represented by concatenating them with . (dot) operators. Alternatively, triples can be concatenated with ; (semicolon) when the following triple uses the same subject, or with a , (comma) when the following triple uses the same subject and predicate.*

In order to query an RDF graph, SPARQL uses an abstraction of the triples, called triple patterns, which allow variables in every element of the triplet. Typically, a query is a composition of triple patterns, in what is known as basic graph patterns.

Definition 3.1.3 (Triple pattern and Basic graph pattern) *a triple pattern t is defined as a triplet from $(s, p, o) \in (U \cup V) \times (U \cup V) \times (U \cup L \cup V)$, where U, L, V are disjoint infinite sets of URIs, literals and variables. Accordingly, a basic graph pattern B is a finite set of triple patterns.*

Basic graph patterns constitute the basis of SPARQL pattern matching against RDF graphs. In this matching procedure, variables are replaced by a solution function that maps query variables to RDF terms. In order to define a graph pattern match, we represent variable bindings as a mapping function $\mu : V \rightarrow U \cup L$, that relates variables in the basic graph pattern to URIs and literals. As an abbreviation, we denote $\mu(B)$ as the set of triples obtained by replacing the variables of the triples in B according to the mapping function μ.

Definition 3.1.4 (Basic graph pattern matching) *Considering B as a basic graph pattern, and G as an RDF graph, a mapping function μ is a solution for B from G, when there is a pattern instance mapping of B such that $\mu(B)$ is a subgraph of G.*

Basic graph patterns only consider a match when the entire pattern has a solution mapping, however due to the irregular structures often found in RDF graphs, SPARQL allows to define optional grouping blocks of basic graph patterns where the match is not rejected if the optional block

does not have a complete binding. Moreover, SPARQL also provides FILTER expressions, which eliminates the solutions that, when substituted into the expression, evaluate to false or produce an error.

In addition to the basic definitions, it is important to note that SPARQL has four query forms, namely SELECT, CONSTRUCT, ASK, DESCRIBE; but the most common form is SELECT, which returns the variables bound in query pattern match. A comprehensive treatment of the definitions and evaluations of SPARQL can be found in [22, 59, 60].

3.2 Motivating Example

In order to understand the challenges of REST API integration, in this section, we present an example that illustrates how different APIs may converge in our integration middleware. An example that demonstrates the need for automated service-integration systems can be seen in the application domain of research-program evaluation. The consideration of several sources of evidence, including research outputs, impact metrics, presentations, and grant funds, is crucial to a comprehensive evaluation. While many organizations, such as digital libraries and funding agencies, offer Web APIs that provide programmatic access to metadata of millions of publications and grants, each one of them offers only a fragmented view of a researcher's activities and contributions. None of them contain all the publications of an individual author (publisher-specific repositories are unlikely to include publications in venues beyond the ones owned by the publisher) nor do they contain all the information relevant to program budgets assigned to researchers.

In order to illustrate our approach, we consider a typical scenario that involves collecting the publications of a particular researcher. We will assume there are two relevant service providers, *MCA* and *EEEI*, and the developer only knows the name of the author. The information managed by the two service providers is similar, but has differences in coverage, response structure, and interface. In the rest of this document, we will use a set of operations from each service provider, exposing data through REST services. *MCA* has one operation: `getPapers`, which receives as an input a person's email, and returns a list of academic papers created by the person. And *EEEI* has two operations: `searchAuthors`, which receives as an input a person name, and returns a set of authors having that specific name; and `getDocuments`, which returns a list of academic papers created by an author, whose identifier is received as an input. This scenario highlights our motivation, since it can show how an input that is seemingly incompatible with the document-related service operations, can be transformed into a service composition task to find semantically linked resources that can be used, in conjunction, to achieve the goal. In the following sections, we describe how our approach is used to describe and compose semantically linked resources on the Web.

3.3 The LRA Data Model

The essential component in any framework for automatic (or semi-automatic) discovery and composition of services is the machine-understandable descriptions of the underlying resources. Hence, the main design consideration in developing the specification language for these descriptions is the approach to map the semantics of the data provided by the web services to the corresponding concepts of the mediating schema.

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 basic graph patterns. LRAs introduce a simple vocabulary for the semantic representation of functional and non-functional attributes of Web APIs; this vocabulary is based on MSM, and incorporates the structural organization of widely adopted syntactic API description formats, such as OpenAPI[1] and RAML[2], thus simplifying their integration in the LRA middleware. The vocabulary is guided by two of the design principles defined for the minimal service model in WSMO-Lite [36]: Proximity to underlying standards, in this case OpenAPI and RAML; and Lightweight minimality, adding only very few new constructs on top of the underlying technologies that are already well-known. The vocabulary defines a *Service* element that acts as the container of the resources and operations of a service provider. Each *Service* has a number of associated *Operations*, which, in turn, are associated with graph patterns that represent the underlying data schema of the Web API operation. The *Operations* are linked to input and output elements within these graphs. Figure 3.1 depicts the LRA model. The rationale behind describing the operation as a connected graph that links inputs and outputs is that simple model references, such as the ones typically used in SAWSDL [10] and OWL-S [30], cannot fully express the required semantics. For example, knowing that, in the `getPapers` operation of the *MCA* service, the input is the person's email, and the output a set of articles, is not enough to identify the person in the input as the author of the papers in the output, since there are many possible relations between person and papers, such as reviewer or editor. Besides, in practice, services may return response documents with information about more than one type of entity that use the same property, like `rdfs:label`, which would cause ambiguities in the interpretation.

Definition 3.3.1 (Linked REST APIs) *LRAs are specified as a set of operations, where each operation S is defined by a 6-tuple*
$S = (E, P, I, O, A, T)$ *of endpoint information, a graph pattern, inputs, outputs, authentication mechanisms, and quality attributes, respectively.*

[1] The OpenAPI specification is a machine-readable description format of RESTful Web services, first proposed in 2010, and originally known as the Swagger specification. Specification available at `https://www.openapis.org/`

[2] The RESTful API Modeling Language (RAML) is a YAML-based language, first proposed in 2013, used for describing RESTful APIs. Specification available at `https://raml.org/`

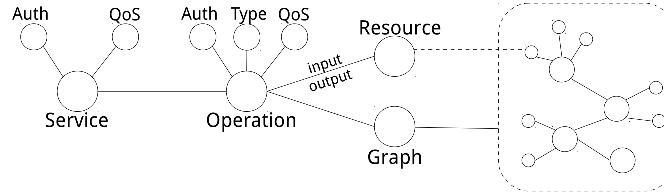


Figure 3.1: Linked REST API Model.

Consider for example the `getPapers` operation. The endpoint $E = \{urlp, method, type\}$ denotes the set of grounding parameters of the service, namely the URL pattern (similar to URI templates in hRESTS [36]), the HTTP request method, and the type of operation. This type of operation is based on Hydra [61], and is used to indicate if an operation results in resources being read, created, deleted, or replaced. In this example, the request method is *GET*, the type is *ReadAction*, and the URL pattern is defined by `http://api.mca.com/persons/papers?email={email}`, where `{email}` is meant to be replaced with an input value.

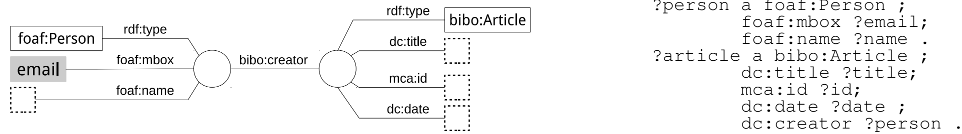
The pattern P defines the graph pattern, composed of triple patterns, representing the data managed by the operation. In our example, the operation `getPapers` relates persons to their articles, exposing information about the name of the person, and the titles, identifier numbers, and publication dates of the articles. Table 3.1 shows a diagrammatic and a textual representation of the graph patterns from the operations used in our motivating example.

The input mapping function I is a partial function I : $\mathsf{nodes}(P) \rightarrow \{itype, iname\}$ that defines a mapping between nodes (elements found in subjects and objects) of the graph pattern to their associated type of input and grounding name. The type of input is defined by establishing the cardinality (required or optional), and the type of match (full or partial, specially for search functionalities that allow to specify a portion of the term). In the example operation `getPapers`, there is only one input, the email, which is required, full match, and is mapped to the url substitution pattern `{email}`.

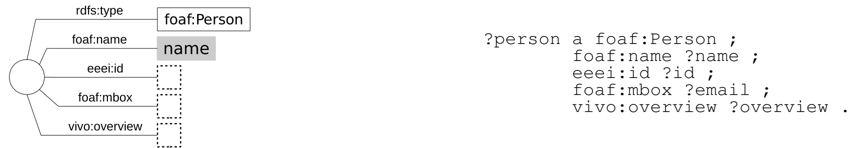
The output mapping function O is a partial function O : $\mathsf{nodes}(P) \rightarrow path$ that defines the mapping between nodes in the graph pattern to their associated location path, represented as an XPath-like expression. In the operation `getPapers`, for example, the title of the documents `?title` is mapped to the expression `/papers/title`, which indicates that the titles are located in the title value, within the papers object in the response document.

The authentication information A denotes the set of parameters used by the authentication mechanism of the operation, which will vary depending on the authentication protocol.

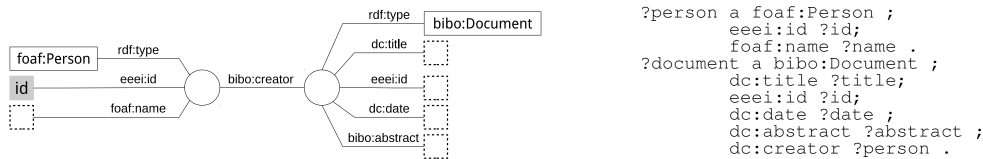
The quality metrics function T is a mapping function $T : metric \rightarrow \mathbb{R}$ that defines associations



(a) Operation `getPapers` from *MCA*



(b) Operation `searchAuthors` from *EEEI*



(c) Operation `getDocuments` from *EEEI*

Table 3.1: Graph representations and textual graph patterns of the data in the example services. The dotted boxes in the diagrams denote the attributes used as outputs, and the gray boxes denote the inputs.

between quality metrics of the operation, such as availability or latency, with its associated value. This information can be used to discard composition chains with operations below some thresholds.

3.4 Service Discovery

In order to leverage the LRA semantic descriptions, SPARQL is used to express a service composition request. Therefore, the SPARQL query graph pattern is used to identify 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.

SPARQL queries consider several operators, such as OPTIONAL, UNION, FILTER, and concatenation to construct graph pattern expressions, which provide an equivalent expressive power to relational algebra [62]. However, Perez et al. [60] demonstrated that the complexity of evaluation of query patterns constructed by using concatenation, FILTER, and UNION operators is NP-complete, and when OPTIONAL operators are included, the complexity is PSPACE-complete. At the same time, restricting operators and imposing restrictions on optional blocks, in what they call *well-designed graph patterns*, yield to more efficient query evaluations.

Although we acknowledge the expressiveness and usefulness of disjunctions, negations, and

property path expressions, LRA SPARQL queries are limited to positive and conjunctive queries. Our approach restricts the queries to well-designed graph patterns, which are very common in practice, and simplify the execution of the query plans. More specifically, LRA SPARQL queries are union-free basic graph patterns, restricted to a single optional block.

Definition 3.4.1 (Query graph pattern) *A query graph pattern expression for LRA, representing a declarative service composition request, is defined as a conjunction of a basic graph pattern, denoted by Q_R, and an optional basic graph pattern, denoted by Q_O. Then, using SPARQL's grouping syntax, $Q = Q_R \text{ OPT}\{Q_O\}$.*

Therefore, a query from our motivating example, which retrieves the titles and (optionally) publication dates of the documents authored by 'John Smith' (a fictional name), can be represented as the following query:

```
Q : SELECT ?title ?date
    WHERE {
          ?author a foaf:Person ;
               foaf:name "John Smith".
          ?doc a bibo:Document ;
               dc:creator ?author ;
               dc:title ?title
          OPTIONAL { ?doc dc:date ?date }
    }
```

Once the query has been defined, the first step towards a service composition is to identify service operations that may provide relevant information, by finding the operations whose graph patterns can be mapped, at least partially, on the query pattern. Such a mapping is defined based on the equivalence of triple-pairs. Informally, the concept of triple equivalence implies that, even though a triple may not have an exact counterpart, an equivalent triple can be established based on inference rules. For instance, a query triple `?x a bibo:Document` can be mapped to a triple `?x a bibo:Article`, since `bibo:Article` is subsumed by `bibo:Document`.

Roughly speaking, the basic inference rules in LRA are based on a subset of properties of RDFS, namely *subPropertyOf* (sp), *subClassOf* (sc), and *type* (type), and two properties of OWL, *equivalentClass* (ec) and *equivalentProperty* (ep). *subClassOf* and *subPropertyOf* are binary properties reflexive and transitive that, when combined with *type*, specify that the type of an individual or property can be lifted to that of a superclass or superproperty, respectively. In turn, *equivalentClass* (and *equivalentProperty*) states that two classes (or properties) have the same class (or property) extension. Formally, entailment properties are expressed as first order formulas:

$$\forall c_1, c_2, c_3(\text{triple}(c_1, \text{sc}, c_2) \wedge \text{triple}(c_2, \text{sc}, c_3) \supset \text{triple}(c_1, \text{sc}, c_3))$$
$$\forall s, c_1, c_2(\text{triple}(s, \text{type}, c_1) \wedge \text{triple}(c_1, sc, c_2) \supset \text{triple}(s, \text{type}, c_2))$$
$$\forall c(\text{triple}(c, \text{type}, \text{Class}) \supset \text{triple}(c, \text{sc}, c))$$
$$\forall p_1, p_2, p_3(\text{triple}(p_1, \text{sp}, p_2) \wedge \text{triple}(p_2, \text{sp}, p_3) \supset \text{triple}(p_1, \text{sp}, p_3))$$
$$\forall s, p_1, p_2, o(\text{triple}(s, p_1, o) \wedge \text{triple}(p_1, sp, p_2) \supset \text{triple}(s, p_2, o))$$

$\forall s, c_1, c_2(\text{triple}(s, \text{type}, c_1) \wedge \text{triple}(c_1, \text{ec}, c_2) \supset \text{triple}(s, \text{type}, c_2))$
$\forall s, p_1, p_2, o(\text{triple}(s, p_1, o) \wedge \text{triple}(p_1, \text{ep}, p_2) \supset \text{triple}(s, p_2, o))$
$\forall p(\text{triple}(p, \text{type}, \text{Property}) \supset \text{triple}(p, \text{sp}, p))$

There are three reasons why this work is restricted to these properties only: first, the prevalence of the selected properties in semantic data representations, which according to Tomaszuk [63] account for around 40% of the most used terms at LOD; second, the relatively low computational complexity of the equivalence implied by the properties; and third, the impossibility to reason on instances (e.g. *owl:sameAs*) or enforce constraints based on instances (e.g. *minCardinality*) due to the autonomy and data authority of Web APIs that restrict full access to the data they provide. For simplicity, we will designate the subsumption and equivalence properties with the symbol $\models$, which denotes that a triple can entail another triple and lays the foundation to formally define triple equivalences.

Definition 3.4.2 (Triple equivalence) *A triple t' from a query graph pattern is considered equivalent to triple t from a service operation graph pattern, denoted by $t \equiv t'$, if $t' = t$ or $t \models t'$, where the particular variable names are ignored, and URIs and Literals of t' can be mapped to variables in t.*

Hence, the mapping of a service operation to a query is defined as the set of equivalent triple-pairs of their corresponding graph patterns, as follows.

Definition 3.4.3 (Query graph mapping) *Given a query graph pattern Q, a mapping μ is a partial function $\mu : T_Q \rightarrow T_{S_i}$, which indicates how triples of Q are mapped to triples in an service operation graph pattern S_i. Two mappings μ_1 and μ_2 are considered compatible when for all $t \in \text{domain}(\mu_1) \cap \text{domain}(\mu_2)$, it is the case that $\mu_1(t) \equiv \mu_2(t)$.*

In order to represent the semantics of query-operation match evaluation, we adapt the relational algebra definitions of the operations of join and left join of two sets of mappings m_1 and m_2. Intuitively, $m_1 \bowtie m_2$ is the result of extending the mappings in m_1 with the compatible mappings in m_2; and $m_1 ⟕ m_2$ extends the elements in $m_1 \bowtie m_2$ with the mappings in m_1 that are incompatible with the mapping of m_2, in order to support the optional operator.

$$m_1 \bowtie m_2 = \{\mu_1 \cup \mu 2 \mid \mu_1 \in m_1, \mu_2 \in m_2 \text{ , and } \mu_1 \text{ and } \mu 2 \text{ are compatible}\}$$
$$m_1 ⟕ m_2 = m_1 \bowtie m_2 \cup \{\mu_1 \in m_1 \mid \forall \mu_2 \in m_2 \text{ , and } \mu_1 \text{ and } \mu_2 \text{ are incompatible}\}$$

Considering the definitions of query graph mappings and join operations, our approach determines whether an API operation may provide data related to the query graph pattern, as follows.

Definition 3.4.4 (Query-API subgraph match) *The evaluation of the subgraph of a query graph pattern Q, over a subgraph of a service graph pattern S_i, denoted by $[\![Q]\!]_{S_i}$, is defined recursively as follows.*

1. *If Q is a triple pattern t, then* $[\![Q]\!]_{S_i} = \{\mu \mid \text{domain}(\mu) = t\}$.
2. *If Q is (Q_1 AND Q_2), then* $[\![Q]\!]_{S_i} = [\![Q_1]\!]_{S_i} \bowtie [\![Q_2]\!]_{S_i}$.
3. *If Q is (Q_1 OPT Q_2), then* $[\![Q]\!]_{S_i} = [\![Q_1]\!]_{S_i} ⟕ [\![Q_2]\!]_{S_i}$.

For example, considering the query presented previously, which retrieves the titles and publication dates of the documents authored by 'John Smith', with respect to the `getPapers` operation, the following match is identified:

```
?author a foaf:Person → ?person a foaf:Person
?author foaf:name "John Smith"→?person foaf:name ?name
?doc a bibo:Document → ?article a bibo:Article
?doc dc:creator ?author → ?article dc:creator ?person
?doc dc:title ?title → ?article dc:title ?title
?doc dc:date ?date → ?article dc:date ?date
```

Besides identifying query triples matching service operation triples, only the subgraphs produced by $[\![Q]\!]_{S_i}$ that are part of a connected component are considered to be a valid match, since disconnected components can lead to inconsistent compositions. Hence, in a connected subgraph match $[\![Q]\!]_{S_i}$ their triples form a connected component, like in the previous subgraph match example.

Finally, unlike RDF data graphs, data from Web APIs is only accessible when a list of required inputs and authentication credentials are in place. Thus, in addition to being a connected subgraph match, the match needs to comply with the input interface of the API operation. For instance, although the query and the `getPapers` operation form a connected subgraph match, the service operation requires the email as the input, which is not provided in the query, thus causing the match to be incompatible. Then, in order to consider a Web API as relevant and accessible, the API graph pattern has to form a compatible subgraph match with respect to a subgraph pattern of the query.

Definition 3.4.5 (Compatible subgraph match) *Given a service operation graph pattern S_i, a connected subgraph match is considered compatible if the node mapping function M, that relates query pattern nodes with their mapped service operation pattern nodes based on μ, satisfies that all the required input resources, denoted by $I^*_{S_i}$, have a mapping in M to a concrete value. More formally, when $\forall n, n \in \text{domain}(I^*_{S_i}) \wedge M(n) \in (U \cup L)$, where U and L denote URIs and Literals, respectively.*

In summary, The LRA data model provides a description that enables automated discovery, composition and invocation of REST APIs. This description relies on a semantic specification of

the input-output functionalities of the APIs and their non-functional attributes, described in terms of Linked-Data. The LRA data model for the semantic description of web services builds on lessons learned from a long history of semantic web services and service-description formats.

4

Service Composition and Orchestration

The main objective of a discovery and composition middleware is to leverage the service descriptions in order to automate the process of reusing these services in software applications, and to reduce the manual work required by software developers. In order to improve the current state of service integration, we present the LRA middleware, where developers specify the data needs of their applications in SPARQL queries, and in response, it provides a consolidated data graph, produced through the composition of possibly multiple REST API chains, taking into account the application developer's credentials and desired quality requirements.

This chapter introduces the methodology and algorithms to leverage the semantic descriptions for automatic service composition, and explores the required considerations for composition enactment. Additionally, the chapter discusses the challenges to validate sophisticated methodologies with existing data collections, and in response, proposes a service test collection, which is used to validate the Linked REST API approach.

4.1 LRA Composition Algorithm

In the context of the LRA framework, the service-composition problem is formulated as that of finding a sequence of API operations $S = \langle S_1, \cdots, S_n \rangle$, having as inputs a set of service operations $R = \{S_1, S_2, \cdots, S_m\}$ described as basic graph patterns, and a query graph pattern Q. The produced sequence of API operations must meet the following constraints:

– Each service operation graph pattern has a non-empty mapping with the query graph pattern,

– Each required query triple is mapped by at least one of the operation triples,

– The variables in the SELECT query are included as the output in one of the service operations, and

– The service operations, when executed in sequence, form compatible subgraph matches. This implies that after each step in the composition chain, a new query, referred to hereinafter as the expanded graph, is created by extending the query graph pattern with the output triples of preceding service operation graph patterns, which at this point can be used as inputs.

In our example, since only the name of the author is available at the beginning, then only `searchAuthors` is compatible and matches part of the query that asks for the publications of an author. After `searchAuthors` is invoked, the composition algorithm can use the id and email returned by the operation in the next step of the composition, and invoke the service operations `getDocuments` and `getPapers`, which would both cover the initial query, resulting in two alternative composition chains.

The composition of LRA operations consists of an iterative process with three main steps: *pairwise matching*, *blocking*, and *pruning*. These steps are repeated until *convergence* in a limited-depth exploration. These steps are described in more detail next, but it is worth keeping in mind that *pairwise matching* is used to ensure the semantics of the composition, while *blocking* and *pruning* are used to achieve good performance. As an additional step, the application of a set of *heuristic rules* refines the compositions and discards redundant composition chains.

4.1.1 Pairwise Matching

At a high level, the composition problem can be represented as a graph-search problem, where the global graph $G = (V, E)$ consists of a (finite) set of service operations denoted by V, and a collection E, of ordered pairs $\{u, v\}$ of distinct elements from V, where $\{u, v\}$ exists if the inputs in the current query that lead to u, make v a compatible subgraph match of the query. After each node is visited, the expanded query is updated by adding the outputs of the current node.

Consequently, the basic step of service composition is pairwise matching, reusing the discovery methodology from the previous section, and then deciding whether or not a service is a compatible subgraph match and can contribute to the overall goal. Our approach checks that the service is a compatible subgraph match, using an adaptation of Ullman's algorithm [64] that considers subsumption and equivalent triples. Subgraph isomorphism is a NP-complete problem, and the complexity of Ullman's algorithm is exponential on the number of vertices of the input graphs.

For example, due to compatibility, only `searchAuthors` is considered as the first node in the composition chain; and thanks to subsumption and equivalence, the concepts in the query

(particularly, `bibo:Document`) are matched to the concepts in `getPapers` (which uses the concept `bibo:Article`).

4.1.2 Blocking

Pairwise matching until query coverage ensures the desired goal of service composition, but may be quite inefficient and even infeasible for a large set of services. The main cause of inefficiency is that pairwise matching may require an exponential number of comparisons to decide which subgraphs are isomorphic matches. Our method, inspired by the work of Garcia et al. [65], attempts to reduce the search space by using a blocking function that retrieves only the services that can use at least one of the inputs in the query.

For example, an attempt to match the initial query that retrieves documents from 'John Smith' with the operation `getPapers` would be unnecessary, since at that point, the email of the author is unknown, and thus, the operation cannot be invoked.

4.1.3 Pruning

As the global graph is being explored, some branches can be discarded without examining them. The pruning technique for LRA examines negative matches and passes on this information to its ancestor nodes, in order to avoid future exploration of those nodes, which cannot possibly influence the final decision, due to the extension principle of expanded queries.

Consider the exploration of the composition chain $S = \langle a, b, c\rangle$, where only a and b but not c were successfully matched. This finding indicates to node a that, when using the same query, an exploration of a composition chain of the form $S' = \langle a, c, ...\rangle$ will fail. The general principle is that, as matches only add triples to the query (and possibly, flag nodes as outputs), then if c cannot be matched to the original query plus the added triples from a and b, then it will not be matched to the original query plus the added triples from a, which is only a subset of the previous case.

4.1.4 Convergence

The exploration of the global graph of service operations may end up in an unbounded graph problem, preventing the comprehensive creation of composition chains for a query, if full traversals are used. Hence, our approach uses a depth-limited search that prevents impractical exploration by limiting the depth of the search to some value l.

The implementation of the algorithm follows a recursive approach to dynamically compose the services, based on graph matching, as outlined in Algorithm 1. The process starts by extracting the triples, using Apache Jena, and identifying the potential inputs (line 6). The first composition step, described in the *blocking* section, aims at restricting the number of pairwise comparisons, by

only considering service graphs that can use at least one of the inputs in the query graph (line 7). Then, the algorithm checks if the candidate operation is in the blacklist for the parent or successive operation levels, and, if so, discards it, implementing the *pruning* step (line 9). The next step uses subgraph isomorphism to determine whether the query graph is present within one of the service graphs, as explained in the *pairwise matching* section (line 11). As long as the match covers new elements of the query graph, the process keeps exploring composition chains from the current service (lines 12-21). Otherwise, the service operation is added to the blacklist of the grandparent level for future pruning (line 23). If the addition of the current service covers all the required elements of the query graph, then the service is added to the composition chain, and the algorithm stops exploration from that service (lines 13-14). The algorithm is applied recursively, creating an expanded query that integrates the output of previous services in the composition chain, until all the composition chains that cover all the required triples in the initial query graph are found within a given depth (lines 16-20).

Algorithm 1: Composition through graph matching

Global : An array of blacklisted operations for each depth level $pruning$ (initially empty).
Input: A query graph Q, A finite set $R = \{S_1, S_2, ..., S_n\}$ of Web API operation graphs, a maximum exploration depth l, and a current depth $depth$.
Output: A list of composition chains S.

```
1  Function compose(Q, R, l, depth)
2      S = ∅
3      if l > max then
4          return S
5      while Q is not covered do
6          in = getInputs(Q)
7          operations = getCandidates(R, in)
8          for each S_i in operations do
9              if S_i ∈ ⋃_{i=level-1}^{depth} pruning[i] then
10                 continue
11             match = getMatch(Q, S_i)
12             if match covers new elements of Q then
13                 if match covers the rest of Q then
14                     add S_i operation to S
15                 else
16                     Q+ = Q + output of S_i
17                     next = compose(Q+, R, l, depth + 1)
18                     if next is not empty then
19                         prepend S_i to chains in next
20                         add chains in next to S
21                 pruning[depth] = ∅
22             else
23                 add S_i operation to pruning[depth − 2]
24     return S
```

4.2 Refinement Heuristics

The previous steps can produce a large number of composition chains that may be redundant or suboptimal. That effect was observed in the first implementation of the middleware, in collaboration

with IBM Canada, in the context of a project that aimed to analyze the activities and productivity of their collaborations with members of academic institutions. Based upon the experience with the aforementioned project, we have devised four heuristic rules, shown below, that attempt to reduce the number of composition chains, while maintaining the same data coverage.

- *Duplicates chains are eliminated.* Since some composition chains can be configured in parallel execution, at least in part of the chain. Hence, a composition may exhibit commutative properties where $S_1 = \langle a, b \rangle$ and $S_2 = \langle b, a \rangle$ both can be included in the solution of the aforementioned algorithm. Since the structural properties of the composition pattern would produce the same result, then this rule filters out chains that are permutations of a previously processed chain, in this case discarding S_2.

- *Shorter chains are preferred.* This rule discards the compositions that are superset of other chains, considering that shorter composition chains are more desirable because they preserve data semantics at a lower cost. For example, when two composition chains $S_1 = \langle a, b \rangle$ and $S_2 = \langle a, b, c \rangle$ are identified, then S_2 is discarded.

- *Chains involving fewer providers are preferred.* This rule discards a composition chain that involves multiple service providers, if there are other shorter chains that already extract data from such providers. For example, assuming that services a and b are provided by R_1, and c is provided by R_2, if three composition chains $S_1 = \langle a \rangle$, $S_2 = \langle b, c \rangle$ and $S_3 = \langle c \rangle$ are identified, then S_2 is discarded, because the data from R_1 and R_2 can be extracted more directly with S_1 and S_3.

- *Chains where all providers are involved in shorter chains are eliminated.* Assuming that if a service provider already participates in one chain, then there is no need to consider the provider in other chains, unless it is used to extract data from other provider that is not already part of one of the chains. For instance, assuming services a, b, and c are provided by R_1, and service d is provided by R_2, if we find three composition chains $S_1 = \langle a, b \rangle$, $S_2 = \langle c \rangle$ and $S_3 = \langle a, d \rangle$, then we can keep chains S_2 and S_3, because S_2 is the shortest composition that allows to extract data from R_1, and S_3, although is also using a service from R_1, is used as the only way to extract data from R_2, .

4.3 Service Orchestration

The LRA middleware encompasses discovery, composition and enactment of Web services, based on the methodology described in the previous sections. The process starts with a declarative composition request, expressed as a SPARQL query, and optionally a set of desired quality thresholds. The query is then parsed, extracting the graph pattern and the parameters used as inputs and outputs.

The query is passed to the composition engine, where the query graph pattern is matched against a service repository containing LRA service descriptions, following the procedure outlined in Sections 3.4 and 4.1. After a set of composition chains have been identified, the access control and quality component discards some composition chains, by comparing the service authentication requirements and expected quality of the chains against the credentials and quality thresholds provided by the developer. Subsequently, the enactment component invokes the individual service elements of each chain, lifts the response data with semantics, and consolidates the data from multiple composition chains.

In this section, we focus on the description of the components that complement the data model and the matchmaking methodology, and enable the invocation of LRA composition chains.

4.3.1 Access Control

Protection against unauthorized access is an essential part of service composition. The LRA framework ensures that only users with the proper credentials on a service can include its operations in their applications.

In the early days of Web APIs, access control was implemented through the use of basic authentication, or by embedding username and password information in each API invocation. However, these authentication methods give the intermediate software access to user's private account information, which can be subject to misuse or unintentional exposure. In order to overcome the deficiencies of basic security schemes, a simple method, via API Key received upon account creation, became popular among web-service providers. Nevertheless, this mechanism does not have any security measures for confidentiality, and thus, provides limited security. Eventually, OAuth, a protocol for making authenticated HTTP requests using a token, was proposed. In OAuth, whenever there is a need to access web services, the requestor is asked to provide the relevant credentials to the invoked resource, while in the background, OAuth creates a token, which can be used by subsequent API calls. Then, the username and password are kept private and unavailable for third-party applications. A few years ago, Maleshkova et al. [66] conducted a survey of web service authentication mechanisms, finding that more than 80% of the APIs have some kind of authentication, with the API key mechanism being the most popular.

In LRA descriptions, we propose a comprehensive description of services, including annotations for authentication requirements, in such a way that the framework knows which security credentials have to be provided by the user, in order to invoke the web services. Then, the composition chains generated from the previous step can be refined. For example, if *MCA* requires an API Key, but the user does not provide it, then, the system would not be able to create a valid request, therefore, invalidating all the composition chains where a service from *MCA* appears, at design time.

4.3.2 Provenance and Transaction Logging

Information comes from numerous sources, each exhibiting a different level of quality. In the LOD project, for example, it is common to find inconsistencies among different versions. Similar problems with inconsistencies and variable data quality are expected when integrating web services. This leads to the general optimization problem of selecting a composition of web services for each query so that the provenance can be used to estimate and maximize its overall quality and cost requirements. Provenance-aware composition has been addressed extensively by other authors [67, 68], proposing several approaches based on exact algorithms or elaborate heuristics.

The LRA framework proposes a simple approach for quality assessments, where developers or information-system managers declare a proposal with desired levels of quality, for attributes such as response time and availability. Then, these features, along with provenance information, can be used to rank and discard composition chains that, in conjunction with other services in the chain, fail to provide the quality specified by the contract, or have a low rank. For example, if the user specifies a maximum latency of 1.5 seconds, but the estimated composition of the services in *EEEI* are of 1 second for `searchAuthors`, and 1 second for `getDocuments`, then the composition chain should be discarded.

Additionally, the LRA framework records the transformation that a SPARQL query undergoes, logging the generated composition chains and their invocation responses, in a transactional chain that developers can inspect by request. The middleware will use the information extracted from the responses to create summary statistics, for example, the service availability or average number of records returned, in order to improve the quality of the composition.

4.3.3 Data Extraction and Lifting

When the composition chains are refined, the invocation manager uses the service description, along with the inputs and security credentials as grounding information. Once it receives the responses from the invoked API, the middleware ‘lifts’ the data contained in the structured documents. *Lifting* refers to the transformation of the API responses, from its representation at the syntactic level into its corresponding semantic representation, in RDF. In some web service formalizations, such as SAWSDL, the description includes special attributes for a lifting link, which was intended to link to an Extensible Stylesheet Language Transformations (XSLT) script. But, XSLT is quite complex and has never been widely adopted. JSON-LD provides a good alternative to ‘lift’ service responses, but still it is not popular among current web service implementations. Hence, in our implementation, we rely on location path expressions, that can act on XML and JSON documents.

Location paths were informally presented in the example introduced in Section 3.3. In the example, the location path expression `/papers/title` indicates that the particular output data is

located in the title value, within the papers object in the response document. Formally, a location path expression $\pi = /s_1/\cdots/s_n$ consists of a number of locations steps (denoted by s_i), where s_1 starts at the root element of the structured document. A location step has two parts: a *node name*, which specifies the name of the node selected by the location step, and zero or more *predicates*, which use arbitrary expressions to further refine the set of nodes selected by the location step. The syntax for a location step is the node name, followed by zero or more expressions in square brackets. For example, in `/papers[type="Book"]`, `papers` is the node name and `[type="Book"]` is a predicate. The set of values selected by the location step is the set of values produced by the node name, and then filtering those values by each of the predicates in turn. If a step in the location path returns an empty set of values, then the result of executing a location path is empty as well. Additionally, location paths can also be expressed as simple functions $f(\pi_1, \cdots, \pi_n)$ that perform operations on the values returned in the values of the location paths in the arguments, and return a set of values. Examples of functions are `concatenate` and `tokenize`.

An output model for a service operation S_i can be formed by combining all the location paths included in the operation description, denoted as $\Pi = \bigcup_{\pi \in S_i} \pi$. The output model forms a tree where the leaf nodes represent the values that will be extracted from the document. Then, given an output model Π and a structured document D, a function $extract(\Pi, D)$ explores D, reconstructing the result document as a set of tuples, according to Π.

4.3.4 Record Linkage

When the Semantic Web was envisioned, URIs were proposed as global identifiers that cross-reference entities across repositories, and the `owl:sameAs` primitive was meant to establish identity between synonymous URIs. Beyond LOD repositories, however, and for most domains, there are no such global identifiers even when there is significant overlap in the repository contents, which, in effect renders these repositories into isolated silos. The problem is exacerbated by naming-convention differences, different representation formats, and errors, such as typos. The LRA middleware employs blank nodes to represent objects whose URI is not provided, and in the post-processing step, it uses the literal values associated to the nodes in order to deduplicate the records in the generated graph, in a process known as record linkage.

The term *record linkage* denotes the task of identifying records across a number of different data sources that refer to the same real-world entity. In order to integrate the results across the composition chains, we implement a simple clustering algorithm that considers accidental or deliberate errors, based on the work of Hernández and Stolfo [69]. This clustering technique improves the pure quadratic time process (i.e., comparing each pair of records) by sorting the entire data set in order to bring the matching records close together. The algorithm can be summarized in three phases: Create Key, Sort Data, and Merge.

The *Create Key* phase computes a key for each record by extracting relevant fields or portion fields that allow to partition the data in such a way that restricts the candidates for matching to a small portion of the records. Therefore, clustering becomes more efficient when applying quadratic time algorithms to each portion. Evidently, a good candidate key is the attribute with the most discriminatory power, which considering the schema-less nature of our dataset, is the type of resource.

The second phase, *Sort Data*, simply sorts the records using the key from the previous phase. In our dataset, each record represents a referent (non-literal) node, and contains all the literal values associated to it.

And finally, the *Merge* phase scans the sorted list of records, limiting the comparisons for matching records to those records with the same key. The equivalence of records during this phase is based upon the computation of a distance function (i.e., edit distance) applied to the common attributes of the pair of records being compared. Two records are considered equivalent if there is at least one common attribute, and the number of equivalent attributes is above a specified relative threshold.

4.3.5 Caching

In addition to the complexities of lifting and linking records, fetching the data may become a time consuming task, in the presence of multiple data sources. Commonly, data integration systems consider two categories of architectures: *warehousing*, and *virtual integration*. In the *warehousing* architecture, the data from the individual data sources are loaded and materialized into a physical database, where queries over the data can be answered. On the other hand, in a *virtual integration* architecture, the data remain in the sources and are accessed as needed at query time [70]. Sometimes, when the data sources are highly dynamic, a full virtual integration architecture is ideal. Nevertheless, most of the time, the system can relax its requirements, and tolerate slightly older data in exchange for better performance. In the LRA framework, we seek to improve performance, by taking advantage of cached records.

The LRA framework supplements the primary database by loading frequently accessed read data in main memory. The caching mechanism is implemented following a Least recently used (LRU) replacement policy, and can be configured to use heap memory or an external caching system, such as Redis [71].

The framework provides configurable caching for the most common types of data used during query processing and service invocation:

- **Ontology:** During pairwise matching, the framework needs to determine whether a subgraph

between a query and a service operation is isomorphic, which entails the consideration of ontology-based inferences. In order to speed up inference resolution, this cache stores the known subclasses and equivalences of ontology resources.

- **Service operation:** The framework makes extensive use of service descriptions to evaluate the relevance of a service operation to respond a query. This entails a frequent access to the repository of service descriptions. Hence, this cache maintains in memory the description of the most used service operations, reducing the access to disk to retrieve such descriptions.

- **Query plan:** Since SPARQL is a declarative query language, when a query is submitted, the framework needs to evaluate several alternative composition chains that may provide an answer to a query. Inspired by traditional database systems, in our middleware when a SPARQL statement is executed, the middleware first looks through the query plan cache to verify that an existing execution plan for the same SQL statement exists. This cache enables the framework to reuse existing plans, reducing the overhead of reevaluating the SPARQL statement.

- **Service response:** Commonly, service providers implement the use of special response headers that let developers know when caching their responses is appropriate. The headers, including the *Cache-Control* and *Expires* header, can declare whether the data should not be cached, and define the lifetime of data in the cache. The reference framework implements a web cache that, in conjunction with the information provided by response headers, maintains a temporary storage of service responses to reduce latency, by responding to recurrent requests that satisfy cache entry lifetime conditions.

4.4 Performance Evaluation

Although research initiatives have contributed to an extensive literature of description models and algorithms for semantic web services, the development of benchmarks that enable the comparison of the approaches have received significantly less attention. Typically, researchers validate their approaches with illustrative examples, or on synthetic datasets under questionable assumptions [11, 55, 72, 73]. The situation is aggravated by the unavailability of some of such synthetic datasets for an eventual comparative study, and the lack of publicly available semantic web services. This limitation precludes scientific progress in the area and casts doubts on the feasibility of such approaches in the industry.

This section starts by discussing the state of the art of semantic web service test collections, and our motivation to create a new evaluation dataset. Subsequently, the evaluation methodology is described, and applied to LRA. Finally, the results and limitations are discussed.

Table 4.1: Overview of Semantic Web Services Test Collections

Name	Formalism	# Services	# Requests	Semantic Scope	Last Update	Available	Synthetic Majority
OWLS-TC	OWL-S	1,083	42	Input/Output*	2010	Yes	No
SAWSDL-TC	SAWSDL	1,080	42	Input/Output	2010	Yes	No
WSC	Custom WSDL	3,500+	5	Input/Output	2010	No	Yes
SWS Challenge	Natural Language + WSDL	30	18	Natural Language	2009	No	Yes
JGD	Natural Language + WSDL	203	3	Natural Language	2009	Only the services	No
SWS-TC	OWL-S	240	-	Input/Output	2006	Yes	No
Assam	OWL-S	164	-	Input/Output	2005	Yes	No

* OWLS-TC version 4 includes preconditions and effects for a subset of services (180 offers, 18 queries).

4.4.1 Semantic Web Service Benchmarks

The most prominent test service collections for evaluative purposes are OWLS-TC[1], SWS-TC[2], Assam[3], JGD[4], WSC, and SWS. A summary of the test collections is presented in Table 4.1.

OWLS-TC is the most popular service test collection, with over 1,000 service descriptions in OWL-S. Besides containing service descriptions, this service collection offers a set of requests with their associated relevant service offering. Although, the OWLS-TC 4 release contains 42 requests with binary and graded relevance judgments, the authors specify that not all possible combinations have been rated, thus the relevance sets can not be considered as complete. Despite its earlier popularity, their authors have stated that the test collection can be considered just as a starting point for any activity towards achieving such a standard collection by the community as a whole. In fact, the test collection exhibits some flaws and limitations that have led to its disuse. The main limitation is inherent to the OWL-S descriptions, which restrict the semantic expressiveness of the description to inputs and outputs[5]. Additionally, the use of a specific encoding formalism prevented other approaches to use the same test collection, which derived in SAWSDL-TC, a SAWSDL test collection based on the description and requests of OWLS-TC. Apart from the limitation of the description, Küster [74] states that a substantial share of the test-collection contains errors, are unrealistic, and for some of them, the semantics of the services is incomprehensible even for a human expert. Furthermore, the responses offered in the test requests only consider the selection of a single service, which prevents its use in complex composition scenarios. Finally, this test collection was actively maintained until 2010, after which, its use in research publications has been reduced considerably. The variants of OWLS-MX and iSeM were evaluated with OWLS-TC.

The SWS-TC test collection is also based on OWL-S, comprising 240 services. Due to their semantic description format, restricted to inputs and outputs, this dataset suffers from the same

[1] `http://projects.semwebcentral.org/projects/owls-tc/`
[2] `http://projects.semwebcentral.org/projects/sws-tc/`
[3] `http://www.andreas-hess.info/projects/annotator/owl-ds.html`
[4] `http://fusion.cs.uni-jena.de/fusion/activity/jena-geography-dataset/`
[5] In version 4, a small subset of 180 service offers include definitions of preconditions and effects, but their semantic value is barely insignificant, since, for example, preconditions are mainly limited to require authentication and restrict cities for location-based services.

limitations as OWLS-TC, regarding semantic expressiveness. Although their authors claim that the test collection can be used for testing the accuracy of matchmaking algorithms, their lack of requests and responses, that can be used as ground truth, precludes its use in accuracy evaluations. This dataset was collected by Ganjisaffar et al. [75], in order to test the performance of their service discovery approach, based on a linear combination of the signature and textual similarity of services. Since its release in 2006, the test collection has not been updated.

The Assam test collection is a set of 164 OWL-S service descriptions, assembled by Hess et al. [76]. The collection was created by gathering WSDL descriptions from the now-extinct service listings *salcentral* and *xmethods*, and then by injecting semantic annotations using Assam WSDL Annotator [76]. As in the aforementioned test collections, the Assam collection only provides semantic annotation for inputs and outputs, and it does not provide sample requests and reference response services. In addition, the semantic annotations have not been updated in more than 10 years, and have not been used in other projects. METEOR-S was one of the approaches evaluated with this test collection.

The Jena Geographic Dataset (JGD) is a collection of 203 service operations in the domain of geography, extracted from public service listings like *seekda*, *xmethods*, *webservicelist*, *programmableweb*, and *geonames*. The services include natural language documentation of the input and output parameters, as well as a description of the service functionality. Besides the collection of 200 services, the authors provide subcollections of 150, 100 and 50 service descriptions. The subcollection of 50 services was used at the 2009 S3 Contest on Semantic Service Selection. Approaches like WSColab and IRS-III were among the top ranked systems in this contest. Although the authors stated that the relevance judgments would be available after the contest, only the requests in natural language, without reference judgments, were published in [74]. We tried to communicate with the authors, but we did not receive a reply.

In the second half of the previous decade, during the heyday of semantic web service research, a few collective initiatives organized events to provide forums where researchers evaluate and compare the performance of matchmaking and automated composition software. One of such forums was the Web Service Challenge (WSC), an annual event organized by IEEE between 2005 and 2010. WSC started as a syntactic composition challenge, and slowly transitioned into a subsumption-based matchmaking competition. The dataset and ontology were synthetically generated, with an emphasis on the creation of complex search spaces, which in practice are unrealistic. For instance, the authors indicated that, in the 2008 challenge, one solution required 37 services with an execution depth of 17. The dataset was revised and adjusted for every competition year. In particular, WSC09 [77] comprised more than 35,000 services, distributed in 5 challenge sets, and described in a custom WSDL format that allows to include semantic references for inputs and outputs, in a similar way as OWLS-based test collections. Since the descriptions were generated synthetically, the semantics of

the services were incomprehensible for developers. Hence, the solutions were not manually curated, instead, a BPEL checking software evaluated the correctness of solutions based on inputs, outputs, and solution path. The limited semantic information and their focus on performance time suggests WSC was essentially a search, and not a semantic composition, competition. Currently, the web page of the challenge is not available, but after communication with some of the organizers, we could obtain the dataset for WSC09 and WSC08. Some recent approaches have used WSC datasets for validations, such as [49, 44, 78], but their simplistic composition algorithms rely only on inputs and outputs, which negatively impacts the semantic correctness of their solutions.

The Semantic Web Service Challenge (SWS) [79] defined discovery problem scenarios that served as the basis comparison of approaches participating in the challenge. The competition defined three problem scenarios, one concerned with shipping of packages, the second dealing with requests to purchase computer hardware, and the third with logistics management. Each problem scenario comprised a set of levels of subproblems, with increasing level of difficulty. The evaluation of solutions was performed by a committee, who manually examined the correctness of message exchanges. The scenarios are described in English and supported with the provision of WSDL specifications. In total, the scenarios involved around 20 services and 30 service requests. The most notable approach that participated in SWS was DIANE, which was certified by the SWS committee as the most complete solution. Currently, the service descriptions and the original website are not available. After the last edition of the challenge, in 2009, research with this test collection was abandoned.

In addition to the aforementioned challenges, the S3 contests on Semantic Service Selection was an annual service discovery competition that took place between 2007 and 2012 [80]. The S3 Contest was based on the OWLS-TC and SAWSDL-TC test collections, thus the composition approaches are limited with respect to the underlying description formalism. The winner of the latest edition was iSeM [45], presented in 2012.

After the season of semantic web service competitions, the development of benchmarks has halted, which has especially affected the authors of recent semantic composition approaches, that rely on semantics that expands the limits of simple signature annotations. This has forced them to validate their approaches on example use cases, which are often reverse engineered from the solution. Approaches like LIDS, Karma, RESTdesc, and SPARQL Micro-Services are validated with a small number of examples, in some cases artificially created. The extension of RESTdesc [81] was evaluated using a synthetic dataset generated by the authors, without human-understandable semantics, and with a questionable quality, since it has not been used to evaluate other approaches.

We argue that a more realistic evaluation methodology is necessary, if we wish our methods to be adopted and have impact in practice. This is why we (a) created a new dataset and (b) designed

an evaluation study around a realistic use case of software-engineering teams building applications that integrate multiple web services.

4.4.2 The LRA Web API Test Dataset

In designing the evaluation method for our approach, we considered using as a basis the OWLS-TC [82] test collection, which is by far the most frequently used and cited in the literature. However, besides being considered relatively unrealistic, erroneous, and incomprehensible [74], the service specifications of this data set do not express the specific relationships between input and output concepts, which is necessary for our method. This problem is exacerbated by the lack of natural-language descriptions for most of the services, which precludes us from disambiguating these relationships. We also considered the other test collections, but none of these data sets were appropriate because of unavailability, incompatibility in their descriptions or lack of reference requests that allows to evaluate the accuracy of our approach.

Our evaluation is designed around a realistic use case, as described by Brueckmann et al. [83], where a developer, who has previously identified a set of relevant service providers, is searching for a web service, or a combination of services, that provides functionality needed in the application under development. This use case can be regarded as a special case of an information-retrieval problem, which is usually evaluated in terms of precision and recall. Accordingly, the evaluation involves three elements: a set of documents (represented by a set of service operations), a set of information needs (denoted by queries against the service operations), and a set of relevance judgments (represented by a set of composition chains that should be retrieved by the system for each query) [84]. The use case, along with the guidelines of information-retrieval evaluations and semantic web service evaluation considerations summarized in [74], led to our evaluation process that comprises the following four elements.

Dataset

The evaluation is based upon a test collection of services described by the documentation that a software developer would use when selecting a service, namely the natural language documentation. Providing natural language service descriptions, as opposed to only formal semantic service descriptions, allows the reuse of the test collection in other approaches, and avoids any bias that may be introduced by the abstraction process involved in creating formalizations of the data.

The service-operation collection was extracted from real service offerings, in order to represent the variety in terms of design patterns and resources found in real environments. We selected the domain of scholarly APIs, and used the list of APIs published by the University of Alberta Libraries[6],

[6] `https://www.library.ualberta.ca/about-us/open-data/api`

which is similar to the lists collected in other institutions, such as MIT[7] and UC Berkeley[8]. Then, guided by an expert in the area, the number of service providers was narrowed down to 13 (57 operations) based on the relevance of the resources offered and the quality of the documentation. The relevant service operations were selected, removing the input parameters used for control and authentication, such as pagination parameters and API keys, in order to help relevance judges by eliminating noise that may interfere with the comprehension of the functional specification. It is important to note that the LRA middleware deals with these issues, and supports service invocation and complete service-composition chain enactment.

The APIs were organized into a website using Swagger UI, keeping the documentation content intact, and allowing easy navigation for judges and developers during the evaluation (the APIs, questions, and relevance judgments are available at `http://cs.ualberta.ca/~serranos/lraeval.html`). In addition to their natural language documentation, the test collection of services is described by their formal semantic descriptions. The following provides a high level description of each Web API source, and each of them is assumed to be considered relevant by the application designer:

- *arXiv* is an electronic archive and distribution server for research articles, maintained and operated by the Cornell University Library.
- *ANDS* provides access to data relating to Australian research grants and researchers. In our evaluation, this data source will be regarded as the local authority providing grants information.
- *AltMetric* provides metrics and qualitative data that are complementary to traditional, citation-based metrics, such as mainstream media coverage and mentions on social networks such as Twitter.
- *CORE* aggregates all open access research outputs from repositories and journals worldwide and make them available to the public.
- *Crossref* is a not-for-profit association that enables persistent cross-publisher citation linking from a variety of content types, including journals, books, and conference proceedings.
- *Elsevier* provides programmatic access to journals and books published on ScienceDirect, and citation data and abstracts from Scopus.
- *IEEEXplore* allows access to the articles published by IEEE, through a single operation.

[7] `http://libguides.mit.edu/apis`
[8] `http://guides.lib.berkeley.edu/information-studies/apis`

- *Mendeley* is a reference manager and academic social network that can help in the organization of research works. It provides access to an extensive catalog of documents.
- *Nature Blogs* provides an interface to enable developers to search for blogs, posts, stories, and papers that are published on Nature Network or indexed in the blog catalog.
- *NSF* allows users to access information about awards granted by the National Science Foundation.
- *ORCID* offers an API that allows to connect to their proprietary registry, which includes information about works, education, and personal data.
- *PatentsView* offers programmatic access to the database of the federal agency for granting US patents and registering trademarks.
- *PubMed E-Utilities* is the public API to the NCBI Entrez system and allows access to the database of PubMed.

Queries

Complementing the collection of services, a domain expert defined a set of representative queries. These queries are of different levels of difficulty, and can be answered through a number of potential providers, thus providing valuable data for comparing the compositions generated by LRA and the relevance judges. In this collection, the expert who defined the queries is a librarian with substantial expertise in research-evaluation metrics, and hands-on experience in producing research impact assessments. The expert was not involved in the selection of APIs in the test collection, so as to ensure that the queries or services were not manipulated to match a particular request.

The list of eight queries, with their corresponding inputs and outputs, devised by the expert can be seen at Listing 4.1. The listing also includes the time spent encoding the natural language query into a SPARQL query. These queries cover 4 different aspects in the assessment of research projects and collaborations:

- *Research Production Indicators*: In bibliometrics, the derived measures or metrics are typically counts of the events or activities of specified types that are observed [85]. An important research output is publications, which include academic papers, blogs posts, patents and more. Queries 1, 4, and 5 fall under this aspect.
- *Link Analysis*: In evaluative link analysis, citation counts are used as indicators of quality, importance, influence, or performance, of researchers and their research output [86]. Evaluative link analysis is used to answer questions of the type, Whose research or influence is better, or has greater impact, than whose?. This subject is covered with Queries 6 and 7.

- *Input-Output Studies*: Some studies have attempted to correlate research and development expenditures, with publication output, also examining the relationship between the quality of scientists and key financial characteristics of the corporations in which they work [87, 88, 89]. These studies have become recurrent in institutions evaluating researcher's performance. This aspect comprises Queries 2 and 3.
- *Collaboration Metrics*: The study of collaboration, specifically through bibliometric analysis, is largely influenced by the work of Katz and Martin [90]. Those studies have shown that research collaboration can bring co-authors greater research productivity and research impact. For that reason, measuring sociability, defined simply as the number of coauthors, is important for researcher's evaluation. Query 8 addresses collaboration metrics.

Listing 4.1: Queries in natural language

1. *Which patents have been assigned to a person?*
In: Person name. **Out:** Country, date, number, type, assignee's name, title.
Encoding time: 4 min.

2. *Which awards have been allocated to an institution?*
In: Institution name. **Out:** Award title, amount*, date*, person name, and agency name.
Encoding time: 2 min.

3. *Which awards have been allocated to a person?*
In: Person name, ORCID id*. **Out:** Award title, amount*, date*, and agency name.
Encoding time: 1 min.

4. *Which are the publications of a person?*
In: Person name, ORCID id*, year range*. **Out:** Publication title, date, DOI*, abstract*, link*, collection name (journal).
Encoding time: 3 min.

5. *Which articles have been published in a specific venue?*
In: Venue name, date range*. **Out:** Same as 4.
Encoding time: 1 min.

6a/b. *Which/How many publications have cited a publication?*
In: Publication title. **Out:** Citing publication title, date, athors, and venue name / Citation count.
Encoding time: 3 min.

7a/b. *Which/How many publications have cited the publications of an author?*
In: Author name, citing publication date*. **Out:** Same as 6a/b.
Encoding time: 1 min.

8. *Which are the coauthors of a person?*
In: Person name. **Out:** Coauthor name, affiliation*, and publication title.
Encoding time: 2 min.

* *Optional attributes*

4.4.3 Relevance Judgments

In order to evaluate the responses to the queries, the ground truth, namely the "desired service compositions" are defined for each query. Other authors, such as [42, 91, 92], have proposed graded relevance scales for service retrieval evaluation, but they lack clear definitions and dimensions that enable domain experts to judge compositions chains unambiguously. By analyzing previous expert-judgment approaches, and our previous experience with expert judgments in [93], we designed a clear and simple relevance model, that attempts to reduce the ambiguities and judgment inconsistencies found in other relevance models. The relevance model is based on two aspects: *semantic correctness*, to determine whether the composition would yield results that are sound, and *usefulness*, to decide if the quality of the composition and its perceived cost make the composition worth it. It is important to note that, while the dimensions are different, they are not completely independent. For example, a *semantically incorrect* chain should not be judged as *useful*.

The relevance scale is defined as a one-dimensional scale that defines four levels:

- *SignificantMatch:* semantically correct, low or moderate cost, and represents the preferred option to extract data from the given providers.
- *PossibleMatch:* semantically correct, low or moderate cost, but is not the preferred option to extract data from the given providers.
- *InsignificantMatch:* semantically correct, but the composition is costly or is not expected to contribute a significant amount of valuable data.
- *NoMatch:* semantically incorrect and ineffective.

In addition, we mapped the values of this one-dimensional scale into a binary relevance criterion, where *NoMatch* and *InsignificantMatch* are classified as *Irrelevant*, and the other levels as *Relevant*.

Four doctoral students of Computer Science and Information Studies from the University of Toronto, with experience in some of the data sources and service-oriented architectures, served as relevance judges. In order to perform the judgments, we held a 3-hour session with the judges. In the first part of the session, the relevance judges received a document with the 8 natural language queries, the description of the relevance scale, and access to the Swagger interface with information about the available APIs. In the following 2 hours, the expert judges were asked to find all the compositions that may answer each of the queries, using the relevance scale to classify the chains. Finally, given the relatively short time of the session, and that considering every single combination of services may be beyond human limits (e.g. compositions of length 3 require to analyze more than 140,000 combinations), we asked the judges to assess the compositions produced by the LRA method, that were not included in their initial responses.

4.4.4 Service and Query Formalization

Although the test collection is designed based on the natural language queries and descriptions in order to support reuse across other formalisms and approaches, eventually, the services and queries have to be formalized in a particular semantic model.

A prominent ontology that can represent semantic concepts in the domain of our evaluation is the VIVO ontology [94], a unified, formal, and explicit specification of information about researchers, organizations, and the scholarly activities, outputs, and relationships that link them together. VIVO also promotes interoperability by reusing established Linked Data ontologies wherever possible, such as FOAF [95] and BIBO [96].

Based on the Swagger descriptions in YAML, and using the VIVO ontology [94], an expert in LRA extended the Swagger files with semantic annotations. The service descriptions are created by experts in the particular description formalism, in this case LRA, in order to avoid suboptimal descriptions. Table 4.2 shows the number of operations per service provider, and the time spent by the expert in formalizing the semantics of the API. In average, the expert took almost 7 minutes to annotate the semantics of each operation, but as only one person created the descriptions, and there is significant potential to learn something from the ontology and similar API design patterns, the Web APIs described at a later stage are benefited from knowledge acquired by describing previous APIs, a phenomenon known as *learning effect* [97]. For example, since the APIs for *arXiv* and *IEEE* are similar, the time to describe the semantics of the latter is decreased considerably.

Next, the natural language queries previously defined, were encoded in SPARQL. In order to mimic realistic environments, the developer that expressed the queries did not use any information beyond the provided queries in natural language, along with the ontology documentation, thus avoiding manipulation of queries to ensure correct retrieval. Listing 4.1 shows the time spent by the developer in encoding the queries. In average, the developer spent 2.13 minutes per query, but the formulation of such queries also suffer from the *learning effect*. Table 4.3 in Section 4.4.6 shows the number of triples and joins used to represent the queries, which provide indication of the relative complexity. The characteristics of the queries are in line with the number of triples, joins, operators, and patterns found in the characterization of real-world queries extracted by Gallego et al. [98].

4.4.5 Results

The service annotations are fed into the LRA middleware, and queries are executed as expressed by the developer. This process results in lists of composition chains for each query, along with runtime and retrieval performance measures.

Query 1 is a high selectivity query, since only one of the data sources provide information about patents. In addition, the patent class is at the lowest level of hierarchy, and thus, it does not take

Table 4.2: Effort in LRA formalizations

Service Provider	Num. Operations	Effort (min.)
arXiv	1	26
ANDS	1	18
AltMetric	4	35
CORE	4	31
Crossref	6	40
Elsevier	13	65
IEEEXplore	1	10
Mendeley	4	22
Nature Blogs	4	24
NSF	1	10
ORCID	9	47
PatentsView	1	13
PubMed	3	16

advantage of subclass inferences. Accordingly, the experts identified only one composition chain, which was successfully identified by the LRA composition method.

Query 2 is a multiple-source query that fetches attributes from three different entities (grant, agency, and investigators), based on the institution name. The query introduces optional output attributes, represented by the awarded amount and date of the grant. In this case, although the grant entities are provided by three data sources (*NSF*, *ANDS*, and *ORCID*), one of the data sources does not associate an institution to the grant (*ORCID*). Consequently, the experts determined two service operations as *SignificantMatch*, and the LRA method recognized the same two operations.

Query 3 is a multiple-operation query, since some of its composition chains have more than one operation. It adapts Query 2 to search the same data, but based on the name of an specific investigator. This time, as *ORCID* grants are associated to persons, it can be used by composing two operations, *search*, followed by one of *record*, *activities*, or *funding*. Initially, the experts identified 3 composition chains as *SignificantMatch*, and 2 chains as *PossibleMatch* due to increased transmission costs. In particular, the experts found three equivalent chains using *ORCID*, but determined that one of them was more desirable than the other, due to the expected lower size of the data to be transmitted. For its part, the LRA method identified 3 composition chains, the top 2 corresponding to *SignificantMatch*, and the other to *PossibleMatch*.

Query 4 is an inference query with optional attributes, since it retrieves data from two entities, a document (in general), and a journal, making use of optional attributes in the input and output. Although this query asks for publications in general (potentially using inference rules), since blogs and patents do not include some of the required output attributes, the resulting composition chains should not include such entities. The experts identified 9 composition chains as *SignificantMatch*,

but 2 of them were incorrect, since they did not provide the required output in its entirety. In total, the LRA method returned 8 composition chains, the first 6 corresponding to *SignificantMatch*, and the last chains, considered by experts as *PossibleMatch* and *InsignificantMatch*, respectively. In particular, the LRA composition method only returned one chain using *Elsevier*, due to the heuristic that eliminates chains where all providers are involved in shorter chains; however, this data source provides access to two different databases (ScienceDirect and Scopus), which implies that there are two valid composition chains from the same provider. Another notable case appears with the use of *CORE*, where the API does not provide information about the publication venue, but the LRA method tries two alternatives to find such information, in the first, by joining it with *AltMetric*, and in the second, using *Nature* and *AltMetric*. When the experts were confronted with these compositions, they assessed the chains as *PossibleMatch* and *InsignificantMatch*, respectively.

The date range restriction in the query is expressed as a post filter, which is a concept borrowed from ElasticSearch where a filter is applied at the very end of a request, in this case, after receiving the response documents from the data sources. Although some operations, such as *search/catalog* from *Mendeley*, can accomplish a better performance and coverage by using the input parameters that define start and end dates, currently our description model only supports equality assignment on URL input parameters. Additionally, this limitation may prevent the system from obtaining complete answers, since APIs often restrict the access to the data by limiting the number of records returned, which decreases the likelihood of containing the records that meet the filter conditions. In order to overcome this limitation, in the future, we plan to associate a relational operator to each input parameter in the description.

Query 5 is conceptually the same as Query 4, also an *inference* query, but uses the venue name as the input, which has meaningful implications on the returned composition chains, since some of the services used in the previous query do not take the venue name as input, which makes impractical their participation. The experts identified 6 *SignificantMatch* composition chains, but one of the chains was incorrect, due to the lack of required inputs for the operation. The experts noted that two of the *PubMed* chains are mutually exclusive, since the overhead and data they return can be considered equivalent. The LRA method returned 4 composition chains, corresponding to all the correct chains identified by the experts, but including only one of the *PubMed* chains, which was the expected the result, due to their mutual exclusivity. The same considerations of Query 4, with respect to blogs, patents, and date range, apply for this query.

Query 6 serves as a negative control, because it is not supposed to result in valid composition chains. The query introduces interesting challenges through the distinction of independent entities of the same type used in the input and output, which may cause ambiguities in traditional description formalisms, such as SAWSDL. Thus, the LRA method has to recognize that although the *cites* relationship is present, the direction in the descriptions is used to indicate the references of a

publication, and not the citing publications. Accordingly, the experts and the LRA method did not find any results for this query. However, the experts noted that some of the sources offered a citation count (*Elsevier* and *Crossref*)[9], and that providing such count would be valuable. Then, the original query, denoted as Query 6a, is relaxed into "How many times a publication has been cited?", which we will refer to as Query 6b. In Query 6b, the experts identified 1 *SignificantMatch* chain, while the LRA method identified 2 chains, one of them being the chain identified by the experts. When confronted with the chains produced by the LRA method, the expert recognized they missed one *SignificantMatch* chain.

Query 7 is conceptually similar to Query 6, but uses the author name as the input. Similarly, the experts proposed to relax its formulation in order to include only citation counts. This time, *Elsevier* had to use intermediaries to search for publication identifiers that can be used to find citation counts. For Query 7b, the experts identified 6 chains as *SignificantMatch*. However, after carefully discussing the added value of the compositions, following the same rationale of the heuristic that prefers shorter chains, they recognized that 4 of their chains are *InsignificantMatch*, thus leaving only 2 *SignificantMatch* chains, which is exactly what the LRA method detected.

Query 8 is an isomorphic query, since the inputs and outputs of the query are of the same type. This query obtains data about the coauthors of a specific person, making use of publications in general, including papers, patents, and blogs. However, since the data source contributing blogs does not provide authorship information, the resulting compositions should not include such entity. In the same way as Query 6, this query uses independent entities of the same type in the input and output, but in this case, connected through an intermediate entity, a publication. The experts identified 7 *SignificantMatch* composition chains, while the LRA method identified 8 chains, including the 7 chains determined by the experts among the top results. When the experts were confronted with the additional chain reported by LRA, they realized they failed to consider patents as documents. Accordingly, the experts classified the chain as *SignificantMatch*.

We analyzed our results in terms of the traditional F_1*-measure* [99], which combines *precision* and *recall* with an equal weight. Additionally, since all the composition chains are not of equal relevance, we also use the *normalized discounted cumulated gain* (nDCG) [100] to reward retrieval of highly relevant items early, and penalize irrelevant chains. Thus, we assigned gain values of $\{3, 1, -1, -2\}$ to the four-level scale, only considering the first two levels in the ideal gain. Table 4.3 shows the retrieval correctness results from the evaluation, as measured by the F_1-measure with the binary scale, and by the nDCG with the four-level scale.

The runtime performance was measured on an Intel Core i7 (2.00GHz) machine with 8 GB RAM running Ubuntu 14.04. Results for each query are shown in Table 4.3, presenting a set of

[9] Social media mentions from *AltMetric* were not considered as citations.

columns with the elapsed time at different exploration lengths. Each measurement corresponds to the average time, in milliseconds, of seven runs of the same kind of query, where the first two runs were discarded. The last columns indicate the F_1-measure, nDCG, and the maximum length of the composition chains found when the exploration length $l = 4$.

4.4.6 Discussion

Firstly, it is important to note that none of the returned composition chains was judged as *NoMatch*, which indicates the high quality of the chains produced by the LRA composition method. Additionally, the 3 errors and 2 omissions made by the experts demonstrate that this task is complex and error-prone, even for experienced developers, which amplifies the potential impact of automating this task. The LRA method successfully identified 27 *SignificantMatch* chains, out of the 30 chains detected by the experts, only returning 3 *PossibleMatch* and 1 *InsignificantMatch*. Considering only *SignificantMatch* chains, the LRA method showed an average F_1-measure of 0.93, showing the same performance as the expert judges before they were confronted with the chains produced by the LRA method.

Our results are not directly comparable to the results of other approaches due to the differences in the test dataset. However, when contrasted with other approaches, such as [41, 101, 40], that use JGDEval, a similar dataset in terms of size, our approach is superior, exhibiting a significantly higher accuracy. Moreover, the quality of the top results is emphasized by the high average nDCG of 0.96. Unlike the results presented in [74], where the choice of the relevance scale (binary or graded) influenced the results moderately, in our evaluation we found that our scale definitions provided similar results, indicating our scales can be considered stable. The stability is consequence of the high accuracy, and we also hypothesize that it may be due to the the expertise of the relevance judges, which were not only experts in technical aspects, but also experienced in the specific domain of interest; while other approaches typically involve only technical experts for their relevance judgments. Additionally, the approaches in JGDEval Benchmark [74] averaged a running time of 998 milliseconds for service discovery, and our approach performs significantly better, averaging a running time of 15 milliseconds ($l = 1$), and 896 milliseconds for compositions chains of length $l = 3$. The time performance is aligned with the results of Baccar et al. [81], in what they call reasonable limits for realistic composition scenarios. This performance was enabled by the blocking function and pruning, which could reduce the number of pairwise comparisons to less than 20% for $l = 3$ and less than 1% for $l = 4$, when compared to the brute force approach that considers every permutation.

Regarding the runtime performance, it is evident from Algorithm 1 and the reference executions in Table 4.3 that the LRA method exhibits an exponential runtime complexity $O(n^l)$, where n is the number of service operations in the repository, and l the exploration depth. Despite the negative

Table 4.3: Runtime and retrieval performance

Query			Time per Exploration Length (ms.)				F-measure	nDCG	Max. Length
Id	# Triples	# Joins	1	2	3	4			
1	11	3	5	22	209	2185	1.00	1.00	1
2	10	3	15	86	841	37413	1.00	1.00	1
3	12	3	17	187	2791	110150	0.75	0.75	2
4	14	3	21	231	1085	22637	0.87	0.88	2
5	13	3	21	231	1085	22637	1.00	1.00	2
6a	9	4	12	25	125	490	1.00	1.00	0
6b	5	2	11	92	930	29836	1.00	1.00	1
7a	7	4	17	78	390	1984	1.00	1.00	0
7b	12	3	16	124	1315	36359	1.00	1.00	2
8	15	5	21	69	196	486	1.00	1.00	2

implications of such high complexity, we found that in practice the length of the composition chains tends to be low (the distribution of relevant chains are 83% and 17% for lengths 1 and 2, respectively), which resembles the *broken telephone effect*, where long message transmission chains are typically discouraged because they lead to an increased expenditure of resources and error accumulation. In addition to low values for depth exploration, Dalvi et al. [102] showed that in order to have a significant coverage of data in a particular domain, a large number of data sources is not necessary. Consequently, this implies that in order to have good coverage of a domain, the number of service operations n will be rather small, in which case, our approach becomes practical. Moreover, since runtime performance is highly affected by code optimizations and caching techniques, then more mature implementations of our approach may provide better results.

We are well aware of the threats to the validity of our findings, due to the relatively small size of the dataset, the small numbers of queries, and their potentially limited complexity. Still, we believe that this work should encourage other approaches to make conclusions based on realistic data, rather than on the basis of reflection on ad-hoc examples, as has been the case with many recent approaches. For this reason, it is important to create realistic service collections that reflect the current state of the API ecosystem and can be used to compare the performance of diverse approaches.

5

The LRA Workbench

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 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 LRA, a higher-level query system is needed to ease the learning curve and allow developers to formulate queries in a natural way.

This chapter begins by identifying the desired features of our development environment, through a systematic exploration of Linked-Data query systems. Then, the proposed query environment and its features are described, and evaluated by an empirical study, involving 20 software developers.

5.1 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-compliant 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 26 highly relevant publications, summarized in Table 5.1. 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.

Table 5.1: Linked Data query systems description matrix

	Interface					Interpretation			Structure			Usability		Rendering			Availability	Active
	Text	IR	Form	Visual	NLI	Navigation	Composition	Approx.	Browser	Step	Path	Casual+	Expert*	List	Table	Diagram		
OpenLink Virtuoso (2006)	×						×								×		×	×
Querix (2006)					×			×				1		×				
iSPARQL (2007)				×			×								×		×	
NLP-Reduce (2007)					×			×				1	3		×		×	
Sindice (2007)		×						×						×				
Twinkle (2007)	×						×								×		×	
Watson (2007)		×						×						×				
gFacet (2008)			×	×			×			×					×	×	×	
K-Search (2008)		×	×			×			×	×		2	2		×	×		
MashQL (2008)			×	×			×		×	×					×		×	
Nitelight (2008)				×			×		×						×			
GQL (2009)				×		×	×				×				×			
Ginseng (2010)					×			×		×		1	3	×				
Konduit VQB (2010)			×				×			×					×			
RDF-GL (2010)				×			×								×			
Semantic Crystal (2010)				×			×		×	×		3	1		×			
SPARQL2NL (2013)					×			×						×			×	
Filter/Flow (2014)			×	×		×	×		×	×					×		×	
SemFacet (2014)		×	×			×				×				×	×		×	×
SPARQLBuilder (2014)			×	×			×				×				×		×	
Squebi (2014)	×						×			×					×		×	
YASGUI (2014)	×						×			×					×	×	×	×
AffectiveGraphs (2015)				×			×			×		3	1		×			
QueryVOWL (2015)				×			×			×					×		×	
AskNow (2016)					×			×						×			×	
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.

5.1.1 Interface

Query interfaces typically appeal to visual representations and natural language constructs for expressing the domain of interest and convey different kinds of requests. These systems rely on the human capacity to draw perceptual inferences, allowing encoding and interpretation of large quantities of information that improve human-computer interaction. The elements of the interface are structured in different ways, depending on the characteristics of the representation model, ideally using comfortable representations, since users prefer to interact with something similar to the reality

they live in, ignoring the underlying abstract model. In order to classify the external interfaces of these systems, we consider the formalization adopted by users to define the elements of interest and the applicable operators. Accordingly, systems are organized into five categories depending on the adopted formalism:

Text editors

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[1] and Bio2RDF[2], and can be employed to compose queries using SPARQL directly. Some of them, such as OpenLink Virtuoso [103] and Twinkle [104], provide only a simple text box for the user to type a complete SPARQL query. Others, such as YASGUI [105] and Squebi [106], 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* [107, 108], because in order to formulate semantically meaningful queries, users need to know, not only the syntax of the query language, but also the structure of the data.

Information retrieval (IR)

These approaches allow users to pose queries to RDF repositories as *bags-of-words*, just as they would to a search engine, such as Google or Bing. The dataset is preprocessed to create an inverted index; when a query is submitted, this inverted index is used to match input-query words to classes, properties, or instances, of the RDF dataset. The most notable approaches of information retrieval on RDF datasets have been Sindice [109] and Watson [110]. At the peak of its activity, Sindice had an index of over 700M pages, processing 20M pages per day, but nowadays, neither Sindice nor Watson are supported any longer. *These approaches only act as locators of RDF resources that return pointers to remote data sources, and are not considered as actual query systems.*

Form-based interfaces

These interfaces capitalize on common user-interface design patterns, adopted by most of modern computer systems, thus increasing the initial familiarity with users. In these approaches, the queries are specified through the use of text boxes, drop-down menus, check boxes, and other well known form elements. These systems typically follow a faceted search interface, enabling users to explore a data set by applying multiple filters. This type of interfaces are frequently used in

[1] Available at `http://dbpedia.org/`
[2] Available at `http://bio2rdf.org/`

e-commerce sites, such as Amazon or eBay. K-Search [111], Konduit VQB [112], SemFacet [113], and PepeSearch [114] are some of the systems adopting such a faceted-search interface.

In order to improve the user-interaction process, some systems combine elements from other interaction models. For example, K-Search and SemFacet provide an initial IR interface that enables users to select an initial subset of the data, consisting of some initial answers and facets. Other approaches rely on more complex visual elements, such as K-Search which combines the form-based query entry with an ontology tree view, and SPARQL Builder/TogoTable [107, 115] that communicates candidate relationships between concepts through a graph visualization.

From the aforementioned approaches, only SemFacet and PepeSearch were available and have been maintained recently. Although TogoTable is still accessible, the SPARQL Builder module that suggests paths between concepts is not available. In addition, none of these tools appear to have been adopted by a considerable number of users.

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 [116].*

Natural-Language Interfaces (NLIs)

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 AskNow [117], SPARQL2NL [108] and NLP-Reduce [118]. Due to linguistic ambiguities, understood as the different ways in which statements can be interpreted and composed, some NLIs restrict the user input to a controlled/structured natural language, constrained in its grammar and lexicon. For instance, Ginseng [119] uses a controlled input language akin to English, and Querix [120] engages users in dialogues controlled by the system in order to resolve linguistic ambiguities. 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*. On the other hand, unrestricted NLIs suffer from low effectiveness due to the aforementioned variability and ambiguities of the language.

Visual approaches

Visual approaches adopt a diagrammatic language, consisting of geometric shapes, textures, and connections between them, to convey the query semantics. Visual-query languages attempt to improve the effectiveness of query-representation languages, by using a wider range of communication media, relying on the high bandwidth of the human-perceptual channel. The diverse approaches in this category vary depending on their adopted graph data model. For example,

iSPARQL [121], Nitelight [122], RDF-GL [123], Affective Graphs [124], Semantic Crystal [119] and QueryVOWL [125] represent RDF triples using nodes for subjects and objects, and links for the predicates that connect them. gFacet [126] and GQL [127] use a UML-like representation of classes with their properties inside a box. MashQL [128] and Filter/Flow [129] use 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, MashQL, and Filter/Flow.

Most of these approaches were abandoned shortly after they were published; only iSPARQL, QueryVOWL and Filter/Flow have a working online demo, but they are not being actively maintained.

Past usability studies have shown that expert users prefer graph-based interfaces, due to their reduced complexity and high expressiveness [116]. Accordingly, we aim to design a workbench that 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.

5.1.2 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

Navigation approaches focus on a concept (or a group of concepts) and enable users to browse the underlying ontology by following links from the current focal concept(s) to other concepts of interest. Typically, in this strategy, users are presented with property values of the selected 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* [116].

Composition

Composition is the most common interpretation approach, where the configuration of elements in the user's input can be unambiguously mapped 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.

Similarly, visual approaches establish a direct correspondence between graphical elements and query constructs. 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 [116]; for this reason, our workbench adopts a composition approach for its query-interpretation step.

Approximate Mapping

These approaches are based on the idea of inferring the semantic structure of the query, in order to locate possible matches in the data store. Approaches using this strategy begin by identifying entities that can be mapped to classes or instances in the dataset, by applying NLP techniques such as lemmatization, word-sense disambiguation, and named-entity recognition. The system then tries to identify triple structures among the entities, aided in some cases by grammatical relations extracted through natural-language parsers, and in others by preconceived query patterns. Subsequently, the identified triples are joined and translated into SPARQL statements. The simplest approaches in this category are based on the bag-of-words technique from IR approaches, while natural-language 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 complex structures involving disjunctions and aggregations (e.g. SPARQL2NL).

In these approaches, *there is frequently a mismatch between the users' expectations and the capabilities of the system, which has been referred in the literature as the* habitability problem *[130]*: although they 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. This results in low response accuracy or forces these systems to rely on large amounts of background domain knowledge. In addition, as stated previously, most of the systems following this strategy focus on simple factoid queries, which limits the complexity of queries and answers that can be produced with these approaches.

5.1.3 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

Casual users are characterized for their lack of 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 [131, 132, 119]. 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

These type of 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. [116] 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. In addition, experts appreciated the ability to see the translated SPARQL query, since it increased their confidence in what they were doing. Besides, this feature enables users to directly manipulate the SPARQL query syntax, and increase the expressiveness of the query system as perceived by expert users.

5.1.4 Structural-Discovery Support

In order to efficiently use Linked-Data databases, users need to explore the relationships among the stored resources. Ontologies, however, are represented at different levels of complexity, and users may not always know all the element properties and the relationships between them, which may hinder the formulation of SPARQL queries. 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

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, MashQL, and Filter/Flow 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 [133].

Step-Recommendation

This simple method explores the neighboring classes and predicates that may be reached from a focus object, and suggest a list of plausible query elements. For example, YASGUI and Squebi 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.

Our workbench 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

Path discovery capabilities attempt to find connections between two resources, where an immediate relationship may not exist. For example, in the VIVO ontology, the affiliation relationship between

a researcher and his institution is mediated by a role instance, which may not seem intuitive for users unfamiliar with the intricacies of the ontology.

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 SPARQL-Builder and GQL propose a path discovery feature, 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. It is worth noting that neither SPARQL Builder nor GQL provide a reference implementation or a demo of their path-discovery interface.

Our workbench improves upon previously proposed approaches by incorporating semantic knowledge into the emergent schema and ranking the paths based on proximity.

5.1.5 Results rendering

The presentation of the information returned is also an important component of the query system. Elbedweihy et al. [116] 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**, although IR approaches rely on a simpler technique, using a **list** of snippets.

However, taking into account the context of the data, an appropriate **diagrams** 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 gFacet can use maps to display information about geographic coordinates. Additionally, YASGUI incorporates YASR [105], which supports visualization via Google Charts, including line, bar and scatter plots, geographical maps, and several others. *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 [134].*

5.2 Workbench Design

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

5.2.1 The Query Console

For experienced users, direct manipulation of SPARQL queries is provided through a Query Console (Figure 5.1), similar to other SPARQL query editors. The Query Console provides the following representations of the responses:

- A table renderer for the projected variables in the SELECT clause, along with a column that specifies the data sources of the row (Figure 5.1-I).
- The raw JSON-LD response as it would be consumed by third-party applications using LRA as a service.
- A graph visualizations that represent entities as nodes, and their relationships as edges (Figure 5.1-II and Figure 5.1-III).

The graph visualizations assist users in their exploration of the returned result. The first visualization offers a *force directed layout* (Figure 5.1-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* (Figure 5.1-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 [135]. Although the visualizations do not scale to an extremely large number of nodes (in the order of thousands), the responses from web services are typically restricted to a few dozens of records, which is suitable for the scale supported by the graph visualizations.

5.2.2 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.

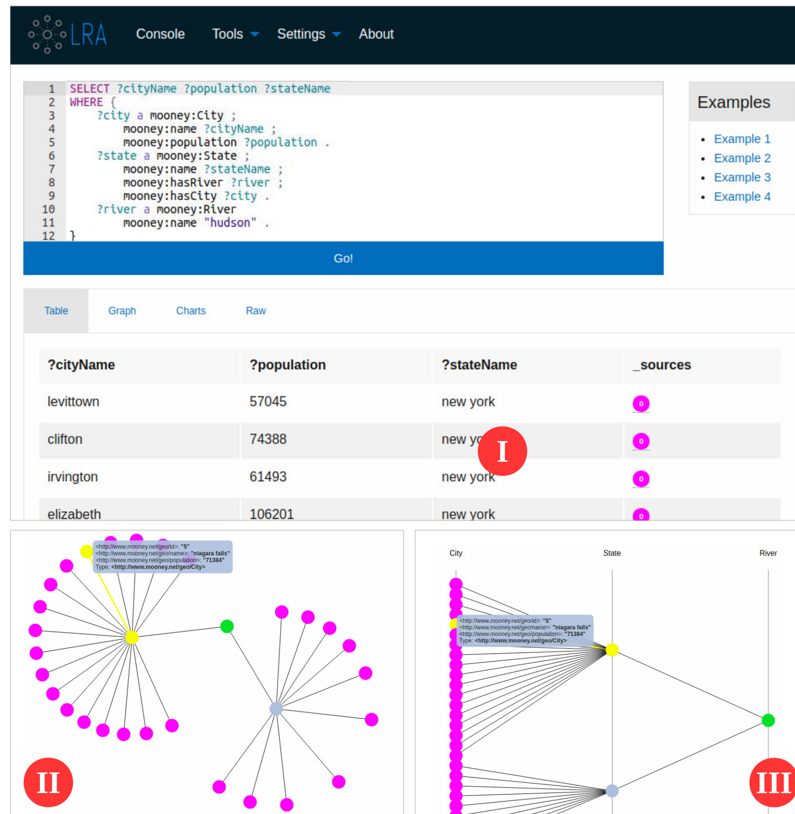


Figure 5.1: LRA Query Console.

The design of the Visual Query Assistant is guided by three key laws of Gestalt Theory in Visual Screen Design [136]: 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 Figure 5.2-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 [137, 107], 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 (Figure 5.2-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 (Figure 5.2-II), which then can be used

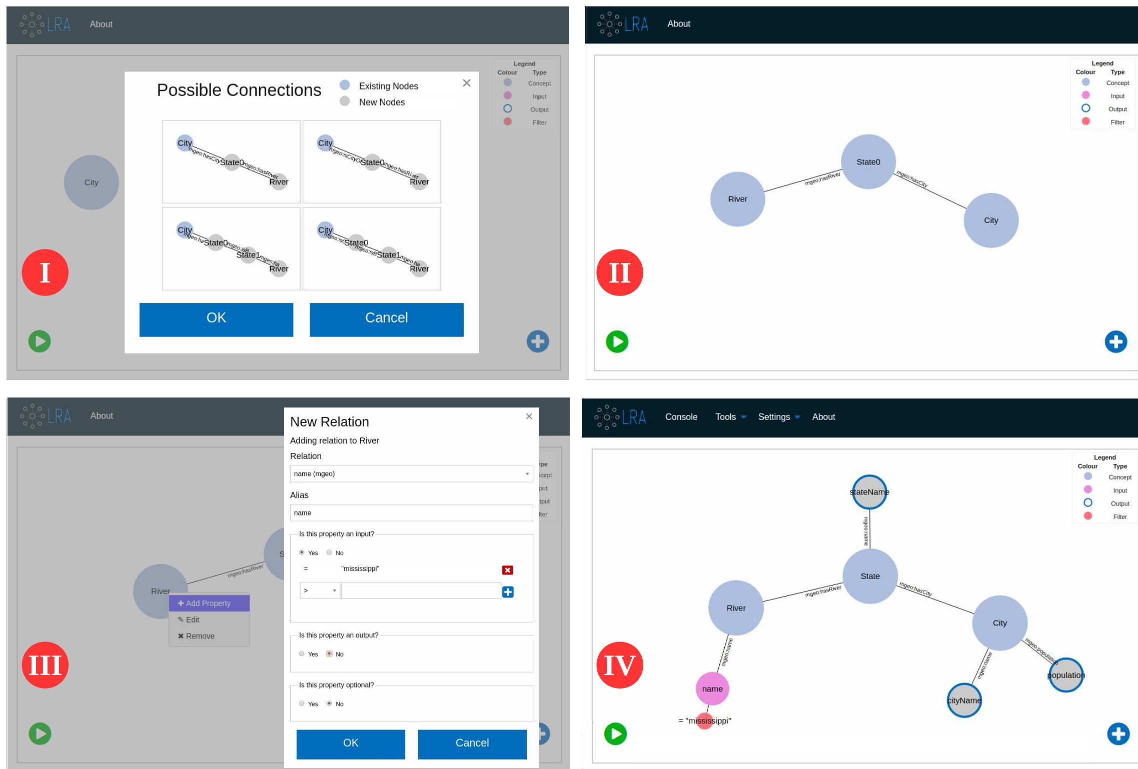


Figure 5.2: Visual Query Assistant.

to add properties to the elements (Figure 5.2-III and Figure 5.2-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.

5.3 Usability Evaluation

This section presents an empirical study to assess the accuracy, efficiency, and perceived usability of developers when asked to produce queries using the workbench. Furthermore, the performance of developers using the LRA Workbench is compared against that of developers using YASGUI, a popular SPARQL interface, used as front-end by a large number of Linked Data publishers.

5.3.1 Empirical Study Design

An empirical study was conducted in order to respond the research question: *Does the LRA Workbench improve the effectiveness of developers formulating LRA-compliant 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.

Hypothesis

In this study, we hypothesize that enabling developers to interactively formulate LRA-complaint queries, through a visual-based interface, can significantly improve their performance. In order to investigate our research question, we outlined two individual null hypotheses:

> $\boldsymbol{H_{01}}$ **:** Developers spend an equal amount of time formulating LRA queries using VQA and YASGUI.
>
> $\boldsymbol{H_{02}}$ **:** Developers provide equally accurate queries, in terms of task solution correctness, using VQA and YASGUI.

The two independent variables of this study, V_{VQA} and V_{YASGUI} represent the tasks designed to evaluate the performance of developers using VQA and YASGUI, respectively.

Variables

Considering the hypotheses H_{01} and H_{02}, our experiment has two dependent variables on which our treatments are compared:

> $\boldsymbol{V_t}$ **:** Time developers spend solving each task.
>
> $\boldsymbol{V_a}$ **:** Developers' accuracy solving each task.

Tasks

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. [98] 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 [138], 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 [119, 116], and found they were based on a small dataset containing geographical information about the US[3], which also contains predefined English language questions. For our study, besides using the same dataset, we considered the questions that these studies selected and

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

their SPARQL translation. The above considerations resulted in the following query-specification tasks.

T_1 : Which are capitals (name and population) of the USA?

T_2 : What are the cities (names) in states through which the Mississippi river runs?

T_3 : Which are the lakes (names and area, if available) in states bordering Minnesota?

T_4 : Which rivers (name and length) run the states that border the state with the capital Atlanta?

T_5 : Which states (names) have a city named Springfield with a city population over 100,000?

Tasks T_1, T_2, and T_5 correspond to questions found in [116]. Tasks T_3 and T_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.

Participants

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 [139].

Data Analysis

In this study, we assume a normal distribution of our variables based on previous work on similar user interface experiments [131, 119]. Additionally, we consider an acceptable probability of 0.05 for Type-I error, i.e. rejecting the null hypothesis when it is true.

5.3.2 Empirical Study Protocol

The protocol of the study was divided in two stages. Stage 1 (Training) involved an introductory tutorial on graph data models and the particular features of the tool assigned. Stage 2 (Working session) involved a working sessions in which 20 participants solved the tasks assigned for the system of the study.

Training Session

The training session was structured in a 30 minute session. The first half of the session involved a tutorial on the use of assigned tool. The second half consisted of a laboratory workshop that provided hands-on experience with the query system.

The 20 participants received a video presentation that introduced an example case study. The case study was used to explore the features of VQA or YASGUI. Additionally, participants completed a set of training tasks that allow them to familiarize with the query system. During this stage, the study coordinator answered questions and helped participants as they completed the guide.

Working Session

Previously, participants were randomly divided into one of two groups, where one group was assigned to use VQA, and the other group to YASGUI.

Each participant took part in an independent working session. Developers were provided with a dedicated working station consisting of a desktop computer where both, VQA and YASGUI, were previously installed and deployed.

At the beginning of each session, participants received a 5 minute presentation on the structure of the dataset of study, which also included 5 minutes of questions and final setup. Next, the queries were presented to the participants one at the time in the same order. 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.

Data Collection

In order to asses 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. 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 [140], a standardized usability test that has proven to be very useful when investigating interface usability [141].

The working sessions were instrumented with LimeSurvey[4], 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. 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.

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

5.3.3 Results

The results for both tools, VQA and YASGUI, are presented in Table 5.2. The first two columns present the evaluation criterion and the number of the task, 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.

Table 5.2: 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_1-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**

The results show that VQA users took significantly less time overall and required fewer attempts to formulate the queries than YASGUI users (95% confidence), thus rejecting hypothesis H_{01}. In addition, queries formulated through VQA were more accurate than queries created with YASGUI, although the difference was not statistically significant and hypothesis H_{02} could not be rejected. Furthermore, VQA presents higher subjective usability ratings than YASGUI.

Our results are not directly comparable to the results of other graph-based approaches using the same dataset [119, 116], 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, compared to AffectiveGraphs users. We hypothesize that this may be due to the *learning effect* [97] 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.

5.3.4 Discussion

In the study, we 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%, in average. 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 average 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.

5.3.5 Threats to Validity

Although the number of participants is limited and they lack of experience on Semantic Web technologies, considering that SPARQL is a fairly unfamiliar language in the software engineering community at large, our pool of participants is representative of prospective LRA application developers. We are aware that this lack of expertise and the learning curve of SPARQL, VQA and YASGUI may impact the developers performance. In order to minimize the impact of this threat to validity, we included an introductory tutorial in the training session of our protocol.

Participants were not allowed to return to a task once it was completed, which might fail to account for the exploratory nature involved in the comprehension of a dataset. The adoption of a

strictly linear answering mechanism was motivated by our desire to precisely measure the efficiency of developers formulating queries. We minimize this threat to validity by including a segment in the instructional videos, with the details of the dataset and its structure. We acknowledge that a more sophisticated answering and instrumentation mechanisms are desirable, however, the linear answering strategy used in this paper is realistic, and has been used in the studies reviewed in the related work (Section 5.1).

The dataset and questions used in this study were developed in a research environment. We cannot claim that the results in this study can be generalized to industrial ecosystems, since ontologies used in real environments are typically more complex. Nevertheless, considering the scope of our research, and the limited availability of test subjects, we believe that the results of this study provide substantial insights on the expected performance developers using Linked REST APIs.

6

Engineering SOA Systems with LRA

After evaluating performance of the composition methodology and the user interface of the development workbench, we turn 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 the research question: *Does the use of the LRA environment result in improvements of developers' efficiency and code quality?*

Accordingly, this chapter presents an empirical study that evaluates the usability and complexity of LRA using the traditional hand-crafted endpoint request workflows as the industry baseline.

6.1 Empirical Study Design

We investigate the quality of software artifacts and performance of developers when implementing applications that integrate data from multiple sources, through the use of Web APIs. We measure the quality of the produced artifacts, and the accuracy and efficiency of developers when asked to develop a *mashup* web application using LRA. In turn, we compare their quality and performance with that of developers using the traditional approach where the developer manually designs a workflow that submit requests directly to the service endpoints.

6.1.1 Hypothesis

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. [83]. 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. In order to investigate our research questions we outlined four individual null hypotheses.

$\boldsymbol{H_{01}}$ **:** Developers produce software artifacts of equal complexity when creating a hybrid application[1] using LRA and the manually-designed workflow strategy.

$\boldsymbol{H_{02}}$ **:** Developers spend an equal amount of time creating a hybrid application using LRA and the manually-designed workflow strategy.

$\boldsymbol{H_{03}}$ **:** Developers provide equally correct software artifacts when creating a hybrid application using LRA and the manually-designed workflow strategy.

$\boldsymbol{H_{04}}$ **:** Hybrid applications spend equal amount of time to load their content when using LRA and the manually-designed workflow strategy.

6.1.2 Variables

Considering the hypotheses H_{01}, H_{02}, H_{03} and H_{04}, our experiment has four dependent variables on which our treatments are compared:

$\boldsymbol{V_c}$ **:** The complexity of the code produced by developers to solve each task.

$\boldsymbol{V_t}$ **:** Time developers spend solving each task.

$\boldsymbol{V_a}$ **:** Developers' accuracy solving each task.

$\boldsymbol{V_l}$ **:** Load time of the application of the isolated application component for each task.

The two independent variables of this study, V_{LRA} and V_{MW} represent the tasks designed to evaluate the quality and performance of artifacts and developers using LRA and the manually-designed workflow strategy, respectively.

6.1.3 Tasks

In this study, we measured the complexity of software artifacts and performance when developers are asked to create applications that retrieve and integrate data from multiple Web APIs. In order to determine the differences in performance and quality, we designed an evaluation study around a

[1] Hybrid applications (also known as *mashups*) are applications that use content provided from more than one source. For our study, we assume that the content is provided through Web APIs, which is the most common data-exchange mechanism.

realistic use case of software-engineering teams building applications that integrate multiple web services, using the approach followed by Brueckmann et al. [83] to create hybrid applications, where developers have previously identified a set of relevant service providers.

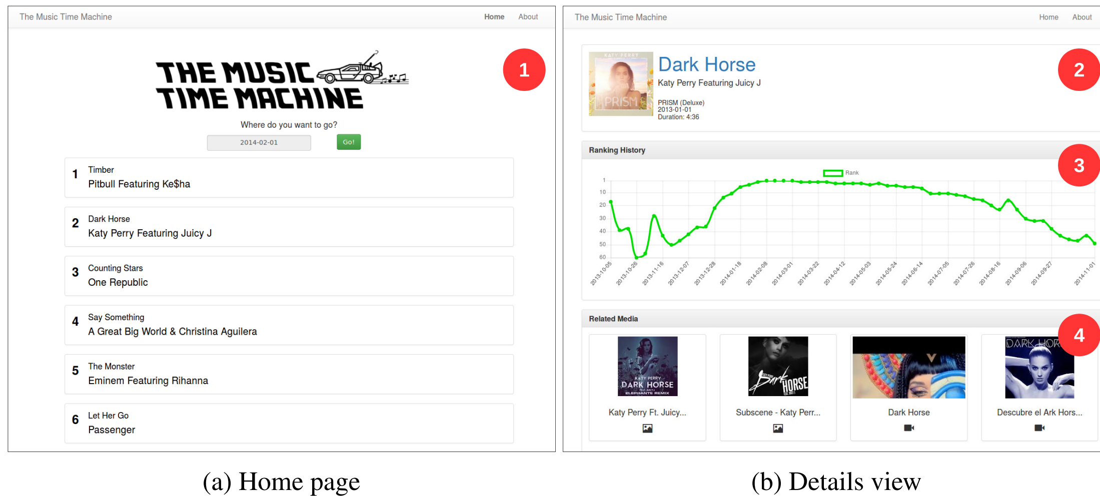
The use case involves the implementation of a test web application, called *The Music Time Machine* (MTM). In essence, MTM shows detailed information about songs that were popular at an specific date, provided by the end-user. The application is similar to other applications, such as The Billboard Hot 100[2], My Birthday Hits[3], or Playback.fm's Birthday Song[4].

The MTM application comprises 4 software components:

- $\boldsymbol{T_1}$ **:** The first component contains a date picker and a button, which loads the top-10 songs at the specified date (Figure 6.1a-1). The ranks of songs correspond to a subset of the Billboard Hot 100, the music industry standard record chart in the United States for singles, originally published by Billboard magazine every week.
- $\boldsymbol{T_2}$ **:** The second component shows information related to the song, including the title, duration, artist names, album name, and album release date. The information also includes a thumbnail image of the album, and a hyperlink that opens a new tab to an external application that streams the song's audio (Figure 6.1b-2).
- $\boldsymbol{T_3}$ **:** The third component presents a line graph showing the ranks of the song over time (Figure 6.1b-3). The line graph summarizes the historical ranks of the selected song in the Billboard Hot 100 chart.
- $\boldsymbol{T_4}$ **:** The fourth component displays links to images and videos related to the song or the artist (Figure 6.1b-4). The links shown in this section feature similar media, which may not be from the same specific song. The links in this component show a thumbnail of the media resource, a title, and an icon showing the type of media (image or video). The media resource item is presented as a hyperlink that redirects the user to the data source.

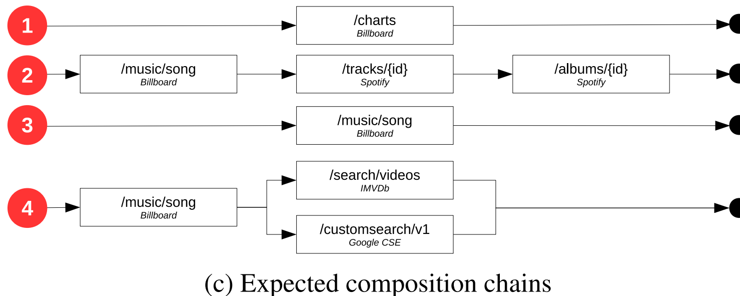
In order to test the performance and quality of the application, we used the previously mentioned components, and the characterization of composition patterns proposed by Jäeger et al. [142], to create a set of tasks that aim at understanding how developers produce service compositions using different 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

[2]http://www.billboard.com/charts/hot-100
[3]https://www.mybirthdayhits.com/
[4]http://playback.fm/birthday-song



(a) Home page

(b) Details view



(c) Expected composition chains

Figure 6.1: The Music Time Machine (MTM) test application

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 two or more independent composition chains are eventually merged. More specifically, this pattern corresponds to a parallel flow with AND split and join [142]. This pattern is used for T_4. Figure 6.1c shows diagrammatically the workflows for each component.

6.1.4 Participants

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 data collection for our study.

All of the participants reported having used the programming language (Javascript) previously.

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.

6.1.5 Data Analysis

Given the small sample size and the expected normal distribution of the data collected in this study, we adopted the Student's t-test, as suggested by de Winter [143], which has also been found to be highly similar to the Mann-Whitney-Wilcoxon test, used for small unpaired datasets with skewed distributions. Since a test of significance for normality may lack power to detect the deviation of the variables from normality, we assume a normal distribution of our variables based on previous work on similar software development experiments [144, 145, 146, 147]. In this study, we consider an acceptable probability of 0.05 for Type-I error, i.e. rejecting the null hypothesis when it is true.

6.2 Empirical Study Protocol

The protocol of the study was divided in three stages. Stage 1 (Preparation) included sessions related to software development process and service-oriented architectures. Stage 2 (Training) involved an introductory tutorial on the features of LRA and its tools. Stage 3 (Working session) involved a working sessions in which 32 participants solved the tasks assigned for the system of the study.

6.2.1 Preparation

The preparation stage was structured in two 2-hour leveling sessions covering topics related to version control systems, front-end programming, and service-oriented architectures. In these sessions, the students are instructed to work in pairs, which is a regular practice in the course, and has demonstrated higher effectiveness in industrial and academic scenarios [148, 149].

The purpose of these sessions is to harmonize the fundamental knowledge required to build web applications. Hence, procedural aspects, such as git repository management, and technical aspects, such as the REST API consumption from Javascript, are covered in these sessions. This decreases possible extraneous variations in the outcome variables, since the participants, as a group, have a comparable understanding of the tools being used. 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.

6.2.2 Training

The training session was structured in a 40-minute session. The first part of the session involved a presentation of the LRA approach, and the semantics used in the specifications of the Web API. The second part consisted of a laboratory workshop that provided hands-on experience with the LRA environment.

The 32 participants received a presentation of the *schema.org* concepts used to describe music-related elements. Then, the task T_1 was used to explore the features of LRA, the visual query assistant, and the differences with the traditional approach explained in the previous stage. One teaching assistant helped the participants to develop the functionality required by the task. At the end of this session, the participant pairs were randomly divided in two groups of equal size. One group was assigned to use LRA, and the other group to the traditional HTTP requests to individual service providers. It is important to note that although the lab sessions are mandatory, the students were asked for their voluntary participation in the study, emphasizing that their consent to participate has no repercussions on their grades.

6.2.3 Working Session

Each pair of participants was assigned to individual workstations with the software dependencies installed, including LRA. At the beginning of the session, participants received a 5 minute presentation on the instructions to submit their responses, following the standard procedures of laboratory sessions of the course.

This stage involved a session with a maximum time restriction of 2.5 hours, where a teaching assistant was available to answer high-level questions about both approaches. The session was instrumented by a survey in eClass, the University's learning management system, powered by Moodle. The survey presents one task at a time, in the same order, until all tasks in the questionnaire are answered. In the survey, the students were asked to submit the commit hash of their git repository, 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 [140], like in the usability study for VQA.

6.2.4 Data Collection

The working session was instrumented with a survey module provided by the University's learning management system. In this session, one member of each pair of participants logged into the survey application using the institutional ID and password. In the survey, first, the participant inputs the institutional ID of his partner for the session. Then, the survey application presents one task at the time until all tasks are answered. The survey does not allow participants to skip tasks, or return

to a task completed previously. The survey application allows to measure the total time spent by the participants in each task, by calculating the time difference between a task is presented to the participant, and the time an answer is submitted. Each task presented in the survey is composed of a short answer question, requesting the commit hash (identifier) that contains the functionality for the given task.

In order to evaluate 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. Given the nature and variety of the complexity metrics, in the following we outline the measures adopted in the study.

Program complexity

Desired quality attributes of software systems, such as maintainability or reliability, cannot be measured precisely until the system becomes available and operational. As a consequence, the software engineering community has devoted a significant effort to establish empirical predictive theories that correlate structural attributes of software representations, found in flow charts or UML diagrams, with quality attributes. These measures are often referred as complexity measures.

Although work in this area has been obfuscated by the lack of consistent evidence regarding

the effectiveness of the prediction models [150, 151], these metrics are still used in academic and industrial environments to estimate the expected quality and validate a great number of technologies.

Given the wide range of metrics used to estimate the complexity of a program, we only include the most common and extensively adopted metrics. In addition, object-oriented complexity measures, such as depth of inheritance tree, were excluded from this study, because the tasks were planned in a way that answers do not require significant considerations regarding the object-oriented design. Extrinsic metrics, such as readability, were also excluded, 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 [152] (vocabulary, length, volume, difficulty, effort, and estimated bugs), cyclomatic complexity [153], and maintainability index [154]. Accordingly, in the following sections, the complexity metrics used in our experiments are described.

Lines of Code (LOC): At the most basic level, software artifacts are expressed in a concrete form and can be described in terms of their size. The most popular measure of program size is the number of lines of code, which has considerable variations depending on what is counted and how it is counted. However, the most widely accepted definition indicates that a LOC is any statement in the program, except for comments and blank lines [155], which is also referred to as noncommented lines (NCLOC) or effective lines of code (ELOC).

A well-known problem with LOC is the inconsistent relationship between a line of code and its associated complexity. In other words, the metric does not acknowledge that some lines are more difficult to code than others. Hence, lines of code alone cannot directly indicate external attributes such as effort. However, Sjøberg et al. [150] have found that size-based metrics are good estimators of maintainability, outperforming more sophisticated complexity measures, such as coupling and depth of inheritance-tree.

Halstead's Metrics: Maurice Halstead proposed the most successful and widely used metrics of complexity based on code size, in what he called *Software Science* [152]. Halstead metrics performs a static analysis of the code, understanding a program as a collection of tokens, classified as either operators or operands. The four basic measures for these tokens are the following:

$$\begin{aligned} \eta_1 &= \text{Number of unique operators} & \eta_2 &= \text{Number of unique operands} \\ N_1 &= \text{Total occurrences of operators} & N_2 &= \text{Total occurrences of operands} \end{aligned} \quad (6.1)$$

These four basic metrics are used to derive metrics for other attributes, such program, vocabulary

(η), length (N), volume (V), difficulty (D), estimated effort (E), and number of delivered bugs (B).

$$\begin{aligned} \eta &= \eta_1 + \eta_2 & N &= N_1 + N_2 & V &= N \times log_2\eta \\ D &= (\eta_1/2) \times (N_2/\eta_2) & E &= D \times V & B &= V/3000 \end{aligned} \tag{6.2}$$

Despite their popularity, the metrics have been criticized by other authors, because the meaning of attributes such as volume and difficulty, were defined arbitrarily by Halstead, without consensus of the community [150, 151]. Additionally, there is no standard definition of operators and operands to use as the counting strategy, which may result in inconsistencies.

Therefore, we have explicitly defined a counting strategy that considers identifiers and constants as operands, and reserved words (e.g. `if`, `for`, `break`) and operators (e.g. =, `;`, >, +) as operators. Some special considerations in our analysis are that a pair of parenthesis is considered a single operator, comments are considered neither an operator nor an operand, and method identifiers are counted as operand in its declaration, and as operator in a call statement. This is strategy is aligned with traditional counting strategies proposed for other programming languages [156, 157].

Cyclomatic Complexity: The cyclomatic complexity metric, introduced by Thomas McCabe in 1976 [153], is a control flow measure that uses the number of linearly independent paths through a program's source code. The cyclomatic complexity is one of the most widely used and well acknowledged complexity metrics.

The control flow measures are usually modeled with flowgraphs, a directed graph where each node corresponds to a block of code that always executes sequentially, and each edge indicates the flow of control from one statement or basic block to another. In McCabe's formulation, the cyclomatic number is calculated as $C(F) = e - n + 2p$, where F represents the program's flowchart with a single connected component, and e and n represent the edges and nodes in the flowchart, respectively. Since for every node representing a predicate there is exactly one collecting node and there are unique entry and exit nodes, it follows that $C(F) = 1 + \pi$ where π represent the number of predicates, showing that the cyclomatic complexity of a program equals the number of predicates plus one [153].

Maintainability Index: The Maintainability Index (MI) is a polynomial composition of three traditional source code metrics (Halstead's volume metrics, McCabe's Cyclomatic Complexity, and lines of code) into a single number that indicates relative maintainability. The original maintainability formula was derived through statistical regression analysis on subjective maintainability scores provided to a number of systems by engineers at Hewlett-Packard in the early 90s.

The most current definition of the maintainability index calculates a value between 0 and 100,

where higher values indicate better maintainability. Throughout the years, the exact formulation has changed [154], and currently the most accepted definition is the following

$$MI = \max\left(0, \frac{(171 - 5.2 \times ln(V) - 0.23 \times C - 16.2 \times ln(LOC)) \times 100}{171}\right)$$

The maintainability index has been promoted by the Software Engineering Institute at Carnegie Mellon University, where they suggested that "the aggregate strength of this work and the underlying simplicity of the concept make the MI technique potentially very useful for operational Department of Defense (DoD) systems" [158]. Additionally, the maintainability index has been adopted as one of the metrics provided in several versions of Microsoft Visual Studio and other popular IDEs.

6.3 Results

In Table 6.1, 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, Figure 6.2 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.

Table 6.1: Results of the Usability Evaluation

	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**		

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

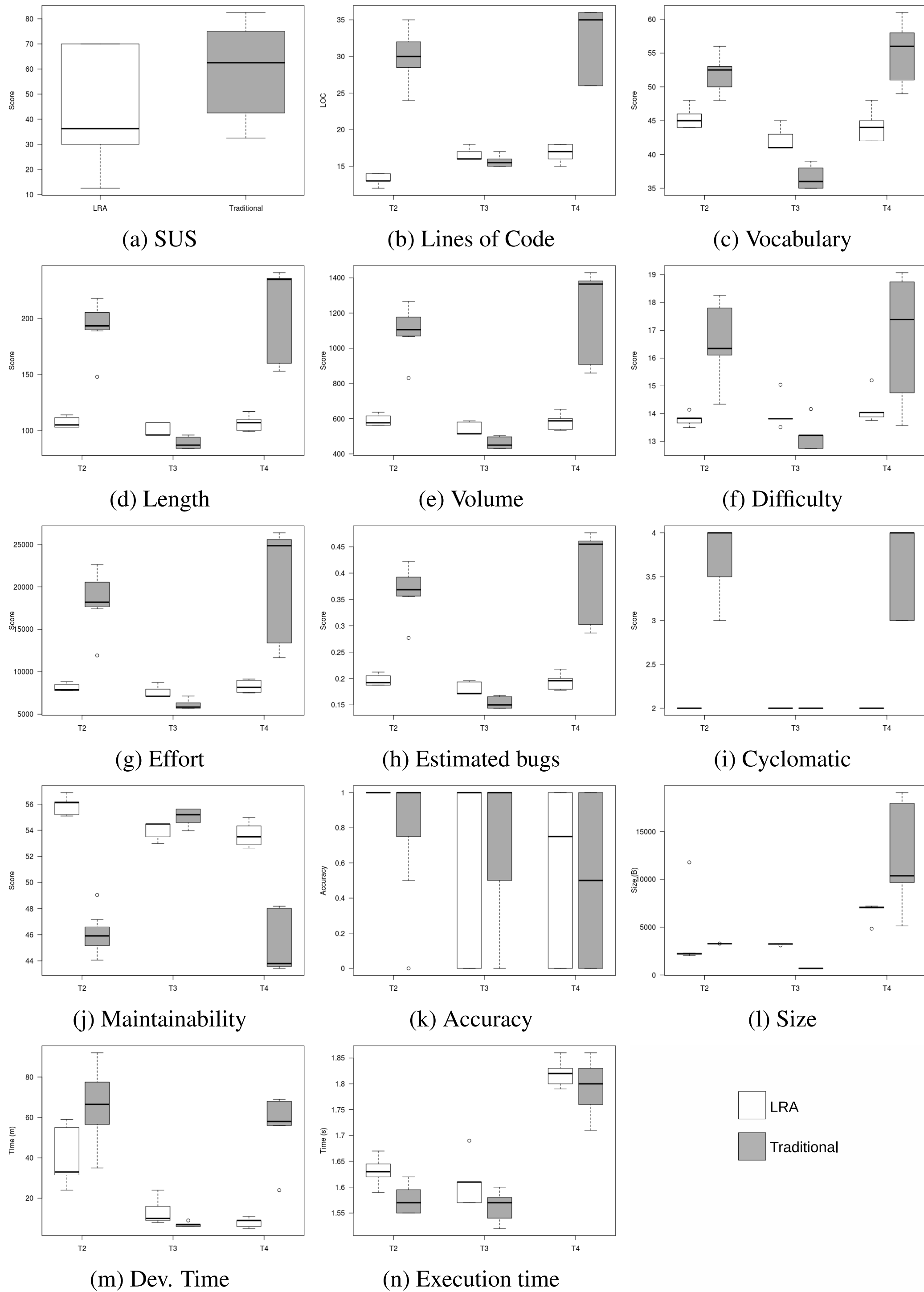


Figure 6.2: Metrics

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 minutes, in contrast with the 60 minutes 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 minutes, as opposed to the 55 minutes 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 [140], 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 [159].

6.4 Discussion

The results 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 minutes in the second task, they spent only 13.4 and 8 minutes 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 [160, 161]. 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.

6.4.1 Threats to Validity

In this study, we measured the complexity of software artifacts and the performance of developers when creating applications that retrieve and integrate data from multiple APIs. We compared two approaches namely, LRA and the traditional manually-designed API-consumption workflow. We

used a test web application to exercise the implementation of three software components of different complexity, in order to investigate the effect of diverse composition patterns.

Moreover, we understand developers' performance in terms of the time they take to complete each task and their correctness, as defined by the 3-level scoring mechanism in Section 6.2.4. The application performance is also empirically analyzed in terms of the execution time and size of the transmitted response documents. Furthermore, structural complexity is quantified using multiple measures, in order to avoid mono-method bias.

The correctness of each expected solution was validated by a software engineer with 7 years of experience in web-based applications and service-oriented architectures. The tasks were designed to cover representative composition patterns of different complexity. 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.

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 [162].

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 (Section 6.2.2), 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 [144, 163]

The study was divided in three stages in a span of three 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 Chapter 5, 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.

7

Conclusions

Nowadays, a substantial amount of Web data is exchanged through Web APIs that expose data in formats such as JSON or XML, which increases the opportunities for creating added value through service reuse and composition. However, these data responses are typically not grounded in semantics, which is a necessary prerequisite for automatically interpreting and interlinking the exchanged content. Thus, developing applications that rely on Web APIs still requires a considerable effort by application developers. They first have to find the APIs that may provide the desired data, which implies understanding the nature of the data these APIs consume and produce. Then, they need to write code for invoking individual APIs, and transforming the data produced by one to the format required in order to be consumed by another. Next, developers need to obtain the necessary access credentials from the providers, and incorporate them into the API requests. Thereupon, responses from multiple data sources are received in different formats and structures, which have to be mapped and adjusted according to the developer's application structure. And finally, in case of duplicates or inconsistent responses, developers have to create rules to reconcile the differences.

A long line of semantic-representation languages has been motivated by the need to simplify and automate this development effort. However, after many years of effort, there is no generally accepted formalism for describing the semantics of web services, or practical frameworks making use of such descriptions. Two main obstacles have curtailed the wide adoption of previous approaches: first, the complexity of their description formalisms, and, second, their limited coordination functionalities, focusing primarily on discovery and approximate service composition based on input-output interface descriptions.

For a long time, the adoption of Semantic Web technologies suffered due to the lack of clear incentives for developers to use them. This aspect has improved, as successful companies, such

as major search engines, have been indexing structured data, encoded as JSON-LD, RDFa, and microformats. Furthermore, the emergence of the API economy has transformed APIs into enablers of new business models and new selling strategies across channels. This represents an opportunity to align business and technology incentives towards the creation of automated service composition frameworks.

Nowadays, the appeal of using online channels to create revenue has contributed to the emergence of applications that bring together huge numbers of data consumers and providers, expanding the market choices available to consumers, and giving producers access to new customers. Companies like Trivago, Hootsuite, and Hopper are examples of applications that develop their business model on the aggregation of data produced by third-party companies. In addition, Web APIs have had a significant impact on the Business-to-Business (B2B) commerce on the Internet. A common growth technique for companies is to create Web APIs that allow programmatic interactions with their service catalogue, as in the case of Google Ads, eBay, and Yelp.

The work presented in this thesis advances the field of service-oriented software engineering by leveraging Semantic Web technologies to introduce a semantic description model for Web APIs, along with a graph-based composition methodology that creates sequences of Web API operations that can answer data queries, expressed as a conjunctive and well-designed pattern in SPARQL. To manage a possible aversion to Semantic Web technologies, we use less disruptive tools, such as the LRA Workbench, to gradually introduce developers to the principles and practices of Semantic Web. Based upon the results found in the performance evaluation (Chapter 4), and the effectiveness empirical study (Chapter 6), we believe that the use of Linked REST APIs may have a positive impact in several B2B productivity and operational elements of value, such as the simplification of tasks, effort reduction, and integration with other businesses, and may also be used to easily bring together a large number of data consumers and providers under one virtual roof.

7.1 Contributions

In this section, the contributions of this thesis to tackle the shortcomings of previous approaches are briefly summarized.

C1: Semantic Service Model Specification

This thesis presents Linked REST API, a semantic service description model that allows the annotation of functional semantics of REST APIs, using Linked-Data ontologies (Chapter 3). The semantic description builds on lessons learned from a long history of semantic web services and service-description formats, and leverages the effectiveness of semantic-representation languages used in well-known data integration projects (such as LOD and *schema.org*), and the usefulness

of widely adopted service description formats (such as OpenAPI and RAML), to specify the implementation details of REST APIs. The description model relies on the flexibility of SPARQL graph patterns to express the data offered by REST APIs. This data representation lays the foundation for automated processing, where the LRA model acts a semantically-enriched abstraction over the concrete service implementation, that exposes the schema of the data provided by the REST API.

C2: Automated Service Composition Methodology

Based on the semantic service model specification, LRA proposes a middleware for REST-service integration, that can automatically compose API calls to respond to data queries, considering quality and access-control constraints (Chapter 4). The middleware implements an automated method for service composition, based on iterative subgraph isomorphism. More specifically, our service-composition process involves (a) the discovery of API operations relevant to the input SPARQL query, (b) the traversal of these APIs through their possible data production-consumption relationships, and (c) the heuristic reduction of the search space, for improved performance.

Through our evaluation, we demonstrated the accuracy and performance of our approach. We used our semantic service model to describe a number of real-world services, and then presented these specifications to the LRA composition method, which composed them into chains to respond to typical queries specified by a domain expert. Our results validate the accuracy and performance of our approach in a realistic scenario, where service descriptions and composition requests are used by developers in the creation of diverse software application features. In our evaluation dataset, our composition methodology achieved an average F-measure of 96.2%, with a processing time of 15 and 896 milliseconds when the maximum composition length is 1 and 3, respectively.

C3: Semantically Annotated Evaluation Dataset

Previous research methods for automated service composition have often been evaluated with illustrative examples and synthetic datasets. In addition, the existing test datasets do not express the specific relationships between input and output concepts, lack natural language descriptions of the services, and for some of them they are no longer available, or there are no reference requests used on the collections. In this thesis, we present a dataset and an evaluation methodology around a realistic use case of software-engineering teams building applications that integrate multiple web services (Section 4.4.2). In this use case, a developer, who has previously identified a set of relevant service providers, is searching for a web service, or a combination of services, that provides functionality needed in the application under development. The test collection was extracted from 52 real service offerings, in order to represent the variety in terms of design patterns and resources found in real environments. Additionally, the service operations are complemented by a set of representative queries, defined by a domain expert. The evaluation methodology also includes the

expected composition chains for each query (as determined by 4 service-oriented system experts), and the queries and services expressed in the LRA model, using the VIVO ontology in conjunction with SPARQL and Swagger.

C4: Integrated Development Environment for Linked REST APIs

The methodological shift proposed by LRA entails the use of a query domain-specific language, namely SPARQL, which has proven to be a powerful tool in the hands of experienced users, but is considered as difficult to understand for average application developers. This thesis introduces the LRA Workbench, which enables developers to express their information needs without extensive 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 (Chapter 5). The LRA Workbench 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.

We evaluated the usability of the LRA Workbench through an empirical study that investigates the performance of developers when formulating LRA-compliant queries. Our results demonstrate that developers perceive the LRA Workbench as considerably more usable than YASGUI, a popular SPARQL interface used as front-end by a large number of Linked Data publishers. The results show that LRA Workbench users took significantly less time overall (in average 201 seconds, compared to 740 seconds for YASGUI users), and required fewer attempts to formulate the queries than YASGUI users (in average, 1.175 and 7.257 attempts for LRA Workbench and YASGUI, respectively). In addition, queries formulated through VQA were more accurate than queries created with YASGUI, although the difference was not statistically significant (in average, F-measure of 0.92 and 0.69 for LRA Workbench and YASGUI users, respectively). Furthermore, VQA presents higher usability ratings than YASGUI.

C5: An Empirical Study on End-to-End Automated Service Composition

This thesis also contributes an empirical study that evaluates the impact of LRA in the software-development process, using the traditional hand-crafted endpoint request workflows as the industry baseline (Chapter 6). In this study, we measured the quality of the produced software artifacts, and the accuracy and efficiency of developers when asked to develop a *mashup* web application using the LRA environment. 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.

The results show that, when implementing service compositions, LRA developers produce significantly less complex code, when compared to developers using classic development method-

ologies. 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.

7.2 Future Work

Our future avenues of investigation include the development of tools to assist in the semantic annotation of web services. Currently, LRA services are encoded by manually enriching the syntactic service description with graph patterns that represent the schema of the data provided by the service. This labor-intensive task was identified as one of the causes of failure of Semantic Web Services in the past. However, the syntactic description of services, which includes definitions in natural language, can be used to infer ontology relations between inputs and outputs, in order to promote those descriptions into LRA service descriptions, which would simplify the development life cycle with semantically-enhanced web services.

In addition, a natural extension of our work includes software engineering approaches to increase the performance. For example, distributed search space exploration by branching out the execution after pairwise matching at each level; or the inclusion of more sophisticated pruning mechanisms based on heuristics, for instance, early discard of composition chains that include operations that are present in shorter composition chains, when the duplicated sources heuristics is activated. Additionally, in industrial scenarios the middleware will need to address replication for best throughput and availability, and fault-tolerant mechanisms to recover from failure in distributed environments.

Finally, the software-engineering community requires access to realistic service test datasets, empirical evidence regarding the advantages of automated composition techniques, and truthful insights about its limitations, in order to invest in semantically-enriched description efforts. Our future work includes investigating the characteristics of automatic service integration and composition scenarios in non-academic applications. We would like to evaluate the performance of developers integrating data from multiple service providers for industrial-size applications using LRA.

Bibliography

[1] J. M. Smith, P. A. Bernstein, U. Dayal, N. Goodman, T. Landers, K. W. T. Lin, and E. Wong, "Multibase: integrating heterogeneous distributed database systems," in *Proceedings of the May 4-7, 1981, national computer conference*, ser. AFIPS '81. New York, NY, USA: ACM, 1981, pp. 487–499.

[2] H. G. Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom, "The TSIMMIS approach to mediation: Data models and languages," *J. Intell. Inf. Syst.*, vol. 8, no. 2, pp. 117–132, 1997.

[3] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava, "The information manifold," in *In Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments*, pp. 85–91.

[4] V. d. Subrahmanian, S. Adali, A. Brink, R. Emery, J. J. Lu, A. Rajput, T. J. Rogers, R. Ross, and C. Ward, "HERMES: A heterogeneous reasoning and mediator system," 1995.

[5] B. Iyer and M. Subramaniam, "The strategic value of APIs," *Harvard Business Review*, p. 1, 2015.

[6] C. Pedrinaci and J. Domingue, "Toward the next wave of services: Linked services for the web of data," *Journal of Universal Computer Science*, vol. 16, no. 13, pp. 1694–1719, jul 2010.

[7] M. Schmachtenberg, C. Bizer, and H. Paulheim, "State of the LOD cloud 2014," *University of Mannheim, Data and Web Science Group [en ligne]*, vol. 30, 2014.

[8] R. Guha, "Introducing schema.org: Search engines come together for a richer web," https://googleblog.blogspot.ca/2011/06/introducing-schemaorg-search-engines.html, 2011.

[9] C. Feier, A. Polleres, R. Dumitru, J. Domingue, M. Stollberg, and D. Fensel, "Towards intelligent web services: The web service modeling ontology (WSMO)," 2005.

[10] J. Kopecky, T. Vitvar, C. Bournez, and J. Farrell, “SAWSDL: Semantic annotations for WSDL and XML schema,” *Internet Computing, IEEE*, vol. 11, no. 6, pp. 60–67, 2007.

[11] S. Speiser and A. Harth, “Integrating linked data and services with linked data services,” in *The Semantic Web: Research and Applications*. Springer, 2011.

[12] M. Lanthaler and C. Gutl, “A semantic description language for RESTful data services to combat semaphobia,” in *Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on*, May 2011, pp. 47–53.

[13] M. Lenzerini, “Data integration: A theoretical perspective,” in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS ’02. New York, NY, USA: ACM, 2002, pp. 233–246.

[14] M. Friedman, A. Levy, and T. Millstein, “Navigational plans for data integration,” in *In Proceedings of the National Conference on Artificial Intelligence (AAAI*. AAAI Press/The MIT Press, 1999, pp. 67–73.

[15] R. Fagin, L. M. Haas, M. Hernández, R. J. Miller, L. Popa, and Y. Velegrakis, “Clio: Schema mapping creation and data exchange,” in *Conceptual Modeling: Foundations and Applications*. Springer, 2009, pp. 198–236.

[16] Y. Arens, C. Y. Chee, C.-N. Hsu, and C. A. Knoblock, “Retrieving and integrating data from multiple information sources,” *International Journal of Intelligent and Cooperative Information Systems*, vol. 2, no. 02, pp. 127–158, 1993.

[17] A. Halevy, A. Rajaraman, and J. Ordille, “Data integration: the teenage years,” in *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 9–16.

[18] A. Doan, P. Domingos, and A. Y. Halevy, “Reconciling schemas of disparate data sources: A machine-learning approach,” in *ACM Sigmod Record*, vol. 30, no. 2. ACM, 2001, pp. 509–520.

[19] H.-H. Do and E. Rahm, “COMA: a system for flexible combination of schema matching approaches,” in *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 610–621.

[20] J. Madhavan, P. A. Bernstein, and E. Rahm, “Generic schema matching with cupid,” in *VLDB*, vol. 1, 2001, pp. 49–58.

[21] T. Berners-Lee, J. Hendler, O. Lassila *et al.*, “The semantic web,” *Scientific American*, vol. 284, no. 5, pp. 28–37, 2001.

[22] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 concepts and abstract syntax," *W3C Recommendation*, 2014.

[23] D. Brickley and R. Guha, "RDF Vocabulary Description Language 1.0: RDF Schema," World Wide Web Consortium, W3C Recommendation, February 2004.

[24] D. L. McGuinness, F. Van Harmelen *et al.*, "OWL web ontology language overview," *W3C recommendation*, vol. 10, no. 10, p. 2004, 2004.

[25] C. Bizer, T. Heath, and T. Berners-Lee, "Linked data: the story so far," *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, pp. 205–227, 2009.

[26] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer *et al.*, "DBpedia–a large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, vol. 6, no. 2, pp. 167–195, 2015.

[27] C. Martinez-Cruz, I. J. Blanco, and M. A. Vila, "Ontologies versus relational databases: are they so different? a comparison," *Artificial Intelligence Review*, vol. 38, no. 4, pp. 271–290, 2012.

[28] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana *et al.*, "Web services description language (WSDL) 1.1," 2001.

[29] M. J. Hadley, "Web application description language (WADL)," 2006.

[30] D. Martin, M. Burstein, J. Hobbs, O. Lassila, McDermott *et al.*, "OWL-S: Semantic markup for web services," *W3C member submission*, vol. 22, pp. 2007–04, 2004.

[31] S. Battle, A. Bernstein, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. Mcilraith, D. Mcguinness, J. Su, and S. Tabet, "Semantic Web Services Framework (SWSF) Overview," Tech. Rep., 2005.

[32] R. Akkiraju, J. Farrell, J. A. Miller, M. Nagarajan, A. P. Sheth, and K. Verma, "Web service semantics - WSDL-S," 2005.

[33] J. Köpke, J. Eder, and D. Joham, "Towards path-based semantic annotation for web service discovery," in *Information Systems Engineering in Complex Environments*. Springer, 2014, pp. 133–147.

[34] A. P. Sheth, K. Gomadam, and J. Lathem, "SA-REST: Semantically interoperable and easier-to-use services and mashups," *IEEE Internet Computing*, vol. 11, no. 6, p. 91, 2007.

[35] J. Kopecky, T. Vitvar, D. Fensel, and K. Gomadam, "hRESTS & MicroWSMO," *CMS WG Working Draft*, 2009.

[36] D. Roman, J. Kopeckỳ, T. Vitvar, J. Domingue, and D. Fensel, "WSMO-Lite and hRESTS: Lightweight semantic annotations for web services and RESTful APIs," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 31, pp. 39–58, 2015.

[37] T. Vitvar, J. Kopecký, J. Viskova, and D. Fensel, *The Semantic Web: Research and Applications: 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, ch. WSMO-Lite Annotations for Web Services, pp. 674–689.

[38] M. Hepp, "Products and services ontologies: a methodology for deriving OWL ontologies from industrial categorization standards," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 2, no. 1, pp. 72–99, 2006.

[39] M. Klusch, B. Fries, and K. Sycara, "Owls-mx: A hybrid semantic web service matchmaker for owl-s services," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 7, no. 2, pp. 121–133, 2009.

[40] M. Gawinecki, G. Cabri, M. Paprzycki, and M. Ganzha, "WSColab: Structured collaborative tagging for web service matchmaking." in *WEBIST (1)*, 2010, pp. 70–77.

[41] L. Cabral and J. Domingue, *Ontology Based Discovery of Semantic Web Services with IRS-III*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 191–202.

[42] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara, "Semantic matching of web services capabilities," in *International Semantic Web Conference*. Springer, 2002, pp. 333–347.

[43] M. C. Jaeger, G. Rojec-Goldmann, C. Liebetruth, G. Mühl, and K. Geihs, "Ranked matching for service descriptions using OWL-S," in *Kommunikation in Verteilten Systemen (KiVS)*. Springer, 2005, pp. 91–102.

[44] J. Huang, Y. Zhou, Q. Duan, and C. c. Xing, "Semantic web service composition in big data environment," in *IEEE Global Communications Conference 2017 (GLOBECOM 2017)*, Dec 2017.

[45] M. Klusch and P. Kapahnke, "The iSeM matchmaker: A flexible approach for adaptive hybrid semantic service selection," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 15, pp. 1–14, 2012.

[46] U. Kuster and B. Konig-Ries, "Semantic service discovery with DIANE service descriptions," in *2007 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Workshops*, Nov 2007.

[47] A. A. Patil, S. A. Oundhakar, A. P. Sheth, and K. Verma, "METEOR-S web service annotation framework," in *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004, pp. 553–562.

[48] G. C. Hobold and F. Siqueira, "Discovery of semantic web services compositions based on SAWSDL annotations," in *Web Services (ICWS), 2012 IEEE 19th International Conference on*. IEEE, 2012.

[49] P. Rodriguez-Mier, C. Pedrinaci, M. Lama, and M. Mucientes, "An integrated semantic web service discovery and composition framework," *IEEE Transactions on Services Computing*, vol. 9, no. 4, pp. 537–550, July 2016.

[50] A. Heß, E. Johnston, and N. Kushmerick, "ASSAM: A tool for semi-automatically annotating semantic web services," in *The Semantic Web–ISWC 2004*. Springer, 2004, pp. 320–334.

[51] E. M. Maximilien, H. Wilkinson, N. Desai, and S. Tai, *Service-Oriented Computing – ICSOC 2007: Fifth International Conference, Vienna, Austria, September 17-20, 2007. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, ch. A Domain-Specific Language for Web APIs and Services Mashups, pp. 13–26.

[52] R. Alarcón and E. Wilde, "From RESTful services to RDF: connecting the web and the semantic web," *CoRR*, vol. abs/1006.2718, 2010.

[53] M. L. Sbodio, D. Martin, and C. Moulin, "Discovering semantic web services using SPARQL and intelligent agents," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 8, no. 4, pp. 310–328, 2010.

[54] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, E. Mannens, R. Van de Walle, and J. G. Vallés, "RESTdesc — a functionality-centered approach to semantic service description and composition," in *Proceedings of the Ninth Extended Semantic Web Conference*, 2012.

[55] M. Taheriyan, C. A. Knoblock, P. Szekely, and J. L. Ambite, "Rapidly integrating services into the linked data cloud," in *ISWC 2012*. Springer, 2012, pp. 559–574.

[56] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: A semantic web rule language combining OWL and RuleML," World Wide Web Consortium, W3C Member Submission, 2004.

[57] S. Gupta, P. Szekely, C. A. Knoblock, A. Goel, M. Taheriyan, and M. Muslea, "Karma: A system for mapping structured sources into the semantic web," in *The Semantic Web: ESWC 2012 Satellite Events*. Springer, 2012, pp. 430–434.

[58] Facebook, "GraphQL," 2017. [Online]. Available: http://graphql.org/

[59] E. Prud, A. Seaborne *et al.*, “SPARQL query language for RDF,” *W3C Recommendation*, 2006.

[60] J. Pérez, M. Arenas, and C. Gutierrez, “Semantics and complexity of SPARQL,” *ACM Transactions on Database Systems*, vol. 34, 2009.

[61] M. Lanthaler and C. Gütl, “Hydra: A vocabulary for hypermedia-driven web APIs.” *Linked Data on the Web Workshop (LDOW)*, 2013.

[62] R. Angles and C. Gutierrez, “The expressive power of SPARQL,” *The Semantic Web - ISWC 2008*, pp. 114–129, 2008.

[63] D. Tomaszuk, “Inference rules for RDF(S) and OWL in N3Logic,” *arXiv preprint arXiv:1601.02650*, 2016.

[64] J. R. Ullmann, “An algorithm for subgraph isomorphism,” *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[65] J. M. García, D. Ruiz, and A. Ruiz-Cortés, “Improving semantic web services discovery using SPARQL-based repository filtering,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 17, pp. 12–24, 2012.

[66] M. Maleshkova, C. Pedrinaci, J. Domingue, G. Alvaro, and I. Martinez, “Using semantics for automating the authentication of web APIs,” in *International Semantic Web Conference*. Springer, 2010, pp. 534–549.

[67] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng, “Quality driven web services composition,” in *Proceedings of the 12th international conference on World Wide Web*. ACM, 2003, pp. 411–421.

[68] M. Alrifai and T. Risse, “Combining global optimization with local selection for efficient QoS-aware service composition,” in *Proceedings of the 18th International Conference on World Wide Web*, ser. WWW ’09. New York, NY, USA: ACM, 2009, pp. 881–890.

[69] M. A. Hernández and S. J. Stolfo, “Real-world data is dirty: Data cleansing and the merge/purge problem,” *Data mining and knowledge discovery*, vol. 2, no. 1, pp. 9–37, 1998.

[70] A. Doan, A. Halevy, and Z. Ives, *Principles of Data Integration*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[71] S. Sanfilippo, “Redis,” 2018. [Online]. Available: https://redis.io/

[72] F. Michel, C. F. Zucker, and F. Gandon, “SPARQL micro-services: Lightweight integration of web APIs and linked data,” in *LDOW 2018-Linked Data on the Web*, 2018, pp. 1–10.

[73] R. Verborgh, T. Steiner *et al.*, “RESTdesc: A functionality-centered approach to semantic service description and composition,” in *ESWC*, 2012.

[74] U. Küster, “An evaluation methodology and framework for semantic web services technology,” Ph.D. dissertation, Jena, Univ., Diss., 2010, 2011.

[75] Y. Ganjisaffar, H. Abolhassani, M. Neshati, and M. Jamali, “A similarity measure for OWL-S annotated web services,” in *Web Intelligence, 2006. WI 2006. IEEE/WIC/ACM International Conference on*. IEEE, 2006, pp. 621–624.

[76] A. Heß, E. Johnston, and N. Kushmerick, “Assam: A tool for semi-automatically annotating semantic web services,” in *International Semantic Web Conference*. Springer, 2004, pp. 320–334.

[77] S. Kona, A. Bansal, M. B. Blake, S. Bleul, and T. Weise, “WSC-2009: a quality of service-oriented web services challenge,” in *Commerce and Enterprise Computing, 2009. CEC’09. IEEE Conference on*. IEEE, 2009, pp. 487–490.

[78] A. Bekkouche, S. M. Benslimane, M. Huchard, C. Tibermacine, F. Hadjila, and M. Merzoug, “QoS-aware optimal and automated semantic web service composition with user’s constraints", journal="service oriented computing and applications,” vol. 11, no. 2, Jun 2017.

[79] C. J. Petrie, T. Margaria, H. Lausen, and M. Zaremba, *Semantic Web Services challenge: Results from the first year*. Springer Science & Business Media, 2008, vol. 8.

[80] M. Klusch, “The S3 contest: Performance evaluation of semantic service matchmakers. semantic web services–advancement through evaluation,” 2012.

[81] S. Baccar, M. Rouached, R. Verborgh, and M. Abid, “Declarative web services composition using proofs,” *Service Oriented Computing and Applications*, pp. 1–19, 2018.

[82] M. Klusch, P. Kapahnke, B. Fries, M. A. Khalid, and M. Vasileski, “OWLS service retrieval test collection,” 2010. [Online]. Available: http://projects.semwebcentral.org/projects/owls-tc/

[83] T. Brüeckmann, V. Gruhn, W. Koop, J. Ollesch, L. Pradel, F. Wessling, and M. Benner, “Codeless engineering of service mashups – an experience report,” in *Proceedings of the Fourteenth IEEE International Conference on Services Computing 2017 (SCC 2017)*, 2017.

[84] C. D. Manning, P. Raghavan, H. Schütze *et al.*, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008, vol. 1, ch. Evaluation in Information Retrieval.

[85] H. D. White and K. W. McCain, "Bibliometrics," *Annual review of information science and technology*, vol. 24, pp. 119–186, 1989.

[86] F. Narin *et al.*, *Evaluative bibliometrics: The use of publication and citation analysis in the evaluation of scientific activity*. Computer Horizons Washington, DC, 1976.

[87] K. W. Boyack and K. Börner, "Indicator-assisted evaluation and funding of research: Visualizing the influence of grants on the number and citation counts of research papers," *Journal of the Association for Information Science and Technology*, vol. 54, no. 5, pp. 447–461, 2003.

[88] P. R. McAllister and D. A. Wagner, "Relationship between R&D expenditures and publication output for US colleges and universities," *Research in Higher Education*, vol. 15, no. 1, pp. 3–30, 1981.

[89] M. R. Halperin and A. K. Chakrabarti, "Firm and industry characteristics influencing publications of scientists in large american companies," *R&D Management*, vol. 17, no. 3, pp. 167–173, 1987.

[90] J. S. Katz and B. R. Martin, "What is research collaboration?" *Research policy*, vol. 26, no. 1, pp. 1–18, 1997.

[91] U. Küster and B. König-Ries, "Evaluating semantic web service matchmaking effectiveness based on graded relevance," in *Proceedings of the Second International Conference on Service Matchmaking and Resource Retrieval in the Semantic Web-Volume 416*. CEUR-WS. org, 2008, pp. 32–46.

[92] V. Tsetsos, C. Anagnostopoulos, and S. Hadjiefthymiades, "On the evaluation of semantic web service matchmaking systems," in *Web Services, 2006. ECOWS'06. 4th European Conference on*. IEEE, 2006, pp. 255–264.

[93] D. Serrano, E. Stroulia, D. Lau, and T. Ng, "Linked REST APIs: A middleware for semantic REST API," in *IEEE International Conference on Web Services 2017 (ICWS 2017)*, 2017.

[94] J. Corson-Rikert, S. Mitchell, B. Lowe, N. Rejack, Y. Ding, and C. Guo, "The VIVO ontology," *Synthesis Lectures on Semantic Web: Theory and Technology*, p. 3, 2012.

[95] D. Brickley and L. Miller, "FOAF vocabulary specification 0.98," *Namespace document*, vol. 9, 2012.

[96] B. D'Arcus and F. Giasson, "Bibliographic ontology specification," http://bibliontology.com/specification, 2009.

[97] R. Rosenthal and R. L. Rosnow, *Essentials of Behavioral Research: Methods and Data Analysis*. McGraw-Hill, 2008, no. 3.

[98] M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente, "An empirical study of real-world SPARQL queries," in *USEWOD workshop*, 2011.

[99] C. J. V. Rijsbergen, *Information Retrieval*, 2nd ed. Newton, MA, USA: Butterworth-Heinemann, 1979.

[100] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of IR techniques," *ACM Transactions on Information Systems (TOIS)*, vol. 20, no. 4, pp. 422–446, 2002.

[101] D. Wei, T. Wang, J. Wang, and A. Bernstein, "SAWSDL-iMatcher: A customizable and effective semantic web service matchmaker," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 9, no. 4, pp. 402–417, 2011.

[102] N. Dalvi, A. Machanavajjhala, and B. Pang, "An analysis of structured data on the web," *Proceedings of the VLDB Endowment*, vol. 5, no. 7, pp. 680–691, 2012.

[103] OpenLink Software. Virtuoso. https://virtuoso.openlinksw.com/.

[104] L. Dodds. Twinkle: A SPARQL query tool. http://www.ldodds.com/projects/twinkle/. [Online]. Available: http://www.ldodds.com/projects/twinkle/

[105] L. Rietveld and R. Hoekstra, "The YASGUI family of SPARQL clients," *Semantic Web*, vol. 8, no. 3, pp. 373–383, 2017.

[106] T. Kurz. Squebi. https://github.com/tkurz/squebi. [Online]. Available: https://github.com/tkurz/squebi

[107] A. Yamaguchi, K. Kozaki, K. Lenz, H. Wu, and N. Kobayashi, "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*. CEUR-WS.org, 2014, pp. 83–94.

[108] A.-C. Ngonga Ngomo, L. Bühmann, C. Unger, J. Lehmann, and D. Gerber, "Sorry, i don't speak SPARQL: translating SPARQL queries into natural language," in *Proceedings of the 22nd international conference on World Wide Web*. ACM, 2013, pp. 977–988.

[109] G. Tummarello, R. Delbru, and E. Oren, "Sindice.com: Weaving the open linked data," in *The Semantic Web*. Springer, 2007, pp. 552–565.

[110] M. d'Aquin, L. Gridinoc, S. Angeletou, M. Sabou, and E. Motta, "Watson: A gateway for next generation semantic web applications," 2007.

[111] R. Bhagdev, S. Chapman, F. Ciravegna, V. Lanfranchi, and D. Petrelli, "Hybrid search: Effectively combining keywords and semantic searches," in *European Semantic Web Conference*. Springer, 2008, pp. 554–568.

[112] O. Ambrus, K. Möller, S. Handschuh *et al.*, "KonduitVQB: a visual query builder for SPARQL on the social semantic desktop," in *Workshop on visual interfaces to the social and semantic web*, 2010.

[113] M. Arenas, B. Cuenca Grau, E. Kharlamov, S. Marciuska, D. Zheleznyakov, and E. Jimenez-Ruiz, "SemFacet: semantic faceted search over YAGO," in *Proceedings of the 23rd International Conference on World Wide Web*. ACM, 2014, pp. 123–126.

[114] S. Heggestøyl, J. W. Klüwer, and A. Waaler, "PepeSearch: Easy to use and easy to install semantic data search," *The Semantic Web: ESWC 2016 Satellite Events*, 2016.

[115] S. Kawano, T. Watanabe, S. Mizuguchi, N. Araki, T. Katayama, and A. Yamaguchi, "TogoTable: cross-database annotation system using the resource description framework (RDF) data model," *Nucleic acids research*, vol. 42, no. W1, pp. W442–W448, 2014.

[116] K. Elbedweihy, S. N. Wrigley, and F. Ciravegna, "Evaluating semantic search query approaches with expert and casual users," in *International Semantic Web Conference*. Springer, 2012, pp. 274–286.

[117] M. Dubey, S. Dasgupta, A. Sharma, K. Höffner, and J. Lehmann, "Asknow: A framework for natural language query formalization in sparql," in *International Semantic Web Conference*. Springer, 2016, pp. 300–316.

[118] E. Kaufmann, A. Bernstein, and L. Fischer, "NLP-Reduce: A naive but domain-independent natural language interface for querying ontologies," in *4th European Semantic Web Conference ESWC*, 2007, pp. 1–2.

[119] E. Kaufmann and A. Bernstein, "Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 8, no. 4, pp. 377–393, 2010.

[120] E. Kaufmann, A. Bernstein, and R. Zumstein, "Querix: A natural language interface to query ontologies based on clarification dialogs," in *5th International Semantic Web Conference*. Springer, 2006, pp. 980–981.

[121] OpenLink Software. iSPARQL. https://www.openlinksw.com/isparql/.

[122] A. Russell, "Nitelight: A graphical editor for SPARQL queries," in *Proceedings of the 2007 International Conference on Posters and Demonstrations*. CEUR-WS.org, 2008, pp. 110–111.

[123] F. Hogenboom, V. Milea, F. Frasincar, and U. Kaymak, "RDF-GL: a SPARQL-based graphical query language for RDF," in *Emergent Web Intelligence: Advanced Information Retrieval*. Springer, 2010, pp. 87–116.

[124] S. Mazumdar, D. Petrelli, K. Elbedweihy, V. Lanfranchi, and F. Ciravegna, "Affective graphs: The visual appeal of linked data," *Semantic Web*, vol. 6, no. 3, pp. 277–312, 2015.

[125] F. Haag, S. Lohmann, S. Siek, and T. Ertl, "QueryVOWL: Visual composition of SPARQL queries," in *Revised Selected Papers of the ESWC 2015 Satellite Events*. Springer, 2015, pp. 62–66.

[126] P. Heim, J. Ziegler, and S. Lohmann, "gFacet: A browser for the web of data," in *Proceedings of the International Workshop on Interacting with Multimedia Content in the Social Semantic Web*, vol. 417, 2008, pp. 49–58.

[127] G. Barzdins, S. Rikacovs, and M. Zviedris, "Graphical query language as SPARQL frontend," in *Local Proceedings of 13th East-European Conference (ADBIS 2009)*, 2009, pp. 93–107.

[128] M. Jarrar and M. D. Dikaiakos, "MashQL: a query-by-diagram topping SPARQL," in *Proceedings of the 2nd international workshop on Ontologies and information systems for the semantic web*. ACM, 2008, pp. 89–96.

[129] F. Haag, S. Lohmann, S. Bold, and T. Ertl, "Visual SPARQL querying based on extended filter/flow graphs," in *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces*. ACM, 2014, pp. 305–312.

[130] C. W. Thompson, P. Pazandak, and H. R. Tennant, "Talk to your semantic web," *IEEE Internet Computing*, vol. 9, no. 6, pp. 75–78, 2005.

[131] G. Vega-Gorgojo, L. Slaughter, M. Giese, S. Heggestøyl, A. Soylu, and A. Waaler, "Visual query interfaces for semantic datasets: An evaluation study," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 39, 2016.

[132] K. Elbedweihy, S. N. Wrigley, F. Ciravegna, D. Reinhard, and A. Bernstein, "Evaluating semantic search systems to identify future directions of research," in *Extended Semantic Web Conference*. Springer, 2012.

[133] F. Frasincar, A. Telea, and G.-J. Houben, "Adapting graph visualization techniques for the visualization of RDF data," *Visualizing the semantic web*, vol. 2006, pp. 154–171, 2006.

[134] M. Powers, C. Lashley, P. Sanchez, and B. Shneiderman, "An experimental comparison of tabular and graphic data presentation," *International Journal of Man-Machine Studies*, vol. 20, no. 6, pp. 545–566, 1984.

[135] E. J. Wegman and Q. Luo, "High dimensional clustering using parallel coordinates and the grand tour," *Classification and Knowledge Organization*, 1997.

[136] D. Chang, L. Dooley, and J. E. Tuovinen, "Gestalt theory in visual screen design: a new look at an old subject," in *Proceedings of the 7th world conference on computers in education conference on Computers in education*. Australian Computer Society, Inc., 2002, pp. 5–12.

[137] M.-D. Pham, L. Passing, O. Erling, and P. Boncz, "Deriving an emergent relational schema from RDF data," in *Proceedings of the 24th International Conference on World Wide Web*. ACM, 2015.

[138] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP2Bench: a SPARQL performance benchmark," in *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 2009, pp. 222–233.

[139] B. DuCharme, *Learning SPARQL: querying and updating with SPARQL 1.1*. O'Reilly Media, Inc., 2013.

[140] J. Brooke *et al.*, "SUS-a quick and dirty usability scale," *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.

[141] A. Bangor, P. T. Kortum, and J. T. Miller, "An empirical evaluation of the system usability scale," *Intl. Journal of Human–Computer Interaction*, vol. 24, 2008.

[142] M. C. Jaeger, G. Rojec-Goldmann, and G. Muhl, "QoS aggregation for web service composition using workflow patterns," in *Enterprise distributed object computing conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*. IEEE, 2004, pp. 149–159.

[143] J. C. De Winter, "Using the student's t-test with extremely small sample sizes." *Practical Assessment, Research & Evaluation*, vol. 18, no. 10, 2013.

[144] F. Hermans and E. Aivaloglou, "Do code smells hamper novice programming? a controlled experiment on scratch programs," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[145] C. F. Kemerer, "Software complexity and software maintenance: A survey of empirical research," *Annals of Software Engineering*, vol. 1, no. 1, pp. 1–22, 1995.

[146] L. Hochstein, V. R. Basili, U. Vishkin, and J. Gilbert, “A pilot study to compare programming effort for two parallel programming models,” *Journal of Systems and Software*, vol. 81, no. 11, pp. 1920–1930, 2008.

[147] P. Baheti, E. Gehringer, and D. Stotts, “Exploring the efficacy of distributed pair programming,” in *Conference on Extreme Programming and Agile Methods*. Springer, 2002, pp. 208–220.

[148] A. Cockburn and L. Williams, “Extreme programming examined,” G. Succi and M. Marchesi, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, ch. The Costs and Benefits of Pair Programming, pp. 223–243.

[149] C. McDowell, L. Werner, H. E. Bullock, and J. Fernald, “Pair programming improves student retention, confidence, and program quality,” *Commun. ACM*, vol. 49, no. 8, pp. 90–95, Aug. 2006.

[150] D. I. Sjøberg, B. Anda, and A. Mockus, “Questioning software maintenance metrics: a comparative case study,” in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2012, pp. 107–110.

[151] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC Press, 2014.

[152] M. H. Halstead, “Elements of software science,” 1977.

[153] T. J. McCabe, “A complexity measure,” *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[154] K. D. Welker, “The software maintainability index revisited,” *CrossTalk*, vol. 14, pp. 18–21, 2001.

[155] S. D. Conte, H. E. Dunsmore, and Y. Shen, *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc., 1986.

[156] D. M. Miller, R. S. Maness, J. W. Howatt, and W. H. Shaw, “A software science counting strategy for the full ada language,” *SIGPLAN Not.*, vol. 22, no. 5, pp. 32–41, May 1987.

[157] N. F. Salt, “Defining software science counting strategies,” *SIGPLAN Not.*, vol. 17, no. 3, pp. 58–67, Mar. 1982.

[158] M. Bray, K. Brune, D. A. Fisher, J. Foreman, and M. Gerken, “C4 software technology reference guide-a prototype.” Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1997.

[159] A. Bangor, P. Kortum, and J. Miller, “Determining what individual SUS scores mean: Adding an adjective rating scale,” *Journal of usability studies*, vol. 4, no. 3, pp. 114–123, 2009.

[160] P. S. Ellen, W. O. Bearden, and S. Sharma, “Resistance to technological innovations: an examination of the role of self-efficacy and performance satisfaction,” *Journal of the Academy of Marketing Science*, vol. 19, no. 4, pp. 297–307, 1991.

[161] T. Kuisma, T. Laukkanen, and M. Hiltunen, “Mapping the reasons for resistance to internet banking: A means-end approach,” *International Journal of Information Management*, vol. 27, no. 2, pp. 75–85, 2007.

[162] A. Luse, A. M. Townsend, and B. E. Mennecke, “The blocking effect of preconceived bias,” *Decision Support Systems*, 2018.

[163] V. Guana and E. Stroulia, “End-to-end model-transformation comprehension through fine-grained traceability information,” *Software & Systems Modeling*, Jun 2017.