FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Mashup-based Linked Data Integration

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Tuan-Dat Trinh
Matrikelnummer 1229761

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung:
O.Univ.Prof. Dipl.-Ing. Dr.techn. A Min Tjoa
Mag. Dr. Elmar Kiesling

Diese Dissertation haben begutachtet:

O.Univ.Prof. Dipl.-Ing. Dr.techn.
A Min Tjoa

Prof. Dr.
Hamideh Afsarmanesh

Wien, 3. März 2016

Tuan-Dat Trinh

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Mashup-based Linked Data Integration

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Tuan-Dat Trinh
Registration Number 1229761

to the Faculty of Informatics
at the Vienna University of Technology

Advisor:
O.Univ.Prof. Dipl.-Ing. Dr.techn. A Min Tjoa
Mag. Dr. Elmar Kiesling

The dissertation has been reviewed by:

O.Univ.Prof. Dipl.-Ing. Dr.techn.
A Min Tjoa

Prof. Dr.
Hamideh Afsarmanesh

Vienna, 3rd March, 2016

Tuan-Dat Trinh

Erklärung zur Verfassung der Arbeit

Tuan-Dat Trinh
Donaufelderstrasse 54/3120, 1210 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. März 2016

Tuan-Dat Trinh

Acknowledgements

I would like to take this opportunity to express my sincere thanks to those who helped me in one way or another on conducting research and the writing of this thesis.

I want to express my heartfelt thanks to my advisor, O.Univ.Prof. Dipl.-Ing. Dr.techn. A Min Tjoa. The work presented in this thesis is not possible without his great support and advice. I also want to thank Mag. Dr. Elmar Kiesling, and Mag. Dipl.-Ing. Dr. Amin Anjomshoaa for participating on my Ph.D. committee and for their insightful feedback, comments and suggestions. I want to sincerely thank them for providing me the opportunity to participate in the Linked Data Lab, which enables me to get to know a brand new field and learn a lot of knowledge and skills.

I would like to show my sincere thanks to my family for their endless support during my 3-year stay at TU Wien University. Starting from the beginning of my study at TU Wien, they kept encouraging and supporting me whenever I felt frustrated for my course work, my research and my life. Without their love and support during the years, I would not have come to where I am.

I thank my friends in the Linked Data Lab: Ba-Lam Do, Peter Wetz, Peb Ruswono Aryan, and Bernhard Ortner for the stimulating discussions, the insightful comments, and for all the fun we have had in the last three years

Last but not the least, I would like to thank all the people that have provided your help and support to me during these years in Austria in the best way. All in all, thanks everyone for your help, support, and love.

Kurzfassung

Das so genannte Web der Daten wächst mit erstaunlicher Geschwindigkeit. Mittlerweile ist eine große Menge an Datenquellen, APIs, Services, und Datenvisualisierungen öffentlich verfügbar. Dennoch ist es nach wie vor eine große Herausforderung, komplexe Informationsbedürfnisse von Anwendern mittels Integration und Verarbeitung von Daten aus heterogenen Quellen zu befriedigen. Zur Lösung dieser Probleme gab es in den letzten Jahren viele Forschungsarbeiten, die sich mit mashup-basierter Datenintegration beschäftigten. Mashups ermöglichen Anwendern, Daten und Services zu integrieren und rekombinieren, um so eigenständig Rich Web Applications zu erstellen. Trotzdem ist es für Anwender ohne technische Expertise schwierig, solche Mashups effizient und effektiv zu entwerfen.

Um dieses Problem zu lösen, stellen wir eine Methode vor, die es ermöglicht, Mashups zu entwerfen, die heterogene Daten auf automatische, kollaborative, und verteilte Weise integrieren. Wir folgen dabei dem Visual Programming Paradigma basierend auf drei Grundsätzen: Offenheit, Vernetzung und Wiederverwendbarkeit. Die Methode fußt auf Semantic Web Technologien und dem Konzept der Linked Widgets, d.h., web widgets, die mit einem semantischen Modell ausgestattet sind. Linked Widgets sind konzipiert, um Herausforderungen der Datenintegration wie folgt zu bewältigen: (i) Die Wiederverwendbarkeit von Datenverarbeitungs-Prozessen wird unterstützt, (ii) Datenintegration mittels einfacher Operationen wird erleichtert, (iii) Es wird Anwendern ermöglicht, relevante Datenquellen auf Basis ihres Kontexts zu erforschen, (iv) Heterogeneität von Daten wird überwunden und (v) automatische Datenintegration erleichtert.

Diese Arbeit stellt ein neues Konzept von semantischen, kollaborativen und verteilten Mashups vor. Den Prinzipien des Semantic Web folgend, können die damit erstellten ad-hoc Datenintegrations-Applikationen heterogene Daten verschiedener Teilnehmer gleichzeitig verarbeiten und kombinieren. Dabei können die Daten von so verschiedenen Geräten wie Sensoren, eingebetteten Systemen, Handys, Desktop Computern, oder Web Servern stammen. Diese Vorgehensweise benötigt keine Server Infrastruktur, um Daten hochzuladen, vielmehr behalten die Teilnehmer Kontrolle über ihre Daten und stellen nur ein minimales Subset anderen Anwendern zur Verfügung. Verteilte Mashups operieren persistent und sind somit ideal für Anwendungsfälle wie Echtzeit-Monitoring oder Verarbeitung von Datenströmen.

Abstract

The web of data is growing at a staggering pace. A large number of data sources, APIs, services, and data visualizations are publicly available. Satisfying users' complex information needs by integrating and processing data from disparate sources, however, remains challenging. In recent years, a large stream of research into mashup-based data integration has emerged. These mashups foster combination and reuse of data and services and thereby have the potential for rapid creation of rich web applications. Nonetheless, users lacking technical expertise still face enormous barriers when trying to develop such mashups efficiently and effectively.

To address this issue, we introduce an approach to compose mashups that integrate heterogeneous data sources in an automatic, collaborative, and distributed manner. We follow a visual programming paradigm and aim for three guiding principles: openness, connectedness, and reusability. The approach is based on semantic web technologies and the concept of Linked Widgets, i.e., web widgets backed by a semantic model. Linked Widgets are designed to effectively tackle data integration challenges by (i) fostering reusability of data processing tasks, (ii) easing data integration via simple operations, (iii) allowing users to explore relevant data sources with regard to their context, (iv) tackling data heterogeneity, and (v) facilitating automatic data integration.

This thesis introduces a new model of semantic, collaborative, and distributed mashups. Following semantic web principles for data integration, these ad-hoc mashup-based data integration applications can simultaneously process and combine heterogeneous data contributed by multiple stakeholders. The data can come from various devices such as sensors, embedded devices, mobile phones, desktops, or web servers. The approach does not require server infrastructure to upload data, but rather allows each stakeholder to keep control over their data and expose only relevant subsets to the collaborative group. Distributed mashups can run persistently in the background and are hence ideal for real-time data monitoring or data streaming use cases.

Preface

Major content of this thesis is taken from our published contributions [1, 2, 3, 4, 5, 6, 7, 8] in three recent years. The author of this thesis is also the lead author of these papers.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

API	Application Programming Interface
CSS	Cascading Style Sheets
CSV	Comma Separated Values
EUD	End-User Development
GWT	Google Web Toolkit
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
ISBN	International Standard Book Number
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
LOD	Linked Open Data
OWL	Web Ontology Language
PDF	Portable Document Format
POI	Point of Interest
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
REST	Representational State Transfer
RSS	Rich Site Summary
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol

SPARQL	Simple Protocol and RDF Query Language
TCM	Tag-based Composition Module
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
W3C	World Wide Web Consortium
WAMP	Web Application Messaging Protocol
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language

List of Symbols

C The configuration model of a widget

EE The external edge set of the graph constructed from a given widget set

E The edge set of the graph constructed from a given widget set

G The graph constructed from a given widget set

IE The internal edge set of the graph constructed from a given widget set

I The finite set of graph-based input models of a widget

M The number of matched output terminals (vertices) for a input terminal (vertex)

N^d The total number of data widgets in a given widget collection

N^p The total number of processing widgets in a given widget collection

N^v The total number of visualization widgets in a given widget collection

N The total number of widgets in a given widget collection

O The graph-based output model of a widget

P The processing function of a widget

R The rate of successful matches between input and output models of all widgets in a given widget collection

V The vertex set of the graph constructed from a given widget set

W A given set of widgets

X The complexity of the input/output model of a widget

w A widget of a given widget set

CHAPTER 1

Introduction

1.1 Motivation

In recent years, organizations and governments have made large volumes of open data [9] available on the web; the data covers a wide range of topics such as economy, currency, geography, entertainment, weather, transportation, etc. Publishers frequently release their data under a license that allows anyone to use, reuse and redistribute it. Such open data, due to its abundant availability, plays an increasingly important role in everyday life and can be used for various purposes [10]. It allows interested stakeholders to analyze the data, put it in a new context, gain insights, and create innovative services.

Open data has the potential to support informed decisions; however, collecting relevant data from various sources and extracting useful information from it has not become easier as the quantity of data made available by organizations and governments has grown. These providers publish their datasets without paying regard to how their data may be used or how it may be combined with data provided by others [11]. Open data is still largely a collection of raw, isolated, and heterogeneous datasets and making effective use of them remains a major challenge [12, 13].

A lot of research has been conducted to facilitate data integration [14] by adding semantics [15, 16] to open data. Linked Data – which refers to "*a set of best practices for publishing and connecting structured data on the Web so that it can be interlinked and become more useful*" [17] – has been adopted and applied by many data publishers [18]. At the very beginning, there were only twelve published datasets, but the so-called Linked Open Data cloud grew rapidly to more than 3,400 datasets with around 85 billion triples by 2015[1]. To be able to manipulate Linked Data, users are required to have knowledge about semantic web technologies as well as the Simple Protocol and RDF Query Language (SPARQL) query language.

[1] `http://stats.lod2.eu/` (accessed Nov. 01, 2015)

Although Linked Data is well-structured and includes millions of links among dispersed datasets, integrating Linked Data is still a challenging issue due to its inherently distributed and inconsistent nature [19]. As of 2015, more than 2,900 vocabularies[2] are used in the Linked Open Data cloud. This shows that many data publishers develop their own vocabularies rather than reusing already existing ones as the best practices would suggest [20]. Linked Data is based on the Uniform Resource Identifier (URI) scheme to globally identify resources by a single identifier; but in reality, many URIs are used for the same concept or entity. Commercial or governmental organizations are unlikely to make use of external URIs from other data sources, because such URIs can be potentially changed or disappear [19].

To sum up, due to ongoing efforts of data publishers and researchers, vast amounts of open data and Linked Open Data are available on the web. The value of this data would dramatically increase if users were able to integrate it. However, they do not know where to find relevant data sources, or they typically do not have the means and skills to collect and process the data. Our motivation is to support non-expert users in obtaining, integrating, and visualizing data from different datasets to gain insights and/or make decisions.

Additionally, we conceive data exploration, integration, and analysis as a collaborative process, as this creates strong potential for both simple ad-hoc data sharing and sophisticated data-driven decision support. Many data integration and analysis tasks are collaborative by nature [21]; they often require the sharing of data held privately by various stakeholders, an – often geographically dispersed – team with a broad skill set, and the agreement on a common interpretation in order to arrive at relevant insights.

1.2 Problem Description

For users to benefit from the value of openly available data, we need to tackle a number of data integration challenges: (i) *Data heterogeneity* makes it difficult to integrate different kinds of data that are in various formats such as Comma Separated Values (CSV), Extensible Markup Language (XML), JavaScript Object Notation (JSON), Resource Description Framework (RDF), or JavaScript Object Notation for Linked Data (JSON-LD) and spread among various storage infrastructures (e.g., databases, files, cloud, personal computers, mobile phones); (ii) Tedious manual data integration processes that users perform to collect, clean, enrich, integrate and visualize data are typically neither *reproducible*, nor *reusable*; (iii) Lack of support for *exploration*, as users often rely on available domain-specific applications that do not allow for the integration of *arbitrary data sources*; (iv) Lack of means for the *identification* of relevant data sources and meaningful ways to *automatically integrate them.*

Gathering data from multiple sources and performing data analysis, integration, and visualization tasks is hence a cumbersome process. End users cannot, yet, tap the full potential of (linked) open data, but rather have to rely on applications built by others.

[2] `http://stats.lod2.eu/vocabularies` (accessed Nov. 01, 2015)

Most existing applications – with the exception of dedicated query and data integration tools – make use of only a single or a limited number of particular open data sources. As a consequence, open data is almost exclusively processed by custom applications tailored to specific use cases or domains and remains inaccessible to end users with more general information needs. This situation is not consistent with the vision of the future web [22, 23], which promised to facilitate easy discovery, sharing, and reuse of highly interconnected datasets across applications.

1.3 Research Questions

The research question addressed in this thesis is:

How can non-expert users be enabled to explore and integrate heterogeneous data sources?

We require that users are able to easily add or remove a data source to facilitate new data integration use cases. To this end, on the one hand, we need a generic and versatile data integration framework that is not tailored towards particular datasets and data formats. On the other hand, the framework must be simple and include appropriate supporting mechanisms to help non-expert users to overcome the technical barriers of complex data integration. Hence, we separate the research question into three sub-questions as follows:

Research Question 1. *How is it possible to support non-expert users in addressing data heterogeneity?*

Research Question 2. *How can non-expert users be enabled to collaboratively integrate data?*

Research Question 3. *How is it possible to automate the data exploration and integration process?*

1.4 Main Contributions

In this thesis we demonstrate that concepts of semantic web and mashups can be combined to facilitate data integration for non-expert users in a flexible and efficient manner. We separate complex data integration tasks into reusable modular functions, which are encapsulated in high-level user interface blocks, i.e., the so-called Linked Widgets. Based on that, users lacking programming skills can visually link widgets to create mashup-based data integration applications. We lift non-semantic data to a semantic level on-the-fly and add explicit semantics to the input and output data of Linked Widgets. Thus we enable users to link disparate data sources, address data heterogeneity, and enrich data from one source with data from other sources to foster new insights.

Semantic web principles allow us to ease and simplify the process of data integration for users; among them are the back-end semantic model of Linked Widgets and mechanisms that leverage the semantics enabling automatic data exploration and integration. We introduce a *matching* algorithm to automatically discover valid connections between

Linked Widgets. This algorithm provides the foundation for the *advanced composition* algorithm that creates meaningful mashups from a given set of Linked Widgets. Finally, we develop a *tag-based composition method* that allows users to compose mashups by specifying them in structured text.

An innovative aspect of our work is the new model of *semantic*, *distributed*, and *collaborative* mashups. There is already a body of work related to semantic and collaborative mashups; however, to the best of our knowledge, there is no research on *mashups* assembled from components that are *distributed* among different nodes (e.g., sensors, embedded devices, mobile phones, desktops, servers) to collect and integrate data. To this end, mashup applications can be composed of both *client* and *server* Linked Widgets. *Client widgets* are executed in the local context of a web browser environment. *Server widgets* can be executed as native applications on various platforms, including personal computers, cloud servers, mobile devices, or embedded systems. *Server widgets* may be used to contribute data from the node they are deployed on to one or multiple mashups. They may also be used to utilize the computing resources of the node to continuously process data in the background. Such architecture allows stakeholders to expose their private data selectively by contributing *server widgets* as functional black boxes. This efficiently facilitates collaborative ad-hoc data integration involving multiple stakeholders that contribute data and computing resources.

We have implemented our concepts in a prototype platform, which is available at `http://linkedwidgets.org`. The data including mashups and semantic models of all widgets is published into the Linked Open Data cloud. It can be accessed via the `http://ogd.ifs.tuwien.ac.at/sparql` SPARQL endpoint.

1.5 Thesis Outline

This thesis is organized as follows:

In Chapter 2, we present the necessary research background on semantic web, open data, Linked Open Data, semantic data integration, and mashups. We then present the state of the art in mashup-based data integration and highlight the research gaps in end-user Linked Data integration.

In Chapter 3, we present the main contribution of our research. We first introduce a modular approach for open data integration that facilitates collaborative work among data publishers, developers, and end-user communities. We then design the architecture of a conceptual mashup-based data integration framework. Next, we present detailed information on *Linked Widgets*, which constitute the basic element of the framework; addressing Research Question 1, Linked Widgets always lift data to a semantic level. We then outline how server and client Linked Widgets interact by means of our *local*, *remote* and *hybrid* protocols. Based on these protocols, we discuss hybrid mashup patterns (i.e., *collaborative*, *persistent*, *distributed*, *streaming*, and *complex* mashups) to address Research Question 2. Finally, we present our mechanisms that leverage the semantic model of Linked Widgets to (i) validate the links between widgets and discover all widgets that can provide data to or consume data from input and output terminals of a widget, (ii) automatically compose mashups from a given set of widgets, and (iii) create auto-composed mashups from structured text. These mechanisms provide the answer to Research Question 3.

Chapter 4 presents the results of our computational experiments on the *terminal matching* and the *automatic mashup composition* algorithms.

In Chapter 5 we illustrate the prototype implementation of the conceptual framework. We discuss the challenging implementation considerations we faced while developing the prototype; these lessons can be useful for developers to build up applications on top of (linked) open data. We then introduce key components of the prototype and demonstrate its capabilities in the context of geospatial data. The prototype allows non-expert users to explore spatial data from various sources (i.e., DBPedia, LinkedGeoData, Geo Names, Wundergound, Nobel Laureate dataset, Flickr, Google Maps, Event Media, and a range of statistical SPARQL endpoints) in heterogeneous formats. By composing a mashup of available Linked Widgets, the users can integrate data in a flexible and effective manner. Finally, we introduce two practical use cases of hybrid mashups.

In Chapter 6 we provide detailed information on a number of mashup tools and frameworks related to our work. With regard to the topic of our research, we categorize the mashup frameworks into widget-based mashups, semantic mashups, embedded mashups, collaborative mashups, automatic mashups, and natural language-supported mashups.

Chapter 7 summarizes the results of this thesis and outlines directions for future research.

CHAPTER 2

Background

In this chapter, we present the background knowledge in the context of our research. We first provide a brief introduction to the semantic web and its potential for data integration in Section 2.1. We then highlight the growth of open data and Linked Open Data in Section 2.3. Next, in Section 2.4, we present the mashup approach to help non-expert users without programming skills to overcome technological barriers and get in touch with these data sources. We outline the state of the art in *end-user mashup-based data integration* by introducing a number of surveys in Section 2.5. Based on that, Section 2.6 discusses the research gap and specifies our approach to bridge the gap between end users and data integration.

2.1 Semantic Web

The web has considerably changed the way we disseminate, access, and retrieve information. The web contains a lot of information, but the original structural information is typically not available; much of the data on the web is delivered in the form of web pages, which are HyperText Markup Language (HTML) documents generated from databases.

Most of the web content today is designed for human consumption. Typically, users make use of keyword-based search engines, such as Yahoo[1], Bing[2], and Google[3] to satisfy their information needs. These keyword-based engines seek keywords in web pages rather than extracting meaning from the web documents themselves. Consequently, the search results have high recall but low precision [16]. Because the results are single web pages, if the required information is spread over multiple documents, users first initiate a number of queries to collect the relevant documents, and then they need to manually extract the partial information and put it together [16].

[1] `https://search.yahoo.com/` (accessed Nov. 01, 2015)

[2] `https://www.bing.com/` (accessed Nov. 01, 2015)

[3] `https://www.google.com` (accessed Nov. 01, 2015)

The semantic web, which is [24] "*the extension of the World Wide Web that enables people to share content beyond the boundaries of applications and websites*", can address these limitations. The term is comprised of techniques that are expected to [16] "*dramatically improve the current World Wide Web and its use*". "*The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation*" [25].

The semantic web provides an easy way to disseminate, locate, access, share, exchange, reuse, aggregate, and integrate information. It is based on standard formats such as RDF[4], Resource Description Framework Schema (RDFS)[5], and Web Ontology Language (OWL)[6] to foster the interchange of data.

2.1.1 RDF

RDF [16] is a graph-based data model for describing resources and their relationships on the web. It is inspired by the linking structure of the Web. It uses URIs (Uniform Resource Identifiers) to name things and their mutual relationships. The data is represented as a set of *triples*, each consisting of a subject, predicate and object in the form of *<subject, predicate, object>*. RDF is commonly described as a directed and labeled graph. The subject is the source of an edge; the predicate is its label; the object is its target.

The subject and predicate of a triple are always URIs, but the object can be either a URI or a literal. URIs that appear as the subject in a triple can also be the object in another triple. A literal is an atomic value such as number, date, string; it can be either "plain" (without type) or "typed" literal, which is defined by using the XML Datatypes[7]. An example of an RDF graph is shown in Figure 2.1. The graph comprises four triples. One of them specifies that "Joe Smith", who is identified by the URI *http://www.example.org/~joe/contact.rdf#joesmith*, is an instance of the *Person* class defined in the *foaf*[8] ontology. The remaining triples define his given name, family name and his homepage. The RDF/XML serialization of the graph is presented in Listing 2.1. RDF/XML is a normative syntax for serializing RDF; however, other serialization formats such as Notation 3 (N3)[9], TURTLE[10], N-TRIPLES[11], and N-QUADS[12] are used as well. These formats are more compacted compared to RDF/XML.

2.1.2 RDFS

RDF describes resources (i.e., subjects, predicates, and objects) with classes, properties, and values. To define particular classes and properties used for an application, we can

[4] `http://www.w3.org/TR/rdf11-primer/` (accessed Nov. 01, 2015)

[5] `http://www.w3.org/TR/rdf-schema/` (accessed Nov. 01, 2015)

[6] `http://www.w3.org/TR/owl-primer/` (accessed Nov. 01, 2015)

[7] `http://www.w3.org/TR/xmlschema-2/` (accessed Nov. 01, 2015)

[8] `http://xmlns.com/foaf/0.1` (accessed Nov. 01, 2015)

[9] `http://www.w3.org/DesignIssues/Notation3.html` (accessed Nov. 01, 2015)

[10] `http://www.w3.org/TR/turtle/` (accessed Nov. 01, 2015)

[11] `http://www.w3.org/2001/sw/RDFCore/ntriples/` (accessed Nov. 01, 2015)

[12] `http://sw.deri.org/2008/07/n-quads/` (accessed Nov. 01, 2015)

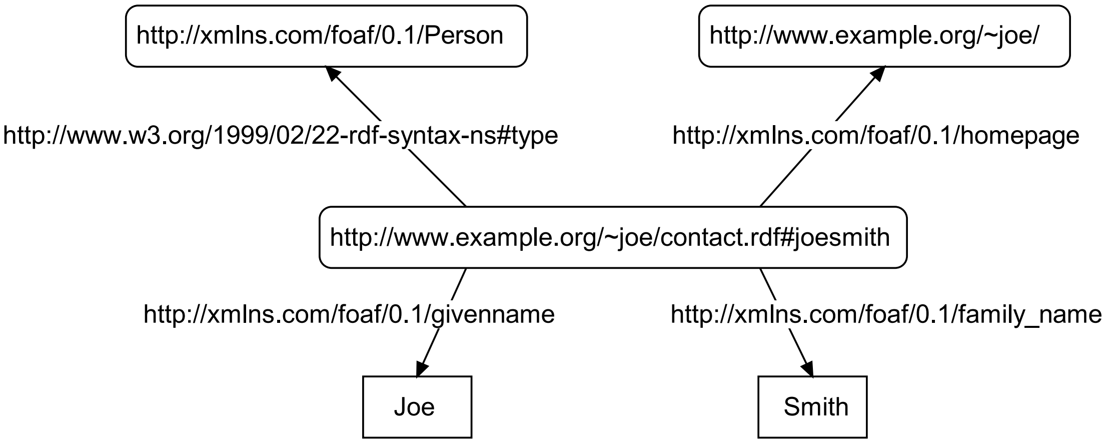

http://xmlns.com/foaf/0.1/Person
http://www.example.org/~joe/
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://xmlns.com/foaf/0.1/homepage
http://www.example.org/~joe/contact.rdf#joesmith
http://xmlns.com/foaf/0.1/givenname
http://xmlns.com/foaf/0.1/family_name
Joe
Smith

Figure 2.1: Example of an RDF graph describing a person

Listing 2.1: The RDF/XML serialization of an RDF graph

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/">
  <foaf:Person rdf:about= "http://www.example.org/~joe/contact.rdf#
     joesmith">
    <foaf:homepage rdf:resource="http://www.example.org/~joe/"/>
    <foaf:family_name>Smith</foaf:family_name>
    <foaf:givenname>Joe</foaf:givenname>
  </foaf:Person>
</rdf:RDF>
```

use RDF Schema (RDFS), which is a semantic extension of the basic RDF vocabulary. RDF Schema does not provide specific classes and properties, but rather the framework to describe these resources. The classes and properties in RDF Schema are similar to the respective concepts in object-oriented programming languages [16].

A class is defined as a sub-class of another class by using the *rdfs:subclassOf* property; a property is defined as a sub-property of another one by using the *rdfs:subpropertyOf* property. *rdfs:domain* and *rdfs:range* are used to define the domain and range of a property, respectively. This allows us to "*specify which properties apply to which kinds of objects and what values they can take, and describe the relationships between objects*" [16].

For example, we can use RDFS to define two classes *Person* and *Dog* as sub-classes of the *Animal* class. The *Animal* class has the *hasChild* property. The *Person* class is linked to the *Dog* class via the *hasDog* property. Three instances of these classes, i.e., a father, a son, and a dog, are then instantiated; the defined vocabulary is used to specify their relationships. The N3 serialization for all of these statements is shown in Listing 2.2.

Listing 2.2: Sample usage of RDFS vocabulary

```
@prefix :     <http://www.example.org/sample.rdfs#> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.

  :Dog      rdfs:subClassOf :Animal.
  :Person   rdfs:subClassOf :Animal.

  :hasChild rdfs:range  :Animal;
            rdfs:domain :Animal.

  :hasDog   rdfs:range  :Dog;
            rdfs:domain :Person.

  :Max      a           :Dog.
  :Abel     a           :Person.
  :Adam     a           :Person;
            :hasSon     :Abel.
            :hasDog     :Max.
```

2.1.3 OWL

The expressivity of RDF and RDFS is very limited [16]; RDF is limited to binary ground predicates while RDFS is limited to class and property hierarchies. OWL is "*a semantic web language designed to represent rich and complex knowledge about things, groups of things, and relations between things*"[13]. It is built on top of RDF; it has a larger vocabulary and stronger syntax than RDF.

OWL comprises three sub-languages, i.e., OWL Lite (which offers simple constraints and a classification hierarchy), OWL DL (which offers maximum expressiveness and retains computational completeness), and OWL Full (which offers maximum expressiveness with free RDF syntax but no computational guarantees). OWL documents are called OWL ontologies; every legal OWL Lite ontology is a legal OWL DL ontology, and every legal OWL DL ontology is a legal OWL Full ontology [16]. OWL 2[14] is the latest version of OWL. It has a similar structure to OWL 1 and adds a number of new features such as (i) richer datatypes, data ranges, (ii) property chains, or (iii) asymmetric, reflexive, and disjoint properties.

Classes in OWL are defined using an *owl:Class* element; *owl:Class* is a sub-class of *rdfs:Class*. There are two predefined classes: (i) *owl:Thing*, which is the most general class and contains everything, and (ii) *owl:Nothing*, which is the empty class. Every class is a super-class of *owl:Nothing* and a sub-class of *owl:Thing*. Equivalence of classes is defined through the use of the *owl:equivalentClass* element. It is possible to define a new

[13] `http://www.w3.org/TR/owl-primer/#Introduction` (accessed Nov. 01, 2015)

[14] `https://www.w3.org/TR/owl2-overview/` (accessed Nov. 01, 2015)

Listing 2.3: Sample SPARQL query

```
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
SELECT ?name ?homepage ?age
WHERE {
  ?x foaf:givenname ?name .
  OPTIONAL {?x foaf:homepage ?homepage.}
  ?x foaf:age ?age
} FILTER (?age < 30)
```

class as the boolean combinations (i.e., union, intersection, complement) of other classes by using the *owl:unionOf*, *owl:intersectionOf*, and *owl:complementOf* elements.

There are two types of properties in OWL: (i) object properties, which relate objects to other objects, and (ii) data type properties, which relate objects to datatype values. *owl:inverseOf* element is used to specify the relationship between a property and its inverse. Equivalence of properties can be defined by an *owl:equivalentProperty* element. *owl:TransitiveProperty* and *owl:SymmetricProperty* are used to specify that an OWL property is transitive and symmetric, respectively; *owl:FunctionalProperty* can be used to define a property that has at most one value for each object.

2.1.4 SPARQL

SPARQL [15] can be used to retrieve and manipulate data stored in RDF format. It is a semantic query language and a protocol that exploits the triple-based structure of RDF to perform graph pattern matching and RDF triple assertion. There are other RDF query languages, such as RDQL[15] (RDF Data Query Language) and SeRQL[16] (Sesame RDF Query Language), but SPARQL is the World Wide Web Consortium (W3C) Recommendation.

SPARQL supports four different query forms: (i) SELECT, which "*returns all, or a subset of, the variables bound in a query pattern match*" (ii) CONSTRUCT, which "*returns an RDF graph constructed by substituting variables in a set of triple templates*" (iii) ASK, which "*returns a boolean indicating whether a query pattern matches or not*", and (iv) DESCRIBE, which "*returns an RDF graph that describes the resources found*".[17] The TURTLE syntax is used to express the RDF graphs in the matching part of the SPARQL query.

Listing 2.3 illustrates an example of a SELECT query. The first line defines a namespace prefix so that the WHERE clause can use the defined prefix to express an RDF graph to be matched. The WHERE clause can include OPTIONAL triples, which are evaluated when they are present, but do not make the matching fail when they are not present. The second line is the SELECT clause, where three identifiers are specified;

[15] http://www.w3.org/Submission/RDQL/ (accessed Nov. 01, 2015)

[16] https://www.w3.org/2001/sw/wiki/SeRQL (accessed Nov. 01, 2015)

[17] http://www.w3.org/TR/rdf-sparql-query/ (accessed Nov. 01, 2015)

each begins with a question mark. The whole query is to get the name, the homepage and the age of all persons in a data source. Constraints can be added for values using a FILTER clause. *FILTER (?age < 30)* is an example of number value restriction to specify that the age of the concerning persons must be less than 30. As an example of string value restriction, *FILTER regex(?name, "Peter")* can be used to return persons whose name contains the string "Peter". The query results can be controlled using the following keywords with the similar meaning to SQL [15]: (i) ORDER BY, which establishes the order of the result, (ii) DISTINCT, which eliminates duplicate results, (iii) OFFSET, which returns the results start after a specified number, and (iv) LIMIT, which puts an upper bound on the number of results returned.

2.2 Semantic Data Integration

In this section, we present a semantic data integration model and an example to illustrate the potential of semantic web for data integration.

2.2.1 Semantic Data Integration Model

Semantic web technologies are useful for integrating heterogeneous data sources; a generic data integration model [26] is illustrated in Figure 2.2. It consists of three main steps: (i) Data in various formats (i.e., XML, JSON, or CSV) is converted into RDF. (ii) Based on the triple graph model, in the second step, a SPARQL engine is used to collect, sort, consolidate, aggregate data and finally convert it to XML format. (iii) In the third step, an Extensible Stylesheet Language (XSLT)[18] engine converts the XML data to HTML and generates the final report or presentation of the integrated data.

In the model, non-RDF datasets are converted into RDF because, due to the triple graph model and the SPARQL language, it is easy to add new nodes to RDF graphs, link the nodes of different graphs to each other, and execute queries over multiple graphs. Thus, disparate sets of RDF data are much easier to combine than disparate sets of data in other common formats. Moreover, as more and more datasets are becoming publicly available in the Linked Open Data cloud (cf. Section 2.3), we can use this external data to enhance our locally stored data and make it richer and more interesting.

A main drawback of the model is that it requires the data integrator to have technical and programming skills. Even though many semi-automatic and automatic RDF and SPARQL tools for data integration have been developed, she should be familiar with these RDF and SPARQL languages. Moreover, despite the simplicity of the model, to practically implement it in a particular use case is not straightforward and takes a lot of time and effort. This approach is hence not applicable for non-expert users that lack technical expertise. To address this issue, as presented in Chapter 3, we modularize and encapsulate the tasks of collecting, processing, integrating, and visualizing data into high-level user interface blocks, i.e., the so-called *Linked Widgets*. By simply connecting

[18] `http://www.w3.org/TR/xslt` (accessed Nov. 01, 2015)

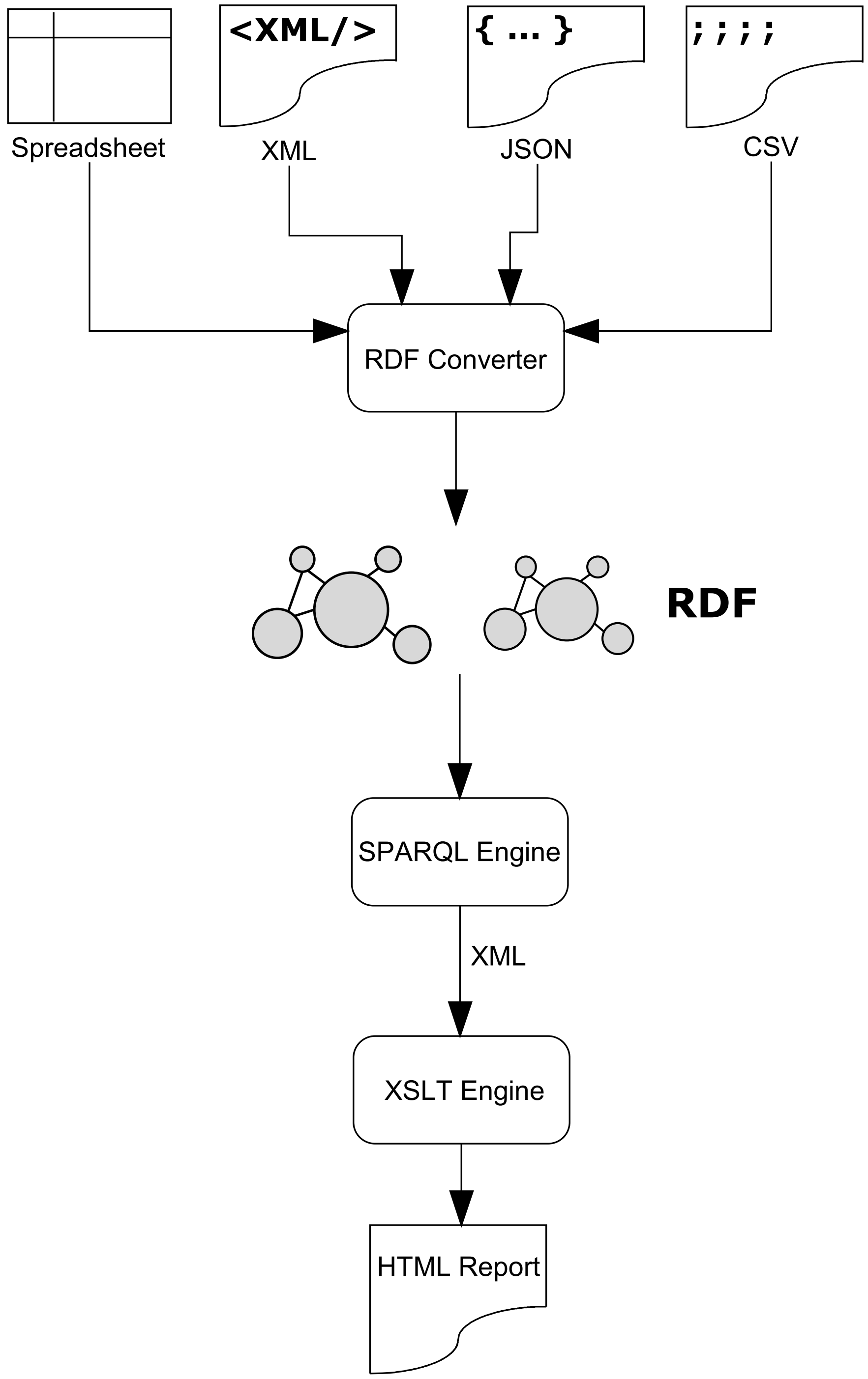


Figure 2.2: Generic semantic data integration model

ID	Author	Title	Publisher	Year
ISBN 0-00-651409-X	id_a1	The Glass Palace	id_p1	2000

Table 2.1: Book table from dataset A

ID	Name	Home page
id_a1	Amitav Ghosh	http://www.amitavghosh.com/

Table 2.2: Author table from dataset A

ID	Publisher Name	City
id_p1	Harper Collins	London

Table 2.3: Publisher table from dataset A

these blocks, end users can create their application without an understanding of the intricate details of queries and transformations executed behind the scenes.

2.2.2 Semantic Data Integration Example

To illustrate the data integration model and the usefulness of the RDF graph triple, consider the following bookstore example, which is adapted from a presentation[19] by Ivan Herman.

In the example, there are two originally tabular datasets: A and B. Dataset A is stored in a MySQL database, consisting of three tables, i.e., *book* table (cf. Table 2.1), *author* table (cf. Table 2.2), and *publisher* table (cf. Table 2.3). This dataset is converted to a set of RDF triples shown in Figure 2.3.

Another bookstore dataset, B, is stored in an Excel spreadsheet. The dataset consists of a detail *book* table and an *author* table (cf. Tables 2.4 and 2.5, respectively). Dataset B does not contain publisher information; it describes the book that is translated from the original book in the dataset A. Dataset B is converted to a triple graph shown in Figure 2.4.

Ideally, the A and B's converted graphs should use the same vocabularies that are commonly used by the semantic web community; to this end, the Linked Open Vocabulary web site[20] lists and visualizes the popularity of more than 500 vocabularies. However, in our example, the two graphs use their own vocabularies. In spite of such inconsistency, it is still easy to integrate two datasets by specifying the *owl:sameAs* relationship between two

[19] `http://www.w3.org/People/Ivan/CorePresentations/IntroThroughExample/` (accessed Nov. 01, 2015)

[20] `http://lov.okfn.org/dataset/lov/` (accessed Nov. 01, 2015)

ID	**Titre**[1]	**Auteur**[2]	**Traducteur**[3]	**Original**
ISBN 2020386682	Le Palais des miroirs[4]	i_abc	i_rst	ISBN 0-00-651409-X

[1] Title

[2] Author

[3] Translator

[4] The Glass Palace

Table 2.4: Book table from dataset B

ID	**Name**
i_abc	Amitav Ghosh
i_rst	Christiane Besse

Table 2.5: Author table from dataset B

identical resources (i.e., properties and instances); the identical nodes (which represent identical instances) in the two graphs are merged into a single node; the merged node inherits all links of the two nodes.

Figure 2.5 shows a triple graph, which is the integration of dataset A, B, and DBpedia. To construct the graph, the original book is first linked to the translated one based on their International Standard Book Number (ISBN) number. The relations between the subjects and objects of our triples "act as glue" to combine separate data sources. The *owl:sameAs* property can be used to specify that the *a:author* property is the same as the *b:auteur* property. Finally, extra information on the authors from the DBpedia SPARQL endpoint[21] is collected and added into the integrated graph.

After the two datasets are integrated, users of dataset B can ask queries as "*What is the title of the original book?*", or "*What is the home page of the original author?*". Such information is not available in dataset B, but can be automatically retrieved by being merged with dataset A. The queries are performed on the relational graph, using the SPARQL language.

To conclude this section, different datasets that (i) are disseminated on the web, (ii) are in different formats (database dump, spreadsheet, etc.), and (iii) use different vocabularies can be combined. The identical URIs (the ISBN in our case) are used to combine the data. By transforming data into RDF graphs and linking them with Linked Open Data (LOD) datasets such as DBpedia, Geonames[22], Freebase[23], etc., completely new ways to execute various queries over a number of data repositories become possible.

[21] `http://dbpedia.org/sparql` (accessed Nov. 01, 2015)

[22] `http://www.geonames.org/` (accessed Nov. 01, 2015)

[23] `http://www.freebase.com/` (accessed Nov. 01, 2015)

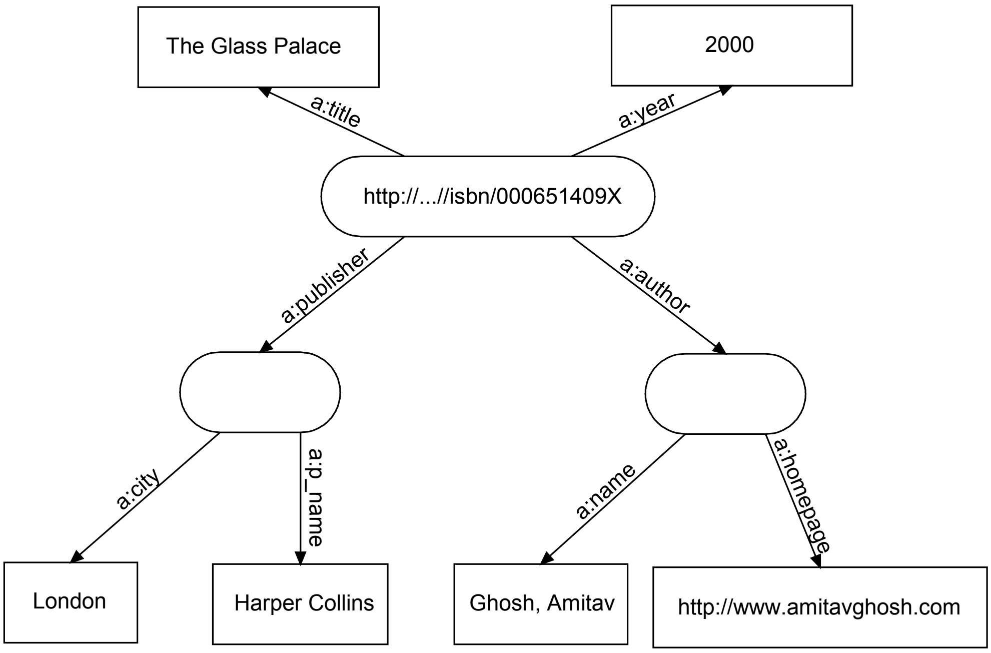


Figure 2.3: Converted triple graph of dataset A

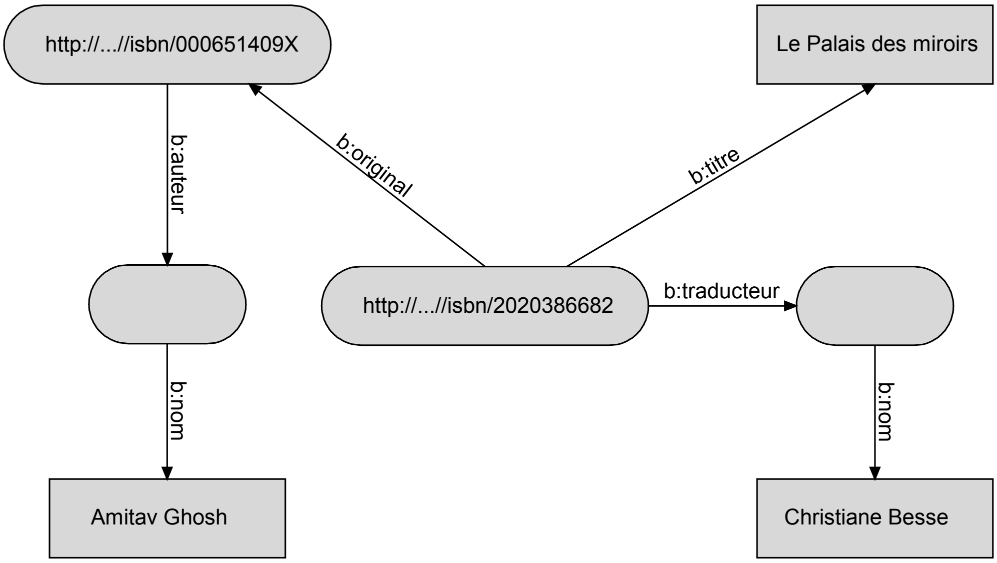


Figure 2.4: Converted triple graph of dataset B

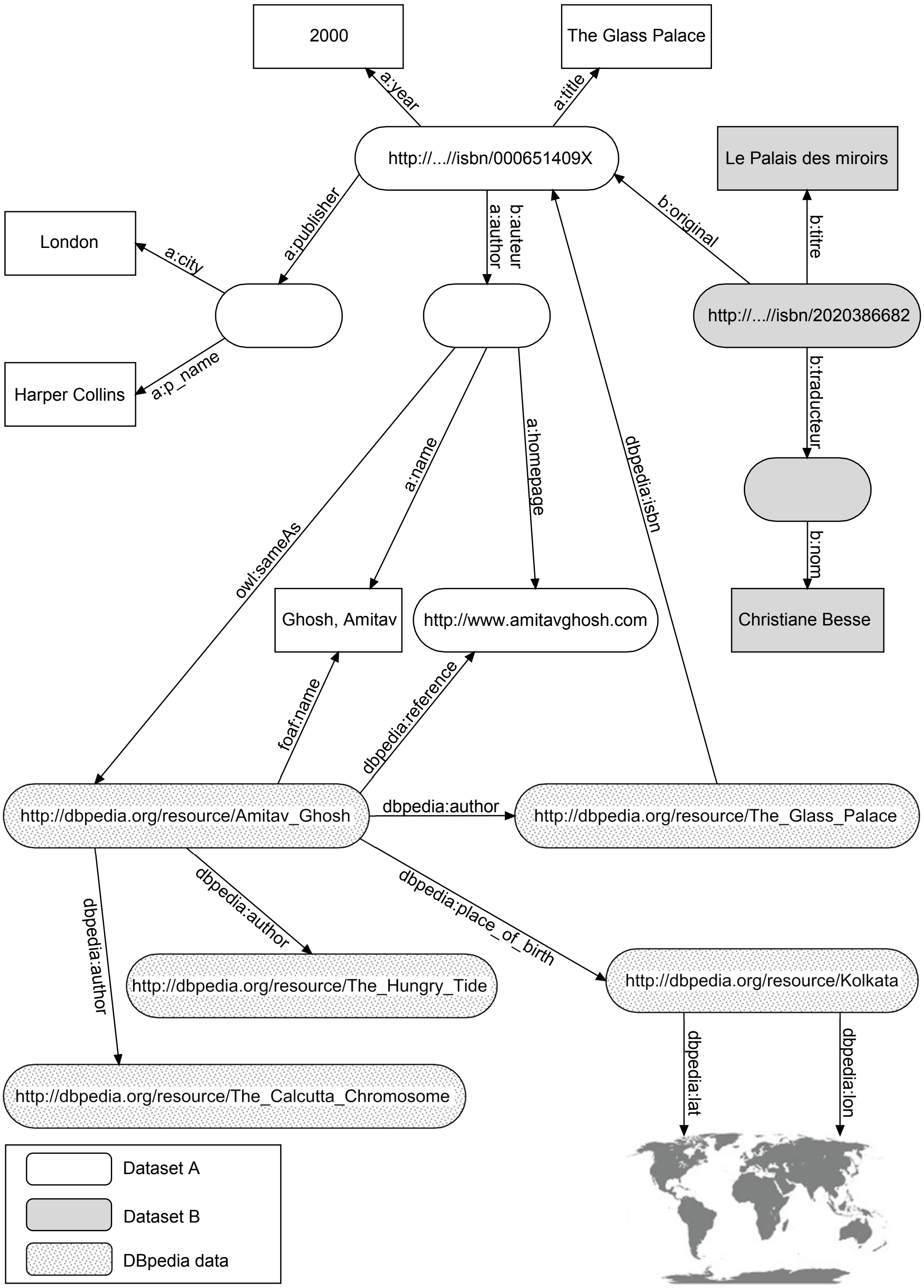


Figure 2.5: Integrated graph of datasets A and B

2.3 Open Data and Linked Data

In recent years, organizations and governments have made large volumes of Open Data available on the web. This allows interested stakeholders to analyze the data, put it in a new context, gain insights, and create innovative services. Publishers frequently release the data under a license that allows anyone to use, reuse and redistribute it.

The inventor of the web and also the initiator of Linked Data, Tim Berners-Lee, suggested a *5 star deployment scheme* for Open Data.[24]

1. The first level (which is rated one star) is to make data available on the web under an open license.
2. The second level is to make data available as structured data.
3. The third level is to use non-proprietary formats (e.g., CSV instead of Excel).
4. The forth level is to make use of URIs to denote things, so that people or applications can dereference and get more information.
5. The fifth level is to link the data to other data sources to provide context.

Linked Data is a set of guidelines for publishing structured data. The metadata is connected and enriched so that applications can find different representations of the same content stored in multiple sources, and finally, establish links between related resources. It is fundamentally about helping the web to transition from a web of documents to a data web. An increasing number of data publishers (e.g., The New York Times[25], the BBC[26], Thomson Reuters[27], the Library of Congress[28], BestBuy[29], Getty[30], the UK government[31], the US government[32]) have adopted Linked Data practices and published a large number of datasets.

According to the formal definition of Linked Data [17], the term "*describes a method of publishing structured data so that it can be interlinked and become more useful. It builds upon standard Web technologies such as Hypertext Transfer Protocol (HTTP) and URIs, but rather than using them to serve web pages for human readers, it extends them to share information in a way that can be read automatical ly by computers. This enables data from different sources to be connected and queried.*"

The publication of Linked Data on the web is based on the following four principles [17]:

1. *"Use URIs as names for things*
2. *Use HTTP URIs so that people can look up those names.*
3. *When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).*

[24] `http://5stardata.info/` (accessed Nov. 01, 2015)

[25] `http://data.nytimes.com/` (accessed Nov. 01, 2015)

[26] `http://www.bbc.co.uk/things/` (accessed Nov. 01, 2015)

[27] `https://permid.org/` (accessed Nov. 01, 2015)

[28] `http://id.loc.gov/` (accessed Nov. 01, 2015)

[29] `https://developer.bestbuy.com/` (accessed Nov. 01, 2015)

[30] `http://www.getty.edu/research/tools/vocabularies/lod/` (accessed Nov. 01, 2015)

[31] `https://data.gov.uk/` (accessed Nov. 01, 2015)

[32] `https://usopendata.org/` (accessed Nov. 01, 2015)

4. *Include links to other URIs, so that they can discover more things"*.

Due to simple principles and basic technologies, Linked Data has been adopted and applied in the LOD project – a community project supported by the W3C – to facilitate the publication of open datasets as RDF. As a result of ongoing efforts, the "*Web of Linked Data*" today comprises of billions of RDF triples and millions of RDF links between datasets. More specifically, at the very beginning, there were only twelve published datasets, but the so-called LOD cloud grew rapidly to 928 datasets with 62 billion triples by 2014[33]. The current statistics[34] are illustrated in the well-known Linked Open Data cloud diagram shown in Figure 2.6. Using the CKAN Application Programming Interface (API)[35], the diagram is generated automatically. The size of a circle corresponds to the number of triples in the respective dataset. If at least 50 links exist between two datasets, there is an arrow; its thickness corresponds to the total number of links.

The datasets published in the LOD cloud now include data on various topics (e.g., music, movies, radio and television programs, books, scientific publications, reviews, proteins, diseases, genes, medicine and clinical trials, people, geographic locations, statistical and census data, companies, and many more other topics). All of the datasets are publicly and openly available on the Web in standard and interoperable formats. This opens up novel opportunities for the next generation of web-based applications; we can aggregate data from different providers as well as integrate fragmentary information from separate sources and hence achieve a complementary and more complete view.

2.4 Mashups

The term "*mashup*" originally comes from the music industry [27]; it is a new music song that is made up from the vocal and instrumental tracks of existing songs.

Analogously, a mashup (application) is a lightweight web-based application that combines data from multiple sources into an integrated and single graphical interface [28]. Mashups are typically based on the web architecture (i.e., HTTP, Representational State Transfer (REST)) and web 2.0 technologies (e.g., AJAX, Rich Site Summary (RSS), JSON). It allows for the fast creation of rich web applications.

There are many other extended definitions of mashups. Daniel and Matera [29] state that mashups must not obligatorily live on the web environment and defined mashups as: "*a composite application developed starting from reusable data, application logic, and/or user interfaces typically, but not mandatorily sourced from the Web*". Yee [30], on the other hand, focuses on the data integration as the key value of mashups, and defines mashups as web sites or web applications that "*seamlessly combine content from more than one source into an integrated experience*". Situational applications, rapid development, and the focus on end users are considered as central concepts of mashups as well [31].

[33] `http://stats.lod2.eu` (accessed Nov. 01, 2015)

[34] `http://lod-cloud.net/` (accessed Nov. 01, 2015)

[35] `http://docs.ckan.org/en/latest/api/` (accessed Nov. 01, 2015)

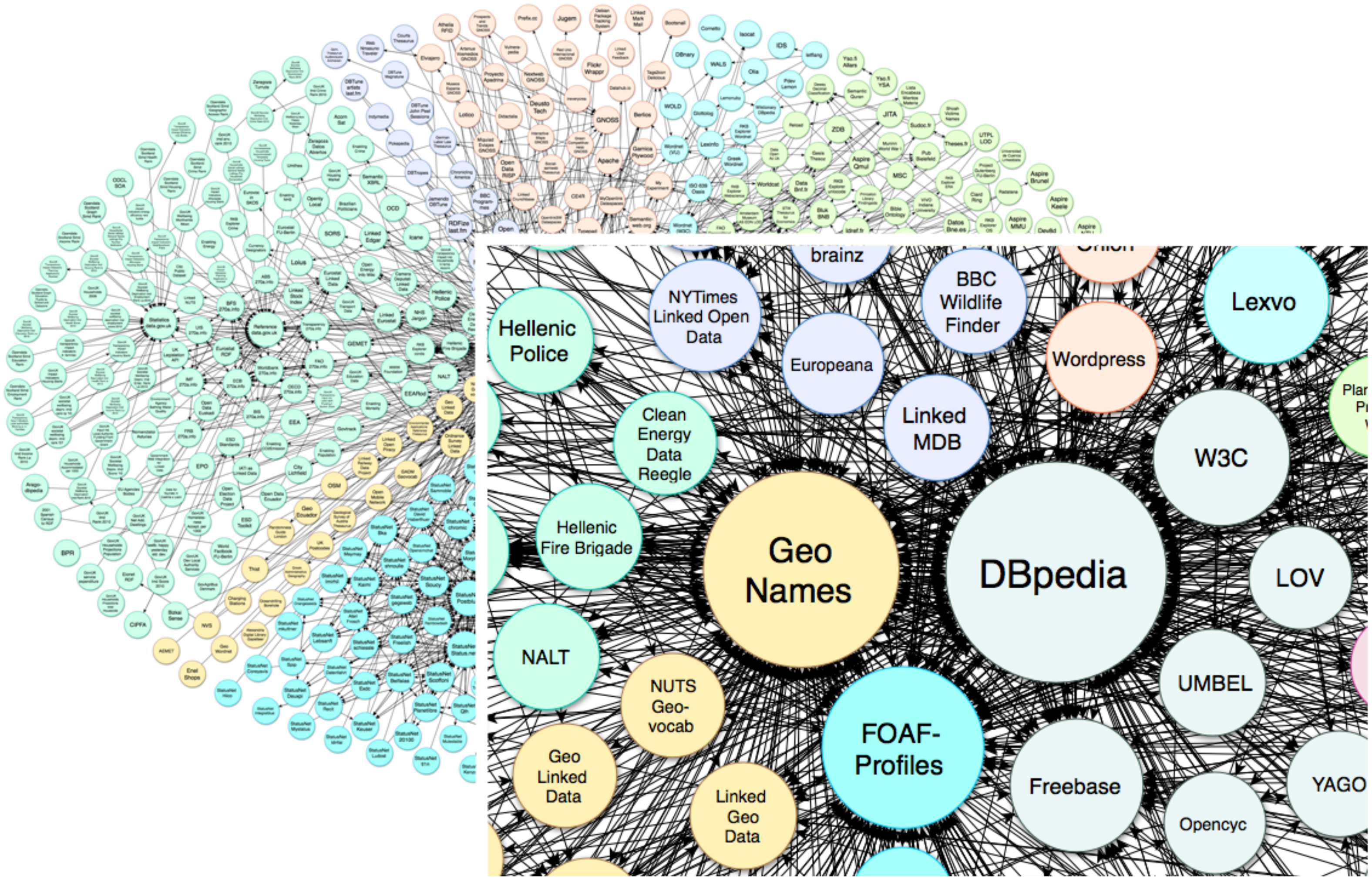


Figure 2.6: LOD cloud as of 2014[a]

[a] Linking Open Data cloud diagram, by Max Schmachtenberg, Christian Bizer, Anja Jentzsch and Richard Cyganiak. http://lod-cloud.net

The term "*mashup*" includes both *data mashups* and *presentation mashups* [29]. An example of presentation-focused mashup is to view customer data[36], stock prices[37], and latest news[38] in a singly integrated interface. Data mashups focus on accessing, transforming and combining different data sources.

Mashups are different from web services composition. While the latter focuses on the composition of business services, the former goes further in that it provides more functionalities and can compose a wide range of heterogeneous resources such as data services, or user interface services [29].

Architecturally, a web-based mashup application is comprised of three elements: (i) content providers, (ii) a web page, and (iii) the client's web browser. Many content providers expose their data through various web protocols (e.g., REST, web services, RSS/Atom, and other web APIs) to facilitate data retrieval. After collecting and mashing the available content from different providers, the mashup is hosted in a web page and provides a new integrated service. Finally, the client's web browser is the consumer of the mashup; it renders the mashup application and allows users to interact with.

Mashups "*come in all shapes and sizes, from the very simple to the complex*" [28]; some combine search results from different search engines, some aggregate tabular data from different sources and display in visual charts, and others integrate geographic data with social news feeds (e.g., Foursquare[39], Twitter[40]) and present all information in a map. Mashup usage is growing rapidly on the web; this can be attributed to the following main reasons [29]:

1. Many influential internet companies including Google, Yahoo, Facebook, Twitter, and Amazon have opened up parts of their data to be combined with other data sources without strict restrictions.
2. Many new tools (e.g., Damia [32], Yahoo! Pipes [33], Microsoft Popfly [34], Google Mashup Editor [35], Exhibit [36], Apatar[41], MashMaker [37], Marmite [38], Super Stream Collider [39], DERI Pipes [40], MashQL [41], Information Workbench[42], ResEval Mash [42], Mashroom [43], Husky[43], Intel Mash Maker [44], Vegemite [45], Presto[44], and mashArt [46]) – which strive to enable users to easily create various types of mashups regardless of their lack of technical knowledge – have been introduced in recent years.

The IT departments of many enterprises and organizations are starting to make use of mashups as a quick, simple, and cost-effective solution to their data integration issues [47]. No big investment is required due to the common technologies used in mashups;

[36] `http://www.programmableweb.com/category/customer-service` (accessed Nov. 01, 2015)

[37] `http://www.programmableweb.com/category/stocks/mashup` (accessed Nov. 01, 2015)

[38] `http://www.programmableweb.com/category/news-services` (accessed Nov. 01, 2015)

[39] `https://foursquare.com/` (accessed Nov. 01, 2015)

[40] `https://twitter.com` (accessed Nov. 01, 2015)

[41] `http://www.apatar.com/` (accessed Nov. 01, 2015)

[42] `http://www.fluidops.com/en/portfolio/information_workbench/` (accessed Nov. 01, 2015)

[43] `http://www.husky.fer.hr/` (accessed Nov. 01, 2015)

[44] `http://mdc.jackbe.com/enterprise-mashup` (accessed Nov. 01, 2015)

mashups are designed to reduce the cost and development time of web applications. By leveraging existing APIs, web services, and visualization technologies, the development time of mashups can be measured in hours or days, rather than in weeks or months [47].

Programmable Web[45] is a repository that provides latest information about internet-based APIs and mashups. Figure 2.7 shows the growth of top ten web API categories in Programmable Web since 2009. *Mapping* is now the most popular category with more than four thousands available APIs.

Mashups can add new value to available data and APIs in an ad-hoc manner; however, there are challenging issues as follows: (i) First, mashups heavily rely on one or multiple third parties; if the API providers stop their services, the mashup would be immediately unavailable as well. (ii) To use a web service of a company, developers always have to check the "*terms of service*". Many providers charge usage fees based on the number of calls made to their server; most of them require developers to ask for permission before using the APIs for the commercial purposes. (iii) Because mashups can add new value to data, new issues such as "*who owns the integrated data*" and "*who can profit from the mashup*" arise.

2.4.1 Mashup Types

Daniel and Matera [29] propose a cube-based mashup classification as illustrated in Figure 2.8. Three perspectives are: *composition*, *domain*, and *environment*.

1. The *composition* perspective classifies mashups based on the layer where the mashup integration is performed; in the typical three-tier architecture, applications are divided and developed in three separate layers, i.e., a data layer, a logic layer, and a presentation layer [48]. This development model makes it easier to implement and maintain different parts of the applications, as these parts are independent but capable of interoperability. Possible types of mashups in this perspective are *data mashups*, *logic mashups*, *User Interface (UI) mashups*, and *hybrid mashups* that span multiple layers of the application stack.
2. The *domain* perspective classifies mashups based on the functionalities that the mashups provide, or in other words, the purpose of the mashups. Various categorized domains of mashups are: *mapping*, *social*, *tools*, *mobile*, *photos*, *financial*, etc.[46]
3. The *environment* perspective classifies mashups based on their deployment context. *Web mashups* and *enterprise mashups* are identified as the two types of mashups in this perspective. *Web mashups* (which is also known as *consumer mashups*) are for internet users. They focus on the functionalities rather than non-functional requirements (e.g., security, compliance with laws and regulations) imposed in the enterprise environment.

[45] http://www.programmableweb.com/ (accessed Nov. 01, 2015)

[46] http://www.programmableweb.com/category (accessed Nov. 01, 2015)

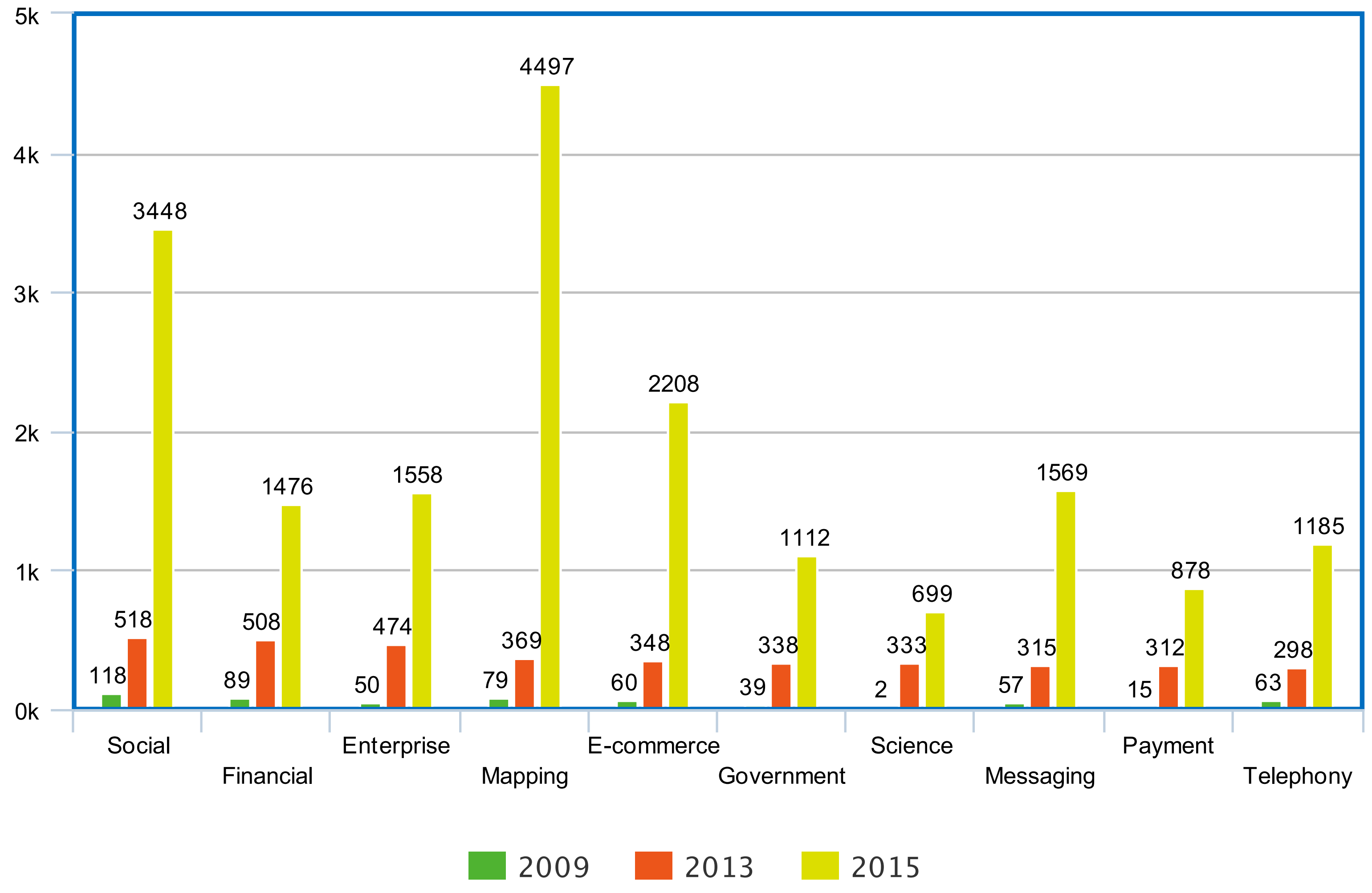


Figure 2.7: Growth of top ten web API categories in Programmable Web since 2009[a]

[a] `http://www.programmableweb.com/api-research` (accessed Nov. 01, 2015)

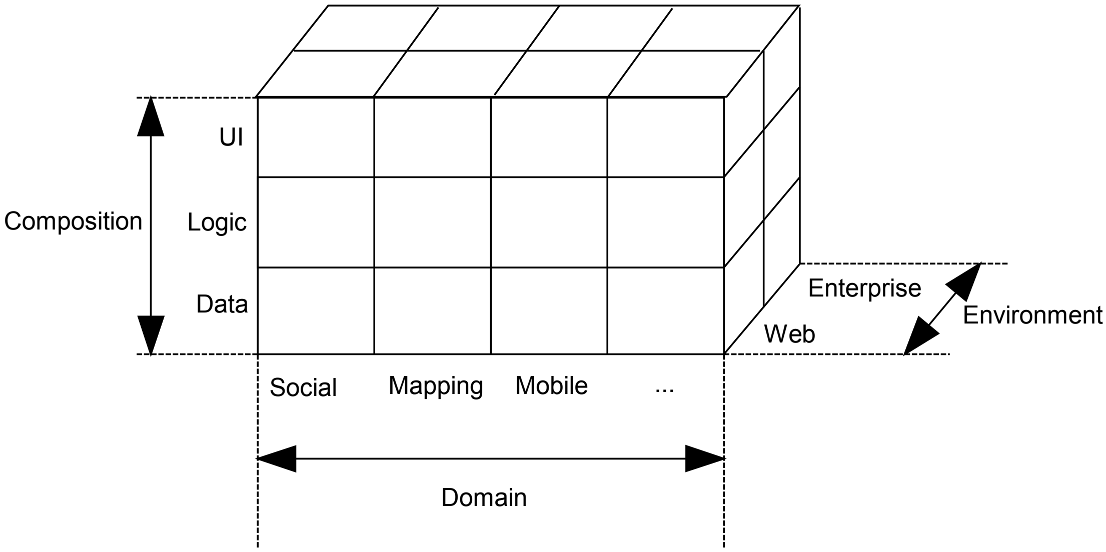


Figure 2.8: Mashup classification dimensions [29]

2.4.2 Mashup Components

A mashup component is the atomic part that mashups are composed of. Daniel and Matera [29] define that "*A mashup component is any piece of data, application logic, and/or user interface that can be reused and that is accessible either locally or remotely*".

Components can be built on top of very different technologies, from Simple Object Access Protocol (SOAP) web services and RESTful services to web APIs, and from RSS feeds to UI widgets. Mashup components are very heterogeneous with regard to their behaviors within a mashup (e.g., sequential, asynchronous, or synchronous interactions) as well as the access methods through which they are published (e.g., protocols). In order to lower the barriers for general users by automating mashup development, we need an abstract model of mashup components that can address such heterogeneity.

Mashup components can be created by wrapping existing resources. A wrapper can be simple and easy to implement if the resource provider already provides all necessary functionalities. The wrapper can be more complex if the resource provider exposes data only and developers themselves have to process and visualize the data.

Components can be classified into three categories that cover the three layers of an application: (i) *data components* which provide access to data, (ii) *logic components* which provide access to business logic or functionalities, (iii) and *user interface components* which present and visualize the data.

2.4.3 Web Widgets

A web widget is a piece of dynamic content that can be easily embedded into a web page. Web widgets can encapsulate one or more APIs. An example of a web widget is shown

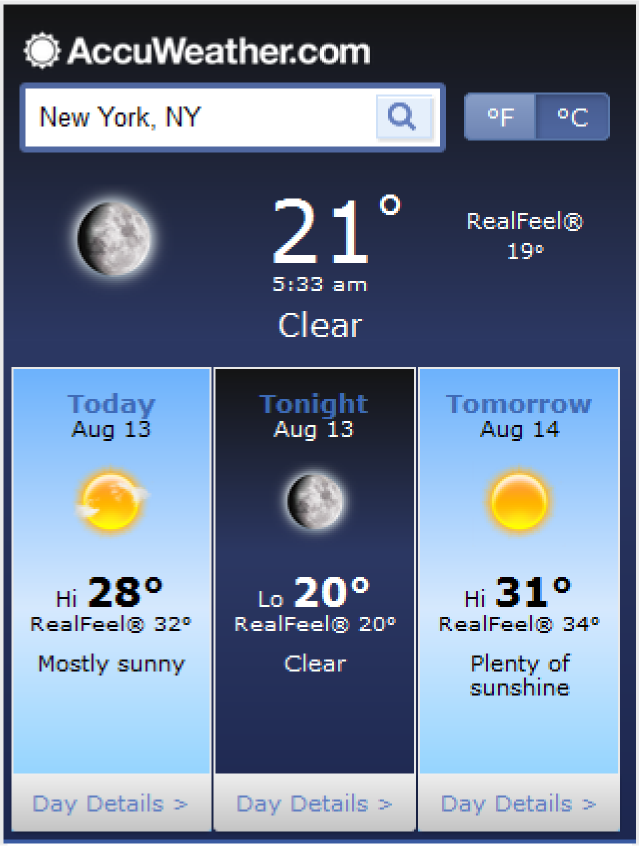


Figure 2.9: Example of the AccuWeather web widget

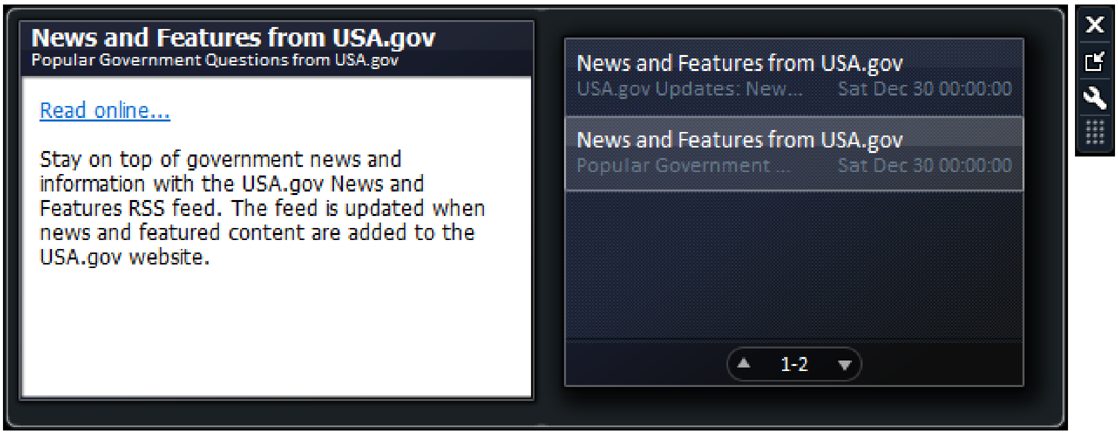
in Figure 2.9. Such web widgets can be placed in a blog to provide readers packaged and intuitive information on common topics (e.g., weather information, blog's statistics, recent news, or stock quotes) in a single interface.

Different vendors use different terms for widgets: badges, gadgets, flakes, snippets, minis, blocks, modules, and capsules [49]. Widgets can be as simple as a single HTML fragment; they can be more complex if they are written in a server programming language (e.g., Java, .NET, PHP). Some of them are "*mashable*" widgets, which can pass and receive events, so that they can be combined to compose a mashup. Running on the web browser environment, web widgets are independent of operating systems. There are many catalogs of online widgets such as Deitel[47], and Widgipedia[48], which provide an extensive catalog of widgets and gadgets for a variety of platforms.

W3C defines web widgets as "*end-user's conceptualization of an interactive single purpose application for displaying and/or updating local data or data on the web, packaged in a way to allow a single download and installation on a user's machine or mobile*

[47] `http://www.deitel.com/ResourceCenters/Web20/Widgets/tabid/1993/Default.aspx` (accessed Nov. 01, 2015)

[48] `http://www.widgipedia.com/` (accessed Nov. 01, 2015)



News and Features from USA.gov
Popular Government Questions from USA.gov

Read online...

Stay on top of government news and information with the USA.gov News and Features RSS feed. The feed is updated when news and featured content are added to the USA.gov website.

News and Features from USA.gov
USA.gov Updates: New... Sat Dec 30 00:00:00

News and Features from USA.gov
Popular Government ... Sat Dec 30 00:00:00

Figure 2.10: Example desktop widget (or "gadget") from Window OS 7

device. A widget may run as a standalone application (meaning it can run outside of a web browser), or may be embedded into a web document"[49]. W3C also standardizes the packaging format and configuration for web widgets.[50]

Web widgets are different from desktop or mobile widgets. The latter runs as a native application while the former must be rendered within a web browser. Desktop and mobile widgets cannot only present useful information at a glance but also allow for quick access to frequently used tools or applications. For instance, we can use desktop widgets to automatically display a slide show of pictures in our computer, or check continuously updated headlines (cf. Figure 2.10).

Web widgets are used as mashup components in many frameworks (cf. Section 6.1). They typically respond to or act upon data events and/or user interactions. They foster reusability because a single widget can be reused in multiple mashups; each mashup is one of numerous possible combinations of widgets.

2.4.4 Semantic Mashups

It is not easy to build up mashups without using any supporting tool or service, because developers need to locate relevant data sources and APIs, understand their structure as well as their semantics, and finally combine all components into a single interface. Semantic mashups can be used to ease such effort, and foster API discovery and integration.

A semantic mashup is "*a data mashup using RDF(S) as data model and SPARQL services to implement the behavior*" [50]. It is the combination of the two visions of future web, i.e., the semantic web and web 2.0 [51]. On the one hand, the vision of semantic web is to give information well-defined meaning and "*better enable computers and people to work in cooperation*" [25]. As illustrated in Section 2.2, if data is represented in semantic format, we can use SPARQL to query over the triple graph models and easily integrate

[49] `http://dev.w3.org/2006/waf/widgets-land/` (accessed Nov. 01, 2015)

[50] `http://www.w3.org/TR/widgets/` (accessed Nov. 01, 2015)

multiple data sources [52, 53]. On the other hand, the vision of web 2.0 is to turn users from "content consumers" into "content providers", which can be partly fulfilled via the mashup concepts and technologies [54, 55]. Research that focuses on semantic mashups will be presented in Section 6.2 as our related work.

2.4.5 End-User Mashup Development

In the traditional software development process [56], we typically start with specifying user requirements. Based on the requirements, we design and implement the overall system, and finally we receive the users' feedback to modify, maintain, and upgrade the software. These steps can be performed differently in a variety of software models (e.g., Waterfall, Incremental, Spiral, or Agile model).

It is not easy to develop software that satisfies all user requirements. First, we can hardly identify user requirements precisely as they are so diverse and complex. Many users work on tasks that rapidly change on a monthly, weekly, or daily basis. Moreover, it is difficult to meet all requirements when developers have limited domain knowledge [57].

To address these issues, an end-user approach in developing application software aims to emancipate end users from the dependency on custom applications tailored to specific use cases or domains. The approach is based on *user-driven innovation*, where a service provider equips users with innovation toolkits so that users themselves can build a small software product for their personal and situational purpose. Making use of these toolkits, users are capable of carrying out the entire development process to realize their own ideas with limited up-front learning time investment [58]. They themselves can also be able to continuously adapt and upgrade the systems to their needs.

Lieberman et al. [57] define End-User Development (EUD) as a set of "*methods, techniques, and tools to allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artifact*". In particular, based on the traditional Human-Computer Interaction principles [59], EUD particularly focuses on the capability of systems to empower users to create their applications, or at least allow them to design and customize the functionality as well as the user interface of software. This is necessary because end users themselves know their context and requirements better than anyone else. An EUD system should be flexible to support users in performing tasks to develop their own software. It should be easy to understand, to use, to learn, and to teach.

By providing simple controls for people without formal programming training, EUD shows considerable potential to make computers more useful in various contexts. It can effectively reduce the time to complete a project [57]. EUD software system is now becoming more popular, powerful and easier to use. The most widely adopted EUD tool is the spreadsheet that can efficiently manipulate tabular data. Spreadsheet is most commonly used in a business context; many companies require their employees to have good spreadsheet skills.

End-User Mashup Development is a branch of End-User Development; its flexibility and task variability are promising for users to create innovative solutions. From the EUD perspective, a mashup appears as an "*end user driven recombination of web-based* *data and functionality*" [60]. Due to the growing number of publicly available services and APIs on the web, to quickly develop a new software, users can locate relevant high-level services, "glue" them together, and create web user interfaces rather than start from scratch as in the traditional programming model. The former method involves less technical activities than the latter, and can be designed to fit with the restricted capabilities of end users. Furthermore, because mashups are based on web technologies, they can facilitate sharing.

Cappiello et al. [61] identify two scenarios where end users are involved in developing mashups:

1. To quickly deliver ad-hoc applications, *mashup templates* (which are frequently requested applications) first are created by expert developers. This means end users are not directly involved in the mashup development process; however, they can modify mashup parameters or adapt the mashup to their context. This approach is not flexible; there is not much room for users' creativity.
2. Expert developers implement and deploy a framework that allows anybody to create their own mashups. In this scenario, end users are directly involved in the creation of their mashups. Exemplified approaches are spreadsheets, programming by demonstration, and visual programming.

2.5 State of the Art in Mashup-based Data Integration

To facilitate data integration, researchers have been developing mashup-based tools and frameworks for years. Many of them are geared towards end users and allow them to efficiently create applications by connecting simple and lightweight components. This section presents various mashup surveys from different perspectives.

Fischer et al. [62] introduce three mashup development approaches, which are manual, semi-automatic and automatic mashup creation. They categorize mashup frameworks as follows:

1. programming paradigm (e.g., IBM WebSphere sMash[51] and BungeeConnect[52]),
2. scripting languages (e.g., Web Mashup Scripting Language (WMSL) [63], Dynamic Fusion of Web Data [64]),
3. spreadsheets (e.g., Extensio Excel Extender[53]),
4. wiring paradigm (e.g., Yahoo! Pipes [33], Marmite [38])
5. programming by demonstration (e.g., Karma [65] and Potluck mashup tool [66]),
6. and automatic creation of mashups (e.g., Composite Application Framework [67] and Service Composition Framework [68]).

The authors identify three user types of such tools; they are *casual users*, *power users* and *developers*. A *casual user* has basic skills; she is able to surf the web and uses the functionality of the browser only. A *power user* cannot program, but has detailed

[51] `http://www.ibm.com/developerworks/ibm/zero/` (accessed Nov. 01, 2015)

[52] `http://www.bungeelabs.com/` (accessed Nov. 01, 2015)

[53] `http://www.extensio.com/index.html` (accessed Nov. 01, 2015)

knowledge of a particular tool or a set of tools. By contrast, a *developer* is skilled at programming and can work directly with web technologies and web APIs.

The authors give an overview of the introduced tools in the six categories and identify their limitations. The "*automatic creation of mashups*" is ideal for non-programmers; however, its problem lies in the high risk of composing irrelevant mashups based on given requirements. By contrast, the others create a barrier to *casual* and *power* users because they do not possess the necessary skills required by the respective tools to create mashups.

Grammel and Storey [60] analyze six mashup tools (i.e., Microsoft Popfly [34], Yahoo! Pipes [33], IBM Mashup Center[54], Google Mashup Editor [35], Serena Mashup Composer[55], Intel Mash Maker [44]) from the end user development perspective. The features of the tools are compared across six criteria, i.e., *levels of abstraction*, *learning support*, *community support*, *searchability*, *UI design*, and *software engineering techniques.*

The authors define three levels of abstraction – which are high, intermediate and low levels – to measure how much computer programming knowledge is required. On the high level, it is easy for users to learn and use the tool, but they are restricted to parameterize and reuse mashups. By contrast, the low level offers users great flexibility. The authors hence suggest to combine different levels of abstractions in a tool because it allows users to select the relevant level to their skill and goal. This fosters reusability of both low and high level software components.

Learning support helps users to get used to the mashup environment of the tool. Common means are a help system, a tutorial, or a video screen capture. The authors discuss the possibility to use context-specific suggestion feature. For example, it can recommend users suitable mashup components based on their current mashup structure.

The authors note that community features were essential to the success of an end-user-development mashup tool. These features include capabilities to share mashups, and collaboratively categorize mashups (e.g., tagging, and rating). A discussion forum and a social network system are proposed as a board to exchange the user experience and their created mashups.

Text-based search is a common method to find and locate mashups or mashup elements. Users can sort the result based on the rating of these artifacts. The authors suggest using the structural information of a mashup. It helps users to easily find example mashups that use one or several specified mashup components.

UI design is the feature to support users in creating appropriate UIs for their mashups. It determines the usability of the mashups. The authors outline three mechanisms, i.e., *automatically generating the UI*, *selecting and customizing the UI*, and *UI composition.*

Finally, software engineering techniques consist of debugging, version control, and testing. Even though the support for these techniques is very important in the EUD research [69], it is still limited. Mashups can introduce new security threats, especially in the enterprise environment. The debugging support can help users to avoid incorrect

[54] `www.ibm.com/software/info/mashup-center/` (accessed Nov. 01, 2015)

[55] `http://www.serena.com/index.php/en/` (accessed Nov. 01, 2015)

and unsafe mashups.

Hendrik et al. [70] do a survey of semantic mashup frameworks from the perspective of knowledge workers, who usually carry out data analyses but are not familiar with technical details of data integration and big data. They consider semantic mashups as the extension of traditional mashup approaches, where semantic web means play their role to facilitate building mashups for novice users. Similar applications of semantic web can be seen in the semantic service approach, which can automate service discovery and service composition. Semantic web services target IT experts. By contrast, semantic mashups are intended for non-expert users, who are incapable of working with raw data and services, but still want to create situational applications on top of existing resources.

The authors define three requirements that a mashup tool should fulfill. An ideal mashup tool should be general, powerful but simple; it should cover various application domains and be able to tackle the complex logic of arbitrary problems, but should still be easy for novice users. Based on these requirements, they classify Black Swan Events [71] and Super Stream Collider [39] as data analytic tools, and DERI Pipes [40], MashQL [41], and Information Workbench[56] as generic tools. Whereas data analytic tools focus on specific domain and make use of their own data processing patterns, generic tools offer generic functions to be used in various problem solving.

Aghaee and Pautasso [72] provide a detailed overview of mashup approaches. Similar to other surveys, based on the end-user programming technique that existing mashup tools use, the authors classify them into spreadsheets (e.g., Mashroom [43], Husky[57]), programming by demonstration (Intel Mash Maker [44], Vegemite [45]), domain-specific language (e.g., Swashup [73]), visual programming (e.g., Yahoo! Pipes [33], Presto[58]), and model-based automation [67, 74].

To enable end users to compose and develop mashups in a lightweight manner, the simplicity and usability have priority over quality and completeness. However, it still requires significant technical skills to develop mashups. The authors discuss open research challenges for end-user mashup development as follows:

1. *Simplicity and expressive power tradeoff.* To be easy-of-use, most current mashup tools focus on the simplicity feature. The tools hence are not powerful enough to create complex mashups that involve various data sources in a complicated work flow.
2. *Mashup components heterogeneity.* The challenge is to facilitate mashup composition based on the abstraction of heterogeneous mashup components such as web APIs, web data sources, and web widgets.
3. *Mashup composition techniques.* The challenge is to fully support three levels of mashup development, which are process integration, data integration, and user

[56] `http://www.fluidops.com/en/portfolio/information_workbench/` (accessed Nov. 01, 2015)

[57] `http://www.husky.fer.hr/` (accessed Nov. 01, 2015)

[58] `http://mdc.jackbe.com/enterprise-mashup` (accessed Nov. 01, 2015)

interface integration [47].

4. *Mashup evolution.* The challenge is to support users in maintaining a mashup, if either the mashup components change or the mashup needs to be re-engineered due to the update of user requirements.
5. *Online communities.* We can use online communities and social networks to foster the sharing of mashups as well as technical problem discussion. Moreover, the communities should be able to develop their mashups in a collaborative process.

Mashup components are heterogeneous and needed to be well modeled to lower the barriers of developing mashups for end users. Aghaee and Pautasso [75] propose a framework to evaluate mashup platforms with regard to the heterogeneous mashup component support. The framework consists of six dimensions as follows:

1. *Discovery.* Initially, users search for relevant components that they may need before starting to build up a mashup. A mashup tool should model the mashup components in a manner that facilitates this discovery task; it can choose among three approaches, that is semantic discovery, syntactic discovery, or keyword discovery.
2. *Input/output data type.* Components in a mashup interact in a work flow; they transfer output and receive input into and from each other. The component model hence needs to capture these input and output models. Their types can be either primitive types (e.g., String, Integer, Boolean) or Multipurpose Internet Mail Extensions (MIME) types (e.g., XML, JSON, RSS, or any standard formats found on the web).
3. *Access method.* The dimension specifies the method to access mashup components. Such methods are heterogeneous and can be categorized into language-dependent, protocol-based, database, and non-standard.
4. *Recursion.* If a created mashup can recursively become a mashup component to constitute a new mashup, it considerably reduce the effort to develop mashup and foster reusability.
5. *Output.* A mashup component can return functional output which is delivered as a service, data output, or visual output.
6. *Behavior.* During the mashup execution, the behavior of a mashup component can be either event-based or task-based. A task-based component is passive; it executes only when called, and given an input, it provides an output. By contrast, an event-based component is more active. It is activated when a specific event is fired. The component hence can trigger a sequence of tasks defined in other components.

Based on these dimensions, the authors evaluate a number of mashup frameworks and identify shortcomings of these approaches. Among these shortcomings are lacks of support for (i) event-based behavior, (ii) component discovery features, and (iii) language-dependent mashup components.

Another survey in mashup literature [76] reviews six different approaches and identifies potential areas of improvement and future research. For instance, the authors suggest that context-specific suggestions could support learning of how to build and find mashups. Regarding user interface improvements, they note that designing mechanisms such as automatic mashup generation to provide starting points to end users would enhance usability drastically.

The authors propose using the model of six types of barriers in end user programming systems [77] to identify typical issues users encounter when composing mashups. It helps researchers to conduct a user study and locate the important area of end user mashup development that needs to be improved first.

Di Lorenzo et al. [78] analyze the strengths and weaknesses of popular mashup tools, i.e., Damia [32], Yahoo! Pipes [33], Microsoft Popfly [34], Google Mashup Editor [35], Exhibit [36], Apatar[59], MashMaker [37], with respect to data integration. The evaluation is carried out based on eight dimensions covering various aspects of data integration, that is (i) data format and access, (ii) internal data model, (iii) data mapping, (iv) data flow operators, (v) data refresh, (vi) mashup output (vii) extensibility, (viii) and sharing.

A mashup tool is designed to handle and manage data available on the web. Despite its usefulness, a disadvantage is that we cannot access and use desktop data in the same way as web data. This is serious because users typically have a lot of data on their desktop; they do not intend to publish it on the web to provide input data sources for their mashup. The authors hence emphasize the need to support local data processing on users' desktops.

The authors summarize several notes at the end of their survey as follows: (i) Most of the tools use internal XML data model. (ii) The available operators for data manipulation and data integration are still limited; they are designed mainly based on the main goal of a tool (e.g., visualization oriented or data processing oriented tool). (iii) The majority of tools do not allow for the reuse of composed mashup. (iv) The corporation between tools is impossible or limited.

All of the evaluated tools are server-side applications, meaning that mashups and the data involved both are hosted on the server of the application provider. This may result in problems due to communication overload when a mashup creates too many requests to the servers. Most importantly, the survey claims that each tool requires a considerable level of programming effort by the user to build a mashup even though the tool is supposed to target "non-expert" users.

Daniel et al. [79] discuss the concept of *process mashups* by introducing three dimensions, i.e., *multi-user support*, *multi-page navigation*, and *workflow support. Multi-user support* enables users to collaboratively and concurrently build up the same mashup. *Multi-page navigation* means organizing mashup components into a navigation structure that are explored by users via hyperlinks. Finally, *workflow support* allows users to define a control and data flow.

From that, the authors classify mashups into eight classes, based on combinations of these dimensions. They are:

[59] `http://www.apatar.com/` (accessed Nov. 01, 2015)

1. *simple mashups*, which support single user, single page and no workflow (e.g., mashArt [46], Yahoo! Pipes [33], MashMaker [37])
2. *multi-page mashups*, which support single user, multi-page, and no workflow (e.g., EzWeb [80])
3. *guided mashups*, which support single user, single page and control-flow (no tool exists)
4. *page flow mashups*, which support single user, multi-page, and control-flow (e.g., ServFace Builder [81])
5. *shared page mashups*, which support multi-user, single page and no workflow (no tool exists)
6. *shared space mashups*, which support multi-user, multi-page, and no workflow (e.g., IBM Mashup Center[60])
7. *cooperative mashups*, which support multi-user, single page and workflow (e.g., Gravity[61])
8. *process mashups*, which support multi-user, multi-page and workflow (e.g., MarcoFlow [82]).

Blichmann et al. [83] present their vision of collaborative mashups by specifying three challenges. They are:

1. to develop a mechanism for unified handling of (non)collaborative components, which means all components should be connectable and synchronizable despite their various implementation or built-in support for collaboration,
2. to synchronize differently implemented components with identical functionality, which allows users to be able to select their preferred components and devices,
3. and to support fine-grained sharing of mashup composition parts, which allows for unstructured collaboration.

They then present their preliminary solutions with their CRUISe platform. The platform uses Semantic Mashup Component Description Language [84] (SMCDL) to uniformly describe mashup components. It currently does not support either component synchronization or individual sharing parts of a mashup.

Endres-Niggemeyer [85] introduces mashup applications for each of her mashup classifications, i.e., media mashups, ubiquitous mashups, DBpedia mashups, web of things mashups, mathematical mashups, speech mashups, urban mashups, travel mashups, end-user mashups, and *semantic mashups*. Most applications are limited to specific domains. Intelligent Book Mashup [86], RDF Book Mashup [87], Black Swan Events [71], and DERI Pipes [40]) are introduced as semantic mashup applications. As ontologies and mashups are respectively the pillars of the semantic web and web 2.0 [51], these applications strive to foster mashups by combining them with inherent semantics of the data. They collect and mashup the data, lift it to a semantic level, perform reasoning

[60] `www.ibm.com/software/info/mashup-center/` (accessed Nov. 01, 2015)

[61] `http://www.sdn.sap.com/irj/scn/weblogs?blog=/pub/wlg/17826` (accessed Nov. 01, 2015)

and return appropriate data to users.

The author also lists a number of available languages and standards to model and describe mashups as follows:

1. Enterprise Mashup Markup Language – EMML[62]
2. Mashup Component Description Language – MCDL [88]
3. Web Mashup Scripting Language – WMSL [63]
4. Universal model based on MetaObject Facility standards [89]
5. Universal model of components and composition [46]
6. ResEval Mash [42, 90] with a domain-specific description language
7. UML2 model for a set of integrated mashups [91]

For end user mashup development, the author identifies four possible types of mashup interfaces. The interface can be based on a flow chart-model, follow a spreadsheet approach, use a tree pattern, or be integrated into a browser as a plug-in. Mashups become increasingly popular because of their attractive properties such as simplicity, loose coupling, data resource reuse, user activity and creativity.

2.6 Research Gap

To sum up, various mashup-based data integration tools and frameworks have been introduced by developers, researchers, and practitioners. Some of them have been discontinued, many of them are domain specific. Based on the introduced surveys [60, 62, 72, 76, 79], mashup tools and frameworks can be classified as shown in Table 2.6.

Mashup tools	Classification
Damia [32], Apatar[63], Marmite [38]	Wiring paradigm [62]
Presto[64]	Wiring paradigm [62], Visual programming [72]
Microsoft Popfly [34]	Wiring paradigm [62], Information mashups [60, 76]
Yahoo! Pipes [33]	Wiring paradigm [62], Information mashups [60, 76], Visual programming [72], Simple mashups [79]
IBM Mashup Center[65]	Wiring paradigm [62], Information mashups [60, 76], Visual programming [72], Shared space mashups [79]
Google Mashup Editor [35]	Information mashups [60, 76], Scripting Languages [62]
Serena Mashup Composer[66]	Process Mashups [60, 76]

[62] `http://www.openmashup.org/omadocs/v1.0/index.html` (accessed Nov. 01, 2015)

[63] `http://www.apatar.com/` (accessed Nov. 01, 2015)

[64] `http://mdc.jackbe.com/enterprise-mashup` (accessed Nov. 01, 2015)

[65] `www.ibm.com/software/info/mashup-center/` (accessed Nov. 01, 2015)

[66] `http://www.serena.com/index.php/en/` (accessed Nov. 01, 2015)

Intel MashMaker [44]	Process Mashups [60, 76], Programming by demonstration [62], Simple mashups [79]
Black Swan [71], Super Stream Collider [39]	Data analytics mashups [70]
Information Workbench[67], DERI Pipes [40], MashQL [41]	Generic mashups [70]
Husky[68], Mashroom [43]	Spreadsheets[72]
Vegemite [45], Karma [65]	Programming by demonstration [72]
ServFace [81]	Visual programming [72], Page flow mashups [79]

Table 2.6: Mashup tools and frameworks

A limited number of frameworks such as Super Stream Collider [39], DERI Pipes [40], and MashQL [41] aim at semantic data processing. Those frameworks, however, do not leverage semantic web means to facilitate automatic data integration for non-expert users [70], neither do they provide mechanisms to integrate semantic with non-semantic data. Thus our objective is first to focus on semantic mashups and overcome such limitations. We address the challenge to design the semantic models for mashup components and leverage the semantics to foster mashup-based automatic data exploration and integration. The input data of mashup components can be in different formats (e.g., CSV, XML, JSON, or RDF), but the semantic models impose the semantic format on the output data and hence can tackle data heterogeneity.

The surveys show that it is difficult for non-expert users to make use of the mashup frameworks to compose mashups and integrate data. On the one hand, a high-level and problem-oriented framework is easier to use than a low-level one, because it does not require users to be familiar with special technological and programming concepts to perform data integration tasks. On the other hand, we need a generic framework that can tackle the increasing number of heterogeneous web resources rather than be tailored towards specific problems and resources. These design objectives lead to a trade-off [72]. A versatile mashup framework typically consists of a large number of predefined components; users generally have a clear idea of what they are trying to achieve, but they do not know which components they need and how to correctly combine them in order to reach their goal. This "*simplicity and expressive power*" trade-off [72] is a challenging issue that we need to address in this research.

The two surveys [60, 72] discuss the importance of the communities to the success of end-user development tools. An open and collaborative model – which ties together three stakeholders (i.e., data publishers, developers, and end users) – enables each stakeholder

[67] `http://www.fluidops.com/en/portfolio/information_workbench/` (accessed Nov. 01, 2015)

[68] `http://www.husky.fer.hr/` (accessed Nov. 01, 2015)

to contribute and share their work to the open data community. Based on available data sources provided by different data publishers, developers are encouraged to create and deploy mashup components that are free to use or reuse. By combining such components, users can create mashup applications to obtain, process, and visualize open data in a dynamic and creative manner. The applications finally can be shared or reconfigured among the user communities to foster reusability. However, to the best of our knowledge, current research focuses on user communities only and allows them to share, comment, or rank their mashups. There is no mashup-based data integration framework that can facilitate collaborative work among users, data publishers and developers and encourage widespread use of (linked) open data.

The survey [78] poses a challenge to integrate data that is stored in different devices and not available on the web yet. In the literature so far, there is no research on *mashups* assembled from components that are *distributed* among different nodes (e.g., sensors, embedded devices, mobile phones, desktops, servers) to collect and integrate data. Such *distributed mashups* can facilitate *collaborative* data integration in which each stakeholder contributes their data and computing resources to the shared processing flow.

CHAPTER 3

Conceptual Framework

This chapter[1]presents the main contribution of our research, i.e., a conceptual framework for exploring and integrating heterogeneous data sources. The framework is based on the concept of Linked Widgets, which are semantic modules that allow non-expert users to access, integrate, and visualize data in a creative, collaborative, distributed, and automatic manner.

The chapter is organized as follows: In Section 3.1, we present the modular approach for open data integration to facilitate collaborative work among the data publishers, developers, and end-user communities. Section 3.2 discusses the architecture of our conceptual framework. Section 3.3 provides detailed information on client and server Linked Widgets. The semantic model of Linked Widgets is discussed in Section 3.4. We present the message protocol to construct and execute mashups in Section 3.5 and Section 3.6, respectively. Section 3.7 introduces five mashup patterns (i.e., collaborative, persistent, distributed, streaming, and complex mashup pattern) that can be used for various use cases. Section 3.8 presents the mechanism to encapsulate a mashup into a new widget to foster reusability. We develop algorithms for automatic data exploration and data integration via Linked Widgets in Section 3.9. Finally, in Section 3.10, we develop a tag-based composition mechanism that enables non-expert users to compose mashups by specifying them in structured text.

3.1 Modular Approach for Data Integration

The modular approach illustrated in Figure 3.1 combines *Web Services* and *Service-Oriented Architecture (SOA)*[92] concepts. Whereas services target developers, we transform them into *blocks* aimed at end users. To utilize data from different (linked) open data sources, publishers and developers can create three types of *blocks*: (i) *data blocks*,

[1] Parts of this chapter also appear in [1, 3]. The author of this thesis is also the lead author of these papers.

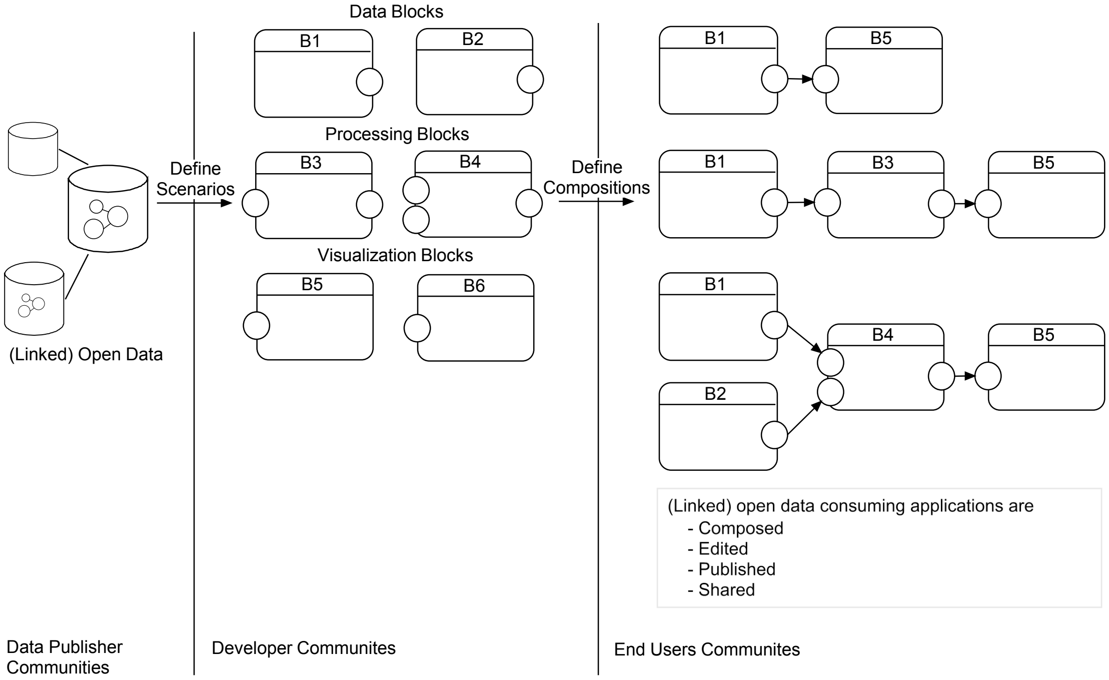


Figure 3.1: Modular approach for open data integration

which collect data from one or multiple datasets; (ii) *processing blocks*, which process and combine data in different ways through enrichment, transformation, and aggregation; and (iii) *visualization blocks*, which display the final data. These blocks are organized into three layers, i.e., *data layer*, *business layer*, and *presentation layer*.

These blocks can be used by end users to compose open data-consuming applications in many ways and create a value chain between open data, developers and end users. This model enhances the reusability of the developers' work; it stimulates end users' creativity to build up dynamic applications; and it inherits various benefits of *SOA* and *Software Component-Based* approaches.

Blocks allow non-expert users to build ad-hoc applications rapidly. Each block can receive input from other blocks to process and return its output which, in turn, can serve as input for another block. To end users, blocks are "*black boxes*" with adjustable parameters that control the *execution function*. Similar to functional programming or web services, blocks can have multiple input terminals, but only a single output terminal.

Two mandatory components for a complete application are *data blocks* and *visualization blocks*; *processing blocks* are optional. A *data block* does not have any inputs; it collects data from an arbitrary data source to provide input for *processing* and *visualization blocks*. *Visualization blocks* display output data of *processing* or *data blocks*. Inside an application, more than one *visualization block* can be applied, because there can be multiple ways to display the same data.

To compose an application, users simply select appropriate blocks and connect the output of a block to the input of another one. The model checks whether the semantics of the particular input and output data models match and only allows for connections between compatible terminals. There are four types of links: (i) links between *data blocks* and *processing blocks*, (ii) links between *data blocks* and *visualization blocks*, (iii) links between two *processing blocks*, and (iv) links between *processing blocks* and *visualization blocks*.

We identify three basic operations of data processing: enrichment, transformation, and aggregation. Corresponding to the three operations, there are three sub-types of *processing blocks*.

1. If blocks get additional data from other datasets and add it to the input, they are called *enrichment processing blocks*. An example is a block in which its input – a list of *Locations*, i.e., *[Locations(lat, long, description)]* – is enriched by sample images for each *Location*, i.e., *[Locations(lat, long, description, [Image])]*.
2. *Transformation processing blocks* transform input instances of an RDF class into output instances of a different RDF class based on the relation between the two classes. Hence those blocks are a crucial element which enables users to leverage the key ideas of LOD, i.e., making use of links between resources from one or more LOD datasets. For example, a block may receive *Locations* and output *MusicEvents* organized at a place nearby.
3. The final type, *aggregation processing blocks*, aggregates data from its multiple input terminals. Therefore, blocks of this type always have at least two input terminals representing different entities of similar classes.

To sum up, major advantages of the block concept are (i) the flexibility in building applications, starting from selecting data sources, applying various processing operations, to visualizing the final data in multiple ways; and (ii) the effective use of links, i.e., inner links and outer links, between LOD datasets.

3.2 Architecture

We define a number of design objectives as follows: First, blocks need to run on heterogeneous platforms and communicate with each other; the cost to develop blocks should be low; anyone should be able to contribute blocks; and it should be easy to share complete applications composed from disparate blocks.

Using modern web technologies, we decided to implement each block as a web widget (cf. Section 2.4.3). Figure 3.2 illustrates the Linked Widgets framework architecture. It serves three major groups of stakeholders: *developers*, *mashup creators*, and *mashup users*.

Linked Widgets (or blocks) constitute the basic elements of the framework; they extend the concept of standard web widgets[2] with a semantic model. This semantic model, which follows the Linked Data principles [17], is used to annotate the input and output of widgets as well as their relations among each other.

[2] `http://www.w3.org/TR/widgets/` (accessed Nov. 01, 2015)

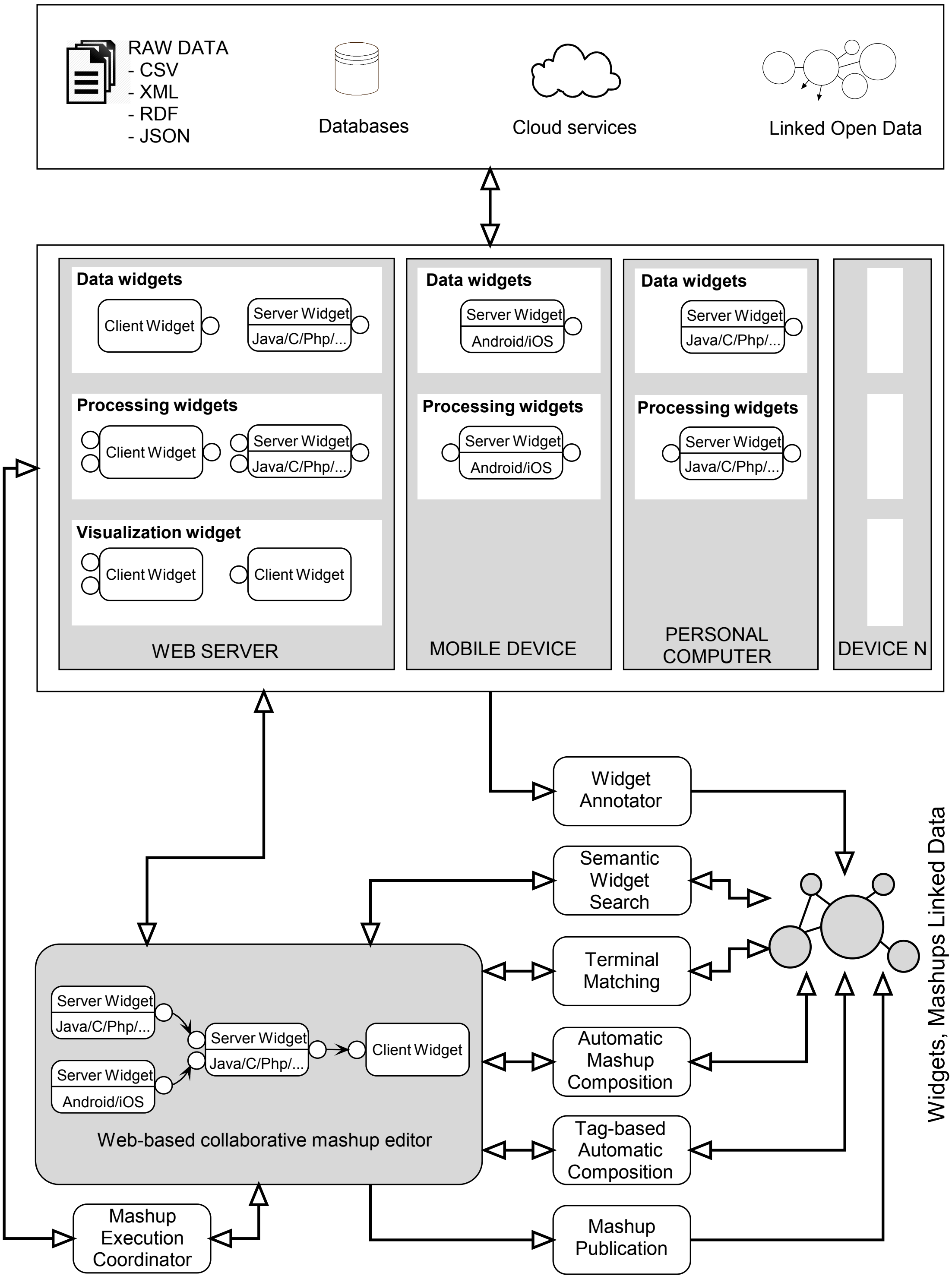


Figure 3.2: Linked Widgets framework architecture

A *Linked Widget* is similar to a web service in that it has multiple inputs and a single output. Important distinctions, however, include that (i) Widgets have a user interface and are hence easier to handle than web services; (ii) Widgets are more versatile than web services, e.g., they can visualize data; (iii) Widgets that are combined properly can collaborate automatically by predefined communication protocols whereas web service composition requires technically additional work, e.g., conformation of parameters; (iv) Connected widget can interact with each other both ways, web service communication is always sequential and unidirectional; (v) Widgets can be deployed on various devices and in various environments whereas web services typically run on a server; and finally (vi) Linked Widgets are associated with a semantic model.

To be able to connect widgets with each other, they have input and/or output terminals. Connecting an input terminal of a widget with an output terminal of another, means the former widget accepts and processes the output data of the latter as its input data. We use JSON-LD[3] for data transmission between widgets. *Linked Widgets* can tackle the challenge of data heterogeneity. They standardize data and lift arbitrary data sources to a semantic level. Thus the framework can integrate raw data in CSV, XML, JSON, or HTML format; furthermore, data can be collected from databases, cloud/API services, or the Linked Open Data cloud.

Depending on their execution mode, widgets may be classified as *client* or *server widgets*. From a functional point of view, we can furthermore categorize *Linked Widgets* into (i) *data widgets*, (ii) *processing widgets*, and (iii) *visualization widgets.* Those widgets are highly reusable and can be parametrized per mashup.

Developers can contribute *client widgets* to the framework, run *server widgets* on their own infrastructure, or provide them for deployment on personal computers, mobile phones or other devices. They use the *widget annotator* module to semantically annotate the widget's input and output models and to provide provenance and license information. The annotation is stored as Linked Data and is used by several modules of the framework.

The core of the data integration architecture is a web-based *collaborative mashup editor*. Multiple *mashup creators* (or data integrators) and *mashup users* can compose mashups simultaneously and collaboratively. This also allows users to create mashups that integrate private data with publicly available data sources.

Using *semantic widget search*, *mashup creators* locate and group available widgets of different *developers* into collections that are relevant for a particular application domain. They then connect those widgets and build mashups. Because combining widgets does not require any particular technical skills, average users can become *mashup creators*. Moreover, as *mashup users*, they can adapt widget parameters of existing mashups to their needs before executing them.

We classify mashups into three types: (i) *local mashups* that consist exclusively of client widgets, (ii) *hybrid mashups* that make use of both client and server widgets, (iii) and *distributed mashups* that consist entirely of server widgets, except for the final visualization widget(s).

[3] `http://www.w3.org/TR/json-ld/` (accessed Nov. 01, 2015)

A *local mashup* does not use any resources of the framework server, because it is executed completely inside the client browser. This implies that intermediate and final data are lost once the web browser is closed.

In contrast, widgets in *distributed mashups* are executed remotely as persistent applications; their output can hence be accessed at any time. *Hybrid* and *distributed mashups* can be executed in a distributed manner, which may involve multiple nodes that each executes individual server widgets. This is highly useful, for instance, for streaming data use cases where data must be collected and processed continuously.

The *mashup execution coordinator* is a critical component that enables widgets to cooperate. It links client and server widgets that are executed in different environments, e.g., browsers, Android, iOS smart phones, personal computers, or web servers. For each type of mashups, the coordination mechanisms differ in order to conserve computing resources. Details of the protocols and mechanisms are discussed in Section 3.6.

When *mashup creators* build a mashup, a common task is to connect an input terminal of a widget to an output terminal of another widget. To enforce valid connections (i.e., ensure that the output terminal can provide all data required at the input terminal), creators can use the *terminal matching* module (cf. Section 3.9.1); the module validates connections using the semantic model. It helps creators to speed up the mashup creation process.

The *automatic mashup composition* module (cf. Section 3.9.2) is designed to automatically compose a complete mashup from a widget, or a complete branch that consumes or provides data for a specific output or input terminal. "*Complete*" in this context means that all terminals must be wired, i.e., have a valid connection. Built on top of the *automatic mashup composition* module, the *tag-based automatic composition* module (cf. Section 3.10) allows users to compose mashups by specifying them in structured text.

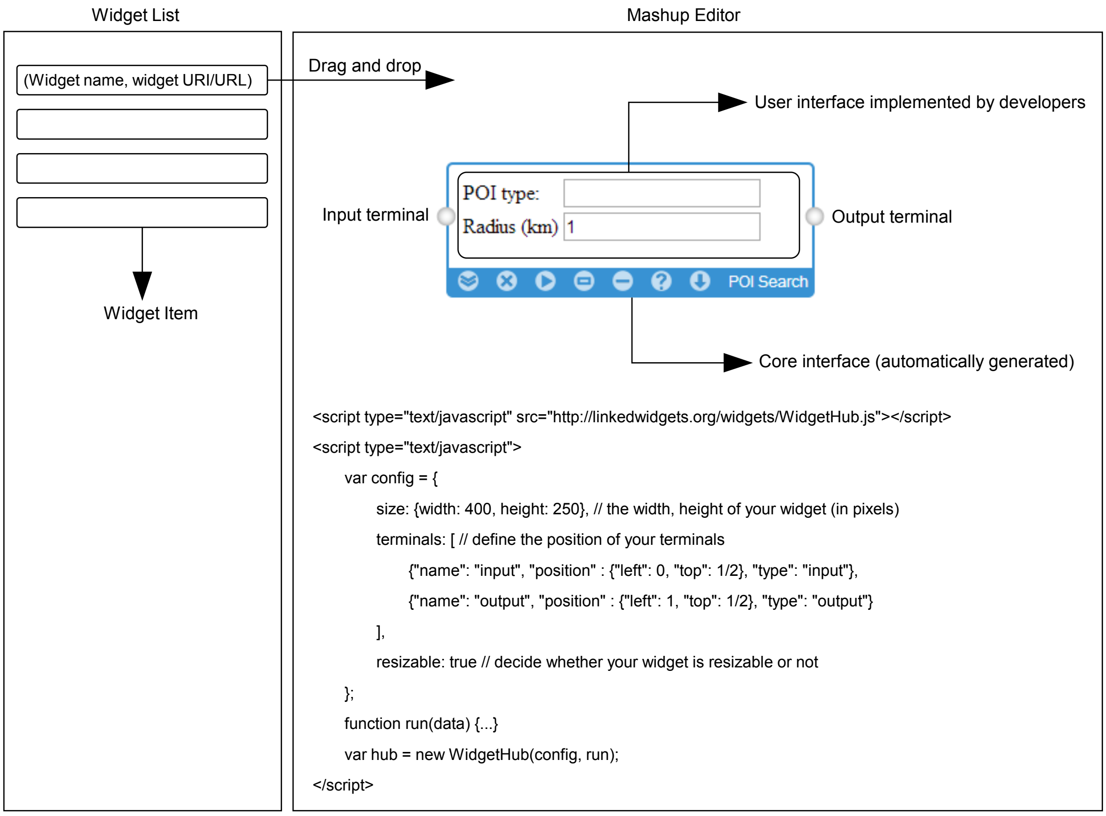
Mashup users can publish mashups on their website by means of the *mashup publication* module. A published mashup shows the final *visualization widget* only and hides all previous data processing steps from the viewer. The mashup itself can also be saved as a new *data widget* using this module. This encourages users' creativity by allowing them to reuse mashups without the need for programming skills.

3.3 Client and Server Linked Widgets

Developers can implement Linked Widgets as either *client* or *server widgets*. The following two sections provide details on each of those types.

3.3.1 Client Widgets

Client widgets are executed on the client side, i.e., they use client memory and processor resources; data is collected and processed on-the-fly in the browser. The server hosting the *client widgets* is not necessarily the framework server, i.e., a mashup may combine widgets hosted on various servers. This makes the framework flexible and allows external parties to host widgets on their own infrastructure.


Widget List
Mashup Editor
(Widget name, widget URI/URL)
Drag and drop
User interface implemented by developers
Input terminal
POI type:
Radius (km)
1
Output terminal
POI Search
Widget Item
Core interface (automatically generated)

```
<script type="text/javascript" src="http://linkedwidgets.org/widgets/WidgetHub.js"></script>
<script type="text/javascript">
    var config = {
        size: {width: 400, height: 250}, // the width, height of your widget (in pixels)
        terminals: [ // define the position of your terminals
            {"name": "input", "position" : {"left": 0, "top": 1/2}, "type": "input"},
            {"name": "output", "position" : {"left": 1, "top": 1/2}, "type": "output"}
        ],
        resizable: true // decide whether your widget is resizable or not
    };
    function run(data) {...}
    var hub = new WidgetHub(config, run);
</script>
```

Figure 3.3: Client widget components

Each *client widget* consists of (i) a semantic model, (ii) an execution function, (iii) input and/or output terminals, (iv) a core widget interface which is automatically generated for users to control the widget, e.g., to *run*, *cache*, *view output data*, *resize*, or *destroy* the widget (cf. Figure 3.3), and (v) its custom user interface programmed by developers. The execution function transforms the received input into an output according to the parameters specified in the interface.

To implement a *client widget*, developers create a user interface in an arbitrary web language and then follow three steps: (i) inject a JavaScript file[4] to facilitate cooperation with other widgets, (ii) define the input and/or output configuration, and (iii) implement a JavaScript function *run(data)* that is invoked when the widget is executed in a mashup. If a widget has no input, then the corresponding *data* object is *null*. Otherwise, upon execution of a widget, output data from all relevant widgets is collected to build the *data* object and pass it to *run*.

A *client widget* is instantiated when users drag and drop a widget item from the widget list into the mashup editor. We associate each widget with an identifier to differentiate

[4] http://linkedwidgets.org/widgets/WidgetHub.js (accessed Nov. 01, 2015)

widgets used in a mashup.

A widget item consists of the widget name, and the URI of the widget. If the widget is already annotated and published, from its URI, we can retrieve its Uniform Resource Locator (URL); and later, to execute advanced functions, such as terminal matching and automatic mashup composition, we use the URI to retrieve the widgets' model and other metadata.

On the other hand, if the widget is still in the developing/testing stage, we use its URL as follows: We first create the core widget interface with an HTML iframe inside. After that, we use the widget URL to load its source code into the iframe and read the input/output configuration to create the corresponding input/output terminals. At this stage, the framework displays a complete widget interface as shown in Figure 3.3.

We decided to decouple input/output configuration from the widget annotation to simplify the widget developing, testing, and maintenance processes. It allows a widget to operate even before it is semantically annotated; however, users cannot use advanced functions as long as semantic annotations are not complete.

3.3.2 Server Widgets

Client widgets are easy to develop and necessary for a lightweight and scalable mashup framework. However, their capabilities are restricted by the web browser execution environment. A web browser is inadequate for hosting mashups that process data from embedded devices or subsystems that are not yet available on the web. Furthermore, web browsers are not designed for heavy data processing tasks. Finally, as soon as a user closes the browser, the mashup output data can no longer be accessed.

To overcome these limitations, *server widgets* shift the execution function from the browser environment to standalone application environments. These server widgets consist of two main parts: (i) a *user web interface*, which is the same as for *client widgets*, but without the *run* function, and (ii) a *remote executor*. The *client interface* is for users to set up the parameter and control the *remote executor*.

When users drag and drop a *server widget* into the mashup editor panel, a *client user interface* of the *server widget* is instantiated. Next, a connection channel between this *client interface* and the *remote executor* of the *server widget* is set up. It is therefore necessary to keep the *remote executor* running persistently and reachable via the internet.

For each *server widget*, we have only a single *remote executor*, but potentially many *client user interfaces* that are instantiated for each instance of the widget that is created for various mashups. When an instance of a *server widget* is executed, the *client interface* sends its parameters to the *remote executor*, which, in turn, creates a *widget job* to process the data received from the predecessor widgets. Details of the communication protocol between *client* and *server widgets* are covered in Section 3.6.

Server widgets introduce the concept of *distributed mashups* – a type of ad-hoc application whose processing tasks are executed in a distributed manner on multiple devices. Such mashups are particularly useful for streaming or real-time data processing applications. Users can close the browser at any time while the backend performs data collection and processing tasks. This distinguishes our framework from previous approaches such as Damia [32], Yahoo! Pipes [33], Microsoft Popfly [34], Google Mashup Editor [35], Exhibit [36], Apatar[5], MashMaker [37], Marmite [38], Super Stream Collider [39], DERI Pipes [40], MashQL [41], Information Workbench[6], ResEval Mash [42], Mashroom [43], Husky[7], Intel Mash Maker [44], Vegemite [45], Presto[8], and mashArt [46].

Server widgets hence provide the following benefits: (i) They can act as a data connector to obtain and provide data on different services, devices, or systems for a mashup; (ii) they facilitate collaborative use cases where each participant contributes *data* or *processing widgets* to a shared mashup; (iii) because their computing tasks are performed within the hosting devices, similar to *client widgets*, they reduce the framework server load; (iv) they can run persistently in the background to collect or process data for data monitoring or data streaming applications; (v) *server widgets* deployed on powerful servers are capable of processing large volumes of data over extended time periods.

There are two sub-types of *server widgets*, i.e., *server data widgets* and *server processing widgets*; we always use *visualization widgets* as *client widgets* at present. Although we can deploy (server) *processing widgets* on any kind of device, servers are the most suitable targets, as they need to be consistently online; this also allows it to offload computationally intense data processing tasks from mobile devices. Mobile devices, however, can be the environment for (server) *data widgets*. They can, for instance, collect and provide data from mobile devices for a mashup. For example, smartphones can act as sensors that periodically provide GPS data, foot steps, temperature data etc.

3.4 Semantic Model for Linked Widgets

As the Linked Widgets framework is based on semantic web technologies, creating an ontology is a requisite step after we have identified the architecture and the requirements of the framework. It enables us to add semantics (cf. Section 2.1) to the model of Linked Widgets, which aims to facilitate automatic operations such as *automatic terminal matching* and *automatic mashup composition*. It also enhances the interoperability between the Linked Widgets framework and other applications; for example, third party applications can access widget annotations as Linked Open Data to implement custom composition algorithms for particular data domains.

In this section, we define ontology requirements and design concepts and relations for the semantic model of Linked Widgets. To this end, we follow the ontology development guideline proposed by Noy and Mcguinness [93]. An ontology is "*a formal and explicit specification of a shared conceptualization*"[94]. It [95]"*defines (specifies) the concepts, relationships, and other distinctions that are relevant for modeling a domain; the specification takes the form of the definitions of representational vocabulary (classes, relations,*

[5] `http://www.apatar.com/` (accessed Nov. 01, 2015)

[6] `http://www.fluidops.com/en/portfolio/information_workbench/` (accessed Nov. 01, 2015)

[7] `http://www.husky.fer.hr/` (accessed Nov. 01, 2015)

[8] `http://mdc.jackbe.com/enterprise-mashup` (accessed Nov. 01, 2015)

and so forth), which provide meanings for the vocabulary and formal constraints on its coherent use".

3.4.1 Definition of Requirements

As a typical step that should be performed before designing an ontology [93], we identify the requirements of the ontology for the Linked Widgets model in the data integration context as follows:

R1 – Reuse of existing ontology This is the common requirement for ontologies to enhance the interoperability among separate systems. To this end, if a concept or a relation that is necessarily included in the Linked Widgets model already exists in some Linked Open Data ontology, we reuse it rather than define the new one.

R2 – Reusability and extensibility It requires that the users of the ontology should be able to easily reuse the defined vocabulary; they can reuse a part of the ontology, clarify a particular concepts by adding one or more attribute, or extend the whole ontology in a more general context. For example, they can create a new type of widgets, or annotate the parameters defined in the user interface of widgets to add the detailed functionality of widgets to the semantic model.

R3 – Lightweight We decide to follow a lightweight approach so that we facilitate our automatic data integration algorithms (i.e., the *automatic data model matching* and the *automatic mashup composition* algorithms presented in Section 3.9.1 and Section 3.9.2, respectively). "*Lightweight*" in this context means that we use as few number of SPARQL queries as possible for frequent tasks such as to load and construct the full input (output) data model of a Linked Widget; the task is performed when we dereference a widget URI. To this end, we use only a small number of concepts and relations, and do not specify complex constraints on them; we mainly focus on the Linked Widget data model. Thus we choose OWL-Lite (cf. Section 2.1.3) language to represent our ontology.

R4 – Change management Because many new APIs and data sources are continuously introduced, developers can modify their widgets' interfaces and functionalities. This may be accompanied by the change in the respective Linked Widgets model (e.g., the change in the input and output data model), which causes the existing mashups to stop working properly. To address this issue, the ontology needs to specify provenance information so that we can track and appropriately react to the widget upgrade (cf. Section 3.4.2). For example, as soon as a widget is modified, we can send an alert to the creators of all mashups that make use of the widget; the mashups then can be reedited and work properly again.

R5 – Interoperability with LOD vocabulary Widgets have input and output terminals that can receive and process data extracted from various LOD datasets. We require that the data model of these input and output terminals must be open, i.e., we can reuse the LOD vocabulary in the input and output data model of Linked Widgets.

R6 – Connectivity The LOD cloud contains many incoming and outgoing links among many LOD datasets. These links are the basis for combining separate sources published by different data publishers. We require the Linked Widget model to be capable of representing such links. To this end, we make use of the links between nodes in the input and output tree model (cf. Section 3.4.3).

R7 – Support for automatic widget discovery and composition The ontology needs to annotate widgets (mashups) in such a manner that we are capable of locating the relevant widgets (mashups) in a particular context; for example, in addition to conventional search methods which are based on keywords, categories, tags, we should be able to search for a widget based on the data type of its input and output data, or the relationship between them. Moreover, the annotated information must be rich enough to allow for the automatic input-output data model matching as well as the automatic mashup composition algorithm.

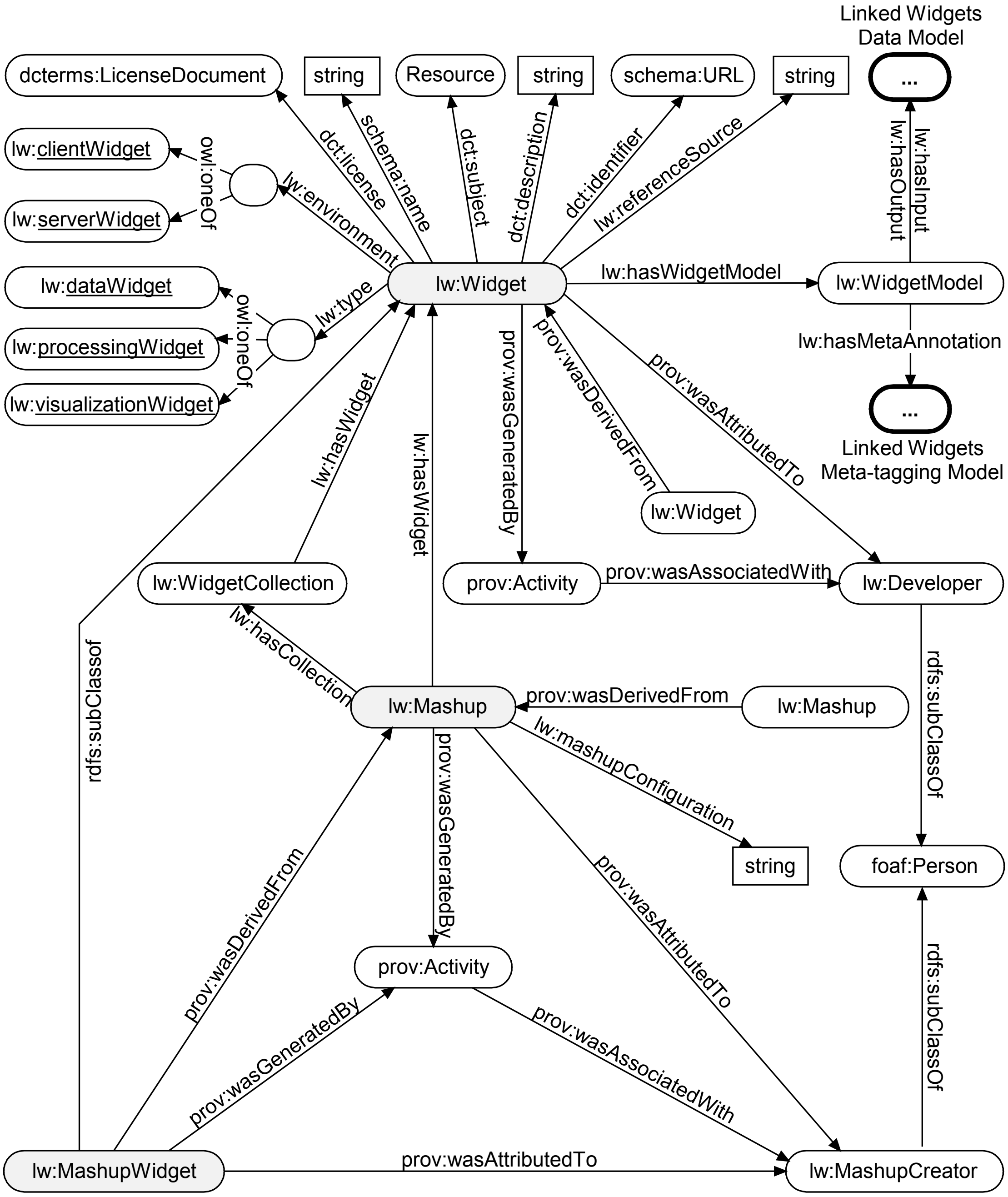
3.4.2 Basic Concepts and Relations

The basic concepts and relations of the Linked Widgets ontology are illustrated in Figure 3.4 and listed in Tables 3.1 and 3.2. Two other main parts of the ontology, i.e., the *Linked Widget data model* and the *Linked Widget meta-tagging model* are presented in Sections 3.4.3 and 3.10.3, respectively; while the *Linked Widget data model* focuses on the expression of widget output and input data model to allow for widget discovery and automatic mashup composition, the *Linked Widget meta-tagging model* offers extra tag-based metadata and aims at mashup composition from structured text.

The central concepts of the ontology are *lw:Widget* and *lw:Mashup*; they are the most important resources of the Linked Widgets framework. Each has a name, a description, one or more licenses, and an address, which are respectively specified by the *schema:name*, *dct:description*, *dct:license*, and *dct:identifier* properties. The address is a URL that we can use to access the user interface of the widget or the mashup.

We use the *owl:allValuesFrom* and *owl:oneOf* properties to specify the restriction on the ranges of the *lw:environment* and *lw:type* properties (which are properties of the *lw:Widget* class). As a result, on the one hand, a *lw:Widget* must be either a *lw:clientWidget* or *lw:serverWidget*; on the other hand, it must be a *lw:dataWidget*, *lw:processingWidget*, or *lw:visualizationWidget*.

To describe additional information such as the URL of a data source used in a data widget, the reference to a special algorithm implemented in a processing widget, or the visualization library used in a visualization widget, we make use of the *lw:referenceSource* property. Finally, using the *dct:subject* property, we define the categories (e.g., Social, Financial, Science, Government, Enterprise) a widget falls into.



lw	http://linkedwidgets.org/ontologies/	**prov**	http://www.w3.org/ns/prov#
dct	http://purl.org/dc/terms/	**foaf**	http://xmlns.com/foaf/0.1/
owl	http://www.w3.org/2002/07/owl#	**schema**	http://schema.org/

Figure 3.4: Linked Widgets ontology

Class	**Description**
lw:Widget	a "mashable" widget, which can pass or receive events so that multiple widgets can be combined to build up a mashup
lw:WidgetCollection	a group of widgets that are in the same domain
lw:Mashup	a mashup of widgets
lw:MashupWidget	a widget derived from a mashup to foster reusability
lw:WidgetModel	range of the *lw:hasWidgetModel* property to associate a widget with its semantic model
lw:Developer	a developer, who creates and adds widgets to the framework
lw:MashupCreator	the creator of a mashup

Table 3.1: Classes of the Linked Widgets ontology

Property	**Description**
lw:referenceSource	the link to additional information of a widget such as the data source, the reference to an algorithm, or the visualization library used in the widget
lw:environment	the execution mode of a widget; widgets can be categorized into client or server widgets
lw:type	the functionality of a widget; widgets can be categorized into data, processing, or visualization widgets
lw:hasWidgetModel	the semantic model of a widget
lw:hasWidget	a list of widgets included in a mashup or a collection
lw:hasCollection	a list of widget collections used in a mashup
lw:mashupConfiguration	the JSON configuration of a mashup such as the saved parameters and the position and size of each widget

Table 3.2: Properties of the Linked Widgets ontology

Widgets that are in the same domain or share the same features are grouped into a *lw:WidgetCollection*. A *lw:WidgetCollection* is linked with a *lw:Mashup* via the *lw:hasCollection* property. All widgets used in a *lw:Mashup* must be in the widget collections of the mashup; the used widgets are listed by means of the *lw:hasWidget* property.

In order to trace the change of both *lw:Widget* and *lw:Mashup*, we use the Provenance ontology[9]. The ontology describes a set of classes, properties, and restrictions that can present provenance information in a wide range of applications and domains (e.g., open

[9] `http://www.w3.org/TR/prov-o/` (accessed Nov. 01, 2015)

information systems, science applications, news, or law). Detailed provenance can be easily created or edited to express the origin of data and the full revision change of data.

We link a *lw:Widget* with a *prov:Activity* and link the *prov:Activity* with a *lw:Developer* (which is a sub-class of the *foaf:Person* class) through the *prov:wasGeneratedBy* and the *prov:wasAssociatedWith* properties, respectively. A new version of the *lw:Widget* is linked with the previous version through the *prov:wasDerivedFrom* property. This overall structure enables us to use SPARQL queries to load the information of the very first to the latest version of the *lw:Widget*. In a similar manner, we link a *lw:Mashup* with a *prov:Activity* and link the *prov:Activity* with a *lw:MashupCreator* to annotate the provenance information of the mashup.

Two out of three stakeholders (i.e., *lw:Developer* and *lw:MashupCreator*) of the framework are modeled in the ontology. We do not model end users (who are the third stakeholder) and store their profile, because they are allowed to edit or run an existing mashup in the framework only. To save a personal mashup, they need to register an account and become a *lw:MashupCreator*.

The *lw:mashupConfiguration* property associates a mashup with a mashup configuration, which is a JSON string. The configuration consists of the saved parameters, the position and size of each widget, as well as the links between input and output terminals. We do not semantically annotate such information because it is not semantically relevant for the mashup; moreover, the simple structure of configuration allows for simple and fast mashup loading.

To facilitate reusability, we enable a *mashup creator* to derive a *lw:MashupWidget* from a *lw:Mashup* (cf. Section 3.8). The *lw:MashupWidget* class is a sub-class of the *lw:Widget* class. Similar to what we have done with the *lw:Widget* and *lw:Mashup* classes, we use the Provenance ontology to annotate different versions of a *lw:MashupWidget*. This allows us to automatically detect which *lw:MashupWidget* are affected as soon as a new version of a particular *lw:Widget* or *lw:Mashup* is released.

3.4.3 Linked Widget Data Model

We semantically annotate the input and/or output data model of Linked Widgets. These semantic I/O models are essential for the subsequent search and composition processes. Furthermore, they are crucial for the effective sharing of widgets. For example, even when the number of widgets available is limited (e.g. 43 for Yahoo! Pipes [33] and 300+ for Microsoft Popfly[10]), finding appropriate widgets needed to build a particular mashup is a difficult task. Many existing mashup frameworks employ a text-based approach for widget search, which is typically ambiguous and hence not particularly helpful to explore and locate widgets.

Client and server Linked Widgets have different execution environments, but they share the same model. Due to similarities with web services, we initially considered to describe Linked Widgets semantically using SAWSDL [96], OWL-S[11], or WSMO[12]. These

[10] `http://en.wikipedia.org/wiki/Microsoft_Popfly` (accessed Nov. 01, 2015)

[11] `http://www.w3.org/Submission/OWL-S/` (accessed Nov. 01, 2015)

[12] `http://www.w3.org/Submission/WSMO/` (accessed Nov. 01, 2015)

languages are well-suited for the formal specification of interfaces, such as Linked Widgets' input and output terminals. They do not, however, allow to establish a well-defined semantic relation between input and output data; they capture the functional semantics and focus on input and output parameters.

As a precondition for advanced widget exploration and automatic mashup composition algorithms, an explicit semantic link between input and output terminals is necessary. To this end, several formal annotation methods using a graph-based model have been developed in the literature [97, 98, 99].

To specify the semantic model of Linked Widgets, we adopt and adapt the Karma model [98]. The idea of the model is to reuse the SWRL vocabulary[13] to define the semantic relation between two nodes in the input and output graphs. Because the model is too complex to represent the links among nodes of the input and output graphs, we decide to simplify it.

Figure 3.5 illustrates the semantic data model of Linked Widgets. Conceptually, both the input and the output data models are trees. The root and intermediate nodes are single objects or arrays of objects, whose dimension is specified by the *lw:hasArrayDimension* property. Each object can have multiple properties. Data properties have leaf nodes with primitive values; object properties consist of sub-trees.

The input and output data model defined in Karma are simply graphs. As graph matching is much more complex than tree matching [100, 101], we decide to transform the Karma graph model into a hybrid of graph and tree model. To this end, we identify two types of links, which are (i) parent-child links, and (ii) links between two arbitrarily related nodes. The former is represented by a single triple whereas the latter is represented by three triples, using the SWRL vocabulary (cf. Figure 3.5). The Linked Widgets model hence is a hybrid of tree and graph structure, which is capable of presenting rich information as a graph, and facilitating model-matching as a tree.

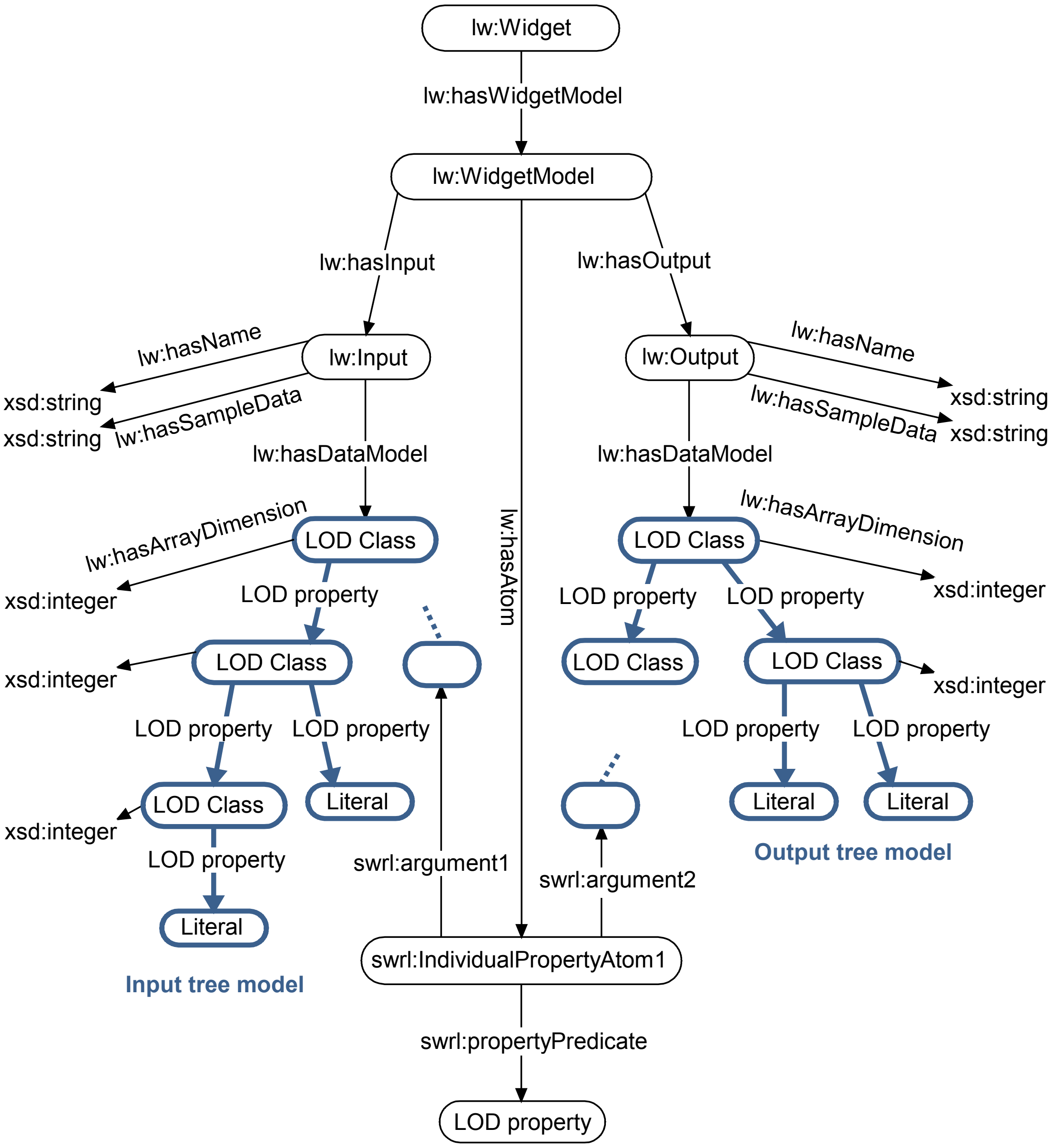
Figure 3.6 illustrates the model of the *Point of Interest (POI) Search* widget that will be used in our sample geospatial data integration use cases (see Section 5.4.4). The widget takes an array of arbitrary objects containing the *wgs84:location* property as input. The range of the property is the *Point* class with two literal properties, i.e., *lat* and *long*. The widget output is an array of GeoNames[14] features satisfying the distance filter specified in the *POI Search* widget.

To specify that input/output is an array of objects, we use the literal property *hasArrayDimension* (0: single element; $n > 0$: n-dimensional array). Because the input of *POI Search* is an "arbitrary" object, we apply the *owl:Thing* class to represent it in the data model.

Using the *lw:hasSampleData* property, we associate the input and output terminals with descriptions that represent their full tree structures in JSON. The example of such information is shown in Listing 3.1. The JSON description does not include the relationships among nodes. It allows us to reconstruct the whole tree model instantly and fosters model matching as presented in Section 3.9.1.

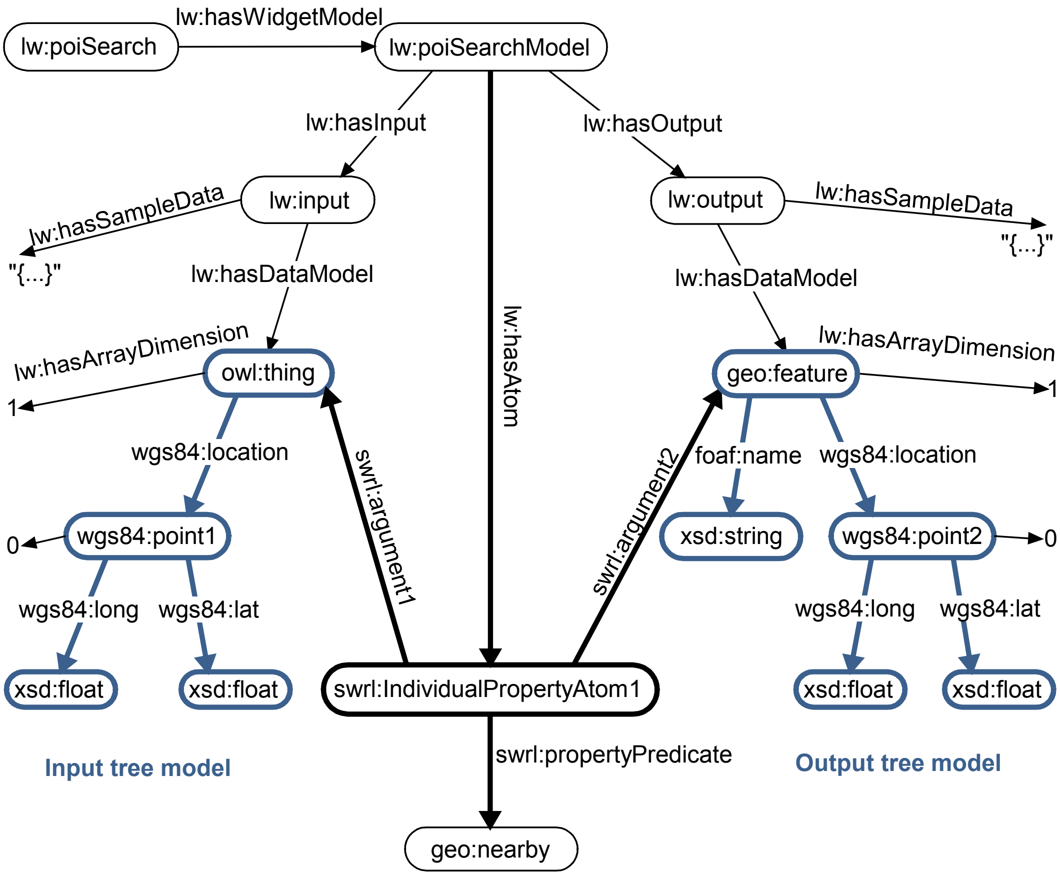
[13] `http://www.w3.org/Submission/SWRL/` (accessed Nov. 01, 2015)

[14] `http://www.geonames.org/` (accessed Nov. 01, 2015)



lw	http://linkedwidgets.org/ontologies/
swrl	http://www.w3.org/2003/11/swrl/
xsd	http://www.w3.org/2001/XMLSchema#

Figure 3.5: Semantic model of Linked Widgets



lw	http://linkedwidgets.org/ontologies/	**wgs84**	http://www.w3.org/2003/01/geo/wgs84_pos#
swrl	http://www.w3.org/2003/11/swrl/	**xsd**	http://www.w3.org/2001/XMLSchema#
foaf	http://xmlns.com/foaf/0.1/	**geo**	http://www.geonames.org/ontology#

Figure 3.6: Semantic model of the POI Search widget

The *point*, *location*, *lat* and *long* terms are available in different vocabularies. However, due to its widespread use, we chose *wgs84*. The *widget annotator* module (cf. Section 3.2), which supports developers in annotating widgets, interactively recommends frequently used terms of the most popular vocabularies to developers. This eases the annotation process and fosters consistency by diminishing the use of different terms to describe the same concepts.

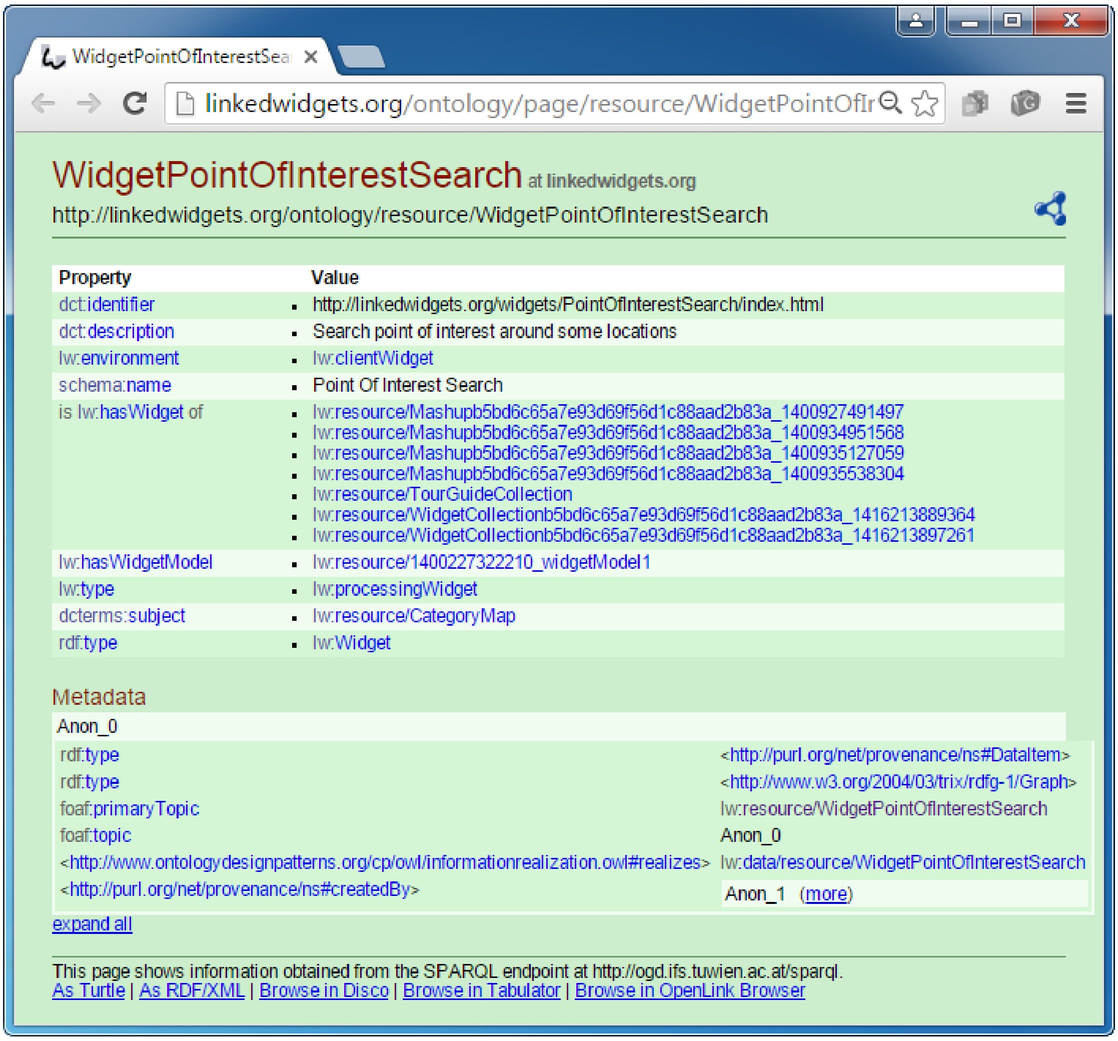
Because we explicitly model the *geo:nearby* relation between the two instances *owl:thing* and *geo:feature* (see Figure 3.6), we know that the output *feature* is nearby the input *location*. If we had used SAWSDL, OWL-S, or WSMO, we would not have been able to specify this input-output relation and hence could not distinguish between this and other possible relations between two locations.

Listing 3.1: Sample JSON structure associated with the input and output models of the *POI Search* widget

```
input = {
   "input": {
      "@context": {
         "lat": "http://www.w3.org/2003/01/geo/wgs84_pos#lat",
         "long": "http://www.w3.org/2003/01/geo/wgs84_pos#long",
         "location": "http://www.w3.org/2003/01/geo/wgs84_pos#location"
      },
      "@graph": [
         {
            "@id": "http://example.com.sampleId",
            "@type": "http://www.w3.org/2002/07/owl#Thing",
            "location": {
               "@id": "http://example.com.sampleId",
               "@type": "http://www.w3.org/2003/01/geo/wgs84_pos#Point",
               "lat": "float",
               "long": "float"
            }
         }
      ]
   }
}

output = {
   "@context": {
      "lat": "http://www.w3.org/2003/01/geo/wgs84_pos#lat",
      "long": "http://www.w3.org/2003/01/geo/wgs84_pos#long",
      "location": "http://www.w3.org/2003/01/geo/wgs84_pos#location"
   },
   "@graph": [
      {
         "@id": "http://example.com.sampleId",
         "@type": "http://www.geonames.org/ontology#Feature",
         "location": {
            "@id": "http://example.com.sampleId",
            "@type": "http://www.w3.org/2003/01/geo/wgs84_pos#Point",
            "lat": "float",
            "long": "float"
         }
      }
   ]
}
```

All widget models are published as LOD and can be accessed using the graph *http://linkedwidgets.org* of the *http://ogd.ifs.tuwien.ac.at/sparql* SPARQL endpoint. Figure 3.7 depicts the human-readable serialization of the *POI Search* widget model.

With SPARQL queries, we can find a widget that receives or outputs an object containing, for instance, geographic information. This is done by means of the *semantic widget search* method presented in Section 5.2.3. Moreover, based on an arbitrary terminal, e.g., the input terminal of the *POI Search* widget, we can find all output terminals that can be connected due to the *semantic terminal matching* method (cf. Section 3.9.1).



WidgetPointOfInterestSearch at linkedwidgets.org

http://linkedwidgets.org/ontology/resource/WidgetPointOfInterestSearch

Property	Value
dct:identifier	http://linkedwidgets.org/widgets/PointOfInterestSearch/index.html
dct:description	Search point of interest around some locations
lw:environment	lw:clientWidget
schema:name	Point Of Interest Search
is lw:hasWidget of	lw:resource/Mashupb5bd6c65a7e93d69f56d1c88aad2b83a_1400927491497 lw:resource/Mashupb5bd6c65a7e93d69f56d1c88aad2b83a_1400934951568 lw:resource/Mashupb5bd6c65a7e93d69f56d1c88aad2b83a_1400935127059 lw:resource/Mashupb5bd6c65a7e93d69f56d1c88aad2b83a_1400935538304 lw:resource/TourGuideCollection lw:resource/WidgetCollectionb5bd6c65a7e93d69f56d1c88aad2b83a_1416213889364 lw:resource/WidgetCollectionb5bd6c65a7e93d69f56d1c88aad2b83a_1416213897261
lw:hasWidgetModel	lw:resource/1400227322210_widgetModel1
lw:type	lw:processingWidget
dcterms:subject	lw:resource/CategoryMap
rdf:type	lw:Widget

Metadata

Anon_0	
rdf:type	<http://purl.org/net/provenance/ns#DataItem>
rdf:type	<http://www.w3.org/2004/03/trix/rdfg-1/Graph>
foaf:primaryTopic	lw:resource/WidgetPointOfInterestSearch
foaf:topic	Anon_0
<http://www.ontologydesignpatterns.org/cp/owl/informationrealization.owl#realizes>	lw:data/resource/WidgetPointOfInterestSearch
<http://purl.org/net/provenance/ns#createdBy>	Anon_1 (more)

expand all

This page shows information obtained from the SPARQL endpoint at http://ogd.ifs.tuwien.ac.at/sparql.
As Turtle | As RDF/XML | Browse in Disco | Browse in Tabulator | Browse in OpenLink Browser

Figure 3.7: Human-readable serialization of the POI Search widget model

3.5 Mashup Construction

Technically, client widgets and the client interface of server widgets are HTML iframes. Such iframes can trigger events, which contain messages. These messages are then consumed by other iframes that have registered a listener for these events. Thus we can implement a library to ease the communication between widgets. The library enables either the *mashup execution coordinator* (which is a critical component that enables widgets running in different environments to cooperate) or widgets to register a remote procedure to be called by each other. Because web browsers do not allow for a direct call from a widget to the other one, the *coordinator* plays the important role to facilitate the information exchange between widgets.

We design the communication interfaces between Linked Widgets and the *coordinator*

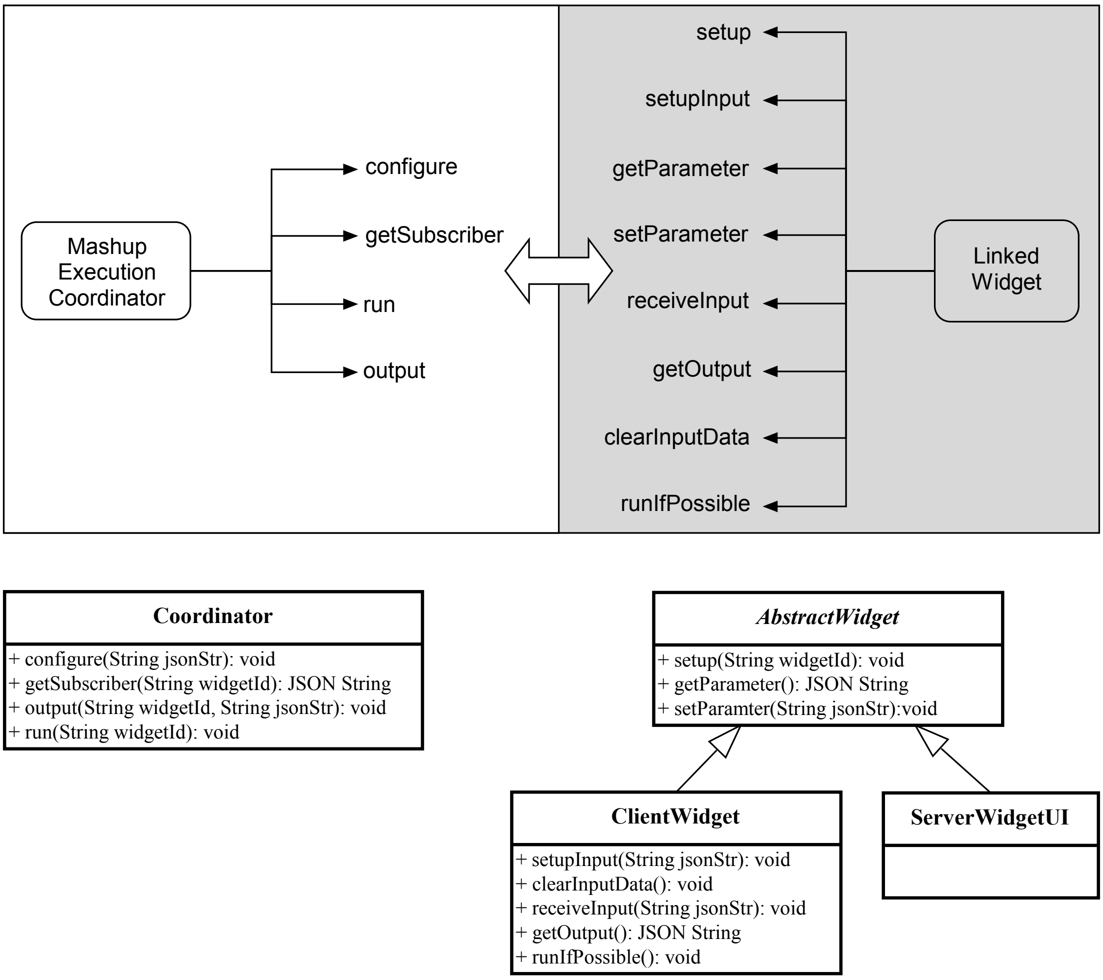

Mashup Execution Coordinator
configure
getSubscriber
run
output
setup
setupInput
getParameter
setParameter
receiveInput
getOutput
clearInputData
runIfPossible
Linked Widget
Coordinator
+ configure(String jsonStr): void
+ getSubscriber(String widgetId): JSON String
+ output(String widgetId, String jsonStr): void
+ run(String widgetId): void
AbstractWidget
+ setup(String widgetId): void
+ getParameter(): JSON String
+ setParamter(String jsonStr):void
ClientWidget
+ setupInput(String jsonStr): void
+ clearInputData(): void
+ receiveInput(String jsonStr): void
+ getOutput(): JSON String
+ runIfPossible(): void
ServerWidgetUI

Figure 3.8: Interfaces between Linked Widgets and the mashup coordinator

as shown in Figure 3.8 and explained in Table 3.3. The interfaces contribute to not only the mashup construction process but also the mashup communication protocols presented in the next section.

To build up a mashup, users drag and drop widgets to the mashup panel. Because widgets are associated with URLs, we can create respective widget pages (which are HTML iframes). When a widget page is completely loaded, the *mashup coordinator* calls the *setup* method of the widget and sets an automatically generated identifier for the widget; the identifier must be unique to differentiate the used widgets.

After receiving its assigned identifier, the widget calls the *configure* method of the *coordinator*. Parameters passed to the method are (i) the widget's input/output configuration and (ii) automatically detected HTML inputs (and their current values) of the widget page. Based on the former information, we can graphically create and display the widget's input/output terminals in the widget interface. The latter information is

Name	**Description**
configure	sends the configuration of a widget (e.g., input, output terminals, and the widget's default size) to the coordinator
getSubscriber	asks the coordinator which widgets are waiting for the output of a widget
run	tells the coordinator a widget wants to run
output	returns the output of a widget to the coordinator
setup	sets an identifier for a widget
setupInput	initializes the input data of a client widget, based on the current wires among widgets
getParameter	asks a client widget or the client interface of a *server widget* for their current parameter settings
setParameter	sets values of the parameters in a client widget or the client interface of a *server widget*. The method is used when we load a saved mashup
receiveInput	sends input data to a client widget. The input data is taken from the output data of the preceding widget of the concerning client widget in the mashup flow
getOutput	asks a client widget for its current output data
clearInputData	clears the input data object of a client widget
runIfPossible	tells a client widget to run if it has no input terminals, or all of its input terminals have been already received data

Table 3.3: Description of mashup communication methods

temporarily stored by the *coordinator*. Once the mashup is saved, the current setting values of the HTML inputs are sent to the *coordinator*. The *coordinator* hence can compare the initial values with the saved values of every single field of all widgets. It then detects and saves the changing fields into the mashup configuration. Consequently, when we load a saved mashup, rather than requiring users to re-enter the mashup parameters, we can automatically set the saved parameters back into widgets, using the *setParameter* method of the Linked Widgets.

As soon as widgets have been instantiated, we can link (or unlink) an input terminal of a widget to the output terminal of another widget. Each time such a link is established (or removed), the input object of the former widget is (re)initialized.

The input data of a widget is dynamic; it is generated based on the number of input terminals of the widget, as well as the number of links of each input terminal to other output terminals. For example, assume that a widget A has two input terminals, namely, "*input1*" and "*input2*". While the terminal "*input1*" is currently linked with two other output terminals, the terminal "*input2*" is linked with only one output terminal. The respective input object of A is then initialized with three empty sub-objects as shown in Listing 3.2. When the mashup is executed, the data received from the output terminals will be filled into these empty sub-objects as illustrated in Listing 3.3. The structure allows us to count the number of input data objects the widget has received from other widgets. If it equals the total number of required input data objects, we can execute the widget.

3.6 Mashup Execution Protocols

This section discusses the communication protocol between widgets used during the execution of a mashup. We design three respective protocols for the three types of mashups, i.e., *local*, *hybrid* and *distributed* mashups. The protocols aim to facilitate efficient communication between independently developed widgets executed on various devices and minimize the framework server load.

3.6.1 Local protocol

Local mashups consist entirely of locally executed *client widgets* that communicate within the client's web browser.

As an example for how the protocol facilitates communication at runtime, consider a mashup with three widgets $A \rightarrow B \rightarrow C$. To construct the mashup, we (i) drag and drop three widgets A, B, and C to the mashup panel, and (ii) link the output terminals of the widget A and B to the input terminals of the widget B and C, respectively.

Figure 3.10 shows all messages transferred between the coordinator and the three widgets. The communication takes place entirely in the client's browser. Because widgets do not know of each other, they have to communicate with the coordinator to obtain their tasks. The first eight messages are delivered during the mashup construction process; they are the prerequisite preparation for the mashup execution.

The mashup execution is triggered as soon as we run the widget C. Typically, the coordinator will clear the input data objects of the three widgets and tell them to execute. A is the only widget that can run in the first phase, because it requires no input data. As soon as its execution is completed, it sends the output to the coordinator, which, in turn, delivers the output to widget B. B can now run as it already has its input data. This process continues until the very last widget (i.e., widget C) receives its input and finishes the execution. Data are transferred between widgets in a successive process.

The local protocol is based on messages and events. This offers a powerful approach for various use cases with rich user interaction. For example, the control flow can be reversed as follows. Assume that A is the *Map Pointer* widget, from which we can define a point on a map. When an event is fired in A, (e.g. users add or delete a Point by clicking on the map), we need to update the final result presented in the widget C. To this end, once A updates its output data, it sends an event to the coordinator by calling the *output* method, which, in turn, updates the input of widget B and informs it to

Listing (3.2) Empty JSON input object

```
input = {
   "input1": [{}, {}],
   "input2": {}
}
```

Listing (3.3) JSON input object when data is filled in

```
input={
  "input1": [
    {
      "@context": {
        "lat": "http://www.w3.org/2003/01/geo/wgs84_pos#lat",
        "long": "http://www.w3.org/2003/01/geo/wgs84_pos#long"
      },
      "@type": "http://dbpedia.org/ontology/Place",
      "location": {
        "@type": "http://dbpedia.org/ontology/Point",
        "lat": 48.2045,
        "long": 16.3806
      }
    },
    {
      "@context": {
        "lat": "http://www.w3.org/2003/01/geo/wgs84_pos#lat",
        "long": "http://www.w3.org/2003/01/geo/wgs84_pos#long",
        "location": "http://www.w3.org/2003/01/geo/wgs84_pos#location"
      },
      "@id": "http://linkedgeodata.org/triplify/node1526007218",
      "label": "Alma Mahler-Werfel-Park",
      "@type": [
        "http://geovocab.org/spatial#Feature",
        "http://linkedgeodata.org/meta/Node",
        "http://linkedgeodata.org/ontology/Leisure",
        "http://linkedgeodata.org/ontology/Park"
      ],
      "location": {
        "@type": "http://www.w3.org/2003/01/geo/wgs84_pos#Point",
        "lat": 48.1966,
        "long": 16.3951
      }
    }
  ],
  "input2": {
    "@context": {
      "lat": "http://www.w3.org/2003/01/geo/wgs84_pos#lat",
      "long": "http://www.w3.org/2003/01/geo/wgs84_pos#long"
    },
    "@type": "http://dbpedia.org/ontology/Place",
    "location": {
      "@type": "http://dbpedia.org/ontology/Point",
      "lat": 48.2239,
      "long": 16.34375
    }
  }
}
```

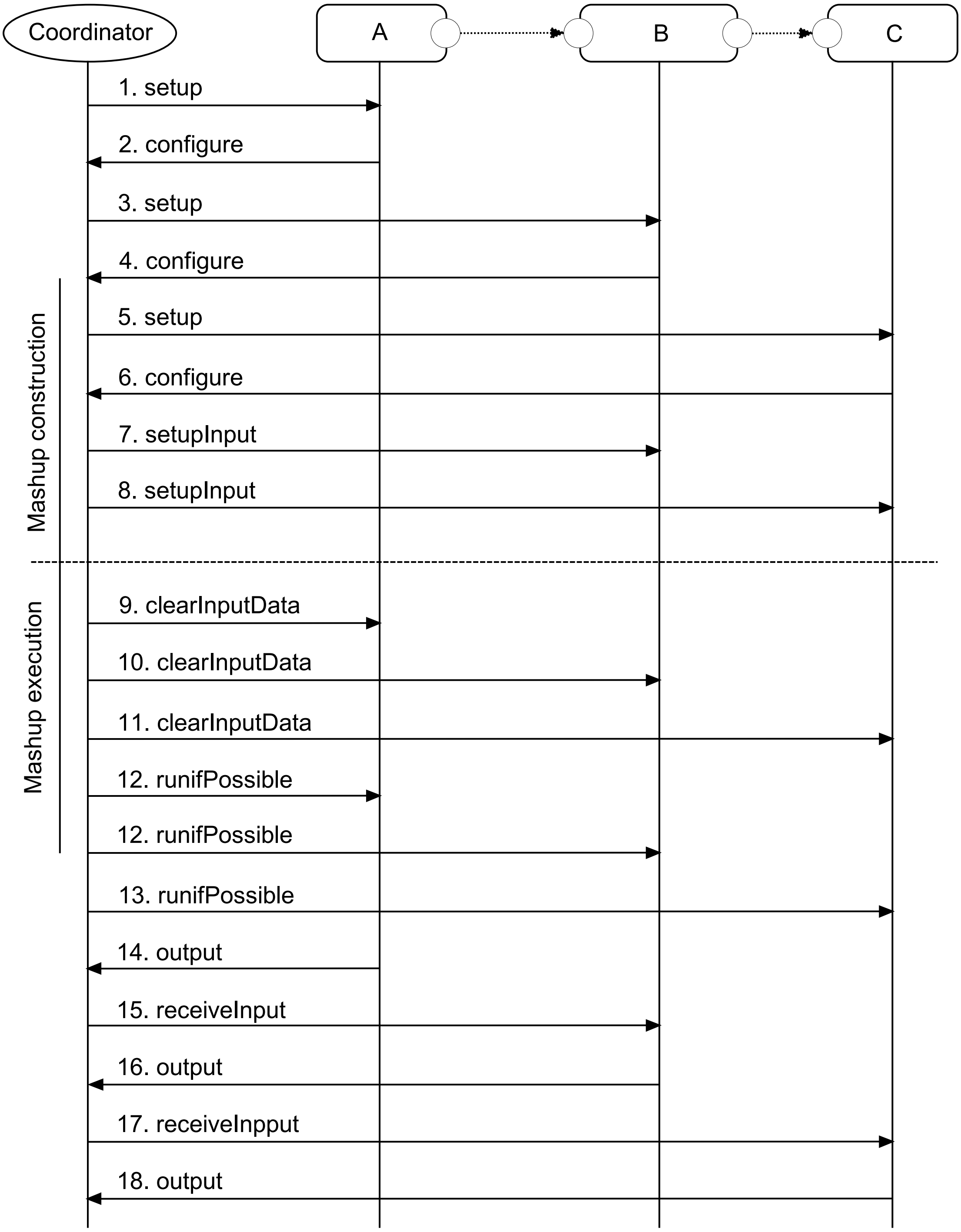


Figure 3.10: Local mashup communication protocol

perform the *execution* function again. Similarly, C runs and finally visualizes the updated data.

The client mashup coordinator is executed locally; this implies that after the mashup has been constructed, the framework server is no longer needed since the coordinator and widgets – which both reside in the client browser – compute the tasks. Framework resources are only required to initiate mashups, but are not used during mashup execution. A client widget can use third party services to collect and/or process data. This reduces server load and improve the performance.

3.6.2 Remote protocol

In order to support *distributed mashups*, which consist of a local visualization widget and a number of remote widgets that may be distributed among nodes, a remote protocol is necessary.

Protocol Specification

For the remote mashup communication protocol, we use the publish/subscribe model and a coordination server. To explain the protocol, consider a sample mashup with four widgets (cf. Figure 3.11): (i) S_1, a server data widget, which runs on a personal computer to get data from a file; (ii) S_2, another server data widget, which runs on an Android phone to obtain its data, e.g., call logs; (iii) S_3, a server processing widget, which runs on a web server; (iv) and C_1, a client visualization widget.

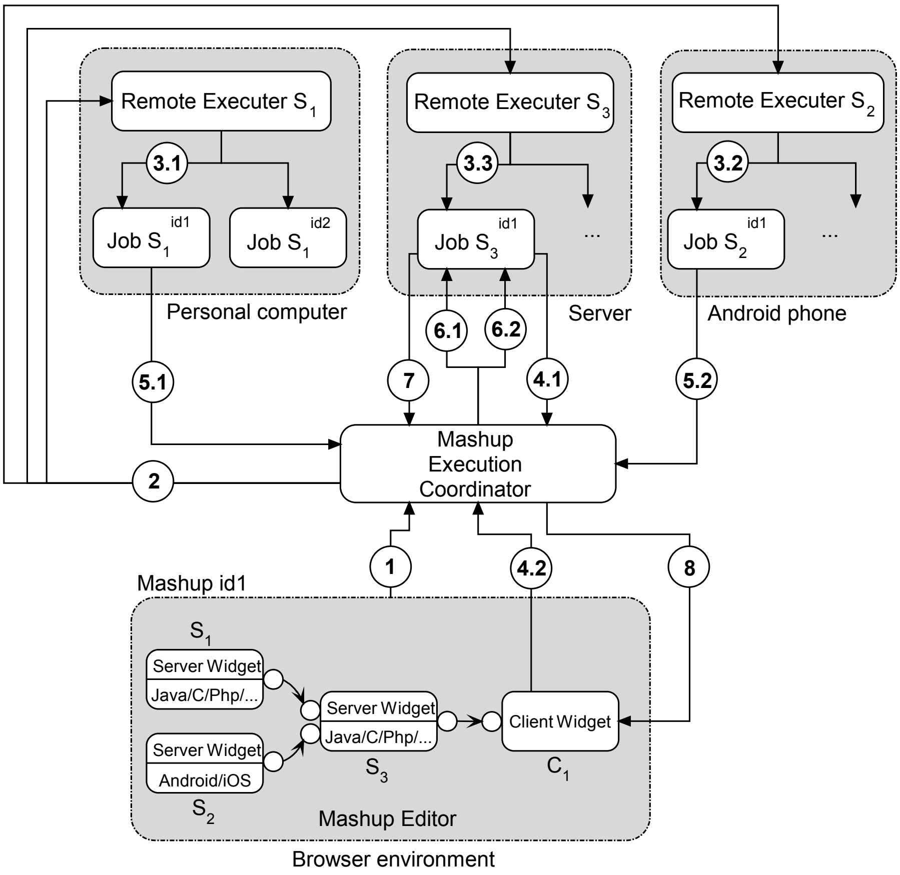
To differentiate multiple instances of a server widget used in multiple mashups, we associate each mashup with a Universally Unique Identifier (UUID), e.g., *id1* in our example.

As soon as a user triggers the execution of a mashup, the framework collects the parameters for all server widgets and forwards them to the coordinator (Step 1 in Figure 3.11). These parameters are (i) the list of URIs of widgets used; (ii) the configuration of the mashup, i.e., all connections between an output terminal of a widget and an input terminal of another widget; (iii) the parameters set by the user in the widget user interfaces (as $parameter, value$ pairs).

Next, the coordinator sends run requests to the *remote executors* of S_1, S_2, and S_3 (Step 2). Details on these executors are provided in Section 3.3.2. Each request contains widget parameters and an identifier of the event the widget should subscribe to. Based on this information, each executor can instantiate a widget job (Step 3).

In our example, we have three such jobs, i.e., S_1^{id1}, S_2^{id1}, and S_3^{id1} for the mashup identified by the UUID *id1*. Next, S_3^{id1} needs to subscribe to the output event of S_1^{id1} and S_2^{id1}, because S_3 requests the output data of S_1 and S_2 as its input data. Similarly, C_1 subscribes to the output event of S_3^{id1} (Step 4).

Jobs S_1^{id1} and S_2^{id1} are executed immediately with the parameters sent from the requests before, because S_1 and S_2 do not need input data from any other widget (Step 5).



1. The editor sends mashup configuration and widget parameters

2. Coordinator requests S$_1$, S$_2$, S$_3$ to initialize widget instances for mashup id1

3.1 S$_1$ initinilizes S$_1^{id1}$ instance

3.2 S$_2$ initinilizes S$_2^{id1}$ instance

3.3 S$_3$ initinilizes S$_3^{id1}$ instance

4.1 S$_3^{id1}$ subscribes to S$_1^{id1}$ output and S$_2^{id1}$ output events

4.2 C$_1$ subscribes to S$_3^{id1}$ output event

5.1 S$_1^{id1}$ runs and publishes S$_1^{id1}$ output event

5.2 S$_2^{id1}$ runs and publishes S$_2^{id1}$ output event

6.1 Coordinator forwards the S$_1^{id1}$ output event to S$_3^{id1}$
S$_3^{id1}$then has output of S$_1^{id1}$

6.2 Coordinator forwards the S$_2^{id1}$ output event to S$_3^{id1}$
S$_3^{id1}$then has output of S$_2^{id1}$

7. S$_3^{id1}$ collects all input. It runs and publishes S$_3^{id1}$ output event

8. Coordinator forwards the S$_3^{id1}$ output event to C$_1$
C$_1$ then has output of S$_3^{id1}$ and visualizes it

Figure 3.11: Remote mashup communication protocol

When these jobs are finished, they publish output events to the coordinator. The coordinator then sends the output to the jobs that have subscribed to the respective output events, i.e., from S_1^{id1} and S_2^{id1} to S_3^{id1} (Step 6). As soon as S_3^{id1} has received its two inputs, it is executed and publishes an output event to the coordinator (Step 7). Finally, because C_1 has subscribed to this event, it receives and visualizes the final data in the browser (Step 8).

This protocol ensures that a user can close the browser and reopen a mashup that is being executed remotely. Upon reopening a *distributed mashup*, the client visualization widget immediately requests and displays the current output data of its predecessor server widget(s). Furthermore, the visualization widget listens for output events and updates its display immediately whenever new data arrives.

Protocol Implementation Consideration

We have identified two major architectural options for implementing the remote mashup protocol. The first uses web services; the second is to implement it on top of a WebSocket [102] infrastructure. We will discuss each of these options in the following.

If we follow the web service approach, the coordinator is a collection of services that server widgets use for communication. A *remote executor* that runs on a web server, can provide an *execution service* and an *output service.* Therefore, we have a bi-directional communication channel between the *remote executor* and the coordinator.

Based on that, the publish/subscribe model can be set up as follows: First, the coordinator calls the *execution service* of S_1 and the job S_1^{id1} is performed. When this job has been completed, it sends the coordinator the token used to receive its output data. The coordinator, in turn, calls the *execution service* of S_3 and sends this token as a parameter to S_3^{id1}. This job uses the token to call the *output service* of S_1^{id1} to load the data. Data, hence, is transferred between and processed inside the *remote executors* themselves. This reduces the server load, because the task of the coordinator is to call the *execution services* only. It does not perform any data processing tasks, neither does it interact with the intermediate widget output data.

However, the web service-based approach is ill-suited for server widgets running in environments such as mobile phones. We should not deploy web services on such devices because of their restricted computing resources and the connectivity requirement (e.g., we need to open a port). This would result in uni-directional communication channels. To simulate bi-directional connections and the publish/subscribe model, we would have to use polling or long-polling. However, this approach has unfavorable scaling characteristics and generates large amounts of unnecessary network traffic when a large number of server mashups are executed concurrently. Furthermore, the coordinator represents a bottleneck in this scenario.

The second potential implementation approach is based on WebSockets. The approach provides full-duplex communication channels over a single *Transmission Control Protocol* connection. Although the technology was conceived as a communication channel between web browsers and web servers, they can be natively implemented in various programming environments and on various devices.

Once a WebSocket connection is established, data frames can be sent back and forth between the client and the server in full-duplex mode. This eases interaction between clients (e.g., browsers) and servers. A server can send content to the browser and allowing for messages to be passed back and forth while keeping the connection open. As a result, we have a two-way communication channel and can easily implement the protocol.

Following the WebSocket approach, the *remote executors* are essentially WebSocket clients, and the coordinator is a WebSocket server. A potential disadvantage of this architecture is that *remote executors* cannot directly transmit data to each other on their own. All data needs to be passed through the WebSocket server. The coordinator therefore is the bottleneck of the architecture. To tackle this problem, we can deploy multiple WebSocket servers and make use of load balancing methods [103]. Furthermore, server widgets that run on a web server may return the URL associated with a token rather than the complete data to the coordinator; the coordinator then forwards the URL to subsequent widgets which may use it to download the output data.

Overall, we opted for the WebSocket approach due to its advantages, which include (i) lower latency compared to HTTP connections, (ii) lower amount of data transferred, (iii) the wide range of supported languages, which provide the basis for various computing environments for server widgets.

3.6.3 Hybrid protocol

Hybrid mashups are an extension of *distributed mashups. Hybrid mashups* do not require that all *data* and *processing widgets* are *server widgets*, i.e., client widgets can be used anywhere in a *hybrid mashup.*

The communication protocol for *hybrid mashups* is similar to the remote protocol. If the predecessor of a client widget is a server widget, it needs to subscribe to the output data event of the *server widget.* If its successor is a server widget and when it returns output data to the coordinator, the coordinator will publish an output event so that the server widget can receive the output data.

3.7 Hybrid Mashup Patterns

This section presents five combination patterns of *client widgets* and *server widgets.* Each pattern is useful in particular data integration scenarios.

3.7.1 Collaborative Mashups

Definition

A collaborative mashup is a type of mashup application that is created and/or operated by more than one user at the same time.

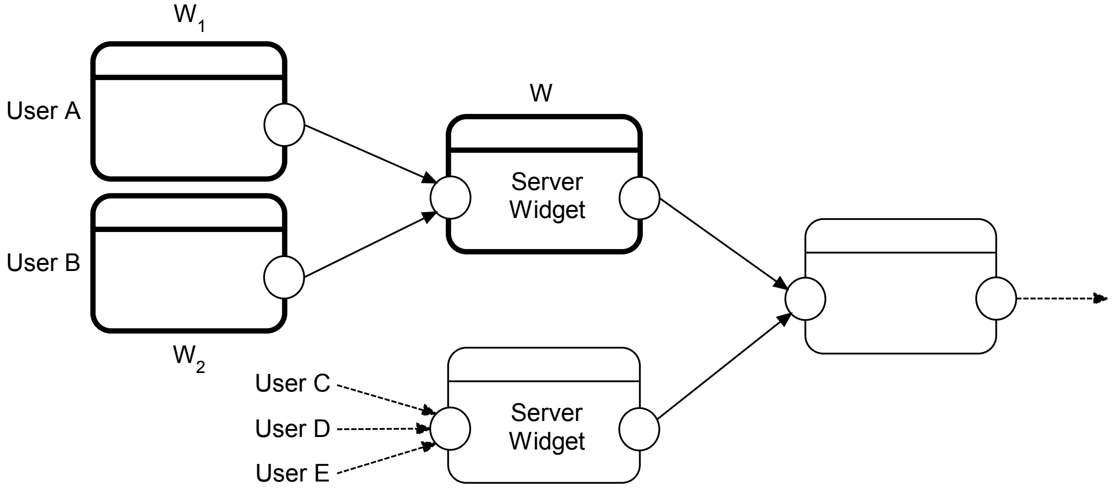


Figure 3.12: Collaborative mashup pattern

Use Case

Collaborative mashups are useful for a group of users. As soon as they agree on a data processing flow defined in a mashup, each participant can update her input data at any time so that all can immediately get the live presentation of the combined data. Tedious and repetitive manual data integration processes (e.g., data cleansing, data uploading, change notifying) are encapsulated in widgets.

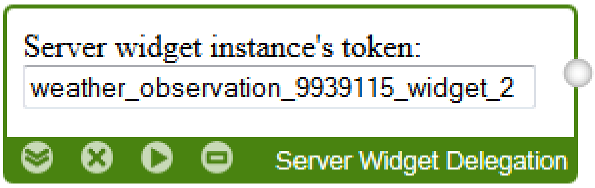
Consider, for example, the simple task of scheduling a meeting between users whose calendars are spread among computers, mobile phones, Cloud services, etc. A widget-based collaborative workflow would allow participants to selectively contribute their calendar using *server widgets* such as locally executed Apps or Cloud-based calendar widgets. They could then simply merge their calendar widgets in a collaborative mashup to identify available timeslots.

Pattern

The Linked Widgets combination pattern of collaborative mashups is presented in Figure 3.12. It involves at least one server widget (e.g., W) and two users; the two users are responsible for their two widgets (e.g., W_1 and W_2), which can be either client or server widgets.

To execute the collaborative mashup, in the first step, an arbitrary participant triggers the run action in the visualization widget. This will also trigger the execution of all preceding widgets (except W_1 and W_2), one after the other. The server widget W, however, cannot run yet because it still waits for the output data of W_1 and W_2.

In the next step, the two users set the values of widget parameters and run their widgets. As soon as both widgets return output data, the server widget W performs



Server widget instance's token:

weather_observation_9939115_widget_2

Server Widget Delegation

Figure 3.13: Delegating widget

the data integration task and returns the result to its succeeding widget. According to the mashup execution protocol presented in Section 3.6, whenever W_1 or W_2 submits its new output data, the final result is recalculated and immediately presented to every participant in the synchronized visualization widget.

A participant can leave the collaborative mashup at any time. To this end, she removes her widget from the mashup so that the input data of the server widget W is reset, and her private data is removed from the mashup. On the other hand, a new participant can easily join the collaborative mashup by entering the collaborative token to load the mashup and adding her private widget into the collaborative mashup editor.

Moreover, we provide each participant with the *delegating widget* (cf. Figure 3.13) that can be persistently connected with an instance of a server widget. When we run this delegating widget, it subscribes to the "*returning output*" event of the server widget instance whose token is specified in the input box. The delegating widget acts as an agent for the server widget instance, meaning that as soon as the server widget returns its new output data, the delegating widget receives the data and immediately returns the same result. The delegating widget is designed for collaborative mashups as follows:

(i) It first allows a participant to prepare her own data for a collaborative mashup. Rather than exposing a large volume of data, she can extract a relevant part of data only, and perform some pre-processing task in a private mashup. She then contributes the data of such mashup to the collaborative group by using a delegating widget. This not only makes data integration more secure (as her data may contain sensitive data that should be removed) but also speeds up the execution of the whole mashup (as the irrelevant part of data is already removed in her private mashup). The private mashup is not a part of the collaborative mashup, but its final output data is used in the collaborative mashup via the delegating widget.

(ii) The delegating widget allows a participant to hide a branch of the collaborative mashup. The Linked Widgets framework enables anybody to develop her own widget; she then can contribute it to the public community or keep it private. A private widget can be used in a collaborative mashup; it is visible to everyone in the shared mashup screen. The hiding feature is hence useful if a participant does not want to expose her private widget used in the collaborative mashup to others. Moreover, it simplifies the overall mashup; each should only see the relevant part of the mashup rather than the part that she can ignore or cannot control.

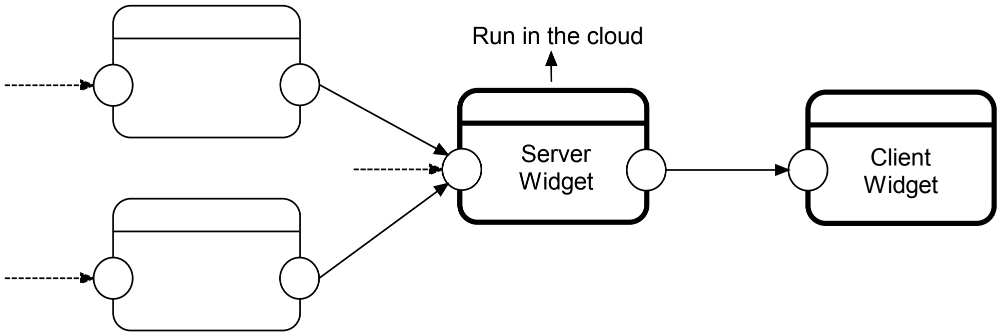


Figure 3.14: Persistent mashup pattern

3.7.2 Persistent Mashups

Definition

A persistent mashup is a type of mashup application that can continuously run in the background and maintain its status and intermediate data.

Use Case

A persistent mashup can be used for data integration tasks that are typically time-consuming (e.g., statistical analysis, data analysis, and data reporting). To this end, the mashup creator composes a mashup, submits her input data, and does not have to care about it any more. At any time, she can reopen the mashup to check the current status and result. As she does not host and run the mashup on her device, she can manage a mashup that performs heavy calculation tasks even with a slow client device. During the waiting time for the calculation, she can turn off her device, or do other work.

Pattern

Figure 3.14 shows the Linked Widgets combination pattern of persistent mashups. It contains a server widget placed before a visualization widget, which is a client widget. Because the server widget performs the processing tasks in the respective server rather than the browser environment, it can persistently maintain the calculation and the intermediate mashup data.

As soon as the mashup is reopened, the client widget requests the latest output data from the server widget and visualizes it. The browser hence acts as a front-end tool that shows up-to-date data from the back-end processing.

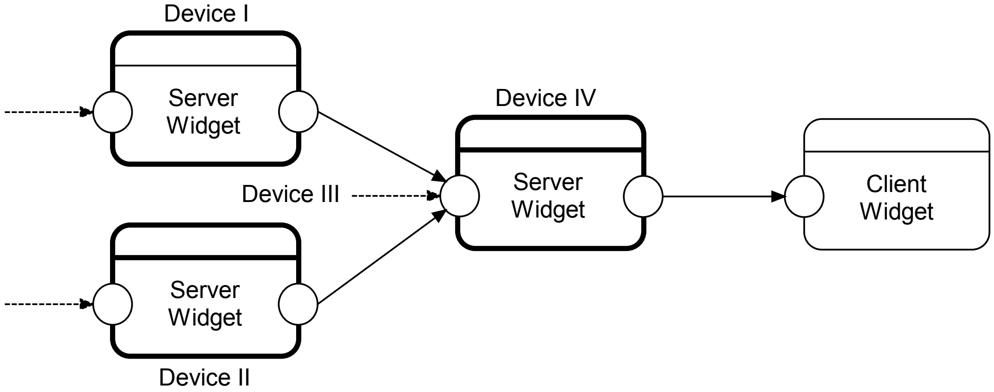


Figure 3.15: Distributed mashup pattern

3.7.3 Distributed Mashups

Definition

A distributed mashup is a type of mashup application in which the involved widgets are hosted in distributed nodes and devices.

Use Case

Distributed mashups first can facilitate data integration of sensors and embedded devices. The data collector tasks run pervasively among server widgets of distributed nodes. To facilitate data integration, those server widgets can clean, formalize, and convert the data into the semantic format before sending it to a central node (which is also a server widget) where data is aggregated before finally being visualized in a client widget. The distributed model typically involves three types of nodes: (i) embedded devices (which provide input data), (ii) a powerful server (which processes data), and (iii) a personal device such as a mobile phone, tablet, or laptop for visualization.

Distributed mashups, moreover, allow us to integrate data from different devices without the necessity to upload the data into a data center. To enrich the data, we can flexibly integrate our local data with open data and LOD datasets by adding widgets into the mashup.

Pattern

Figure 3.15 depicts the widget combination pattern of distributed mashups. The involved server widgets are placed in an arbitrary position of the mashup. Based on the available programming language (e.g., Java, Python, Erlang, C++, etc.) of the hosting device, we implement the respective versions of the server widgets. By adding (removing) a widget into (from) a mashup, we can add (remove) the node into (from) the ad-hoc architecture. It is required that the device is connected to the internet, so that widgets can communicate in the *remote protocol*.

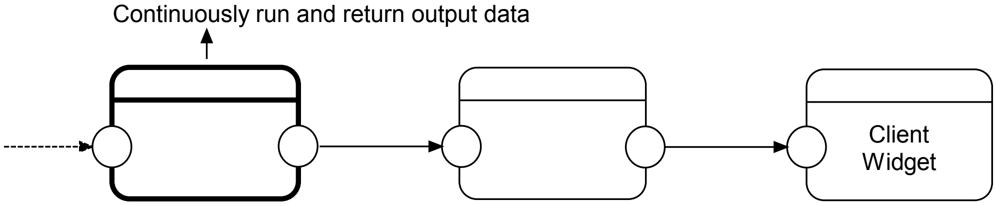


Figure 3.16: Streaming mashup pattern

3.7.4 Streaming Mashups

Definition

A streaming mashup is a type of mashup application in which data flows continuously from a widget to others in the mashup.

Use Case

Streaming mashups can be used for data monitoring use cases. While currently integrated data is presented to users, new data is constantly generated, delivered, and aggregated for presentation. Due to the variety of real-time and streaming data sources available on the web (e.g., weather, public transportation, stock quotes) and the easy-customized feature of mashups, each can build up her own application to process daily data and support decision making. For example, she can compose a mashup that collects the temperature, the pressure, and the wind speed in different places (e.g., her home, her work place) every minute, based on the weather conditions of the nearest Wunderground stations (out of 140,000 stations all over the world). She then can visualize the aggregated data in a chart, which is updated every minute. This example is presented in Section 5.5.1.

Pattern

The Linked Widgets' streaming mashup pattern is illustrated in Figure 3.16. At least one widget continuously runs and returns its output data. We use the term *streaming widget*, which can be either a client or a server widget. A streaming client widget can be used, but the streaming data flow of the mashup will be stopped once we close the browser; a streaming server widget should be used if we intend to make our streaming mashup persistent.

3.7.5 Complex mashups

The purpose of classifying hybrid mashup patterns is to clarify and emphasize different aspects of mashups only; there is no clear boundary between *collaborative*, *persistent*, *distributed*, and *streaming* mashups. We can combine these types in various ways. For example, we can construct mashups that continuously integrate streaming data from distributed sensors of multiple stakeholders.

3.8 Mashup Encapsulation

The basic idea of our approach is to foster reuse of functionality encapsulated inside Linked Widgets. Moreover, we enable users to reuse a created mashup as a new widget without programming. By using both client and server Linked Widgets, there are two mechanisms as follows:

1. Making use of the *delegating widget* (cf. Section 3.7.1 and Figure 3.13), we can use the output of an arbitrary sever widget of an existing mashup as the data source for a new mashup. This not only serves well for collaborative use cases but also simplifies complex mashup composition.
2. We can compose a mashup (which can be a client mashup, server mashup, or hybrid mashup) and save it as a new widget. We name this special type of widget a "*mashup widget*". A *mashup widget* operates in the same way as an ordinary widget does. The *mashup widget* also has an associated URI and a semantic model (which is the same as the model of the last widget lying in the mashup flow).

To define the user interface of the *mashup widget*, in the first step, all HTML inputs (and their labels) of every widget used in the mashup are automatically detected. Next, the default value and the new label for the detected fields can be set; the unnecessary fields can be hidden. These configurations are associated with the annotation of the *mashup widget*, which is finally published as Linked Data.

When a *mashup widget* is dragged and dropped into the mashup panel, the widget configurations are loaded to generate the *mashup widget's* interface. As soon as the mashup (which contains the *mashup widget*) is finished editing and executed, the communication process between ordinary widgets and the *mashup widget* should be performed as usual (cf. Section 3.6). To this end, we define the *execution function* of a *mashup widget* as the whole process of executing the mashup from which the *mashup widget* is derived. This allows us to have multi-level nested mashups.

Figure 3.17 illustrates an example that uses a *mashup widget.* The execution of the *mashup widget* starts with the execution of W_1 and W_2 and ends with the execution of W_4. Because W_4 (which is the last widget of the internal mashup) is a visualization widget, the output data of the *mashup widget* is the input data of W_4; assume that W_4 is a processing widget, the output data of the *mashup widget* would be the output data of W_4.

3.9 Automatic Data Integration

Today, a large and fast-growing number of ready-to-use open datasets and services are available. For an individual application or a single developer, however, it is difficult to make use of these resources; we need a collaborative environment to use the power of the community.

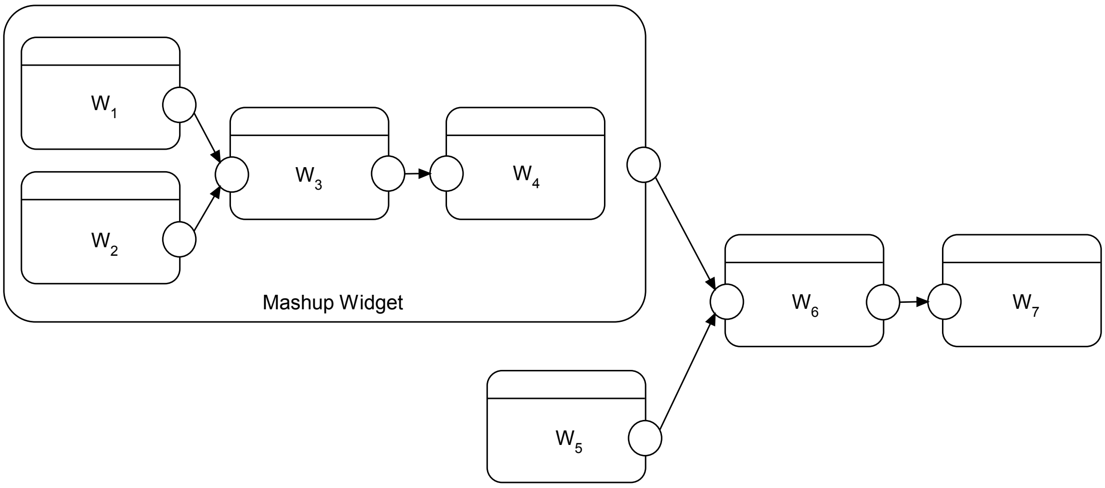


Figure 3.17: Nested mashup

The framework is not tailored towards particular datasets, but integrates data from arbitrary sources. Available widgets can also be combined with widgets that may become available in the future; currently available data can be merged with future data without modifying the existing applications. More importantly, the framework facilitates data integration in a quick and flexible manner as we can simply add or remove a data source to facilitate new use cases. In the following two subsections, we outline how the framework supports data integrators in performing data exploration and automatic data integration tasks.

3.9.1 Semantic Terminal Matching

Because widgets collect, process or visualize data from a source, we can consider widgets as representations of the data itself. Once a widget has been added to a mashup, semantic *terminal matching* allows the user to explore additional widgets that are relevant in a given context. To this end, we query widgets that can be connected to the input and output terminal(s) of a given widget. For instance, we can determine that we are able to connect the output terminal of *POI search* with the input terminal of *Weather Forecast* and hence integrate geospatial data from Linked Geo Data with weather data from Wundergound.

Let i and o denote the root nodes of the input and output tree models (cf. Section 3.4.3), respectively. There are three preconditions for matching input and output models:

1. the RDF classes of i and o must be identical, or the RDF class of i must be a sub-class of that of o;
2. any child of i must correspond to a child of o (i.e., the set of properties required by the input must be a subset of properties provided by the output);

Listing 3.4: A SPARQL query for terminal matching

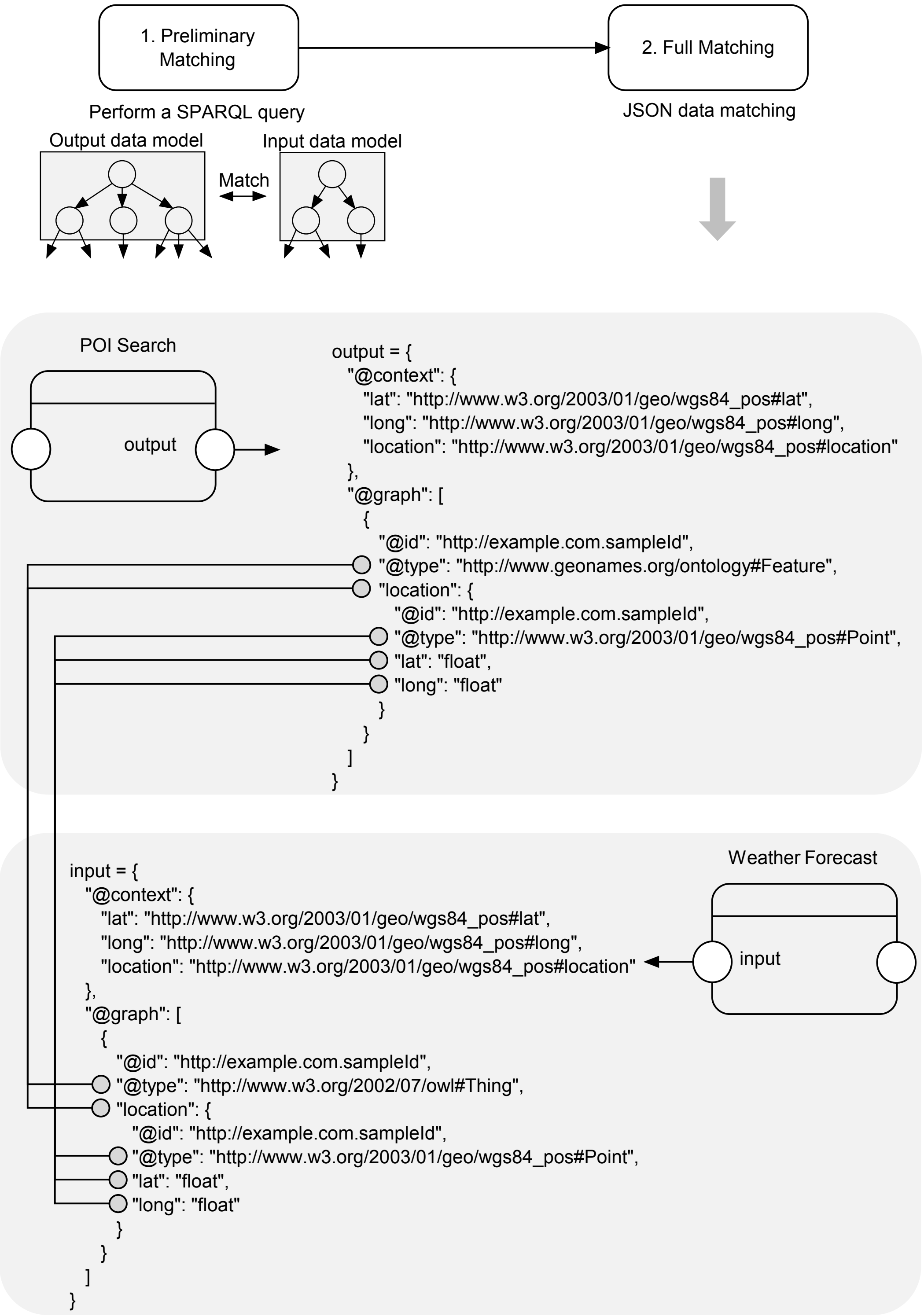
```
PREFIX lw: <http://linkedwidgets.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?iTerminalName ?iWidget ?iJSONData ?oJSONData WHERE {
  <http://linkedwidgets.org/resource/WidgetPOISearch>
                 lw:hasWidgetModel  ?oWModel.
  ?oWModel lw:hasOutput
                 [lw:hasName "output"^^xsd:string;
                 lw:hasDataModel ?oDataModel].
  ?oDataModel a ?type.
  ?oDataModel lw:hasArrayDimension ?dimension.
  ?oDataModel lw:hasSampleData ?oJSONData.

  ?iWidget lw:hasWidgetModel  ?iWModel.
  ?iWModel  lw:hasInput
                 [lw:hasName ?iTerminalName;
                 lw:hasDataModel ?iDataModel].
  ?iDataModel a [rdfs:subClassOf ?type].
  ?iDataModel lw:hasArrayDimension ?dimension.
  ?iDataModel lw:hasSampleData ?iJSONData.

  FILTER NOT EXISTS{
    ?oDataModel ?property ?oValue.
    FILTER NOT EXISTS {?iDataModel ?property ?iValue.}
  }
}
```

3. recursively, the data model of the input object property must match with the data model of the corresponding output object property.

The process of *terminal matching* comprises two steps: *preliminary matching* and *full matching* (cf. Figure 3.18). *Preliminary matching* checks the first two requirements via a single SPARQL query (cf. Listing 3.4).

The query, moreover, allows us to get the JSON sample data associated with the input and output tree data model. *Full matching* hence can check the third requirement by comparing the JSON sample data of the input with that of the output. This is efficient because we do not have to recursively execute a large number of SPARQL queries to build the input and output trees, which is typically time-consuming and inefficient. This pragmatic approach, however, does not allow for semantic processing during the full matching; for example, because we cannot use ontology alignment techniques to match the class (property) of an input node with that of another output node, the classes and properties must be exactly the same in a successful match.

As the example shown in Figure 3.18, the output data model of the *POI Search* is matched with the input data model of the *Weather Forecast* because:



```
output = {
  "@context": {
    "lat": "http://www.w3.org/2003/01/geo/wgs84_pos#lat",
    "long": "http://www.w3.org/2003/01/geo/wgs84_pos#long",
    "location": "http://www.w3.org/2003/01/geo/wgs84_pos#location"
  },
  "@graph": [
    {
      "@id": "http://example.com.sampleId",
      "@type": "http://www.geonames.org/ontology#Feature",
      "location": {
        "@id": "http://example.com.sampleId",
        "@type": "http://www.w3.org/2003/01/geo/wgs84_pos#Point",
        "lat": "float",
        "long": "float"
      }
    }
  ]
}
```

```
input = {
  "@context": {
    "lat": "http://www.w3.org/2003/01/geo/wgs84_pos#lat",
    "long": "http://www.w3.org/2003/01/geo/wgs84_pos#long",
    "location": "http://www.w3.org/2003/01/geo/wgs84_pos#location"
  },
  "@graph": [
    {
      "@id": "http://example.com.sampleId",
      "@type": "http://www.w3.org/2002/07/owl#Thing",
      "location": {
        "@id": "http://example.com.sampleId",
        "@type": "http://www.w3.org/2003/01/geo/wgs84_pos#Point",
        "lat": "float",
        "long": "float"
      }
    }
  ]
}
```

Figure 3.18: Terminal matching

1. The output *geo:Feature*[15] is a subclass of the input *owl:Thing*; both the output and the input are a 1-dimensional array.
2. The output property set (which is {*wgs84:location*[16]}) is the same as the input property set.
3. The input JSON data fits the output JSON data, as the values of the key "*location*" of the two objects are the same.

3.9.2 Automatic Mashup Composition

A mashup, essentially, represents an integration of data from multiple sources; automatic mashup composition hence can be seen as a type of automatic data integration. Despite the similarities between mashup composition and service composition, we can automatize the former more easily than the latter because every widget is associated with a semantic model.

Leveraging the *terminal matching* mechanism, we know what widgets can be connected. This allows us to design an algorithm to determine all possible ways to integrate data based on a specified list of data sources and related Linked Widgets. To design an algorithm for such a process, we first formalize Linked Widgets and related concepts.

Let W denote a set of available widgets.

$$W = \{w_1, \ldots, w_m\} \tag{3.1}$$

Each widget $w \in W$ is represented by a quadruple (I, P, O, C) with a finite set of input data models I for n input terminals, an output data model O, a configuration model C where all widget parameters are defined, and a processing function P. We have not added C to the semantic model of Linked Widgets yet and consider this as future work.

$$I = \{I_1, \ldots, I_n\} \tag{3.2}$$

$$P : I_1 \times \ldots \times I_n \times C \rightarrow O \tag{3.3}$$

Whereas *processing widgets* require that all elements are non-empty (i.e., $I, P, O, C \neq \emptyset$), *data widgets* have no input (i.e., $I = \emptyset$) and *visualization widgets* have no output (i.e., $P, O = \emptyset$).

Next, n_k denotes the number of input terminals for widget $w_k \in W$. O_* is a set of output terminals of all widgets in W where o^k denotes the output terminal of w_k; and I_* is the set of input terminals of all widgets in W.

$$O_* = \{o^1, o^2, \ldots o^m\} \tag{3.4}$$

$$I_* = \{i_1^1, i_2^1, \ldots, i_{n_1}^1\} \cup \ldots \cup \{i_1^m, i_2^m, \ldots, i_{n_m}^m\} \tag{3.5}$$

[15] `http://www.geonames.org/ontology#Feature` (accessed Nov. 01, 2015)

[16] `http://www.w3.org/2003/01/geo/wgs84_pos#location` (accessed Nov. 01, 2015)

Let $W^{'} \subseteq W$ denote a set of widgets used in a *complete mashup* composed from W. *Complete* in this context means that all terminals must be wired. A *complete mashup* is then a pair $(W^{'}, SE)$ where SE is a set of edges; each edge is in form of (o^l, i_k^q), representing the link between output terminal of w_l and input terminal k of w_q.

We require $o^l \in O_*^{'}$, and $i_k^q \in I_*^{'}$ such that $l \neq q$, because the output terminal of a widget cannot be connected to its own input terminal. Furthermore, for each $i_k^q \in I_*^{'}$, there is exactly one corresponding pair in SE. In other words, each input terminal must be connected to exactly a single output terminal.

To specify the automatic mashup algorithm, we perform a pre-processing step to construct a weighted directed graph $G = (V, E)$ for the set of available widgets W. The vertex set V consists of all input and output terminals of all widgets.

$$V = O_* \cup I_* \tag{3.6}$$

$$E = IE \cup EE \tag{3.7}$$

$$IE = \{(o^m, i_1^m, 0), \ldots, (o^m, i_{n_m}^m, 0), \forall w_m \in W\} \tag{3.8}$$

$$EE = \{(o^l, i_k^q, 1), l \neq q, \forall w_l, w_q \in W \, such \, that \, O^l \, match \, I_k^q\} \tag{3.9}$$

The edge set E is the union of IE and EE where IE is the set of internal edges, and EE is the set of external edges. Internal edges connect input and output terminals of the same widget; external edges represent valid connections between the output and input terminals of two widgets. To differentiate between these two types of edges, the weight of internal edges is zero whereas the weight of external edge is one. We use the *terminal matching* algorithm to populate EE.

The direction of external edges is from the input terminal to the output terminal; the direction of internal edges is from the output terminal to input terminal. Due to this directionality rule, widget connections are equivalent to paths in the graph, e.g., a connection between w_1 and w_2 can be represented by the path $o^2 \rightarrow i_1^2 \rightarrow o^1$ (where o^k and i_m^k are the output terminal and the input terminal number m of widget w_k, respectively). Figure 3.19 shows an example graph of seven widgets.

We define a mashup branch as a part of a mashup that consumes (provides) data for a specific output (input) terminal, respectively. Based on that definition, we propose three types of the auto-composition algorithms from a given set W of available widgets:

1. list all complete mashups that contain a particular widget $w_m \in W$,
2. construct all possible complete mashup branches for a particular input or output terminal of $w_m \in W$, and
3. construct all complete branches for all input terminals of $w_m \in W$.

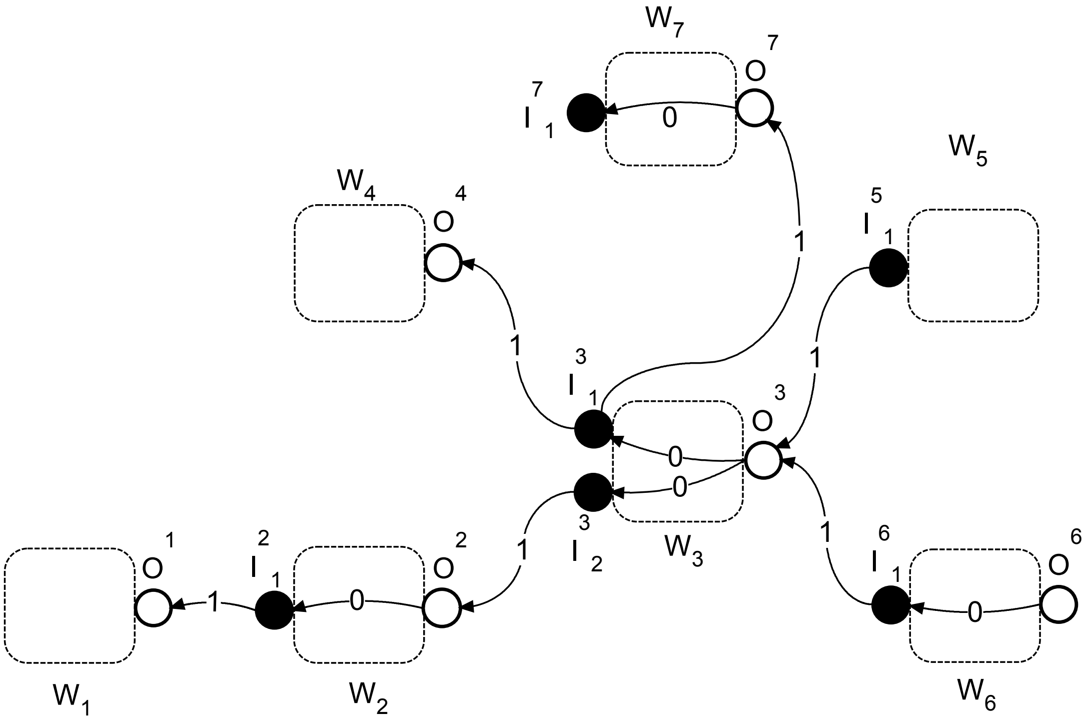


Figure 3.19: Example graph corresponds to a set of widgets

Algorithm 1 specifies the first variant, which is, without loss of generality[17], limited to mashups that use no more than a single instance of each widget.

The auto-composition of mashups is a graph search problem. The input is the preconstructed graph and a starting widget. We then need to identify all mashups that contain this widget. This means we have to find complete paths starting from all output and input terminals of the widget such that all involved input (output) terminals are wired with other output (input) terminals.

The search algorithm traverses the graph in a depth-first manner. We build up branches for input and output terminals in backward and forward search directions, respectively. *sbEdges* and *sfEdges* are two linked lists that store all possible edges during the backward and forward search. *uiVertices* saves all unvisited input vertices of visited widgets; *mashup* stores all mashup edges during the construction process; *isDead* is a flag that indicates dead ends, i.e., when we cannot build up a complete mashup from the current selections.

The algorithm starts by storing all edges ending with output terminal o^m into *sfEdges*. Next, we keep going forward until we find the input terminal of a visualization widget. We

[17] If we allow composed mashups to use a particular widget $n > 1$ times, we replicate all of its vertices n times and adapt the algorithm to avoid duplicate mashups.

Algorithm 1 Automatic mashup composition

```
Input: $G(V, E)$, $i_1^m, \ldots, i_k^m, o^m$
Output: The set of complete mashups
Global Variables: sbEdges , sfEdges , uiVertices , mashup , isDead
procedure autoCompose($G(V, E)$, $i_1^m, \ldots, i_k^m, o^m$)
    isDead ← false
    uiVertices ← $i_1^m, \ldots, i_k^m$
    sfEdges ← edgesEndWith($o^m$)
    while sbEdges is not empty or sfEdges is not empty do
        if sbEdges is not empty then
            e ← last edge of sbEdges
            backtrack(startVertex(e))
            goBackward(endVertex(e))
        else
            e ← last edge of sfEdges
            backtrack(endVertex(e))
            goForward(startVertex(e))
        end if
        if isDead is false then
            visitAllVertices
        else
            isDead ← false
        end if
    end while
end procedure

procedure goForward(Vertex $v$)
    Mark $v$ as visited
    if we cannot go forward from $v$ then
        isDead ← true
    else if $v$ is input k of widget q ($v \equiv i_k^q$) then
        Add $(o^q, i_k^q)$ to mashup                         ▷ $o^q$: output of widget q
        goForward($o^q$)
    else if $v$ is output then
        $S \leftarrow \{e \in E \mid e = (u, v)$ such that $u$ is unvisited$\}$
        Add $S$ to sfEdges to save all possibilities
        for $e \in S$ do
            Remove $e$ from sfEdges and add e to mashup
            goForward(startVertex($e$))
            if isDead is true then
                backtrack($v$)
                isDead ← false
```

Algorithm 1 Automatic mashup composition algorithm (continued)

```
            end if
        end for
    end if
end procedure

procedure goBackward(Vertex v)
    Mark $v$ as visited
    if we cannot go backward from $v$ then
        $isDead \leftarrow$ true
    else if $v$ is output of widget q ($v \equiv o^q$) then
        $I \leftarrow \{i^q_1, \ldots, i^q_{n_q}\}$                    ▷ all inputs of widget q
        Add $I$ to $uiVertices$ and add $\{(v, i) \mid i \in I\}$ to $mashup$
    else if $v$ is input then
        $S \leftarrow \{e \in E \mid e = (v, u)$ such that $u$ is unvisited$\}$
        Add $S$ to $sbEdges$ to save all possibilities
        for $e \in S$ do
            Remove $e$ from $sfEdges$ and add e to $mashup$
            goBackward(endVertex($e$))
            if $isDead$ is true then
                backtrack($v$)
                $isDead \leftarrow$ false
            end if
        end for
    end if
end procedure

procedure visitAllVertices
    for $v \in uiVertices$ do
        if $v$ is not visited then
            goBackward($v$)
        end if
        if $isDead$ is true then
            break
        end if
    end for
    if isDead is false then
        one complete mashup is found, save it
    end if
end procedure
```

Algorithm 1 Automatic mashup composition algorithm (continued)

```
procedure BACKTRACK(Vertex v)
    Remove all edges added to mashup since we visit the vertex v
    Clear visited mark of corresponding visited vertices
end procedure
```

call *visitAllVertices* to form mashup branches for all input terminals of visited widgets.

For instance, assume that we need to compose complete mashups of w_2 (cf. Figure 3.19). Starting with o^2, we go forward to i_2^3, and o^3. From o^3, we can go either to i_1^5 or i_1^6. We try to go to i_1^5 first and temporarily finish going forward there, because i_1^5 is the input of a visualization widget. Next, we have to find connections for the two input terminals i_1^2 and i_1^3. For i_1^3, we go backward to o^7, i_1^7 and get stuck there. We backtrack on i_1^3, then go to o^4 and finish going backward from i_1^3. Next, we go backward from i_1^2 to o^1. Now we have a complete mashup of w_1, w_2, w_3, w_4, w_5; the mashup consists of six edges (i.e., (i_2^3, o^2), (o^3, i_1^3), (o^3, i_2^3), (i_1^5, o^3), (i_1^3, o^4), and (i_1^2, o^1)).

To search for more mashups, we backtrack on o^3 and try to go to i_1^6, and then get stuck at o^6. The algorithm ends here because both *sfEdges* and *sbEdges* are empty; we have no more options to try. The detail steps and the values of the *uiVertices*, *sfEdges*, *sbEdges*, *mashup* in each step are presented in Appendix A.

3.10 Tag-based Automatic Mashup Composition

Automatic mashup composition is aimed at users who are already familiar with the mashup composition environment, but need support in finding relevant combinations of widgets. Its main drawback is that users must have all widgets they need at hand.

To simplify mashup development and ease data integration, we design a Tag-based Composition Module (TCM). The TCM targets novices who have no or little experience in mashup development; they do not know which widgets they need to compose the mashup, but they can specify the data (e.g., Points of Interest, Weather, or Transportation data) they want to collect and process. The TCM automatically locates relevant widgets working with such data and composes mashups from those widgets.

The processing flow of the TCM is illustrated in Figure 3.20. Users enter text and interactively select appropriate tags from LOD resources. The TCM can then identify the mashup context and discover related widgets. It finally composes mashups over detected widgets and displays them to users. We describe the process in the following.

3.10.1 Defining Mashup Context

There are several approaches (e.g., using a subset of natural language, using predefined components, or using a dedicated mashup language) to enable users to define the requirement or the context of their mashups (cf. Section 6.6). In our research, we follow a light-weight approach that makes use of our semantic Linked Widget models and the

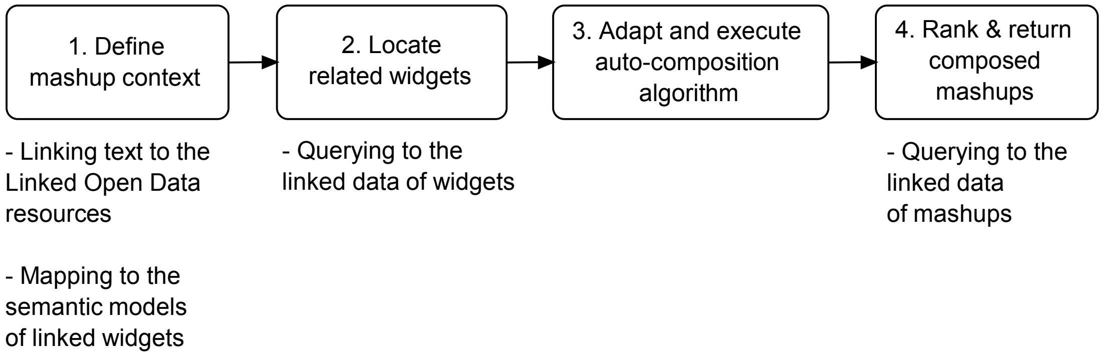

1. Define mashup context
2. Locate related widgets
3. Adapt and execute auto-composition algorithm
4. Rank & return composed mashups
- Linking text to the Linked Open Data resources
- Mapping to the semantic models of linked widgets
- Querying to the linked data of widgets
- Querying to the linked data of mashups

Figure 3.20: Processing flow of the tag-based composition module

automatic mashup composition. We require that the TCM is responsive, i.e., it can compose mashups based on the users' input text in (near) real-time.

To this end, we combine named entity recognition with resource tagging approaches. Users map their chosen words to available widgets through LOD resources. The mapping is based on tagging; this means free text is tagged with resources from the LOD cloud. These tags can then be used to identify corresponding widgets based on their data model. To reduce sources of errors, we use auto-complete to suggest LOD resources for tagging.

Users enter a sequence of characters into an input field; once the users are confident that the typed word defines the context of their information need, they invoke the TCM by pressing *Ctrl + Space*. From the selected word, the TCM generates a query on the LOD cloud. The entities and vocabularies (i.e., concepts and relations) discovered through this query are then returned to the users, who choose the desired resources to tag their word with. They can re-tag a word (or tag a new one) by selecting the word and pressing *Ctrl + Space*.

When annotating widgets, we require developers to tie their widgets to LOD resources. From these annotations and the users' tags, we can perform a mapping to locate widgets related to the user context. The resource-tagging approach removes ambiguities and therefore is preferable over regular text-tagging approaches; each resource is mapped with a URI and is hence unique. The approach can leverage the value of the *owl:sameAs* property. Although the resource tagged to a widget (by developers) is different from the resource tagged to a mashup context (by users), we can still precisely match the widget to the context, if the resources are linked by *owl:sameAs*.

The tagging approach is a simplification of the dialog-based approach. It is simple, because users interactively select their tags; they do not have to answer any questions – which may be vague or difficult to understand – to define their context.

3.10.2 Tagging Techniques

We combine two tagging techniques, i.e., *vocabulary tagging* and *entity tagging*. The former is for locating widgets with respect to their defined concepts and relations in the semantic widget model. The latter is for restricting the context of mashups; it hence filters the set of possible widgets (which are discovered via *vocabulary tagging*) needed to compose users' mashups.

We offer two options for users to enter text: (i) They can use *syntax text* to describe the mashup, following two patterns: "Find A_1", and "A_1 r_1 A_2". A_1 and A_2 are ontology concepts and r_i is the relation between them. Practical examples are: "*Find Park*", "*Find Swimming Pool*", and "*Park near Swimming Pool*". (ii) *Free text* is another option. Users can type arbitrary words in arbitrary order, e.g., "A_1 A_2 A_3 r_1 r_2" and tie these words to LOD vocabularies.

The technique to tie A_i and r_i to LOD vocabularies is called *vocabulary tagging.* If *syntax text* is used, the TCM queries the Linked Data of widgets to locate widgets that output objects of class A_1 for the pattern "Find A_1". For pattern "A_1 r_1 A_2", it locates widgets whose input consists of objects of class A_1 and whose output consists of objects of class A_2; moreover, the input and output objects must be linked via r_1 property. If users follow the *free text* style, the TCM simply finds widgets whose output or input consists of objects of class A_i or whose relations contained in widgets (i.e., widget – hasIndivualAtom – relation) is r_i. The *free text* option is easier to use, but is less precise than the *syntax text* mechanism.

To associate widgets and mashups with a context, we make use of *entity tagging* along four dimensions, i.e., where, when, who, and what. These dimensions refer to a place, a time, a person, or an object, respectively.

We can apply multiple dimensions to tag a widget or mashup. To this end, we need a common dataset for users and developers to tag to; DBpedia [104] – which may be considered the central hub of the LOD cloud – is a suitable option. It extracts structured information from Wikipedia and then stores and represents it as Linked Data. DBpedia supports 125 languages. As of June 2015, the English version describes 4.58 million things; it contains 1.45 million persons and 0.74 million places available for "where" and "who" dimensions. Because these two classes (i.e., Person and Place) are the most frequently used classes in DBpedia, "where" and "who" are typically more common than "when" and "what". We can reuse the publicly available DBpedia Spotlight service [105] for named entity recognition of DBpedia knowledge.

When annotating widgets, developers tie them to DBpedia entities in order to define the widget context; for instance, a widget that returns Nobel Prizes could be tied to *http://dbpedia.org/resource/Alfred_Nobel* and *http://dbpedia.org/resource/Nobel_Prize* for the "who" and "what" dimensions, respectively; a widget which returns POIs in Vienna would be tied to the Vienna entity in DBpedia. Matching user tagging entities with widget tagging entities makes mashup results more precise, as illustrated in the example of Section 3.10.6.

3.10.3 Linked Widget Meta-Tagging Model

To implement the tagging techniques presented in the previous section, we add the meta-tagging information (cf. Figure 3.21) to the Linked Widgets model.

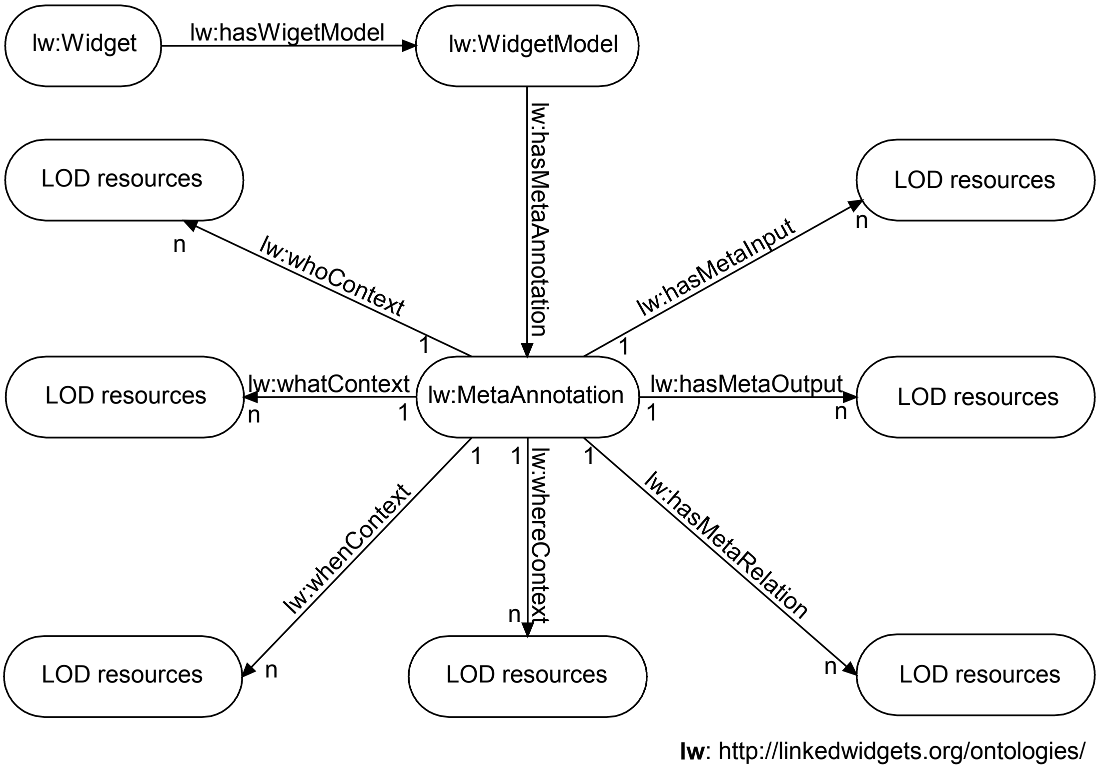


Figure 3.21: Meta-tagging model of Linked Widgets

Each widget model is linked to a *lw:MetaAnnotation* via the *lw:hasMetaAnnotation* property. The *lw:whoContext*, *lw:whatContext*, *lw:whenContext*, and *whereContext* properties realize the *entity tagging* techniques. They associate the *lw:MetaAnnotation* with various LOD resources. For each property, the relationship between the *lw:MetaAnnotation* and LOD resources is $1 : n$. Similarly, the vocabulary tagging is implemented by using the *lw:hasMetaInput*, *lw:hasMetaOutput*, and *lw:hasMetaRelation* properties.

As an example, consider the meta-tagging model of the POI Search widget illustrated in Figure 3.22. The widget returns all points of interest in Vienna from the geospatial input, which is an arbitrary object associated with the *wgs84:location* property. The widget meta-model hence is tied to the *dbpedia:Vienna*[18] resource via the *lw:whereContext* property. Because there is no restriction on the *who*, *what*, and *when* dimension, all of the respective properties link the widget meta-model to the *owl:Thing* resource. Finally, we can add further meta-tagging annotation of the widget's input, output, and relation to the widget model. To this end, we link the widget meta-model to the *dbpedia:Location*, *dbpedia:Place*, *dbpedia:Venue*, *dbpedia:Point*, *geo:Feature*, and *geo:nearby* resources.

[18] `http://dbpedia.org/resource/Vienna` (accessed Nov. 01, 2015)

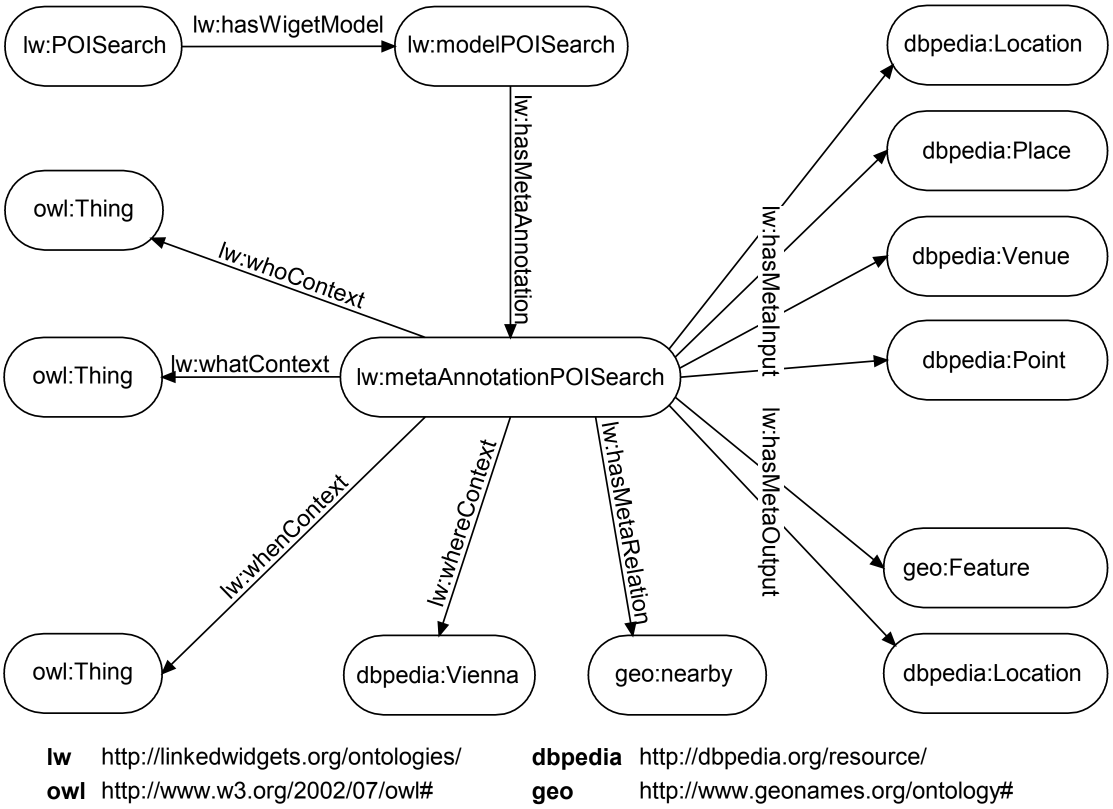


Figure 3.22: Meta-tagging annotation of the POI Search widget

3.10.4 Locating Widgets Based on the Mashup Context and the Meta-tagging Model

Based on the user-defined mashup context and the Linked Data repository of widget meta-tagging model, we need to search for relevant widgets and build up the input of the automatic mashup composition algorithm.

To this end, (i) we first use a number of SPARQL queries to locate widgets for each tagged relation in the input field. (ii) Similarly, we locate widgets that can output instances of the tagged concepts; those widgets will take the responsibility to provide necessary input data for the composing mashups. They can be either *data widget* or *transformation widget*; the latter is a special type of *processing widget* that transforms input instances into output instances of a different RDF class. Because the *transformation widget* has input terminals, to be able to complete the mashups, we need further process to search for additional *data widgets* that can provide input data for our *transformation widgets*. This is done via the *terminal matching*.

An example SPARQL query to locate widgets that can provide information about Swimming Pools in Vienna is shown in Listing 3.5. We put two restrictions on the

Listing 3.5: A SPARQL query to search for widgets that can provide information about Swimming Pool in Vienna

```
PREFIX lw: <http://linkedwidgets.org/ontology/>
PREFIX dp: <http://dbpedia.org/resource/>
PREFIX schema: <http://schema.org/>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?w ?adr ?n ?t WHERE
{
 ?w lw:hasWidgetModel [lw:hasMetaAnnotation ?m].
 ?w schema:name ?n.
 ?w lw:type ?t.
 ?w dct:identifier ?adr.
 ?m lw:hasMetaOutput dp:Swimming_pool.
 ?m lw:whoContext owl:Thing.
 {?m lw:whereContext dp:Vienna. UNION ?m lw:whereContext owl:Thing}.
 ?m lw:whatContext owl:Thing.
 ?m lw:whenContext owl:Thing.
}
```

lw:hasMetaOutput and *lw:whereContext* properties. We use *UNION* operator for the *lw:whereContext* restriction, because appropriate widgets are ones that can output Swimming Pool for either Vienna only or every city in the world.

Finally, for each widget discovered by the tagging techniques, we use *terminal matching* to find possible visualization widgets. If users follow the *syntax text* style, only widgets that can visualize objects of class A_i (which is in a "Find A_i" pattern) are kept.

To sum up, we now finish building up a temporary widget collection that includes:

1. *RW* – widgets whose model contains the tagged relations,
2. *DW* – *data widgets* that can output instances of the tagged concepts,
3. *TW* – *transformation widgets* and *ADW* – their additional *data widgets* that should be combined together to output instances of the tagged concepts, and
4. *VW* – visualization widgets that can visualize the output of the discovered widgets.

3.10.5 Composing and Ranking Mashups

We define T as the set of all user-defined tags.

$$T = \{t_1, \ldots, t_n\} \tag{3.10}$$

t_i can be either a *relation* tag or a *concept* tag. Let W_i be the set of located widgets for the tag t_i.

Next, C is the set of all located widgets.

$$C = RW \cup DW \cup TW \cup ADW \cup VW \tag{3.11}$$

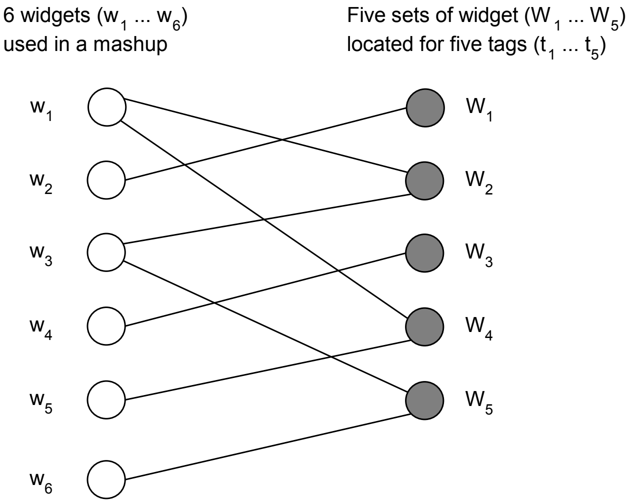

6 widgets (w_1 ... w_6)
used in a mashup
Five sets of widget (W_1 ... W_5)
located for five tags (t_1 ... t_5)
w_1
w_2
w_3
w_4
w_5
w_6
W_1
W_2
W_3
W_4
W_5

Figure 3.23: Tag completeness checking

Based on the constructed widget collection C, as presented in Algorithm 2, we execute the tag-based mashup composition. The algorithm relies on Algorithm 1. In the first step, we construct the graph $G(V,\ E)$ for the collection. Next, we need to determine the starting terminals to prepare the other input of the Algorithm 1.

Because the composed mashups must contain at least one widget in every single set of widgets W_i discovered for each tag t_i (i.e., this ensures that all user tags are considered), the starting terminal can be the output terminal of any widget $w \in W_i$. As any $w \in W_i$ can be a part of our mashups, to be able to compose every possible mashup, we have to start with all of these output terminals, one after another. Therefore, to decrease the number of calls to Algorithm 1 and hence increase the performance of the algorithm, we need to find the tag t_k such that the size of W_k reaches the minimum value.

Because a located transformation widget should always be linked with its respectively additional data widget, we mark the input terminal of the former and the output terminal of the latter as visited. This can considerably improve the efficiency of the automatic composition algorithm. We later need to add the links between these two special types of widgets into the composed mashups (Lines 15–19).

For each composed mashup, we need to perform further processing to ensure that all tags are considered (Line 13). The requirement is illustrated in Figure 3.23. Suppose we have five tags ($t_1 \dots t_5$), five respectively located widget sets ($W_1 \dots W_5$), and a mashup that consists of six widgets ($w_1 \dots w_6$). Each widget w_i can be the element of one or multiple widget sets (e.g., w_1 belongs to both W_2 and W_4).

The mashup is valid only if we can find a *matching* in the graph such that the number of edges in the *matching* is five. A *matching* in a graph is "*a set of pairwise non-adjacent edges; that is, no two edges share a common vertex*" [106]. In our example, the mashup is valid due to the *matching* $\{(w_1, W_2), (w_2, W_1), (w_3, W_5), (w_4, W_3), (w_5, W_4)\}$. The *tag*

Algorithm 2 Tag-based automatic mashup composition

```
Input
    $RW$, $DW$, ($TW$, $ADW$), $VW$
    $T = \{t_1, \ldots, t_n\}$ ← the set of all defined tags
    $W = \{W_1, \ldots, W_n\}$ ← the map that matches each tag $t_i$ with the respective set of
    located widgets $W_i$
Output
    The set of complete mashups
procedure COMPOSE($RW$, $DW$, ($TW$, $ADW$), $VW$, $T$, $W$)
    $C \leftarrow RW \cup DW \cup TW \cup ADW \cup VW$
    $G = (V, E)$ ← the constructed graph for the widget collection $C$
    $mashups$ ← a list that stores all composed mashups
    $t_k$ ← a tag of $T$ such that the size of $W_k$ reaches the minimum value
    For every widget $w \in TW$, mark all of its input terminals as visited
    For every widget $w \in ADW$, mark its output terminal as visited

    for $w \in W_k$ do
        $o$ ← the output terminal of $w$
        $tmpMashups$ ← AUTOCOMPOSE($G$, $o$)                    ▷ cf. Algorithm 1

        for $mashup \in tmpMashups$ do
            $W^{'}$ ← the set of widgets used in $mashup$
            $completed$ ← check if for every single tag $t_i \in T$, there always exists at
            least one widget $w^{'} \in W^{'}$ such that $w^{'} \in W_i$
            if $completed$ is $true$ then
                for $w^{'} \in W^{'}$ do
                    if $w^{'}$ is a transformation widget then
                        Add the respective data widget of $w^{'}$ to the $mashup$, and connect
                        the output of the data widget to the input of $w^{'}$
                    end if
                end for
                Add $mashup$ to $mashups$
            end if
        end for
    end for

    Check and remove duplicated mashups                        ▷ Post-processing
    Rank $mashups$
    return $mashups$
end procedure
```

completeness checking hence turns out to the *maximum-cardinality matching* problem; the largest possible number of edges of a *matching* should be equal to the number of defined tags. We hence can use the blossom algorithm [107] introduced by Jack Edmonds to remove irrelevantly composed mashups.

Finally, we check and remove duplicated mashups. Because the algorithm can result in a large number of composed mashups, it can be useful to rank the results as the final step. The ranking score of a mashup is calculated as the average score of all input and output links between member widgets; the score of a link is the total number of occurrences of that link in all existing mashups of the framework. Mashups with a high ranking score appear on top of the list returned to users. Because the framework is open for anyone to compose mashups, the longer the framework operates, the more precise the ranking algorithm will become.

3.10.6 Example

To illustrate our approach, consider a small set of widgets: (i) *Map Pointer* – a data widget which returns one or multiple points defined by users on the map, (ii) *Vienna POI* – a data widget which returns different types of POIs located in Vienna, (iii) *Weather Condition* – a processing widget which adds weather conditions collected from Wunderground[19] to each of its input locations, (iv) *Flickr Image* – a processing widget which adds sample Flickr Images for each of its input locations, (v) *Air Quality* – a processing widget which adds an air quality index value for each of its input locations, (vi) *Geo Merger* – a processing widget which combines the two input arrays of locations based on a distance constraint, (vii) *POI Search* – a processing widget which queries LinkedGeoData[20] for POIs near the input locations, (viii) *Map Viewer* – a visualization widget which visualizes locations and associated information (i.e., descriptions, images, charts).

As an example, consider the following query: "*Park*[21] *near*[22] *Swimming Pool*[23]". The TCM performs the resource matching task, and discovers *Geo Merger* and *POI search* widgets for the relation *near*.

There are two widgets (i.e., *Vienna POI*, *POI Search*) that can output *Park* and *Swimming Pool*. Suppose that the user has not used *entity tagging* to restrict the mashup context, yet. This means that the *Parks* and *Swimming Pools* that the user is searching for can be located in any city. Thus the *Vienna POI* is irrelevant.

At this point, we have *Geo Merger*, and *POI Search*. Because until now we have processing widgets only, we use *terminal matching* to discover additional data widgets (i.e., *Vienna POI* and *Map Pointer*) to provide input data, otherwise we cannot compose a complete mashup. Since the *Vienna POI* is still irrelevant, we keep the *Map Pointer* widget only. Finally, we take the *Map Viewer* for the visualization and have four widgets to compose mashups.

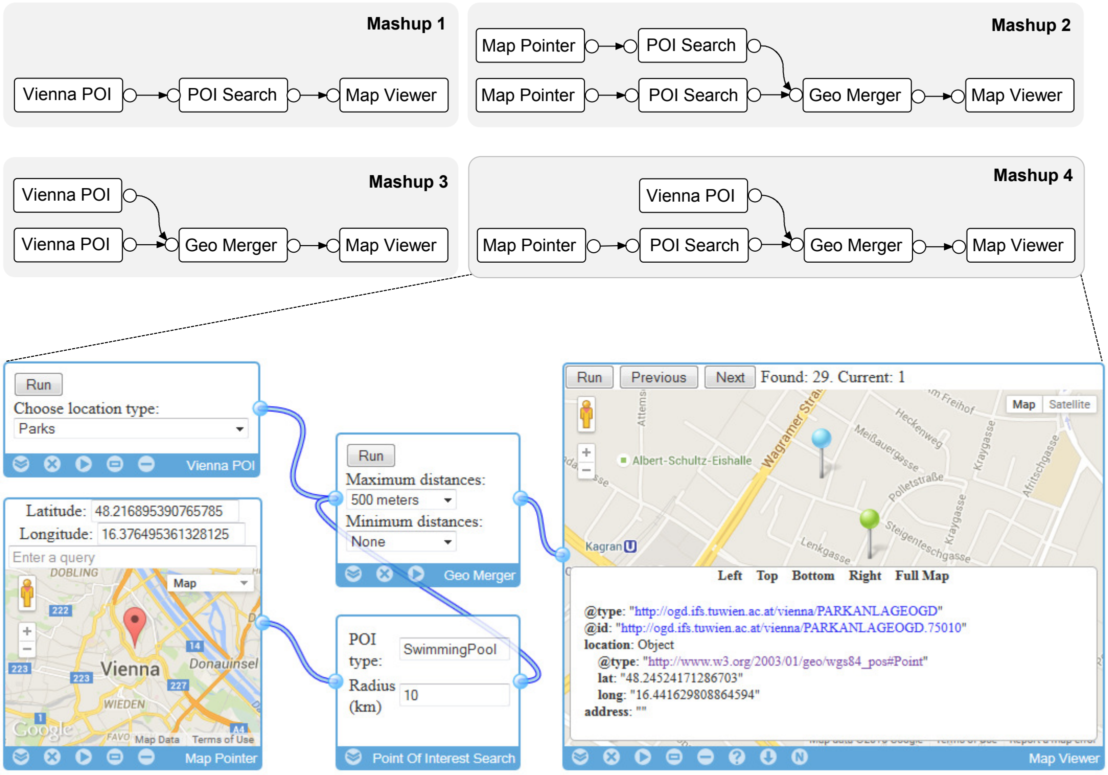
[19] `http://wunderground.org/` (accessed Nov. 01, 2015)

[20] `http://linkedgeodata.org/` (accessed Nov. 01, 2015)

[21] `http://dbpedia.org/resource/Park` (accessed Nov. 01, 2015)

[22] `http://www.geonames.org/ontology#nearby` (accessed Nov. 01, 2015)

[23] `http://dbpedia.org/resource/Swimming_pool` (accessed Nov. 01, 2015)



Choose location type: Parks

Latitude: 48.216895390765785
Longitude: 16.376495361328125

Maximum distances: 500 meters
Minimum distances: None

POI type: SwimmingPool
Radius (km) 10

Found: 29. Current: 1

```
@type: "http://ogd.ifs.tuwien.ac.at/vienna/PARKANLAGEOGD"
@id: "http://ogd.ifs.tuwien.ac.at/vienna/PARKANLAGEOGD.75010"
location: Object
  @type: "http://www.w3.org/2003/01/geo/wgs84_pos#Point"
  lat: "48.2452417128670З"
  long: "16.441629808864594"
address: ""
```

Figure 3.24: Example of tag-based automatic mashup composition

Assume the user now specifies the context like “in Vienna[24]” for the *where* dimension of her desired mashup. Five relevant widgets are identified: *Vienna POI*, *Map Pointer*, *Geo Merger*, *POI Search*, and *Map Viewer*. From this set, the TCM automatically composes four meaningful mashups as shown in Figure 3.24. The fourth mashup is displayed in full detail. It is used to integrate data from Google Maps, Vienna Open Government Data, and LinkedGeoData. There are 29 pairs of parks and nearby swimming pools shown in the map.

[24] http://dbpedia.org/resource/Vienna (accessed Nov. 01, 2015)

CHAPTER 4

Computational Experiments

In this chapter, we report on results of the computational evaluation of the *terminal matching* (cf. Section 4.1) and the *automatic mashup composition* algorithms (cf. Section 4.2) to study the performance characteristics of these algorithms and compare the experimental results with the theoretical model.

We have not carried out computational experiments on the *tag-based automatic mashup composition*, because the algorithm heavily relies on the *automatic mashup composition* algorithm, and we need a real deployment of the framework with a large number of mashups to rank the results. To produce the synthetic input data of the algorithm is a challenging issue and we leave it for future work.

Testing environment To conduct the experiments, we use a 64-bit Window 7 Enterprise computer with the Intel(R) Core(TM) i5-3470 CPU @ 3.20 Ghz processor and 8.00 GB of DDR3 RAM.

4.1 Terminal Matching

4.1.1 Goal of the Experiments

The process of terminal matching comprises two steps, i.e., *preliminary matching* and *full matching.* Pérez et al. [108] show that the evaluation of general SPARQL queries is a PSPACE-complete and NP-complete problem. Because *partial matching* is based on a SPARQL query (cf. Listing 3.18) executed in the Apache Jena[1] engine, it is an exponential time and polynomial space algorithm.

Assume that after performing *partial matching* step, we obtain n candidate models that need to be exactly matched with the querying model. To this end, in the *full matching*, we align every property and class of each model with that of the querying

[1] `https://jena.apache.org/` (accessed Nov. 01, 2015)

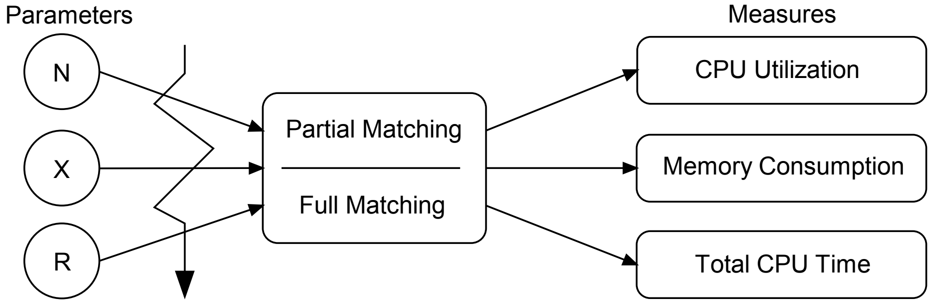


Figure 4.1: Experimental design of terminal matching

model, using string comparison (cf. Section 3.9.1). Full matching is a polynomial time and space algorithm.

The purpose of our experiments is to analyze the runtime characteristics and resource requirements of the terminal matching algorithm and compare the experimental results with the theoretical complexity.

4.1.2 Experimental Setup

Factors that affect the performance of the terminal matching algorithm are: (i) the total number N of available widgets, (ii) the rate R of successful matches between input and output models, and (iii) the complexity X (which is defined in the following) of the input and output model of widgets.

The complexity of the input/output tree model (cf. Section 3.4.3) can be specified by the depth of the tree, the node branching factor (which is the number of children at a node), and/or the total number of involved nodes and edges. In our experiment, we define X (which is a natural number) as the complexity of a tree, if and only if (i) the height of the tree is X, and (ii) the root node has exactly X children. Moreover, to avoid the excessive expansion of the tree, the branching factor of every node but the root is one.

As illustrated in Figure 4.1, we evaluate the effect of each parameter (i.e., N, X, and R) on the terminal matching algorithm with respect to (i) CPU utilization, (ii) total CPU time, and (iii) memory consumption. Because the amount of memory consumed by the algorithm continuously changes during the execution, we observe the total amount of used memory every 0.1 second.

We analyze the CPU utilization and the total CPU time of the *partial matching* and the *full matching* separately and hence can evaluate the contribution of each step to the overall process. For memory consumption, however, we have to combine the two steps and observe the memory consumed by the whole process, because they share the same allocated memory in the same thread.

To evaluate the effect of X and R on the performance of the algorithm, we require that (i) every synthetic input/output model shares the same complexity X, and, (ii) for a given input (output) model, there must be exactly M other matched output (input)

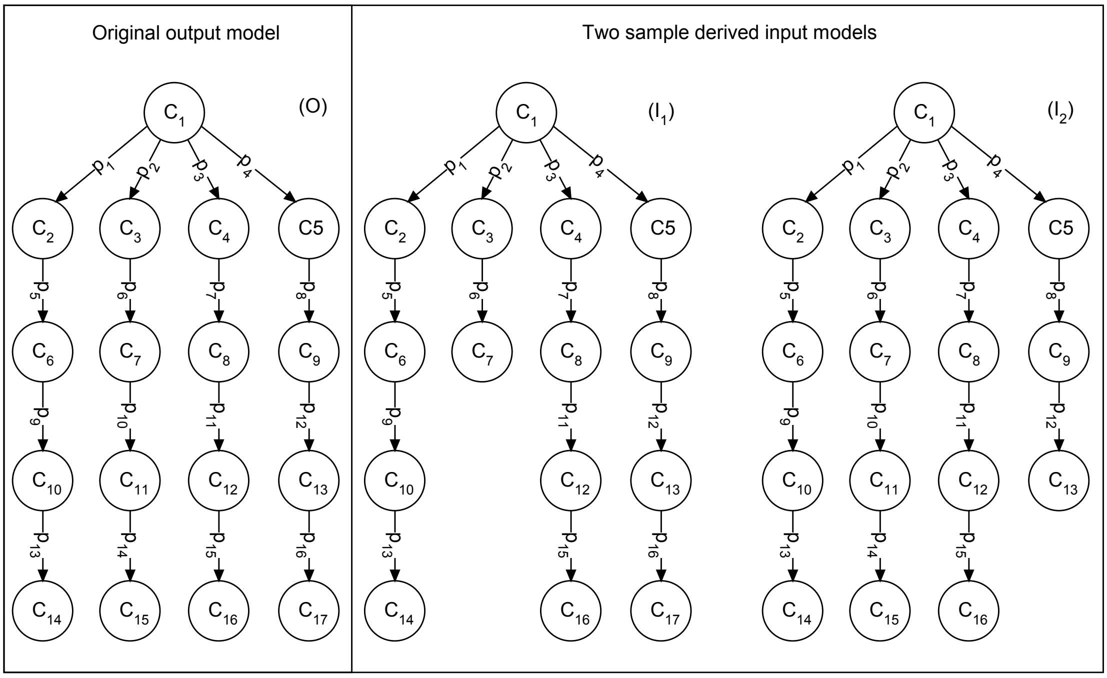


Figure 4.2: Sample input and output models of synthetic widgets

models, where $M = N \times R$. These requirements ensure that all widgets are equal and we can perform the terminal matching algorithm on arbitrary widget input/output model without loss of generality.

During our terminal matching experiments, rather than working directly with widgets, we try to match their terminal models only. The model can belong to a widget of an arbitrary type without affecting the algorithm. Thus we do not have to differentiate between data, processing, and visualization widgets and can synthetically generate processing widgets only.

Based on the discussion above, we generate a synthetic test set of widgets – which is specified by a triple (N, X, R) – as follows:

1. We do not generate widgets separately, but generate a set of $(M + 1)$ processing widgets simultaneously. Each input (output) model of a widget of this set will match exactly M output (input) models of M other widgets. To this end, we first generate an original output model with the complexity X, and take it as the output model of all $(M + 1)$ widgets; while constructing the output tree model, we use different classes and properties for each node and edge. Next, we derive $(M + 1)$ input models from the output model by deleting a random node in a random branch. This deleting method ensures that each derived input tree model matches the output tree model. Figure 4.2 illustrates a sample output model O and its two possibly derived input models, i.e., I_1, and I_2, which are generated by

deleting the node C_{11} and C_{17}, respectively.

2. To generate N widgets, we perform the above step n times, where $n = N/(M+1)$. Because the output model generated in each iteration step uses completely different classes and properties, the arbitrary input model derived in an iteration step never matches the output model generated in any other iteration step. This means the input model of a widget matches exactly M output models of the M widgets generated in the same iteration step.
3. When semantically annotating the generated widgets and their input and output models, we also add extra information (e.g., the name and the address of the widgets) to make the synthetic dataset more real. Finally, we store the data into a local file that will be reloaded in the experiments to evaluate the algorithm. To this end, we use the Jena library[2] for semantic processing.

Parameter Values To carry out the experiments, we need to decide the values of the triple (N, X, R) and generate the respective widget sets. To this end, we take a snapshot of the current prototype implementation of the framework (cf. Chapter 5); as of November, 01, 2015, there are 43 widgets; the average complexity of their input/output models is around 5, and the model-matching rate is about 20%.

As the RAM of the testing PC is limited to 8 GB, we do not generate a set of more than 10,000 widgets; the complexity of the input/output tree model X is limited to 20 and the matching rate R is limited to 0.5. The semantic models of all synthetic widgets generated for $(N, X, R) = (10000, 20, 0.5)$ already consist of more than 12 million triples.

We use the values of the three parameters as follows:

$$N \in \{1000, 2000, 4000, 8000, 10000\} \tag{4.1}$$

$$X \in \{1, 5, 10, 15, 20\} \tag{4.2}$$

$$R \in \{0.1, 0.2, 0.3, 0.4, 0.5\} \tag{4.3}$$

We conduct three experiments, that is (i) N**-experiment**, (ii) X**-experiment**, and (iii) R**-experiment**. In each experiment, we change the value of one parameter while setting fixed value for other parameters. For example, in the N**-experiment**, we change the value of N and use the fixed value of X and R.

To this end, we use (8000, 10, 0.2) as the standard values of (N, X, R). 8,000 widgets is a reasonable number; it is not too large that the respective semantic models may use all of the memory of the testing PC. We assume that 10 and 0.2 are respectively the average values of X and R. For each value of the (N, X, R) triple, we run the algorithm ten times and calculate the median to ensure the reliability of the experiment.

4.1.3 Results

The final results are presented as follows:

[2] `https://jena.apache.org/` (accessed Nov. 01, 2015)

1. N-**experiment**: Figure 4.3 presents the total CPU time of the *preliminary matching* and *full matching* for each $N \in \{1000, 2000, 4000, 6000, 8000, 10000\}$. Figure 4.6a visualizes the memory consumed by the whole process during the execution time.
2. X-**experiment**: Figure 4.4 presents the total CPU time of the *preliminary matching* and *full matching* for each $X \in \{1, 5, 10, 15, 20\}$. Figure 4.6b visualizes the memory consumed by the whole process during the execution time.
3. R-**experiment**: Figure 4.5 presents the total CPU time of the *preliminary matching* and *full matching* for each $R \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$. Figure 4.6c visualizes the memory consumed by the whole process during the execution time.

The box plots of the experiment data are presented in Appendix B. Based on the results, in the following, we will draw a number of conclusions.

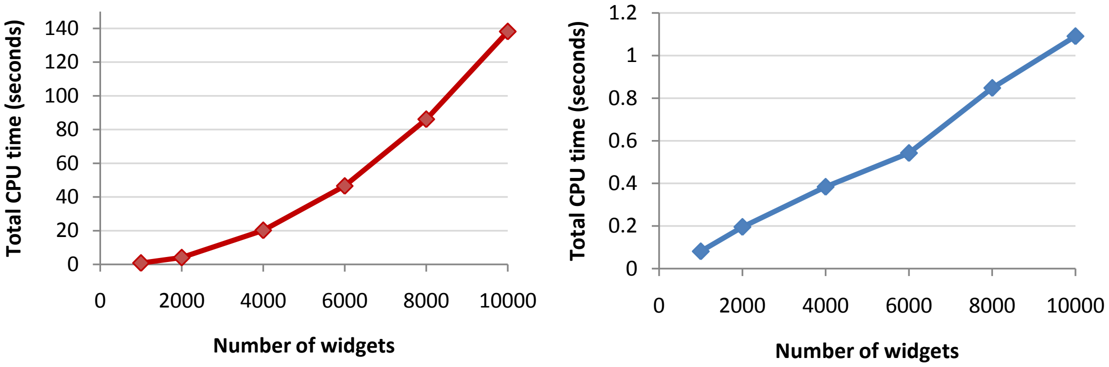
CPU utilization In every experiment, the algorithm always uses approximately 100% of one core of the CPU. As there is no interesting insight, we provide the detailed box plots in Appendix B.

Total CPU time The total CPU time of the *full matching* is roughly hundred times smaller than that of the *preliminary matching* in every experiment. It is because the *preliminary matching* has to process a large amount of data to achieve the list of potential matched models for a given terminal; the *full matching* then performs further processing tasks on these models to confirm the match only.

When we increase the number of widgets from 1,000 to 10,000, we witness the exponential growth in the total CPU time of the preliminary matching and the linear growth in the total CPU time of the full matching (cf. Figure 4.3). Reversely, we observe the linear growth in the preliminary matching's total CPU time and the polynomial growth in the full matching's total CPU time when we increase the complexity of the widget model from 1 to 20 (cf. Figure 4.4). Meanwhile, the total CPU time of both steps rises linearly when we increase the input-output matching rate from 0.1 to 0.5 (cf. Figure 4.5).

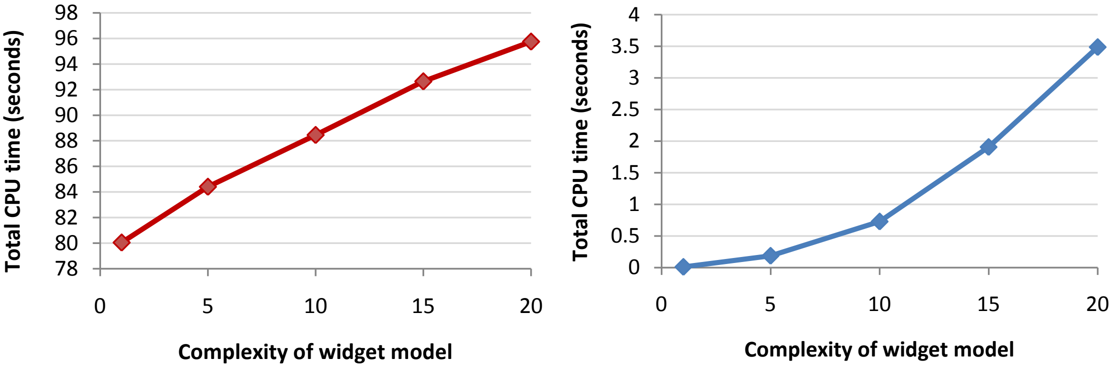
With a large number of widgets, it can take a lot of time for the algorithm to complete the task (e.g., approximate 200 seconds for $N = 8000$, X=10, $R = 0.5$, as shown in Figure 4.5). To overcome this issue, we can use a caching mechanism. We associate each terminal with a list of matched terminals. Each time a new widget is added to the framework, we run the algorithm for the input (output) terminal of that widget, save the result, and update the matching list of all other terminals. As this whole matching data may be large, we can store it in a secondary storage.

Memory consumption In all experiments, the consumed memory periodically increases and decreases between two thresholds during the execution time. When we increase the number of widgets or the complexity of the widget model, the threshold increases steadily (cf. Figures 4.6a and 4.6b). Meanwhile, as we increase the input-output matching rate, the threshold seems stayed (cf. Figure 4.6c).



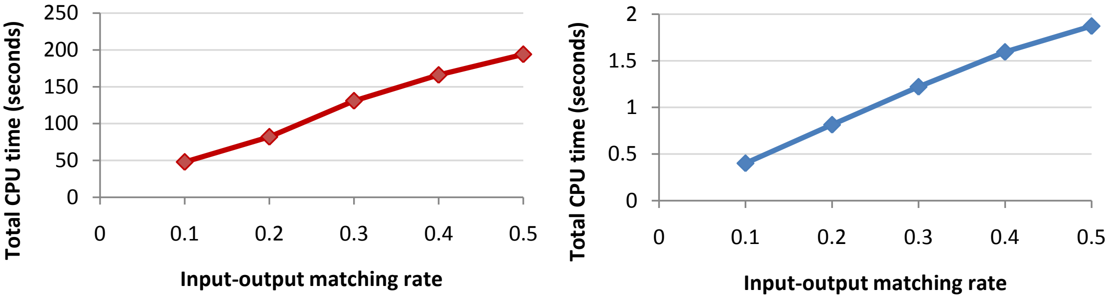
(a) Preliminary matching (b) Full matching

Figure 4.3: Terminal matching: total CPU time as a function of number of widgets (Matching rate $R = 0.2$ and model complexity $X = 10$)



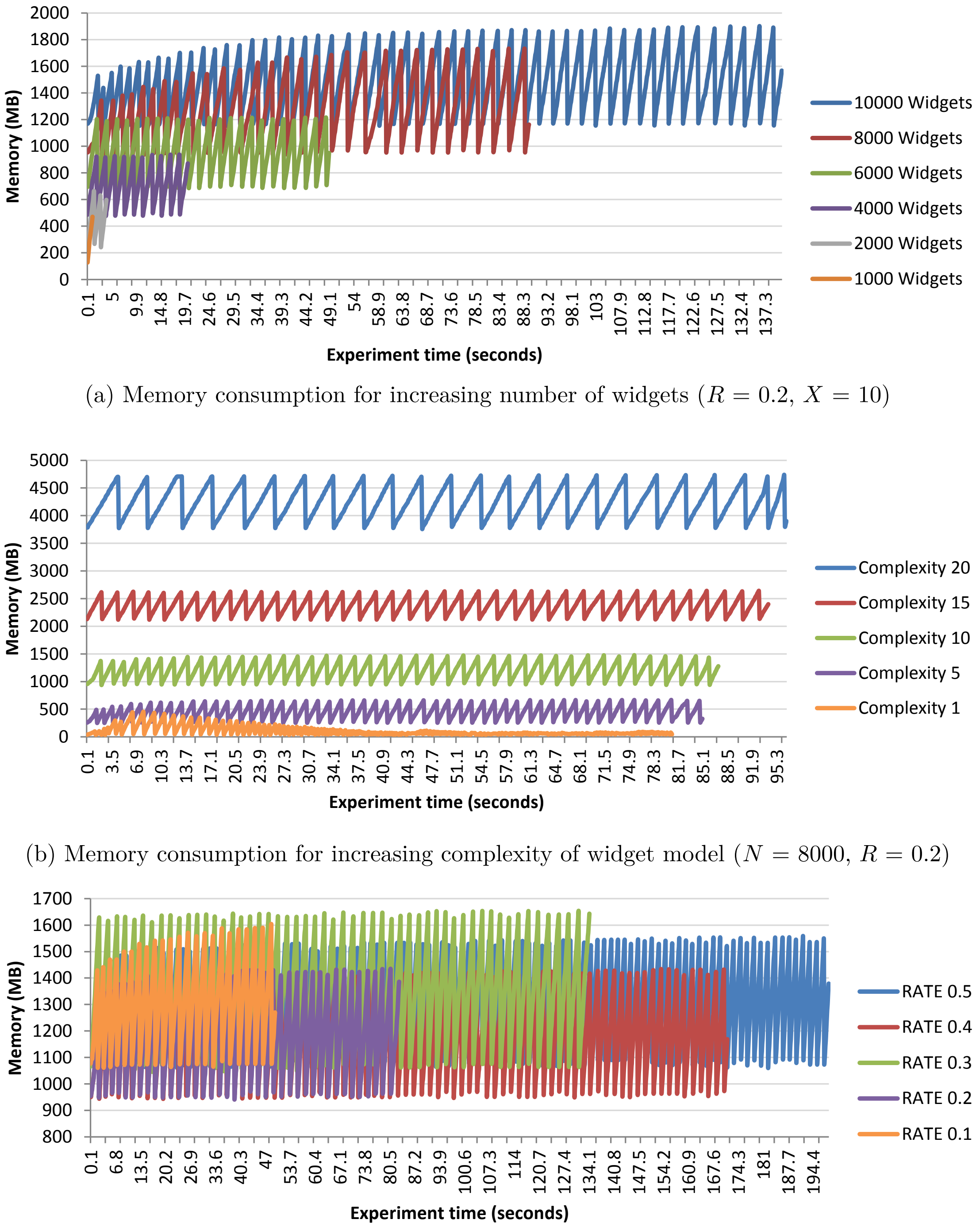
(a) Preliminary matching (b) Full matching

Figure 4.4: Terminal matching: total CPU time as a function of complexity of widget model (Number of widget $N = 8000$ and matching rate $R = 0.2$)



(a) Preliminary matching (b) Full matching

Figure 4.5: Terminal matching: total CPU time as a function of input-output matching rate (Number of widget $N = 8000$ and model complexity $X = 10$)



(a) Memory consumption for increasing number of widgets ($R = 0.2$, $X = 10$)

(b) Memory consumption for increasing complexity of widget model ($N = 8000$, $R = 0.2$)

(c) Memory consumption for increasing input-output matching rate ($N = 8000$, $X = 10$)

Figure 4.6: Terminal matching: memory consumption

To sum up, N contributes most to the total CPU time of the algorithm; N and X have a significant effect on the memory consumption. The experiments shows results as expected in the theoretical model (i.e., the preliminary matching is NP and the full matching is P). To improve the algorithm, we need to focus on *preliminary matching*, because its total CPU time is roughly hundred times longer than the total CPU time of *full matching* in every experiment.

4.2 Automatic Mashup Composition

4.2.1 Goal of the experiments

To evaluate the complexity of the automatic mashup composition, consider a widget collection that consists of one data widget, one visualization widget, and n processing widgets. In this collection, we can link the data widget to all processing widgets, and link an arbitrary processing widget to the visualization widget as well as any other processing widget. Thus the total number of links is $n + n + n(n-1) = n(n+1)$.

An example is illustrated in Figure 4.7 where $n = 3$. The total number N of mashups that can be composed for this widget collection is calculated in Equation 4.4. Mashups are categorized into n groups; the group k comprises mashups that use exactly k processing widgets and hence has $P(k, n)$ mashups, where $P(k, n)$ is the number of k-permutations of n (cf. Figure 4.7).

$$N = P(1, n) + P(2, n) + ... + P(n, n) \; where \; P(k, n) = \frac{n!}{(n-k)!} \tag{4.4}$$

As $P(n, n) = n!$, the total number of composed mashups is greater than the factorial of the size of the input data. To discover, store, and rank all of these mashups are hence non-trivial and we need an exponential time and space algorithm.

In the worst case, the Algorithm 1 introduced in Section 3.9.2 is an EXPTIME and EXPSPACE algorithm. In general, its performance depends on (i) N^d – the number of data widgets (ii) N^p – the number of processing widgets, (iii) N^v – the number of visualization widgets, and (iv) R – the input-output matching rate (which decides the density of the constructed graph).

The purpose of our experiments is to analyze the effect of each parameter (i.e., N^d, N^p, N^v, and R) on the runtime characteristics and resource requirements of the algorithm (cf. Figure 4.8).

4.2.2 Experimental Setup

To generate the synthetic graph as the input of the automatic mashup composition, we first generate a list of N^d data widgets, N^p processing widgets, and N^v visualization widgets. Next, we create one input vertex and one output vertex for each visualization widget and data widget, respectively. Even though a processing widget can have multiple input terminals, in practice, we usually create processing widgets with only one or two input terminals, because an input terminal can accept multiple links from other output

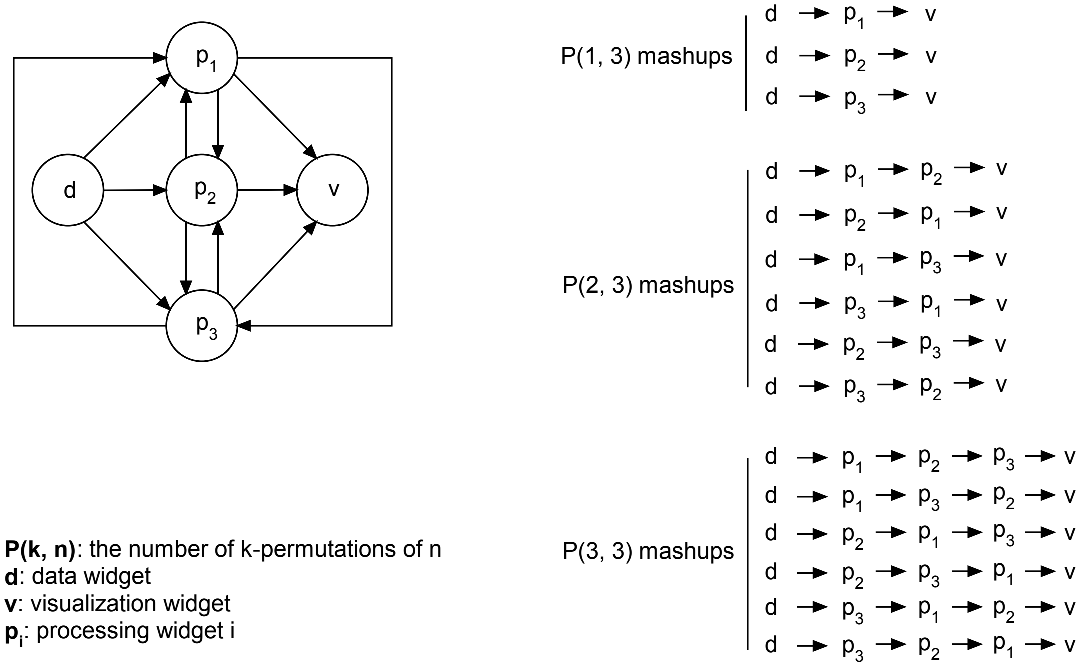


Figure 4.7: Example widget collection

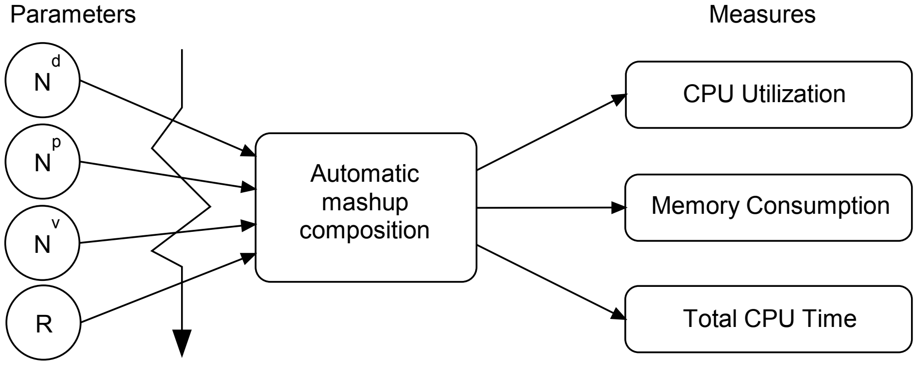


Figure 4.8: Experimental design of automatic mashup composition

terminals. To this end, in a random manner, we create one or two input vertices for each processing widget; we then create an output vertex and link it with the input vertex(es), using the inner edge(s) (cf. Section 3.9.2).

In the next step, we establish the external links between input and output vertices of different widgets, with respect to the parameter R. Because visualization widgets do not have any output vertex, the total number of output vertices is $(N^p + N^d)$. To achieve the rate R, for each generated input vertex, we randomly choose M out of $(N^p + N^d)$ output vertices, where $M = (N^p + N^d) \times R$. Each output vertex has the same

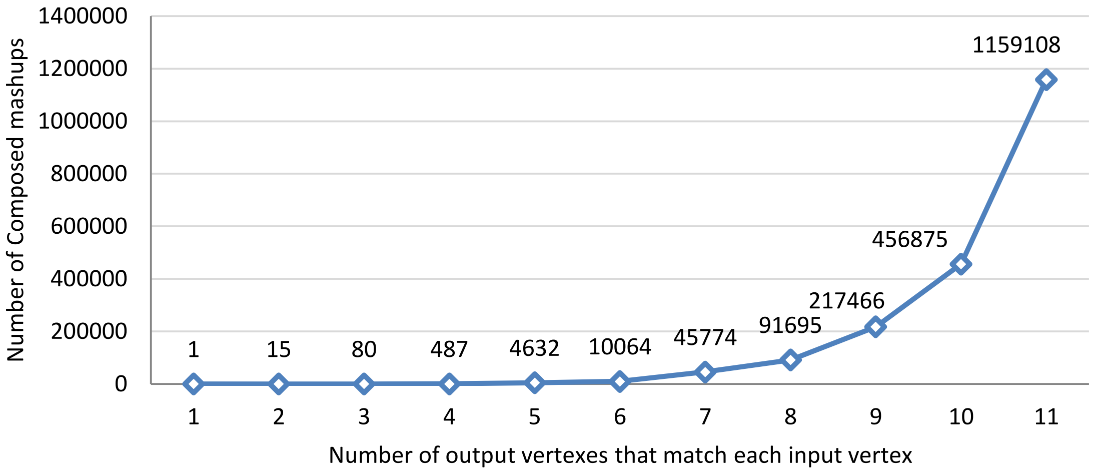


Figure 4.9: Growth in the number of mashups ($N^d = 5$, $N^p = 10$, $N^v = 5$)

probability to be included in the M chosen vertices. For example, assume we need to take 3 out of 5 elements $\{o_1, o_2, o_3, o_4, o_5\}$; if we take $\{o_5, o_3, o_1\}$ in the first turn, the two remaining elements (i.e., o_2 and o_4) should take priority in the second turn. As we need one more element, we can randomly select one of o_5, o_3, or o_1. Suppose we select o_3 and obtain $\{o_2, o_4, o_3\}$ for the second turn, we are only allowed to take 3 elements out of $\{o_1, o_2, o_4, o_5\}$ in the third turn.

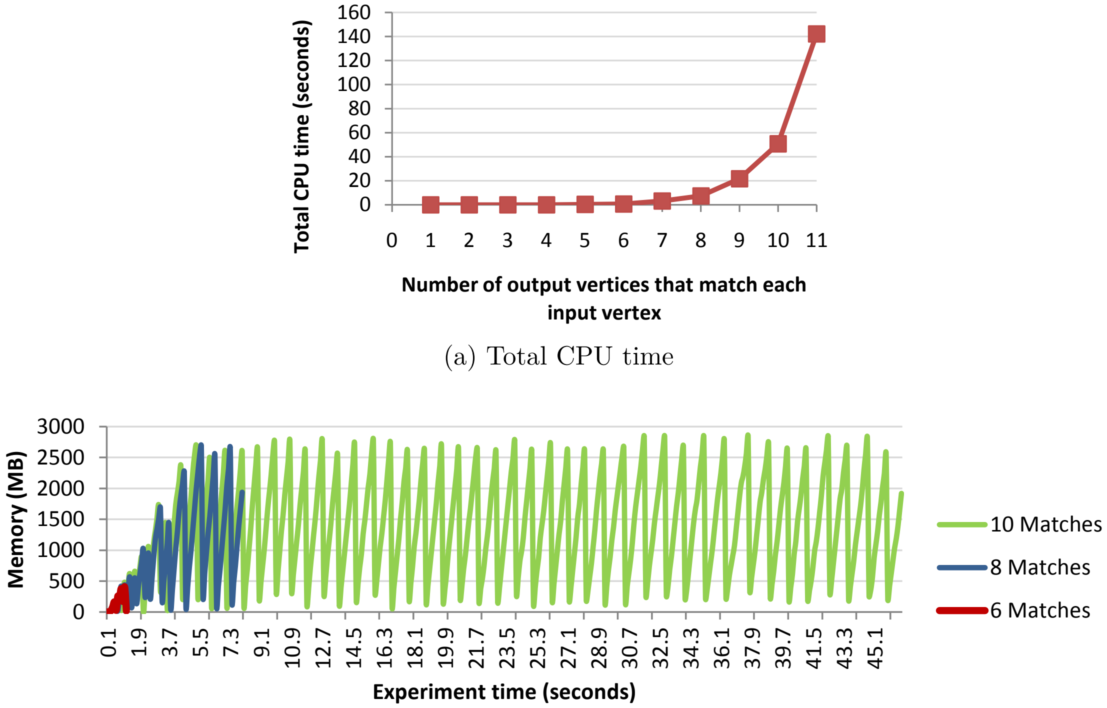
To this end, we use the Colt project[3], which provides a set of libraries for technical computing in Java. It allows us to generate a deterministic and reproducible series of random numbers based on a seed, using a large variety of probability distributions (e.g., Poisson, Gamma, Hyperbolic, or Uniform distribution).

We conduct four experiments to evaluate the effect of each parameter (i.e., N^d, N^p, N^v, or R) on the algorithm, with respect to CPU utilization, total CPU time, and memory consumption. In each experiment (e.g., ***R*-experiment**), we change the value of one parameter (e.g., R) and hold the other parameters constant.

Parameter Values Given a small set of widgets, the number of composed mashups can be very large. Figure 4.9 illustrates the exponential growth in the number of composed mashups for a set of 5 data widgets, 10 processing widgets, and 5 visualization widgets, when we increase the number of matched output vertices for each input vertex.

Our assumption is that the number of processing widgets is typically twice the number of data widgets (or visualization widgets). Thus we choose (5, 10, 5) as the values of (N^d, N^p, N^v) in the ***R*-experiment**. This small set of widgets allows us to observe the behavior of the algorithm until it has completed, even if the input-output matching rate is very high.

[3] `https://dst.lbl.gov/ACSSoftware/colt/` (accessed Nov. 01, 2015)



(a) Total CPU time

(b) Memory consumption

Figure 4.10: Automatic mashup composition: total CPU time and memory consumption for increasing input-output matching rate ($N^d = 5$, $N^p = 10$, $N^v = 5$)

Meanwhile, in the **N^d, N^p, N^v experiments**, because the running time is too long, we only observe the algorithm until 20,000 mashups have been composed. It is reasonable to stop the experiments there, because 20,000 is already a number too large for users to manually try. In these experiments, the standard values of four parameters are as follows: $N^d = 50, N^p = 100, N^v = 50, R = 0.2$.

4.2.3 Results

In the following, we present the results of our four experiments and draw a number of conclusions. The box plots of the experiment data are presented in Appendix C.

R-experiment: Figure 4.10 shows the total CPU time and the memory consumption of the algorithm over increasing number M ($M = (N^p + N^d) \times R = 15 \times R$) of matched output vertices for each input vertex.

N^d, N^p, and N^v experiments: Figures 4.11, 4.12, and 4.13 illustrate the total CPU time of the algorithm over increasing number of data, processing, and visualization widgets, respectively. The total CPU time is measured until 20,000 mashups have been composed. Figure 4.14 shows the memory consumption of the three experiments; the memory is observed every 0.1 second.

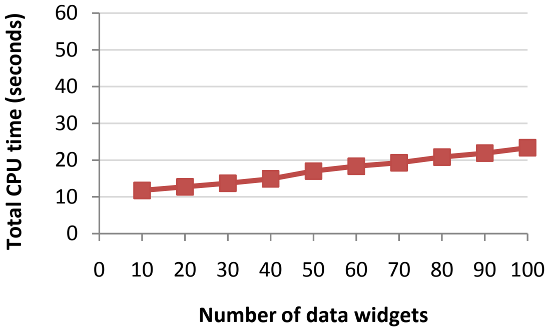


Figure 4.11: Automatic mashup composition: total CPU time (which is measured until *20,000 mashups* have been composed) as a function of number of data widgets (N^p = 100, N^v = 50, R = 0.2)

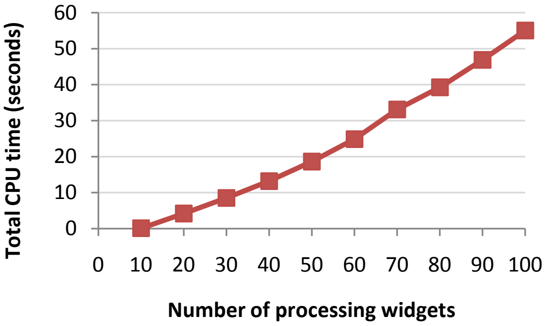


Figure 4.12: Automatic mashup composition: total CPU time (which is measured until *20,000 mashups* have been composed) as a function of number of processing widgets (N^d = 50, N^v = 50, R = 0.2)

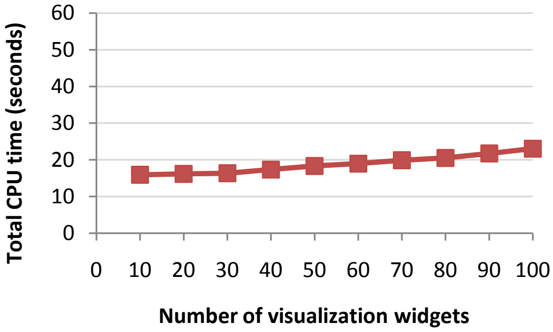
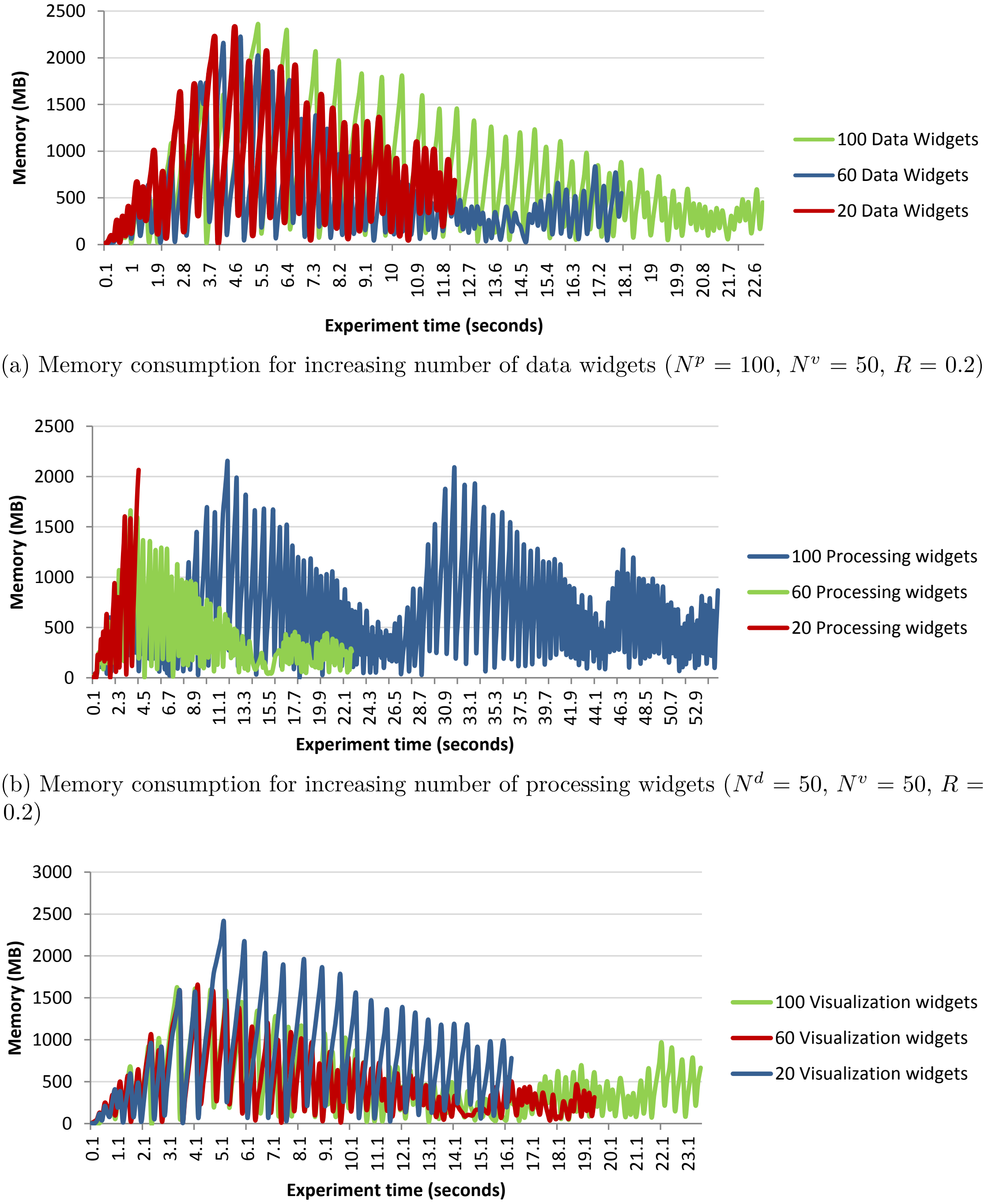


Figure 4.13: Automatic mashup composition: total CPU time (which is measured until *20,000 mashups* have been composed) as a function of number of visualization widgets (N^d = 50, N^p = 100, R = 0.2)



(a) Memory consumption for increasing number of data widgets ($N^p = 100$, $N^v = 50$, $R = 0.2$)

(b) Memory consumption for increasing number of processing widgets ($N^d = 50$, $N^v = 50$, $R = 0.2$)

(c) Memory consumption for increasing number of visualization widgets ($N^d = 50$, $N^p = 100$, $R = 0.2$)

Figure 4.14: Automatic mashup composition: memory consumption (which is observed until *20,000 mashups* have been composed)

CPU utilization The CPU utilization of every experiment is around 100% most of the time. As there is no interesting insight, we provide the detailed box plots in Appendix C.

Total CPU time The R**-experiment** shows that our automatic mashup composition algorithm is an exponential time algorithm (cf. Figure 4.10). On the other hand, as we increase the number of data, processing, or visualization widgets, the required time to achieve 20,000 results will increase linearly. The line in the N^p **experiment** is much steeper than that in the N^d and N^v **experiments**. It is because we have to spend more time searching in wrong routes and need to backtrack afterwards.

Memory consumption In each experiment (e.g., N^p**-experiment**), when we change the respective parameter (e.g., $N^p = 20$, 60, or 100), the shape of memory consumption over the time looks quite similar (cf. Figure 4.14). The memory mainly serves for storing the *sfEdges* and *sbEdges* lists (cf. Algorithm 1); these lists can be empty or include a large number of elements during the execution time. As a result, the consumed memory can increase to 2,500 MB or drop down to nearly 0 MB as seen in the graph.

To sum up, until 20,000 mashups have been composed, the four parameters N^d, N^p, N^v, and R equally contribute to the memory consumption of the algorithm; on the other hand, N^p and R have a major effect on the total CPU time. As expected in the results, the automatic mashup composition is an exponential time algorithm.

Summary of the Experiments The experiment results show that the *automatic terminal matching* and the *automatic mashup composition* can be used in practice, with regard to their total CPU time. We can further improve their performance by using caching mechanisms.

The *automatic mashup composition* produces positive results with only a few number of mashups returned, if the widget matching rate is low; it is easy for a user to locate the mashup she needs. However, as the widget matching rate is getting higher, the number of composed mashups will increase exponentially. Even though all mashups are syntactically correct as all links among widgets in each mashup are valid, many mashups can be meaningless. To address this issue, we can use heuristic search rather than the depth-first search in the *automatic mashup composition* – which is a graph search problem (cf. Section 3.9.2). From a terminal (which is represented by a vertex in the graph), we choose the next terminal to visit in such a way that the link between them has the highest score; the score of a link is the total number of occurrences of that link in all existing mashups of the framework. We consider the heuristic technique as future work to improve the algorithm because it requires a real deployment of the framework with a large number of mashups.

CHAPTER 5

Prototype Implementation of the Framework

In this chapter[1], we present the prototype implementation of the conceptual framework, which is available at *http://linkedwidgets.org.* We first discuss architectural design considerations in Section 5.1. We outline the key components of the prototype in Section 5.2. We then provide implementation details and reflect on the lessons learned while implementing the framework in Section 5.3. Next, in Section 5.4, we show example applications in the geospatial context. Finally, we illustrate the use of hybrid mashups by means of two practical use cases in Section 5.5.

5.1 Architectural Design Considerations

When defining the architecture for our mashup framework, we set out to follow three essential design principles, (i) *openness*, (ii)*connectedness*, and (iii) *reusability.*

First, the framework that fosters the widespread use of open data cannot tap its full potential if it is not open. It should follow an open architecture that enables arbitrary developers and end users to contribute and share their work with the open data community. An example for the benefits of openness is LinkedGeoData,[2] which uses information collected by the OpenStreetMap[3] project and makes it available as an RDF knowledge base according to the Linked Data principles [17]. It allows users to directly edit displayed resources in the map view, which simplifies the process of extending data quantity and improving data quality. Similarly, we encourage developers to implement and add new Linked Widgets to the framework to enable new combinations of open data

[1] Parts of this chapter also appear in [2, 3, 4, 6]. The author of this thesis is also the lead author of these papers.

[2] http://linkedgeodata.org/About (accessed Nov. 01, 2015)

[3] http://www.openstreetmap.org/ (accessed Nov. 01, 2015)

sources. Even end users can create a new widget from composed mashup applications without any programming and contribute it to the community.

Next, we design the architecture around the idea of *connectedness*, which is implemented via two concepts, i.e., *data connection* and *functionality connection*. Because there are a lot of related open data sources, applications should not restrict themselves to a small number of data sources. Instead, combining two or more data sources and enriching the data with additional value creates exciting opportunities.

From the *openness* feature we can derive the *functionality connection* and the *reusability* feature. Anyone can contribute new functionality to an application, however, these should not be separated from each other. It should be possible to connect and reuse them in an effective and efficient manner. Our mashup framework supports reuse in four ways: (i) To compose mashups and integrate data, users can creatively combine Linked Widgets from different developers; (ii) they can reuse mashups from others, but change the parameters of the constituted widgets; (iii) they can reuse a mashup as a new widget; and (iv) based on available widgets, developers can implement new widgets to support new use cases.

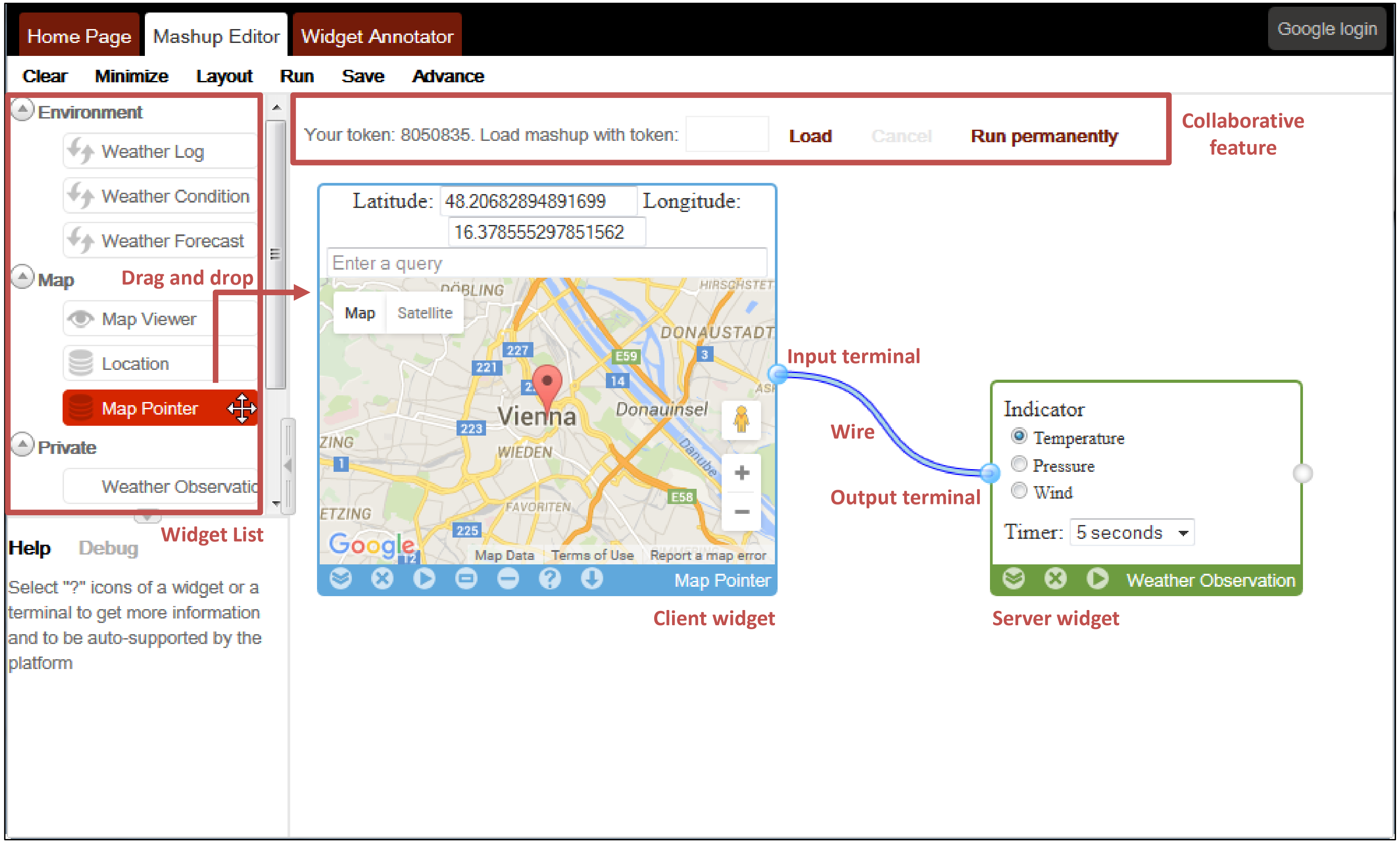
Another architectural design consideration for applications that integrate data from multiple sources, is to decide where the data processing task should be performed. The alternatives are to either do it locally at the client application or remotely at the server side. Because a server of high quality datasets can easily become overloaded with too many requests from clients, we should make use of client resources whenever possible. An example is Linked Data Fragments [109] in which the client itself will execute complex SPARQL queries after receiving the data fragments – corresponding to its defined triples – from the server. In the Linked Widgets framework, data processing can be done on-the-fly in the client's browser. Moreover, to tackle additional types of data, i.e., stream data, real-time data, big data, and long-time processing data, we consider widgets as a user interface on the client and the actual data retrieval and processing tasks, instead, could be performed at the server (cf. Section 3.3.2).

5.2 Key Components

The prototype implementation of the framework provides (i) a tool that supports developers in creating and annotating widgets, (ii) a tool for users to locate relevant widgets, and (iii) a drag-and-drop collaborative mashup editor. We describe each tool in the following.

5.2.1 Collaborative Mashup Editor

The mashup editor (cf. Figure 5.1) is the core component of the framework; it allows users to collaboratively compose, publish, and share their mashup applications. From a selected widget collection placed at the left-hand side, they drag and drop a widget item into the editor to create an instance of the widget. Users can then wire the input of a widget to the output of another one and thus build up a data-processing flow.



Your token: 8050835. Load mashup with token:

Latitude: 48.20682894891699 Longitude: 16.378555297851562

Enter a query

Indicator
● Temperature
○ Pressure
○ Wind

Timer: 5 seconds

Select "?" icons of a widget or a terminal to get more information and to be auto-supported by the platform

Figure 5.1: Web-based collaborative mashup editor

Once a widget has been added to a mashup, semantic *terminal matching* allows users to explore additional widgets that are relevant in the given context. *Terminal matching* is available in the user interface via a click on the question mark symbol when hovering a terminal. Making use of the widget semantic model, we query widgets that can be connected to the input and output terminal(s) of a given widget.

The *automatic mashup composition* module is also included in the editor. It can use all widgets in the current collection and discover all possible ways to construct complete mashup branches starting from any input or output terminal of arbitrary widget.

Multiple users can collaboratively edit the same mashup. Each editing mashup is assigned a UUID which a user can send to other people. This UUID can be used to access and collaboratively create and edit the mashup.

All operations such as *adding/removing a widget to/from the mashup*, *connecting two widgets*, *resizing a widget* are propagated and synchronized in all editors sharing the same UUID. We make use of Web Application Messaging Protocol (WAMP) WebSocket framework with the publish/subscribe model to implement this feature.

The combination of server widgets and the collaborative editor has great potential for collaborative data integration across various devices. Each collaborator can use their own widget collection that may contain private server widgets. These private widgets are black boxes to others and work as a data collector; they can, for example, provide users' private data from their mobile phones, databases, cloud services, etc. selectively for a shared data integration task. This approach gives users full control over the output data of their private widget. As soon as they stop the widget, their data is no longer shared.

5.2.2 Linked Widget Annotator

The *widget annotator* allows developers to create and annotate widgets correctly and efficiently. Developers simply drag and drop and then configure three components called *Widget Model*, *Object*, and *Relation* to visually define their widget models.

Figure 5.2 is an example that illustrates the definition of a semantic model for the *POI Search* widget (cf. Figure 3.6). In the *Widget Model*, we declare input and output terminals of the widget. Their data models – arrays of *Thing* and *Feature* objects – are defined in the *Object* components. We request the *wgs84:location* property from the *Thing* input and define its domain, using another *Object* component. Similarly, we define the properties for the output of the widget. Finally, we make use of a *Relation* component to specify the *geo:nearby* relation between the input *Thing* and the output *Feature* objects.

After that, the system automatically generates the OWL description file for the model as well as the corresponding HTML widget file. The HTML file represents either the source code of the *client widget*, or the *client user interface* of the *server widget*. It includes the injected JavaScript code snippet required for the widget communication protocol and sample *JSON-LD* input/output of the widget according to the defined model. Based on that, developers can implement the widget's processing function of the *client widget* or the *remote executor* of the *server widget*, which receives input from preceding and returns output to succeeding widgets.

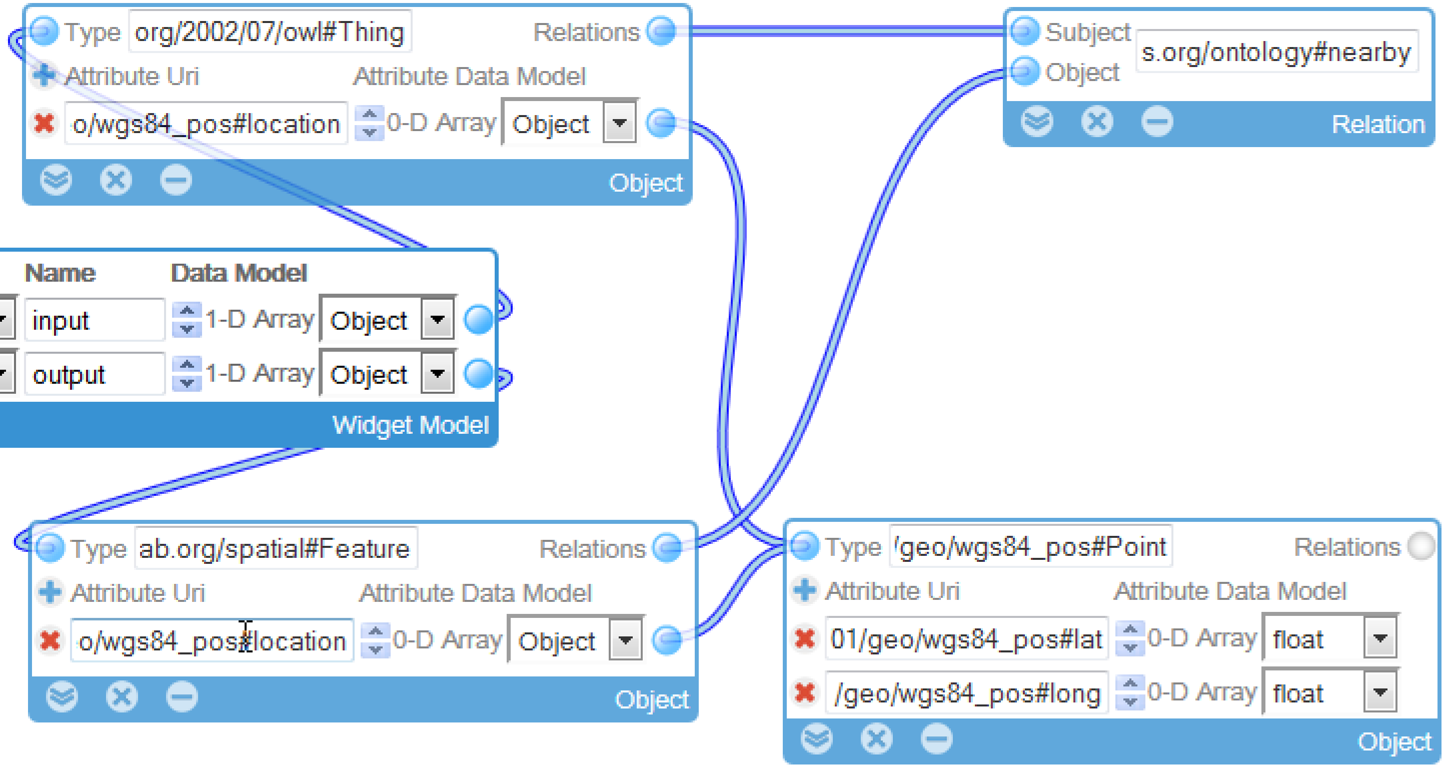


Figure 5.2: Visual model defined for the POI Search widget

Finally, as soon as they have deployed their widgets, developers can submit their work to the framework where it will be listed and can be reused with other available widgets; in particular, widget annotations are published into the LOD repository of widgets which can be accessed via a SPARQL endpoint[4].

5.2.3 Semantic Widget Search

In line with the growth of open data sources, the number of available widgets can also be expected to grow rapidly. In this case, to ensure that users can find widgets on the framework, we provide a *semantic search* feature in addition to conventional search methods which are based on keywords, categories, tags, etc. This and the *terminal matching* module are two effective data exploration tools.

Because the widgets' RDF metadata is openly available via the SPARQL endpoint, other third parties can also develop their own widget-search tool. The search we provide is similar to the annotator tool, but it is simpler and directed at end users.

By defining the model constraints for input/output, they, for example, can find widgets which receive and return objects associated with location properties as shown in Figure 5.3. Based on the defined model, a SPARQL query (cf. Listing 5.1) is generated and executed so that a list of located widgets is returned to users. Users can use short names for classes and properties in their model. Moreover, we provide *tolerant search* for users. Tolerant search uses ontology alignment methods [110] to return widgets that have models similar to the specification. To this end, based on the LOD vocabulary statistics offered by the Linked Open Vocabulary web site[5], we collect a number of common ontologies and make use of the Alignment API[6] to identify equal LOD concepts and properties. This allows us to modify the SPARQL query so that widgets whose models use similar concepts or properties as the ones specified in the user-defined searching model can be located.

5.3 Implementation Details and Lessons Learned

Table 5.1 shows the list of libraries and tools used to implement the Linked Widgets framework.

As presented in Section 3.6.2, we decide to follow the WebSocket approach to implement the server widget's *remote executor* as a WebSocket client, and the *mashup execution coordinator* as a WebSocket server. To this end, we make use of WAMP[7].

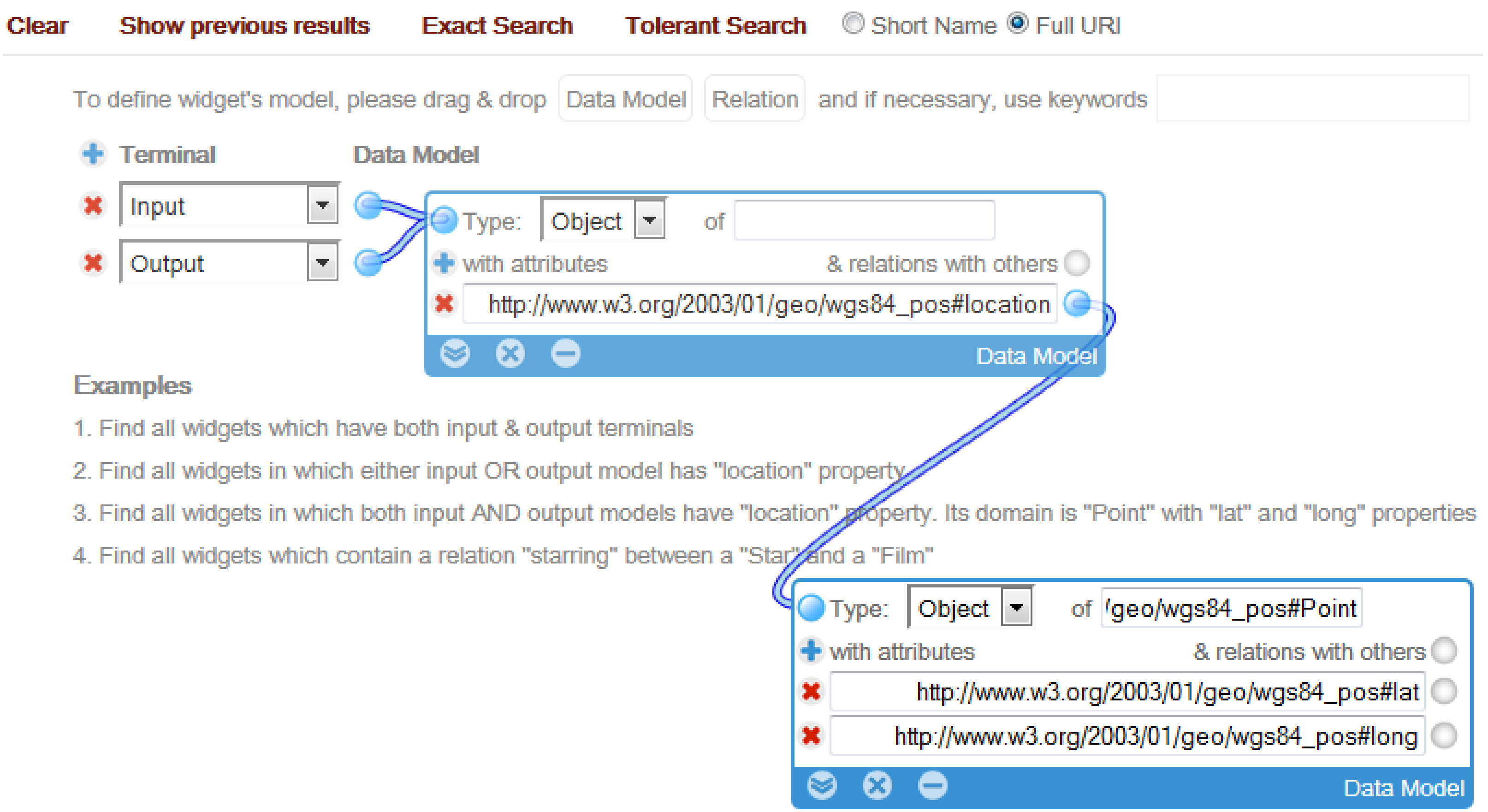
WAMP is an open standard WebSocket subprotocol that provides two application messaging patterns, i.e., remote procedure calls and publish/subscribe. It has client implementations for many programming language such as JavaScript, Java, Python, Erlang, C++, C#, Objective-C (for iOS), or PHP. As a result, it allows to develop

[4] `http://ogd.ifs.tuwien.ac.at/sparql` (accessed Nov. 01, 2015)

[5] `http://lov.okfn.org/dataset/lov/` (accessed Nov. 01, 2015)

[6] `http://alignapi.gforge.inria.fr/` (accessed Nov. 01, 2015)

[7] `http://wamp.ws/` (accessed Nov. 01, 2015)



To define widget's model, please drag & drop Data Model Relation and if necessary, use keywords

Input

Output

Type: Object of

http://www.w3.org/2003/01/geo/wgs84_pos#location

Type: Object of 'geo/wgs84_pos#Point

http://www.w3.org/2003/01/geo/wgs84_pos#lat

http://www.w3.org/2003/01/geo/wgs84_pos#long

Examples

1. Find all widgets which have both input & output terminals
2. Find all widgets in which either input OR output model has "location" property
3. Find all widgets in which both input AND output models have "location" property. Its domain is "Point" with "lat" and "long" properties
4. Find all widgets which contain a relation "starring" between a "Star" and a "Film"

Figure 5.3: Widget search that defines the input and output semantic models

Purpose	**Language, library, and tool**	URL
Integrated Development Environment	Eclipse Luna	`https://www.eclipse.org/`
Main programming language	Java	
Web server	Apache Tomcat 7.0.65	`http://tomcat.apache.org/`
Ontology language	OWL Lite	`http://www.w3.org/TR/owl-features/`
Ontology editor	Protégé 4.3	`http://protege.stanford.edu/`
JavaScript framework for interactive web application	Google Web Toolkit (GWT) 1.9.15	`http://www.gwtproject.org/`
WebSocket protocol	WAMP 2.0	`http://wamp-proto.org/to`
JavaScript implementation of a WAMP client (which is used to implement client widgets)	Autobahn	`http://autobahn.ws/js/`
Java implementation of a WAMP client (which is used to implement server widgets)	Jawampa 0.4.1	`https://github.com/Matthias247/jawampa`
Java implementation of a WAMP server (which is used to implement the *mashup execution coordinator* widgets)	Jawampa 0.4.1	
Web services	RESTEasy 3.0.5	`http://resteasy.jboss.org/`
Widget input and output data format	JSON-LD	`http://json-ld.org/`
JSON data processing	gson 2.2.4	`https://github.com/google/gson`
Semantic processing	Apache Jena 4.2.3	`https://jena.apache.org/`

Table 5.1: List of used libraries and tools

Listing 5.1: A SPARQL query to search for widgets based on a defined model

```
PREFIX lw: <http://linkedwidgets.org/ontology/>
PREFIX wgs84: <http://www.w3.org/2003/01/geo/wgs84_pos#>

SELECT DISTINCT ?widget WHERE
{
 ?widget lw:hasWidgetModel ?widgetModel.
 ?widgetModel a lw:WidgetModel.
 ?widgetModel lw:hasInput [lw:hasDataModel ?iDataModel].
 ?iDataModel wgs84:location
            [
              a wgs84:Point;
              wgs84:lat [];
              wgs84:long [];
            ]
 ?widgetModel lw:hasOutput [lw:hasDataModel ?oDataModel].
 ?oDataModel wgs84:location
            [
              a wgs84:Point;
              wgs84:lat [];
              wgs84:long [];
            ]
}
```

server widgets for various computing environments. We currently use Autobahn for the JavaScript implementation of a WAMP client and Jawampa for the Java implementation of a WAMP client and WAMP server.

As the main programming language is Java, we use Apache Jena for semantic processing, and use *gson* for JSON data processing. Eclipse Luna, Apache Tomcat, and Protégé are used as the integrated development environment, web server, and ontology editor, respectively.

In the following, we discuss lessons learned while implementing the prototype of the framework, which can be useful for developers of applications on top of open data.

5.3.1 Frameworks for Interactive Open Data Web Applications

Web platforms can provide an ideal environment for creating accessible, sharable, extensible, and maintainable applications. Interactive web applications typically rely on JavaScript libraries to provide rich graphical user interfaces on the client side (e.g., drag and drop). Choosing an appropriate library is crucial and may considerably reduce development effort and improve results. This section sets up a guideline for novice developers of open data web applications.

The are many free and open source JavaScript libraries/frameworks. For the implementation of our Linked Widget framework, we evaluate (i) YUI[8] – a free, open source JavaScript and Cascading Style Sheets (CSS) library for building rich interactive web applications, (ii) WireIt[9] – an open-source JavaScript library to create web graph editors for dataflow applications, (iii) Sencha Ext JS[10] – a JavaScript framework with a Model-View-Controller architecture and modern widgets, (iv) GWT[11] – a development toolkit for building complex browser-based applications, and (v) SmartGWT[12] – a GWT-based framework featuring a rich palette of UI elements.

Before deciding to use a library, developers have to address a number of questions. First, how much development time is available; is the result supposed to be a prototype or a ready-to-use product? Next, are there any requirements regarding compatibility with devices and browsers (e.g., touch devices and different browsers and versions)? What is the maximum allowable size for the loaded web resources? Which UI elements will be used in the application? Finally, developers have to read the documentation of potential libraries/frameworks to select the most suitable one for their application.

If minimizing the size of the necessary web resource (i.e., images, CSS and JavaScript code loaded for executing the application) is an important consideration, then YUI and GWT are good options because they allow developers to select the modules they would like to use on their page instead of the whole library. Other frameworks, e.g., SmartGWT, will result in the client browser loading around 4 MB in total, even for a single and simple feature.

After carefully evaluating the alternative libraries/frameworks, we found that GWT meets most of our requirements. Essentially, GWT allows web developers to create and maintain complex JavaScript front-end applications in Java. In addition to its basic user interface elements, which can be inherited and extended easily, a large number of advanced elements contributed by the GWT community are available. Developers write their application in Java, which can then be compiled into optimized JavaScript by the GWT Java-to-JavaScript compiler. The compiler itself ensures that web applications run on different browsers.

Finally, making use of GWT helps developers to not only rapidly develop their prototype applications, but also makes it straightforward to finalize it into a complete product. Using other frameworks to extend already supported UI elements with new features, e.g., maximizing/minimizing a window, developers have to read, understand, and add their new source code to complicated and intricate JavaScript and CSS code of the library, which is difficult and time consuming. GWT easily supports such tasks, because its implementation language is Java – an object oriented programming language in which the concept of inheritance is much clearer than in JavaScript – an object-based language only.

[8] `https://yuilibrary.com/` (accessed Nov. 01, 2015)

[9] `http://neyric.github.io/wireit/docs/` (accessed Nov. 01, 2015)

[10] `http://www.sencha.com/products/extjs/` (accessed Nov. 01, 2015)

[11] `http://www.gwtproject.org/` (accessed Nov. 01, 2015)

[12] `http://code.google.com/p/smartgwt/` (accessed Nov. 01, 2015)

5.3.2 Data Format for Lightweight Semantic Applications

In lightweight semantic applications which include on-the-fly data processing and data transmission, it is important to choose an appropriate data format. A good decision will save a considerable amount of resources, i.e., CPU power, memory, and time.

We evaluated RDF, OWL, XML, JSON, and JSON-LD as potential data formats for our mashup framework. Among those, we found that JSON-LD, which "*combines the simplicity, power, and web ubiquity of JSON with the concepts of Linked Data*" [111] is most appropriate for our needs. Since January 2014 it has been an official web standard recommended by the W3C. Compared to RDF, JSON-LD is more human-readable and takes less memory to present the same information. Additionally, in simple cases of Linked Widget interaction, where the output data model of a widget fits the input data model of another widget exactly (i.e., they have exactly the same structure or the output is a subset of the input), due to the JSON format, widgets can directly receive data from others without further processing tasks. In more complex cases, the output of a widget needs to be modified to be compatible with the input of another widget. In these cases, JSON-LD enables the framework to create a SPARQL query to perform this additional data adaption task.

5.4 Example Applications in the Geospatial Context

Geospatial data plays an increasingly important role in planning and decision-making processes across a broad range of industries and information sectors. The quantity and variety of spatial data is increasing rapidly and there is an abundance of opportunities to integrate them with other sources. The Linked Widgets mashup framework can address data heterogeneity and allow for data exploration and integration in a flexible and automatic manner. In this section, we demonstrate the capabilities of the framework to facilitate integration and reuse of open data sources, in the context of geospatial data. The framework is also applicable to a variety of other domains such as environmental, streaming, statistical, or smart city data.

5.4.1 Introduction to Geospatial Data

Geospatial data is increasingly becoming an integral part of our everyday lives. The amount and variety of such data available as well as the adoption of services that make use of it is increasing rapidly. For instance, Google, Yahoo, and Apple have each developed online GIS systems; various governments, e.g., the United Kingdom, Ireland, and Austria, have published transportation network data, including, for example, the locations of bus stops or metro stations. Location-based services such as Foursquare or Yelp also rely heavily on geographic data and expose their functionalities to developers via APIs. Flickr, as another example, provides APIs to search for photos taken nearby a specific location.

As a result, we can today locate interesting holiday spots all over the world from our desk. To plan a trip, we retrieve images and travel descriptions via search engines; we may also obtain statistical information such as economic indicators, crime rates, traffic accidents, etc., to decide the best places. Next, we check the weather conditions and history to find out the best time for a trip. Finally, we search for points of interest to see along our journey.

This example illustrates that geospatial information plays an important role in linking and combining data through shared locations [112]. It can be a powerful "glue" to integrate information across domains [113]. Realizing this vision, however, is still difficult because data is typically heterogeneous and not well organized.

A lot of research has been conducted to facilitate integration by adding semantics to geospatial data. As of January 2015, the LOD cloud diagram[13] includes 24 published geographical datasets. GeoNames[14] is a central hub of those datasets. It contains more than 10 million geographic names and 9 million unique features. Another dataset, LinkedGeoData[15], enriches the web of data with geospatial information. It publishes POI data from Open Street Map in RDF format. Furthermore, many other LOD sources are linked to these geographical datasets. DBpedia[16] also includes a lot of geographic data, as is highlighted by the fact that the *Place* class is the second most frequently used class in DBpedia (the most widely used class is *Person*). Out of 4.58 million things described in the English DBpedia version, 735,000 are places.

As a consequence, there are abundant opportunities to integrate the massive amounts of available spatial and non-spatial data in innovative ways. To facilitate such integration, however, we need to tackle a number of challenges in the design of an integration framework: (i) to support widespread reuse of geospatial data, it should not be restricted to particular datasets or problem domains, (ii) it should tackle the problem of data heterogeneity, i.e., to connect a geospatial data source with other sources of geospatial or arbitrary data, e.g., weather data, image data, statistics, transportation data irrespective of the formats used – e.g., CSV, XML, JSON, RDF, JSON-LD, (iii) it should cope with a variety of LOD vocabularies – i.e., different URIs for the same term/resource, (iv) it should foster reusability of data processing tasks, e.g., to collect, clean, enrich, integrate and visualize data, (v) it should allow users to dynamically explore, link, and integrate multiple data sources via simple operations, and (vi) it should deliver data integration applications to end users and allow them to explore, use, and share the applications. To demonstrate how the Linked Widgets framework can address these challenges, consider a number of data sources and respective widgets in the next sections.

5.4.2 Data Sources

Based on geographic objects with latitude and longitude properties, datasets in the LOD cloud can be queried dynamically and linked to each other. We use five LOD datasets and three other data sources in our examples:

[13] `http://lod-cloud.net/` (accessed Nov. 01, 2015)

[14] `http://www.geonames.org//` (accessed Nov. 01, 2015)

[15] `http://linkedgeodata.org/` (accessed Nov. 01, 2015)

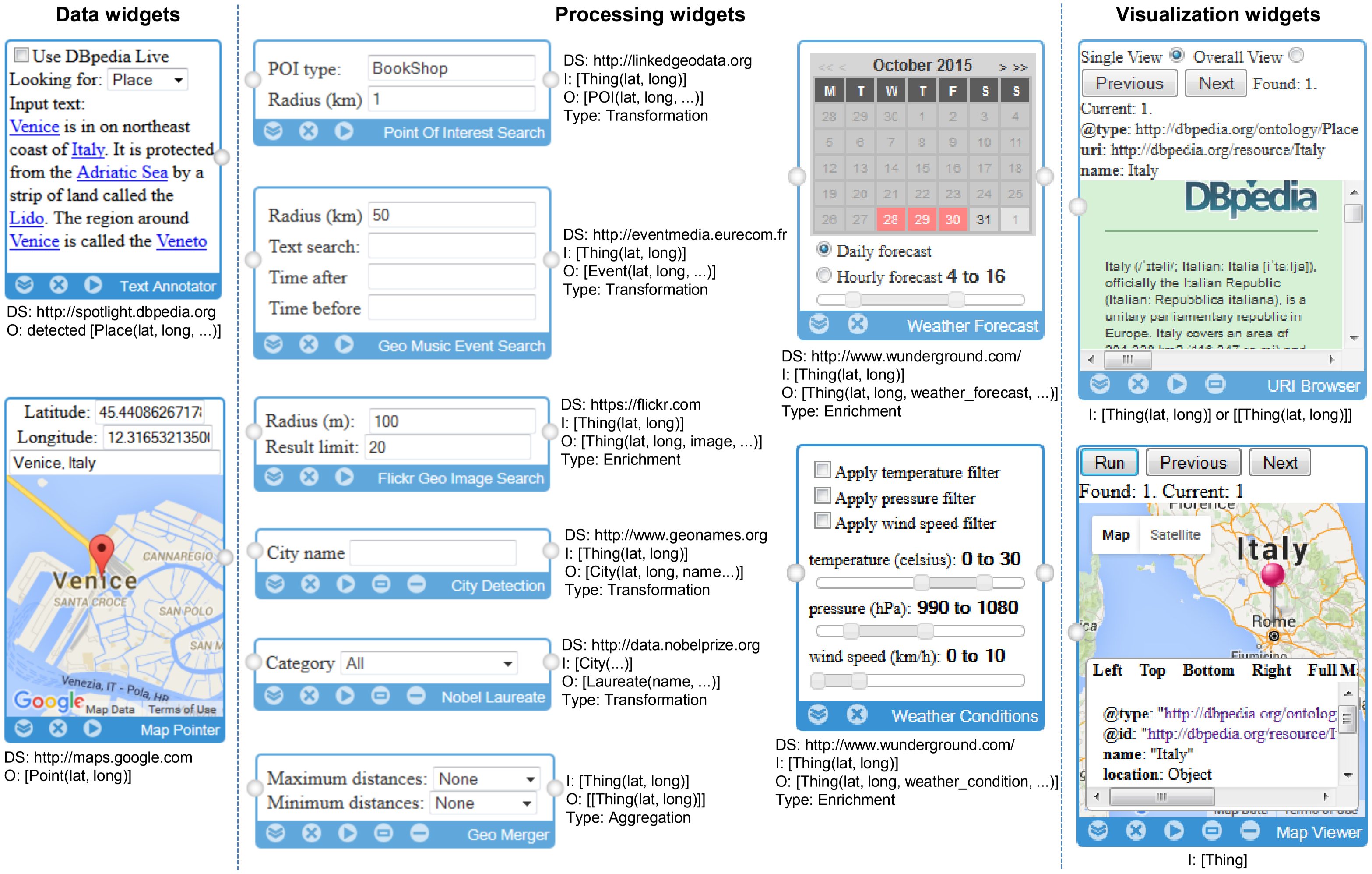
[16] `http://dbpedia.org/` (accessed Nov. 01, 2015)

1. `http://linkedgeodata.org` – Publishes data collected by the Open Street Map project as RDF data.
2. `http://www.geonames.org` – Contains more than 10 million geographical names and consists of over 8 million unique features including 2.8 million populated places and 5.5 million alternate names. *GeoNames* data are linked to DBpedia and other RDF datasets.
3. `http://spotlight.dbpedia.org` – A service looking for approximately 3.5 million things of unknown or 320 known types in text and linking them to their global unique identifiers in DBpedia.
4. `http://eventmedia.eurecom.fr` – A LOD dataset composed of events and media descriptions associated with these events which are obtained from three large public event directories, i.e., last.fm, eventful and upcoming.
5. `http://data.nobelprize.org/snorql` – Contains information about who has been awarded the *Nobel Prize*, when, in what prize category and the motivation, as well as basic information about the laureates.
6. `https://flickr.com` – An image and video hosting website. It provides a free API to access 5 billion photos with valuable metadata such as tags, geolocation, etc.
7. `http://www.wunderground.com/` – A network of more than 140,000 personal weather stations that allows for exact weather forecasts based on actual weather data points. It exposes APIs for developers to access the current weather conditions of every station as well as the weather forecast for every city in the world; the APIs are free for 500 calls per day.
8. `http://map.google.com` – Offers satellite imagery, street maps, and street view perspectives which can be accessed through its API services.

5.4.3 Sample Geospatial Widget Collection

In our example use cases, we use twelve widgets (cf. Figure 5.4) organized into three layers, i.e., data, processing, and presentation layers.

1. **Map Pointer** – a data widget. Users can define a point on a map. The point's latitude and longitude is then returned as output.
2. **Text Annotator** – a data widget. This widget detects a list of Places from text, using the DBpedia Spotlight service.
3. **Point of Interest Search (POI Search)** – a processing widget. This widget leverages the LinkedGeoData repository to find semantically encoded POIs. Users can influence the output by providing parameters. Users can select the type of POI and the radius of retrieved POIs with respect to the incoming location, which is any kind of objects with *wgs84:lat* and *wgs84:long* attributes.
4. **Music Event Search** – a processing widget. This widget receives any kind of objects with *wgs84:lat* and *wgs84:long* attributes and transforms them into music events that were (or will be) organized nearby.
5. **Flickr Geo Image Search** – a processing widget. By using the Flickr Image



□ Use DBpedia Live
Looking for: Place
Input text:
Venice is in on northeast coast of Italy. It is protected from the Adriatic Sea by a strip of land called the Lido. The region around Venice is called the Veneto

DS: http://spotlight.dbpedia.org
O: detected [Place(lat, long, ...)]

Latitude: 45.4408626717
Longitude: 12.31653213500
Venice, Italy

DS: http://maps.google.com
O: [Point(lat, long)]

POI type: BookShop
Radius (km) 1

DS: http://linkedgeodata.org
I: [Thing(lat, long)]
O: [POI(lat, long, ...)]
Type: Transformation

Radius (km) 50
Text search:
Time after
Time before

DS: http://eventmedia.eurecom.fr
I: [Thing(lat, long)]
O: [Event(lat, long, ...)]
Type: Transformation

Radius (m): 100
Result limit: 20

DS: https://flickr.com
I: [Thing(lat, long)]
O: [Thing(lat, long, image, ...)]
Type: Enrichment

City name

DS: http://www.geonames.org
I: [Thing(lat, long)]
O: [City(lat, long, name...)]
Type: Transformation

Category All

DS: http://data.nobelprize.org
I: [City(...)]
O: [Laureate(name, ...)]
Type: Transformation

Maximum distances: None
Minimum distances: None

I: [Thing(lat, long)]
O: [[Thing(lat, long)]]
Type: Aggregation

● Daily forecast
○ Hourly forecast 4 to 16

DS: http://www.wunderground.com/
I: [Thing(lat, long)]
O: [Thing(lat, long, weather_forecast, ...)]
Type: Enrichment

□ Apply temperature filter
□ Apply pressure filter
□ Apply wind speed filter
temperature (celsius): 0 to 30
pressure (hPa): 990 to 1080
wind speed (km/h): 0 to 10

DS: http://www.wunderground.com/
I: [Thing(lat, long)]
O: [Thing(lat, long, weather_condition, ...)]
Type: Enrichment

Single View ● Overall View ○
Previous Next Found: 1.
Current: 1.
@type: http://dbpedia.org/ontology/Place
uri: http://dbpedia.org/resource/Italy
name: Italy

Italy (/ˈɪtəli/; Italian: Italia [iˈtaːlja]), officially the Italian Republic (Italian: Repubblica italiana), is a unitary parliamentary republic in Europe. Italy covers an area of

I: [Thing(lat, long)] or [[Thing(lat, long)]]

Run Previous Next
Found: 1. Current: 1

@type: "http://dbpedia.org/ontolog
@id: "http://dbpedia.org/resource/I
name: "Italy"
location: Object

I: [Thing]

Figure 5.4: Geospatial widget collection

Search API[17] this widget enriches location data with images.

6. **Weather Forecast** – a processing widget. This widget enriches the input locations with the Wunderground weather forecasts.
7. **Weather Conditions** – a processing widget. This widget enriches the input locations with the weather conditions of the nearest Wunderground station.
8. **City Detection** – a processing widget. This widget uses the GeoNames service to find Cities that the input objects belong to and looks up extra information via DBpedia, e.g., area or population.
9. **Nobel Laureate** – a processing widget. This widget takes Cities as input and returns a list of Laureates born in those Cities.
10. **Geo Merge** – a processing widget. This widget merges two or more lists of point data into a single list of pairs based on their distance. Users can specify a minimum and a maximum distance between points. The Geo Merge widget therefore serves two purposes, i.e., merging of two inputs into one output and filtering based on distance constraints.
11. **Map Viewer** – a visualization widget. This widget displays points on a map. It is typically used to display the final results of a geospatial mashup.
12. **URI Browser** – a visualization widget. This widget shows input objects one by one and the objects' corresponding URI-dereferenced page.

Detailed information about the widgets and their annotated input and output models can be found on our prototype implementation[18].

5.4.4 Sample Data Integration Use Cases

There are many ways to compose useful applications from the twelve widgets selected for our examples. Assume, e.g., that we have a touristic text introducing different beautiful spots in a city. The application depicted in Figure 5.5 then presents on overview for those places. Next to detailed information from DBpedia, the locations and images are displayed on the map. From a point specified by a user on the map, the combination in Figure 5.6 finds all pairs of nearby restaurants and banks that are less than 100 meters apart from each other.

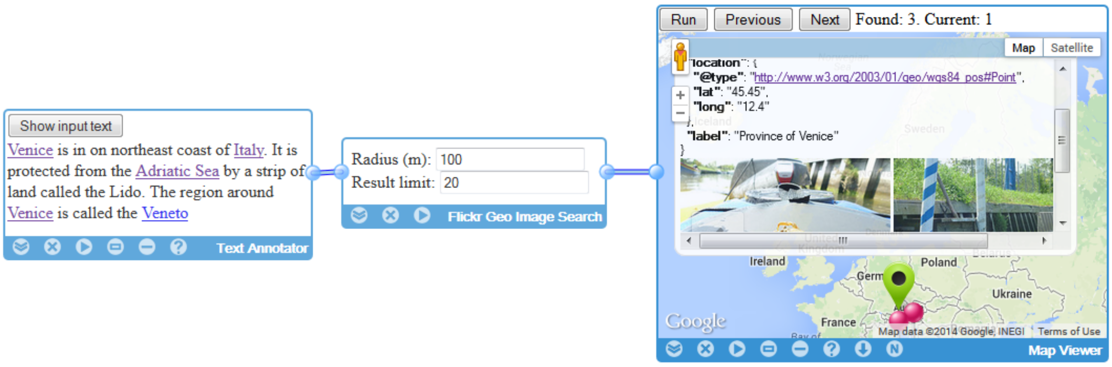
As a more complex data integration use case, the example depicted in Figure 5.7 illustrates how a mashup can be used to choose a suitable park to spend time at in the weekend. We start from a point on the map (e.g., our home location) and use the *POI Search* widget to retrieve nearby parks via a SPARQL query to the Linked Geo Data server.

Next, we add weather forecasts from Wunderground[19] using the *Weather Forecast* and *Weather Condition* widgets. We specify our preferred time period in *Weather Forecast* and select the weather conditions we are interested in (e.g., temperature or pressure) in the *Weather Condition* widget. For each park in the input, this widget obtains measurements

[17] `https://flickr.com/services/api/` (accessed Nov. 01, 2015)

[18] `http://linkedwidgets.org` (accessed Nov. 01, 2015)

[19] `http://www.wunderground.com/` (accessed Nov. 01, 2015)



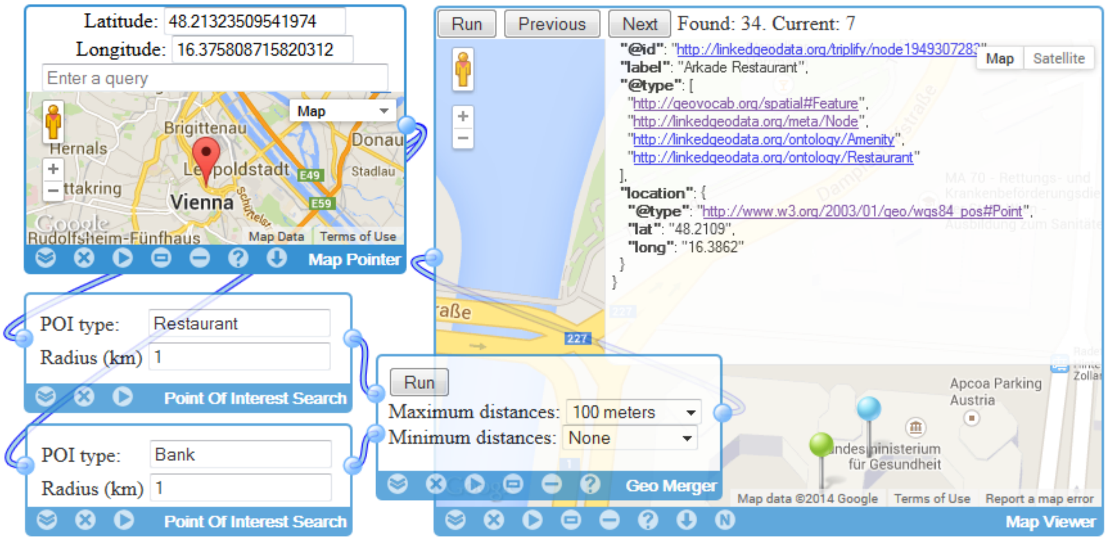
Venice is in on northeast coast of Italy. It is protected from the Adriatic Sea by a strip of land called the Lido. The region around Venice is called the Veneto

Radius (m): 100
Result limit: 20

Found: 3. Current: 1

```
"location": {
  "@type": "http://www.w3.org/2003/01/geo/wgs84_pos#Point",
  "lat": "45.45",
  "long": "12.4"
 },
 "label": "Province of Venice"
}
```

Figure 5.5: Use case 1: display famous Places detected from text on the map



Latitude: 48.21323509541974
Longitude: 16.375808715820312
Enter a query

POI type: Restaurant
Radius (km) 1

POI type: Bank
Radius (km) 1

Maximum distances: 100 meters
Minimum distances: None

Found: 34. Current: 7

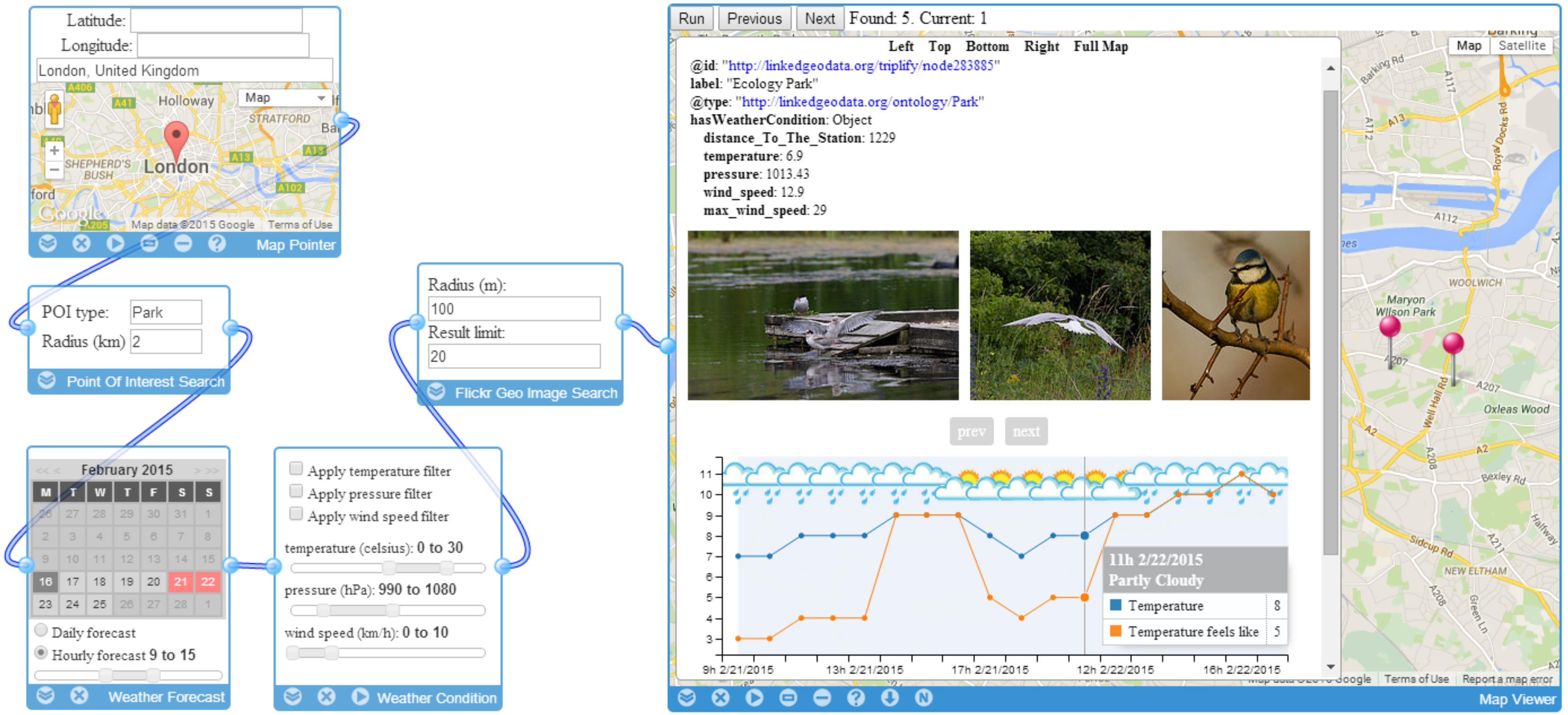
```
 "@id": "http://linkedgeodata.org/triplify/node1949307283",
 "label": "Arkade Restaurant",
 "@type": [
  "http://geovocab.org/spatial#Feature",
  "http://linkedgeodata.org/meta/Node",
  "http://linkedgeodata.org/ontology/Amenity",
  "http://linkedgeodata.org/ontology/Restaurant"
 ],
 "location": {
  "@type": "http://www.w3.org/2003/01/geo/wgs84_pos#Point",
  "lat": "48.2109",
  "long": "16.3862"
 }
}
```

Figure 5.6: Use case 2: display nearby Restaurants and Banks on the map

of the nearest station out of 140,000 Wunderground stations all over the world. We also add Flickr images to the parks based on their location using the *Flickr Geo Image* widget.

Finally, we present the collected information in the generic *Map Viewer* visualization widget. This widget accepts any kind of input with associated geographic information and is hence useful in many spatial applications. It leverages the semantics of the input to appropriately visualize the data. Location input data with *wgs84:lat* and *wgs84:long* properties are displayed as pins on the map. If the *foaf:depiction* property of an input is set, the respective images will be shown in an information window. In particular, if the value of an input property is an instance of the W3C cube *http://purl.org/linked-data/cube#DataSet* class, then the *Map Viewer* will automatically analyze the data and visualize it in a chart.

In the mashup, we display charts of the weather forecast for each park. The validity



Latitude:

Longitude:

London, United Kingdom

POI type: Park

Radius (km) 2

February 2015

○ Daily forecast

● Hourly forecast 9 to 15

□ Apply temperature filter

□ Apply pressure filter

□ Apply wind speed filter

temperature (celsius): 0 to 30

pressure (hPa): 990 to 1080

wind speed (km/h): 0 to 10

Radius (m): 100

Result limit: 20

Found: 5. Current: 1

@id: "http://linkedgeodata.org/triplify/node283885"
label: "Ecology Park"
@type: "http://linkedgeodata.org/ontology/Park"
hasWeatherCondition: Object
  distance_To_The_Station: 1229
  temperature: 6.9
  pressure: 1013.43
  wind_speed: 12.9
  max_wind_speed: 29

11h 2/22/2015
Partly Cloudy
🟦 Temperature 8
🟧 Temperature feels like 5

Figure 5.7: Use case 3: pick a park to visit at the weekend

of all connections between widgets is enforced by the framework based on the underlying semantic model. In this case, the mashup is valid because all widgets only require input as instances of *wgs84:Point* with *wgs84:lat* and *wgs84: long* values.

The final example (Figure 5.8) makes use of the relations between locations and other LOD resources. It collects and displays various types of information that are in some way related to a particular location.

The *Text Annotator* widget uses the DBpedia spotlight service to retrieve a list of recognized places based on the input text. At this point, we split the data flow to further process the detected places. In the first flow, we use the *Music Event Search* widget to send a SPARQL query to Event Media[20] to get nearby music events and display them on the *Map Viewer*. The second flow applies the *City Detection* widget, queries GeoNames[21], and retrieves DBpedia city entities corresponding to the locations. These entities are then passed on as an input to the *Nobel Laureate* widget, which queries the Nobel Prize dataset[22] to obtain all Nobel Laureates born in the detected cities. The final data is Linked Open Data with dereferenceable URIs, so we can browse them in the generic *URI Browser* widget.

As shown in Figure 5.9, there are more ways to combine the example widgets. We can create new applications by replacing two *data/visualization widgets* with each other. Moreover, one or more *processing widgets* can be added between a *data* and a *visualization widget*. By linking the widgets, the data from different LOD datasets are connected. Furthermore, with different values of parameters inside a widget, new use cases can be created. In Figure 5.6, we can replace “Bank” by “Park”, and add one more instance of both *POI Search* and *Geo Merger* to find combinations of restaurant, park and book shop near each other.

5.5 Hybrid Mashup Example Use Cases

In the following, we illustrate the use of hybrid mashups by means of two practical use cases.

5.5.1 Streaming and Persistent Mashups

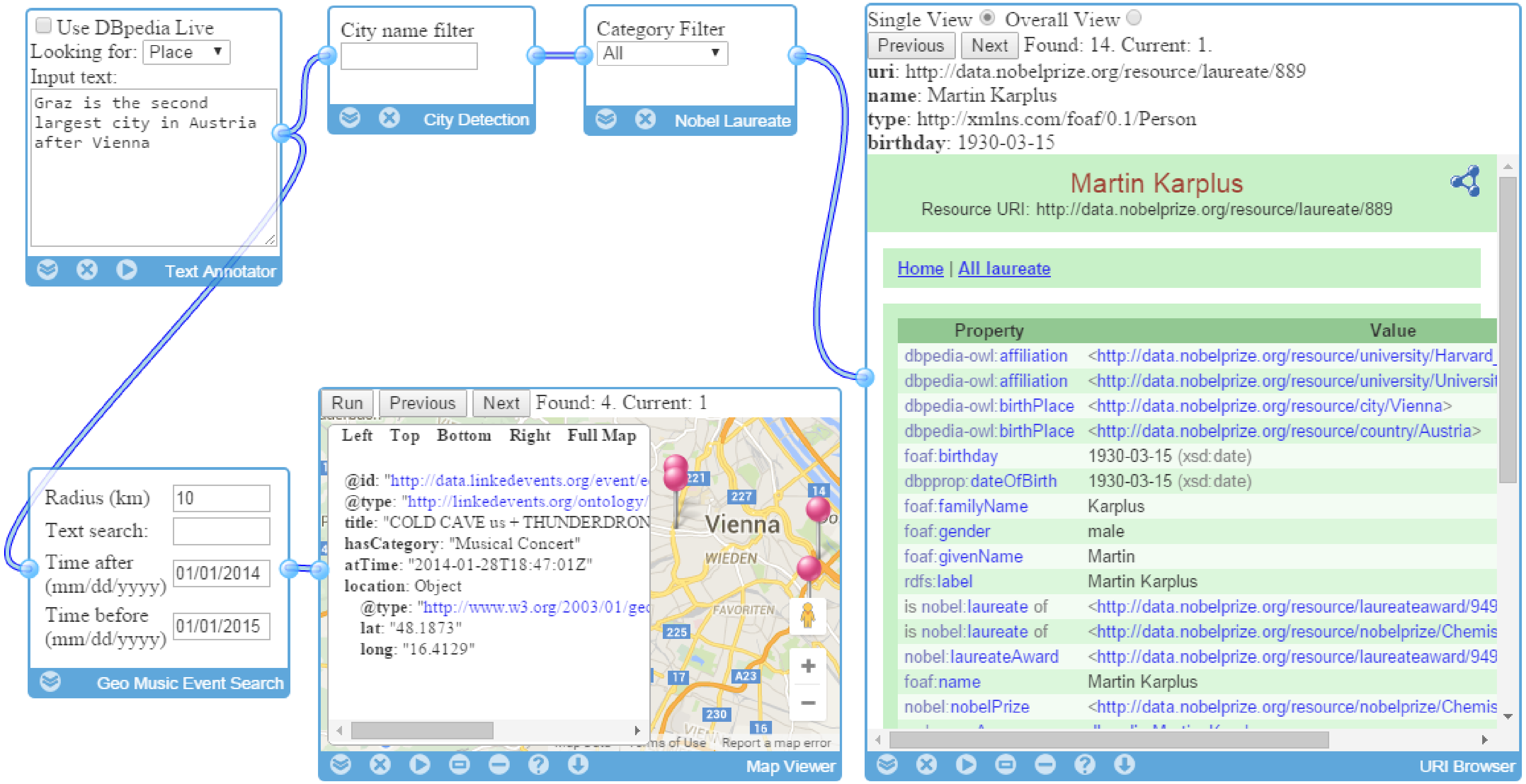
The example depicted in Figure 5.10 illustrates a practical use case of streaming and persistent mashups. The mashup consists of two client widgets (i.e., the *Map Pointer* and the *Chart Viewer* widget) and two server widgets (i.e., the *Weather Observation* and *Cube Merger* widgets).

Using the two instances of the *Map Pointer* widget, we first obtain the latitude and longitude of two arbitrary locations in the world (e.g., Vienna, Austria and Bethlehem, Pennsylvania). Next, the two instances of the *Weather Observation* widget initiate two back-end services that collect the temperature of the input locations every ten minutes;

[20] `http://eventmedia.eurecom.fr/` (accessed Nov. 01, 2015)

[21] `http://www.geonames.org/` (accessed Nov. 01, 2015)

[22] `http://data.nobelprize.org/` (accessed Nov. 01, 2015)



□ Use DBpedia Live
Looking for: Place
Input text:

```
Graz is the second
largest city in Austria
after Vienna
```

City name filter

Category Filter: All

Radius (km): 10
Text search:
Time after (mm/dd/yyyy): 01/01/2014
Time before (mm/dd/yyyy): 01/01/2015

Run | Previous | Next | Found: 4. Current: 1

Left Top Bottom Right Full Map

@id: "http://data.linkedevents.org/event/e
@type: "http://linkedevents.org/ontology/
title: "COLD CAVE us + THUNDERDRON
hasCategory: "Musical Concert"
atTime: "2014-01-28T18:47:01Z"
location: Object
  @type: "http://www.w3.org/2003/01/geo
  lat: "48.1873"
  long: "16.4129"

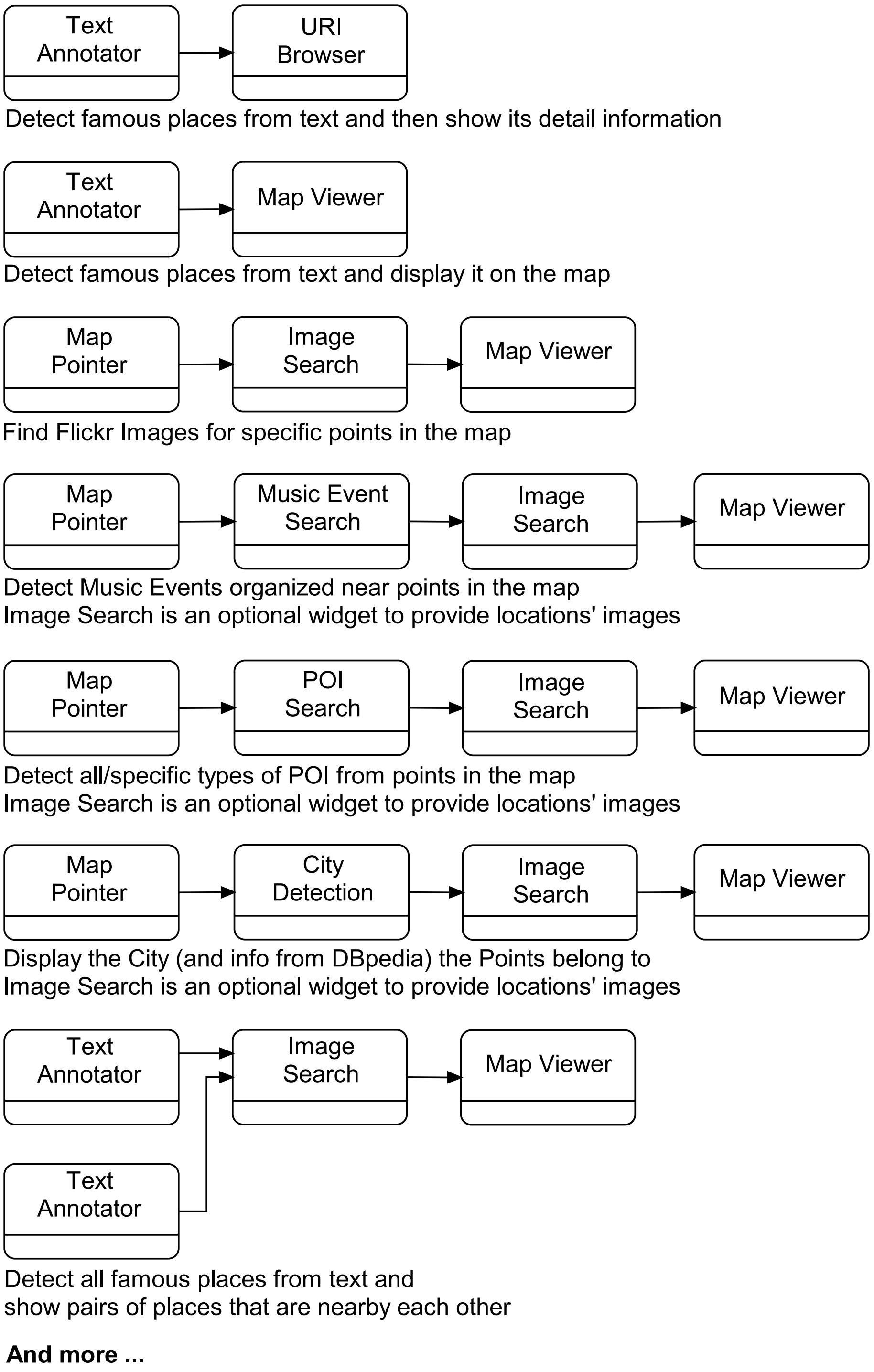
Single View ● Overall View ○
Previous | Next | Found: 14. Current: 1.
uri: http://data.nobelprize.org/resource/laureate/889
name: Martin Karplus
type: http://xmlns.com/foaf/0.1/Person
birthday: 1930-03-15

Martin Karplus
Resource URI: http://data.nobelprize.org/resource/laureate/889

Home | All laureate

Property	Value
dbpedia-owl:affiliation	<http://data.nobelprize.org/resource/university/Harvard_
dbpedia-owl:affiliation	<http://data.nobelprize.org/resource/university/Universit
dbpedia-owl:birthPlace	<http://data.nobelprize.org/resource/city/Vienna>
dbpedia-owl:birthPlace	<http://data.nobelprize.org/resource/country/Austria>
foaf:birthday	1930-03-15 (xsd:date)
dbpprop:dateOfBirth	1930-03-15 (xsd:date)
foaf:familyName	Karplus
foaf:gender	male
foaf:givenName	Martin
rdfs:label	Martin Karplus
is nobel:laureate of	<http://data.nobelprize.org/resource/laureateaward/949
is nobel:laureate of	<http://data.nobelprize.org/resource/nobelprize/Chemis
nobel:laureateAward	<http://data.nobelprize.org/resource/laureateaward/949
foaf:name	Martin Karplus
nobel:nobelPrize	<http://data.nobelprize.org/resource/nobelprize/Chemis

Figure 5.8: Use case 4: transform geospatial data into Linked Open Data resources



Detect famous places from text and then show its detail information

Detect famous places from text and display it on the map

Find Flickr Images for specific points in the map

Detect Music Events organized near points in the map
Image Search is an optional widget to provide locations' images

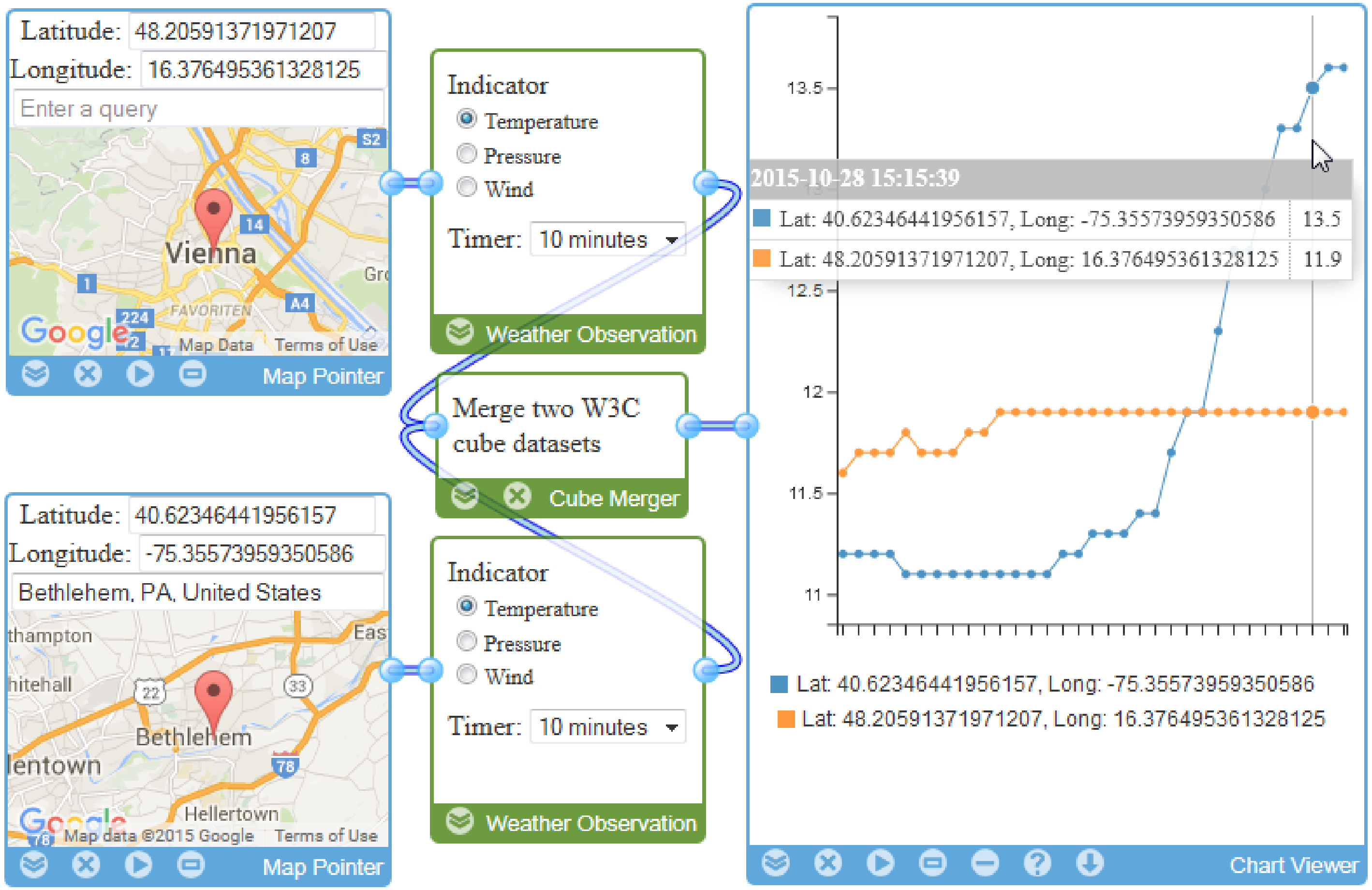
Detect all/specific types of POI from points in the map
Image Search is an optional widget to provide locations' images

Display the City (and info from DBpedia) the Points belong to
Image Search is an optional widget to provide locations' images

Detect all famous places from text and
show pairs of places that are nearby each other

And more ...

Figure 5.9: Combinations of available widgets



Latitude: 48.20591371971207
Longitude: 16.376495361328125
Enter a query

Latitude: 40.62346441956157
Longitude: -75.35573959350586
Bethlehem, PA, United States

Indicator
● Temperature
○ Pressure
○ Wind
Timer: 10 minutes

Indicator
● Temperature
○ Pressure
○ Wind
Timer: 10 minutes

Merge two W3C cube datasets

2015-10-28 15:15:39

🟦 Lat: 40.62346441956157, Long: -75.35573959350586	13.5
🟧 Lat: 48.20591371971207, Long: 16.376495361328125	11.9

🟦 Lat: 40.62346441956157, Long: -75.35573959350586
🟧 Lat: 48.20591371971207, Long: 16.376495361328125

Figure 5.10: Vienna and Bethlehem weather comparison

the temperature is taken from the nearest station (out of 140,000 Wunderground stations all over the world). As soon as these two instances retrieve new temperature data, they send the data in form of a W3C cube dataset (which has two dimensions, i.e., *location* and *observed time*, and one measure, i.e., *temperature* in our example) to the *Cube Merger* widget. The *Cube Merger* widget aggregates this new data with the data it has collected since the mashup started. Finally, the *Chart Viewer* analyzes the aggregated W3C cube dataset and visualizes the final result. Because the *Weather Observation* continuously returns new output data every ten minutes, and because both the *Weather Observation* and the *Cube Merger* are server widgets, the mashup is a streaming and persistent mashup. The web page can be closed; the mashup still runs in the back-end and presents to us the whole observations whenever the web page is opened again.

5.5.2 Collaborative Mashups

For the second example, consider the need to integrate data from several Excel and Google spreadsheets in the enterprise context. The typical process to achieve this goal is to download all of them, copy, delete columns, and create formulas to aggregate the data. These tedious tasks may take a lot of time and have to be repeated whenever the source data changes.

A mashup example that accomplishes such a task collaboratively is illustrated in Figure 5.11. It combines and visualizes sales data for a series of retail points of sale (e.g., ice cream stores).

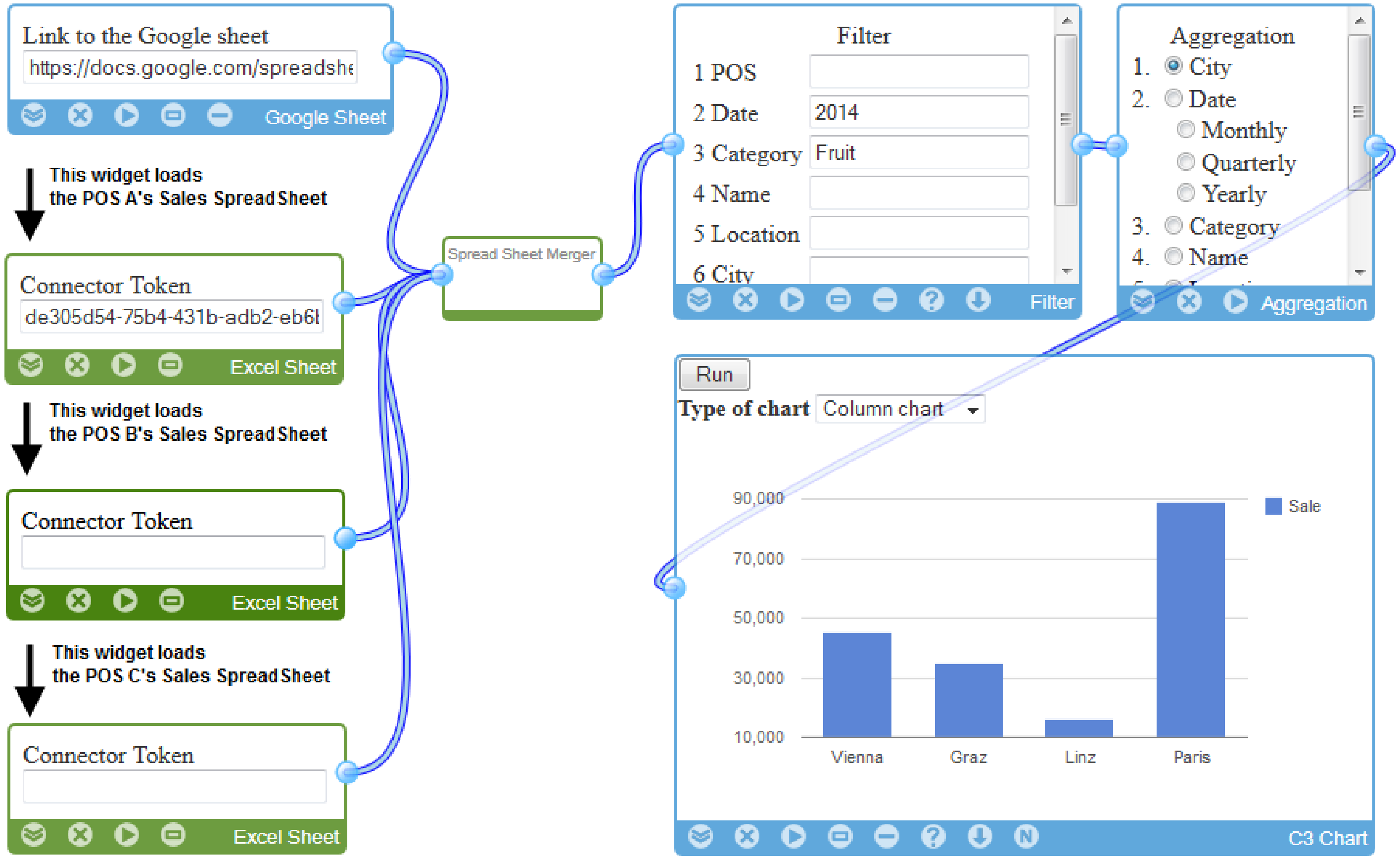
We have two types of spreadsheets: (i) a point of sale (POS) spreadsheet that contains their respective id, name, latitude, longitude, city, country; and (ii) three sales spreadsheets, each listing the number items per category sold per day at that point of sale. Whereas the point of sale spreadsheet is on Google Drive, the three sale spreadsheets of POS A, B, C are stored on personal computers of the local branch managers, who update the data every day by adding new rows into the spreadsheet.

To integrate the data, the branch and headquarter managers may collaboratively build their single shared mashup using our web based mashup editor as follows. They first use the *Google Sheet* widget to load the shop spreadsheet from Google Drive and convert it into datasets that follow the W3C data cube vocabulary[23].

Next, each branch manager contributes their respective *Excel Sheet* widget (which is a *server widget*) to load and convert her sales data into a cube dataset. To this end, they installs the *Excel Sheet* server widget as a standalone application on their device; we hence have multiple *remote executors* of this *server widget.* To differentiate between these respective deployments of the server widget so that a *client user interface* can connect to its respective counterpart (e.g., the client interface of POS A is connected to A's *Excel Sheet* server widget), each deployment is associated with a unique token.

Figure 5.11 is the mashup as seen by point of sale manager A; she has to enter the token of her *Excel Sheet* server widget's deployment into her *client interface* of the widget. The tokens of other shops are filled out by the respective branch managers. To prepare

[23] http://www.w3.org/TR/vocab-data-cube/ (accessed Nov. 01, 2015)



Link to the Google sheet: https://docs.google.com/spreadshe

Connector Token: de305d54-75b4-431b-adb2-eb6l

Connector Token:

Connector Token:

Filter

1 POS:
2 Date: 2014
3 Category: Fruit
4 Name:
5 Location:
6 City:

Aggregation

1. ● City
2. ○ Date
   ○ Monthly
   ○ Quarterly
   ○ Yearly
3. ○ Category
4. ○ Name

Type of chart: Column chart

Figure 5.11: Collaborative mashup example for sales data integration

the data for visualization, the datasets are merged in the *Spread Sheet Merger* widget, and passed through a *Filter* and an *Aggregation* widget. The *C3 Chart* widget visualizes the final data in a chart.

The options inside the *Filter* and *Aggregation* widgets are generated automatically based on the data. They can be manipulate by all collaborators concurrently. Their view of the mashup is synchronized, i.e., changes immediately become visible to all participants.

By changing the automatically generated options inside the *Filter* and *Aggregation* widgets, various analyses can be performed easily. For example, collaborators can "compare all-time sales of all POS", "compare sales of all POS in 2014", "compare sales of fruit, milk, and chocolate items of POS *A* in 2014", "compare aggregate sales in different countries or cities", or "compare sales of fruit items in different cities in 2014" as shown in Figure 5.11.

This is a *hybrid* and *collaborative* mashup where we use four *server widgets* (i.e., three *Excel Sheet* widgets and a *Spread Sheet Merger* widget) and four *client widgets*. The data collecting tasks are performed on multiple devices. Because the *Spread Sheet Merger* widget is a *server widget*, we can access its output data at any time. Points of sale can easily be added to or removed from the mashup.

In this example, the spreadsheet data is converted into W3C cube datasets. To this end, RDF Mapping Language[24] or similar languages can be used; however, to do the conversion, the *Excel Sheet* widget currently identifies the first and the last column of a table as the dimension and measure of the cube dataset, respectively. Based on the semantic format of the cube data, the generic widgets such as the *Filter*, *Aggregation*, and *C3 Chart* widget can process and visualize the data automatically. Moreover, the semantic format can facilitate data integration; for example, we can enrich the sales data with the weather statistics to analyze the effect of the weather conditions on the ice cream sales data.

[24] `http://rml.io/` (accessed Nov. 01, 2015)

CHAPTER 6

Related Work

In this chapter[1], we dive into a number of selected mashup tools. With regard to the topic of this thesis, we categorize the tools into six main groups and present (i) widget-based mashups in Section 6.1, (ii) semantic mashups in Section 6.2, (iii) embedded mashups in Section 6.3, (iv) collaborative mashups in Section 6.4, (v) automatic mashups in Section 6.5, and (vi) natural language-supported mashups in Section 6.6. Finally, because the Linked Widgets framework does not follow the programming-by-demonstration approach, we only present Vegemite as an exemplification of this approach in Section 6.7.

6.1 Widget-based Mashups

6.1.1 Yahoo! Pipes

Yahoo! Pipes [33] is one of the most well-known widget-based mashup tools, and is the inspiration for the Linked Widgets framework and many other mashup frameworks. It is an interactive web-based editor[2] that uses the visual dataflow approach to enable users to combine pre-configured modules into a mashup. Yue [114] conducts a pilot experiment of Yahoo! Pipes and describes its positive results. Students, who are the subjects of the experiment, find it to be useful and interesting; they gain expertise in mashup development quickly, without any major difficulty.

Yahoo! Pipes provides 53 modules organized into nine categories, that is:

1. *Data Sources* modules, which collect data available on the web to further process in the mashup;
2. *User Input* modules, which allow users to define parameters for their pipes;
3. *Operator* modules, which transform and filter data flowing;
4. *URL* modules, which manipulate URLs;

[1] Parts of this chapter also appear in [1, 2, 3, 4, 5, 6, 7, 8]. The author of this thesis is also the lead author of these papers.

[2] `https://pipes.yahoo.com/pipes/` (accessed Nov. 01, 2015)

5. *String* modules, which support users in manipulating and combining text strings;
6. *Date* modules, which define and format dates;
7. *Location* modules, which can convert text to geographic locations;
8. *Number* modules, which provide basic arithmetic operations;
9. *Deprecated* modules, which are not recommended to be used.

A module has one or multiple terminals, which are depicted as small circles in the interface. Users create a mashup (which is also known as a pipe) by dragging available modules to the editor canvas and linking their terminals. The pipe can aggregate and transform data from various web services, pages and feeds. Pipes can be reused as new operators for new pipes. The pipes' final results are displayed in feeds, lists, or on maps, based on the type of the output data.

Yahoo! Pipes makes use of the visual programming technique to hide the complexity of developing a mashup; however, it requires users to be familiar with programming concepts such as *for* loops, *if* conditions, and *regular expressions.* The further limitation of Yahoo! Pipes is that it focuses on web feeds (i.e., RSS, Atoms, and RDF-feeds) and hence is capable of representing news items only. Its main idea is to create new pages by aggregating RSS feeds from various sources; it cannot manipulate other kinds of data in XML or RDF formats.

6.1.2 Apache Rave

Apache Rave[3] is a *web and social mashup engine* for data integration. It combines pre-existing projects (i.e., Apache Shindig[4] – which is an OpenSocial[5] container to host OpenSocial applications – and Apache Wookie[6] – which is a Java-based server application to upload and deploy widgets). Apache Rave, as an extendable, lightweight, and open-standards based platform, intends to host and integrate OpenSocial features with W3C web widgets[7]. It can provide collaborative and context-aware building blocks for multi-side-oriented, content-driven and social-network-oriented websites, services and platforms.

Apache Rave has implemented a lot of crucial features that a social mashup platform should support. It allows for secure and single sign-on access control, which enables users to easily personalize and share their web content via widgets and mashups. It can make use of both OpenSocial and Wookie widgets to construct a widget repository with life-cycle management (e.g., to install, update or remove widgets) and extended metadata (e.g., the categories, comments, or ratings of widgets).

[3] `https://rave.apache.org/` (accessed Nov. 01, 2015)

[4] `https://shindig.apache.org/` (accessed Nov. 01, 2015)

[5] `https://opensocial.atlassian.net/wiki/display/OSD/Specs` (accessed Nov. 01, 2015)

[6] `http://wookie.apache.org/` (accessed Nov. 01, 2015)

[7] `http://www.w3.org/TR/widgets/` (accessed Nov. 01, 2015)

6.1.3 Presto

Presto[8] is an enterprise application and mashup platform. It covers the entire application life cycle, i.e., preparing mashable data, aggregating, transforming, integrating data in mashups, and exposing the final result in various visualizations that are accessible from either desktops or mobiles.

Presto provides users with a *Presto Hub* so that they can turn a wide variety of internal, external, historical, transactional, and real-time sources into *mashable artifacts*. These artifacts can be shared, tagged, rated, or commented among user communities. The data can be collected from applications, systems, databases, or office documents due to multiple types of *Presto Add-Ons*. Presto is capable of working with large datasets; it offers support for data streaming and in-memory management.

Presto supports real-time access to data; it does not permanently save data to any type of devices. Basic user authentication, secure connections and certificates, as well as single sign-on solutions are all supported, which allows for secure access to both information sources and created mashups.

Presto empowers users with various useful tools, i.e., *App Maker*, *App Editor*, *Mashboard*, *Wires*, and *Mashup Editor*. (i) The *App Maker* is a visual wizard to build up basic applications from *mashable artifacts* without programming; based on the data format, users can choose one or multiple views such as table, map, or different kinds of charts. (ii) The *App Editor* is a web-based tool for uploading and updating custom applications. Custom applications can use common web resources such as HTML, JavaScript, and CSS. The editor aims at developers only, allowing them to easily share or package their work. (iii) The *Mashboard* is a visual, drag-and-drop designer to combine multiple applications or views in a dashboard. A wide variety of layout such as *simple columns*, *tables*, *drill downs*, *desktop layouts*, *multiple tabs* or *multiple pages* are available. (iv) The *Wires* is a web-based and drag-and-drop tool to create visual mashups from available built-in and power-packed action blocks. (v) Finally, the *Mashup Editor* is a web-based tool for developers to write and design mashups in Enterprise Mashup Markup Language (EMML) – which is a mashup language specified by the Open Mashup Alliance[9].

6.2 Semantic Mashups

Super Stream Collider (SSC) [39], MashQL [41], and DERI Pipes [40] are three platforms aimed at semantic data processing. Whereas SSC consumes live stream data only, MashQL allows users to easily create a SPARQL query by means of its custom query-by-diagram language. MashQL cannot aggregate data from different sources and its output visualization only supports text and table formats. DERI Pipes requires users to be familiar with semantic web technologies, SPARQL queries, and programming to perform semantic data processing tasks from different data sources.

[8] `http://mdc.jackbe.com/enterprise-mashup/content/about-presto` (accessed Nov. 01, 2015)

[9] `http://www.synteractive.com/News-Events/Pages/Open-Mashup-Alliance.aspx` (accessed Nov. 01, 2015)

Similar limitations apply to a Linked Data Integration Framework [115], a semantics-enabled mashup of existing Web APIs [116], and a web-based method that integrates static and dynamic sources for Linked-Data consuming applications [117]. In other words, those are data integration frameworks aimed at developers, not end users. They can be encapsulated in the *processing widgets* of our model to facilitate new usage scenarios.

The contributions [85] and [118] discuss useful semantic mashup systems, e.g., [87], [71], but lack a systematic approach for all LOD datasets. Although each system can effectively exploit only one or several datasets, they cannot be extended to more LOD datasets and do not utilize LOD interconnections. Yokohama Art Spot [119], for example, is a web mashup application that offers information on art in Yokohama by consuming three LOD datasets (LODAC Museum, Yokohama Art LOD, and PinQA) and is hence a domain-specific solution.

In general, there has so far been limited research on allowing end users to consume data in a dynamic way. Typically, users are expected to use semantic browsers to explore data and collect information by themselves. Alahmari et al. [120] evaluate fourteen semantic browsers (such as Sigma, Marbles, Disco, etc.) with respect to consumption of structured Linked Data. Those browsers do not offer means to combine data from different sources.

6.2.1 Intelligent Book

Intelligent Book [86] is an application for mashing up Amazon and Half.com books. When a user search for a book, a request is sent to these book stores. The result is stored into an ontology. Based on the profile of the user with her interests and preferences, the application can apply its rules and rank the result. It uses Pellet for reasoning.

6.2.2 RDF Book

RDF Book [87] intends to publish book information from multiple web APIs (e.g., Google, Yahoo, Amazon and eBay) as semantic data. Books, bookstores, authors, reviews, purchase offers each have dereferenceable URIs with RDF descriptions. To dereference a URI, the mashup sends queries to the APIs, assembles the query results and returns to client the description in RDF/XML syntax. The majority of vocabularies derive from Dublin Core [10] and FOAF [11]

The description contains outgoing *owl:sameAs* links from the book authors to paper authors in the DBLP[12] database. To generate an outgoing link, the application sends a query to DBLP SPARQL endpoint[13] and matches the full name of the author with the query result. It also contains about 9000 incoming links from DBpedia; DBpedia automatically generates these links based on the ISBN number found in Wikipedia.

[10] `http://dublincore.org/documents/dces/` (accessed Nov. 01, 2015)

[11] `http://xmlns.com/foaf/0.1/` (accessed Nov. 01, 2015)

[12] `http://dblp.uni-trier.de/` (accessed Nov. 01, 2015)

[13] `http://dblp.l3s.de/d2r/sparql/` (accessed Nov. 01, 2015)

6.2.3 Black Swan

Black Swan [71] allows users to find out about correlations between socioeconomic developments and certain historical events. For instance, we can check how a war impacts income per capita. Black Swan uses data from international organizations such as the World Bank[14], IMF[15] and many other projects including DBpedia, Freebase[16], NOAA[17], Correlates of War[18], EM-DAT[19] and BBC Timeline[20]. It can automatically detect the outliers in global statistics together with the appropriately related events.

In the Black Swan system, an event is specified by a date, location, title, and event category; a statistical data point is specified by the numerical value of the data point, the year, location, and type of the statistic. The data is extracted from online data sources in three steps: (i) parsing, (ii) schema matching, (iii) and data cleansing.

The system contains a number of flexible parsers to handle both structured (e.g., CSV, HTML/XML, RDF) and unstructured (i.e., plain text) formats. It uses Apache Tika[21] to parse structured sources and collects more than 43,000 distinct events. The statistical data comprises 1,250,000 individual values of 200 countries over a time span of 200 years.

Black Swan combines information on events with statistical data to discover patterns of the former's influence over the latter. It decreases the number of candidate events that may contribute to an outlier in a statistic. Black Swan uses association rule mining to discover dependencies among variables in a large database.

6.2.4 DERI Pipes

DERI Pipes (or Semantic Web Pipes) [40] allows for the fast development of data-intensive applications manipulating RDF data. It decomposes data integration and data processing tasks into frequently used operators. Those operators foster reusability; they aim to alleviate the repetition of cumbersome, error-prone, and resource-intensive processing tasks in typically semantic systems.

A pipe, which is a set of instances of operators, defines two classical patterns of workflows, that is *split* and *merge*. Two most important operators that can be used in the pipe are *merge* and *split*. The input of the former is a number of RDF graphs represented in RDF/XML, N3[22] or Turtle[23] format; its output is a merged RDF graph of input graphs. The *split* operator is designed to connect a single output to an arbitrary number of inputs of other operators. Available operators are:

[14] `http://data.worldbank.org/` (accessed Nov. 01, 2015)

[15] `https://www.imf.org/external/data.htm` (accessed Nov. 01, 2015)

[16] `http://www.freebase.com` (accessed Nov. 01, 2015)

[17] `http://www.ngdc.noaa.gov` (accessed Nov. 01, 2015)

[18] `http://www.correlatesofwar.org` (accessed Nov. 01, 2015)

[19] `http://www.emdat.be` (accessed Nov. 01, 2015)

[20] `http://news.bbc.co.uk/2/hi/europe/country_profiles/` (accessed Nov. 01, 2015)

[21] `http://tika.apache.org/` (accessed Nov. 01, 2015)

[22] `http://www.w3.org/TeamSubmission/n3/` (accessed Nov. 01, 2015)

[23] `http://www.w3.org/TeamSubmission/turtle/` (accessed Nov. 01, 2015)

1. the *getRDF* and *getXML* operators, which take a URL as input to convert the respective resource into RDF and XML formats;
2. the *XSLT* operator, which performs a XSL transformation on a particular XML input file;
3. the *SPARQL CONSTRUCT* and *SELECT* operators, which can extract relevant part of a big graph, perform basic ontology mapping, or align two separated graphs;
4. the *RDFS* and *OWL* operators, which can apply RDFS or OWL inference rules on the input graph.

All operators have exactly one output, and one or multiple inputs. The output of an operator can link to the single input of another operator. Such links between input and output must be acyclic. Input data should be string text, RDF, or other XML formats.

DERI Pipes invites developers to contribute their work to an open source project[24]. Semantic web developers can implement new operators and add them to the project. The contribution model of DERI Pipes hence is different from that of the Linked Widgets framework. Our framework enables developers to freely develop and deploy widgets on their own infrastructure. On the contrary, DERI operators must be on the same deployment package to be able to cooperate with each other.

To create a DERI pipe, users make use of the *pipe editor*, which is a graphical web-based application that supports drag-and-drop and wiring operations. DERI Pipes intends to enable users to perform semantic data processing tasks from different RDF data sources. However, potential users need knowledge of RDF, SPARQL query results, XML, or HTML formats and how to process them algorithmically to make effective use of the provided operators.

A created pipe can be serialized and stored in XML format; this XML configuration can be loaded into the *pipe editor*. To execute a pipe, there is a *server-side execution engine* that coordinates the processing tasks defined in each operator of the pipe. All remote sources are fetched into the in-memory triple stores of the engine. The engine allows for concurrent execution, as each operator can be performed in a separated task. To avoid repetitive remote-resource fetching and hence enhance the performance, the engine makes use of a caching mechanism.

A saved pipe is available via a stable URL and hence can be reused as an operator in other pipes through a simple HTTP call. Pipes are based on common web standard and can be deployed on any Java web application server. Because of these two features, the DERI Pipes' creators note that pipes can run on a single machine, or be distributed among a number of nodes. Linked Widgets framework also enables these features. Moreover, the base operators (which are Linked Widgets) can be distributed among nodes due to our WebSocket-based remote protocol. By contrast, base DERI Pipes operators must be on the same web application.

[24] http://sourceforge.net/projects/semanticwebpipe/ (accessed Nov. 01, 2015)

6.2.5 Super Stream Collider

Super Stream Collider [39] is a web-based platform[25] to develop mashups of semantically annotated Linked Stream and Linked Data resources. It currently supports a number of live data sources, which are Linked Stream Middleware sensors[26], Twitter streams, DBpedia, and Sindice data sources. Super Stream Collider consists of a drag-and-drop mashup construction tool, a visual SPARQL/CQELS editor, and a visualization tool.

Super Stream Collider designs a set of streaming operators, which are categorized into three classes as follows:

1. *Data acquisition operators*, which use either pull-based or push-based mechanism to collect data from sources and gateways.
2. *Stream processing operators*, which use declarative language (e.g., CQELS) to specify stream processing functionalities.
3. *Streaming operators*, which stream the final output of a mashup to the consuming applications.

Each operator receives n input streams and returns exactly one output stream. While the input format is quite arbitrary, the output format is always RDF.

Super Stream Collider mashups run in a cloud computing infrastructure, which dynamically allocates relevant execution containers to execute every single operator in the deployed mashup. A composed and published mashup can be reused as a data source or an operator; it is assigned to a unique WebSocket URL where third party applications can access to collect the streaming data.

6.2.6 MashQL

MashQL [41] is a query-by-diagram language and tool that supports people in developing data mashups diagrammatically. The idea is to generalize Web 2.0 mashups and consider the internet as a database, where mashup is seen as a query.

MashQL is a generalization and extension of query formulation approaches, i.e., query-by-form, query-by-example, and query-by-filter. A MashQL query is represented as a tree. The root of the tree is the *query subject*. Each branch is a *query restriction*, which specifies a filtering rule on the *query subject*. Users hence can query and mashup web resources by building up such a tree in an interactive and intuitive process. The tree or the MashQL query is translated into and executed as SPARQL queries in the background by means of a MashQL-to-SPARQL translator written in Java. MashQL uses Oracle's SPARQL [121] rather than W3C's SPARQL standard, as the former inherits all SQL functionalities such as grouping or aggregating functions.

To formulate a MashQL query, users can use a web-based editor[27]; the UI of the editor is inspired by Yahoo! Pipes. The editor executes *background queries* to dynamically generate drop-down lists so that users can select and express their queries. For example, when users select a dataset in the *RDF input module*, they can specify the *query subject*

[25] `http://superstreamcollider.org/` (accessed Nov. 01, 2015)

[26] `http://open-platforms.eu/library/deri-lsm/` (accessed Nov. 01, 2015)

[27] `http://sina.birzeit.edu/mashql/` (accessed Nov. 01, 2015)

by choosing one item in the automatically generated list of all classes and instances of the input source. From the chosen subject, users select one of its properties and define relevant restrictions based on the domain and range of the selected property. To create a query, they hence do not need to understand the schema, the structure, or the technical details of the RDF data sources.

A MashQL pipe can be serialized into textual content, based on the MashQL markup language. The language is based on XML and consists of three sections as follows:

1. *Header*, which defines the input sources, prefixes, and other metadata of the query;
2. *Body*, which mainly specifies query conditions;
3. *Footer*, which specifies the result modifiers, ordering preferences, and output styles.

MashQL currently intends to query RDF data sources, using SPARQL language. It can be extended to query XML documents of relational databases, once a XSL stylesheet to translate MashQL markups into XQuery and/or SQL content is developed.

6.3 Embedded, Mobile, and Pervasive Mashups

Salminen et al. [122] focus on mashups for embedded devices. They introduce two environments for users to compose mashups in a procedural and declarative fashion. Both aim at context-aware mashups on embedded devices and can use data of these devices as input data for mashups. Similarly, Mikkonen and Salminen [123] present a runtime environment for embedded devices. It is able to combine data from the web and from the device peripherals and liberate the applications from the restriction of web browsers. The environment is implemented based on Qt [124], which is a cross-platform application framework that offers an extensive set of APIs in various embedded devices.

Chang et al. [125] present a development system for mobile mashups. They design three development tools for PCs, mobile pads, and smartphones.

1. The *PC tool* targets experts with full functionalities for mashup development; it consists of two main modules, i.e., *block builder* and *webapp builder* so that experts can create blocks and mobile web applications. The *block builder* contains multiple editors for different parts of a block (i.e., a *metadata editor* for the block's general information, a *HTML editor* for the block's user interface, and a *JavaScript editor* for the block's execution logic). Blocks can be shared or modified among users. To create a mobile application on top of available blocks, they have to edit the layout and the workflow of the application, using the *layout* and *workflow editor*, respectively.
2. The *pad tool* is similar to the PC tool with simplified UIs for the limited hardware and user interactions; it does not support block creation.
3. The smartphone tool is mainly for non-expert users with considerably reduced functionality and simplified UIs. They cannot directly edit the layout or the workflow of the application, but use templates – which are a mobile mashup application sample. To make a mashup, users search, select a template, and customize it by replacing particular parts of the constituent blocks or changing their attributes.

The runtime environment for mashup execution is a web browser; the browser can be a desktop browser (e.g., Chrome, Safari, Firefox) or a custom web browser for Android and iOS devices. For the latter case, the mashup can access special functions of the mobile devices such as camera, compass, microphone, gallery, or contacts.

Corvetta et al. [126] introduce CAMUS (Context-Aware Mobile mashUpS) as a framework to design context-aware, mobile or web-based applications that collect and integrate heterogeneous sources (e.g., data and services) in a context-aware fashion. To hide the complexity, CAMUS makes use of high-level visual abstractions [127] for context and mashup modeling. It then uses generative techniques to transform the model into running code in various devices. The structure and content of the created mashup is hence decided at the run time, and can be flexibly and personally adapted to users' devices, needs and situations of use.

Ma et al. [128] propose the "*Brick-based and State-driven Mobile Service Composition*" (BSMSC) framework as a mobile mashup approach. The framework serves three groups of stakeholders as follows:

1. *Composite RESTful Service Developer* (CRSD) who develops *composite RESTful services* and provides the respective models by means of JSON description,
2. *Service Brick Developer* (SBD) who develops *Android-fragment-based service bricks* on top of the *composite RESTful services*,
3. *Composite service brick developer* (CSBD) – the end user of the framework – who uses a visual tool to choose service bricks and design her own *brick-based mobile application.*

A *service brick* is a UI component to hide the back-end *composite service* from the end users. It is divided into two types: (i) *web-based service brick*, which is created using HTML, CSS, and JavaScript, and (ii) *native service brick*, which is created using Android native objects. The latter is able to access client-side local resources such as sensor data or address book.

Service bricks can send (receive) events to (from) the *event engine* of the framework. As the engine matches and forwards events to relevant service bricks, if offers a mechanism to connect the *front-end service bricks* and the back-end RESTful services in a state-driven fashion (i.e., there will be a session to maintain and manage shared resources and states among composing services). To this end, the framework is based on the service composition engine JOpera [129]. Because the output of a service needs to be transformed into a valid JSON format before serving as input of another service, the framework contains a *service mediator* as an intermediate layer between JOpera and the service bricks. It offers four major functionalities when orchestrating services: (i) *One-Shot Service Flow Execution*, (ii) *Stateful Service Flow Execution*, (iii) *Service Substitution*, and (iv) *Format Wrapping for Service Output.*

6.4 Collaborative Mashups

6.4.1 PEUDOM

PEUDOM [130] is designed as a platform for multiple devices and collaborative mashups. It allows users to create components on top of REST services and combine these components to build mashups. Similar to the Linked Widgets framework, it creates a live collaboration paradigm. However, a major difference is that the data processing tasks in PEUDOM cannot be assigned to different devices. In fact, these tasks are REST services that will be called in the corresponding components implemented for different devices. PEUDOM hence can be seen as a service composition platform for end users on different devices.

PEUDOM consists of two major environments, i.e., *Component Editor* in which users can register their REST services to the platform, and *Mashup Dashboard*, which is a web environment to create new mashups or modify the existing ones. PEUDOM uses event-driven and publish-subscribe paradigm to coordinate various implementations of components on different devices.

PEUDOM follows a client lightweight execution approach. REST services encapsulated in components are called and processed at the client side. Meanwhile, the server takes care of synchronous and asynchronous communication and manages the resources shared by multiple users.

6.4.2 MoSaiC

MoSaiC [131] is a web-based mashup tool and platform for *collaborative document engineering.* It models enterprise documents as mashups or situational applications of content provision, transformation and publication. MoSaiC allows a team to consolidate content from various sources; the team can collect and share data in the course of collaborative work.

An example use case is from scientific publication; Alice (who is the initiator) first creates a research paper mashup by dragging document elements (i.e., abstract, section, text, figure, or bibliography) into the mashup canvas. She associates each element with a meaningful task description. Alice can define further behavioral rules such as all sections need to be delivered three days before the deadline.

Alice then adds two services to the mashup as follows: (i) a layout format service that automatically generates a document from the mashup, (ii) and a submission service that uploads the created document to a relevant server. Finally, Alice adds some people in her research team to the paper mashup and assigns them particular tasks.

In the following weeks, when evolving the paper, the team can make changes to the initial mashup. They can add new content to their assigned sections, as well as see other work. The event-based reminding rule defined before will trigger a notification to related authors who do not send the content on time, if it discovers that a content element is not in “final” state. As soon as all elements are completed, the whole mashup is sent to the layout service, and finally, a document is created and submitted due to the submission service.

6.4.3 ContextGrid

ContextGrid [132] is a *contextual mashup-based collaborative browsing platform.* The idea is to bring knowledge-sharing services to internet users; from various open APIs, ContextGrid integrates heterogeneous information and assists users to collaboratively navigate the web. To *navigate* means users interact with the web and find their relevant information.

Each user has her bookmark set, which is represented in her personal ontology. A user context is then defined by her ontology and her current web page. ContextGrid matches and identifies the semantic relationships between the user's current web page and her set of bookmarks, and hence can recognize the context.

A user can get browsing information of another user if they are in similar context and share the same preferences. They can exchange knowledge, resources, experiences (e.g., heuristics and know-how) while searching for information. For example, three users are related if one visits the Vienna Wikipedia page, one locates Vienna on a Google map, and the other browses Flickr photos taken in Vienna. To this end, ContextGrid develops a model that first recognizes current context of each user, and then compares those contexts to detect the similarities. Finally, ContextGrid makes use of open API-based mashups to integrate various types of web resources and recommend relevant resources to users, based on their contexts.

6.4.4 Apache Rave Extension

Hertel et al. [133] introduce an extension of the open-source mashup platform Apache Rave[28] to add support for real-time collaboration among users when composing user interface mashups (or widget-based mashups). As users can exchange knowledge and experiences, the collaboration feature is expected to make mashup development more social and encourage users to be more productive.

The challenge is to design a synchronization algorithm and a conflict-resolution mechanism. To this end, the extension makes use of Operational Transformation algorithm [134], which is a concurrency control mechanism that can maintain the consistency of mashup models. To resolve conflict operations, it transforms them into previously applied concurrent operations so that they satisfy the consistency model.

The extension defines six operations for the mashup synchronization as follows:

1. "*Add Widget*", which adds a new widget to the building mashup,
2. "*Move Widget*", which moves a widget to another position in the mashup canvas
3. "*Remove Widget*", which removes a particular widget from the mashup
4. "*Change View Mode*", which sets the display mode of a Apache Rave widget

[28] `http://rave.apache.org/` (accessed Nov. 01, 2015)

5. "*Replace Widget Property*", which replaces the current value with the new value for a property
6. "*Change Connection Setting*", which changes the connection between widgets.

When users interact with Apache Rave platform, such operations are issued to the server. The server may perform necessary transformations if there are concurrent operations. It then updates its global mashup configuration and propagates the operations to other collaborators.

6.5 Automatic Mashups

Due to the similarity of mashups with web services, we consider the rich stream of research on web service composition closely related. Rao and Su [135] review research efforts of automatic web service composition techniques both from the workflow and AI planning research communities. Dustdar and Schreiner [136] present different composition strategies based on existing platforms and frameworks. They argue that – amongst others – specifications, interactions, and non-functional attributes need to be considered.

Fischer et al. [137] present an evolutionary algorithm to generate ad-hoc mashups from semantic web services. The information retrieved from the invoked services is transformed into a semantic representation. Feng et al. [138] propose a service-oriented approach to integrate open data services. They semantically model data services and transform the integration problem into a service composition problem. SMART [139] is a platform for mashing up REST web services. It matches the users' querying keywords with REST web service descriptions in its ontology and allows users to build personalized mashups. Common issues of service mashups lie in (i) the complexity of services to users, and (ii) the high risk of generating irrelevant mashups due to uninformative service descriptions. We hence encapsulate web services and other web resources into Linked Widgets and associate our widgets with a semantic model to facilitate automatic processing.

To be more precise, considerable research effort is directed at automatic mashup composition. Wong and Hong [38] present Marmite, which uses syntactic similarity, such as comparing output and input data types, to couple widgets. Goal-driven approaches such as MARIO (*Mashup Automation with Runtime Orchestration and Invocation*) introduced by Riabov et al. [140] automatically deduce compositions based on user-defined goals. Ngu et al. [141] propose a semantic annotation technique to find sets of functionally equivalent components that can be combined into composite applications. Hybrid approaches go one step further and combine goal-driven and semantics-driven techniques. Elmeleegy et al. [142] present MashupAdvisor, which recommends composition goals based on a user's current composition context. Deutch et al. [143] introduce so called *glue-patterns* which are utilized after the user selects desired components. These glue patterns are used to auto-complete partial mashups. Cappiello et al. [61] propose to incorporate quality considerations when identifying adequate components for mashups. They introduce an assisted composition process which does not require explicit models. Instead, quality becomes a key indicator for recommending complete mashups to users. Our composition technique uses a well-defined semantic model, which needs to be specified before widgets are executed; this technique always ensures compatibility between widgets, and therefore guarantees successful mashup execution. Adding goal-based characteristics to our current approach may be useful in limiting the result space and respecting users' information needs.

Bai et al. [144] introduce mashlets (i.e., online resources including data, functions, and presentations) according to user goals and combined them into new applications. They follow the goal decomposition and refinement approach to fill the gap between low-level mashlets and high-level user goals. Rodríguez et al. [145] implement a Firefox plugin to recommend useful composition patterns to end users during mashup development in Yahoo! Pipes. The composition knowledge is extracted from the repository of existing mashups. Similar work can be found in [146, 147]. The former observes that mashups developed by different users typically share common characteristics. It exploits these similarities to predict potential mashup compositions, given a user mashup specification. [147] is a hybrid recommendation approach, which leverages existing compositions and component descriptions to provide continuous development support for end users.

In the following, we review two platforms that allow for automatic mashup composition.

OMELETTE

Roy Chowdhury et al. [148] introduce a mashup system called OMELETTE[29]. To support users in designing mashups, it follows a hybrid approach, which combines the *goal-oriented solutions* [149, 150] and the *pattern-based development* [141, 151]. The former aims to automatically develop mashups that satisfy user-specified goals while the latter recommends composition patterns to auto-complete a part of mashup.

OMELETTE identifies three main mashup challenges. (i) Users do not know which widgets they need to their intended mashup. (ii) Provided that they are aware of necessary widgets, they still cannot check if these widgets are available (iii) Users typically are not skilled enough to define the data and the mashup control flow. To address these challenges, it extends the widget-based mashup platform Apache Rave by two modules, i.e., *Automatic Composition Engine* (ACE) and *Pattern Recommender* (PR).

ACE provides users with an interactively dialog-based agent that allows them to specify the goal of their mashups. The knowledge base consists of a domain ontology and a set of functions that define the agent's behavior. There are four important functions as follows:

1. *Question Building*, which is language-aware, to establish a conversation flow with users;
2. *Evidence Collection* to collect user response and build a context model on top of the domain ontology;
3. *Widget Selection* to select widgets by matching the collected evidences with the widget registry;

[29] `http://www.ict-omelette.eu/home` (accessed Nov. 01, 2015)

4. *Workspace Configuration* to setup parameters for the workspace and involved widgets.

PR is designed to support users in extending mashups created by ACE, and composing mashups from the scratch. It extracts patterns from the knowledge base of all created mashup models. *Widget co-occurrence* and *multi-widget* are two supported patterns. During the composition process, PR reacts to user operations such as adding, selecting or deleting a widget. It identifies the operation, calculates and generates relevantly ranked recommendations, and returns them to users.

CRUISe

CRUISe [84, 152] is a platform for dynamic composition and execution of adaptive composite applications. Its implementation is based on Eclipse Modeling Framework[30], where application models can be serialized into XML Metadata Interchange (XMI) format and thus become exchangeable and tool-independent.

CRUISe uses services as input to compose a web-based user interface of service-oriented composite components. The basic structure and interface of these components are specified in *component models*. A component model is characterized by three parameters: (i) *configuration*, which stores a set of component states specified by pairs of key and value, (ii) *event*, which is issued by a component to publish to others that its state has been changed, (iii) and *operation*, which is a method of a component triggered by events. CRUISe identifies three component types, i.e., *User Interface Component*, *Logic Component*, and *Service Component*.

CRUISe follows the semantic approach to represent its component models. To this end, it defines *Semantic Mashup Component Description Language* (SMCDL), which uses the principles of SAWSDL [96] and WSMO-Lite [153]. Annotation of components is categorized into three types, i.e., (i) *data semantics*, which match properties and parameters of events and operations to semantic concepts, (ii) *functional semantics*, which combines operations, events, and the component as a whole to define the overall functionality, (iii) and *non-functional semantics*, which describe metadata (e.g., author, price, license) of a component.

Based on the defined component models, from a particular mashup template (which is defined using SMCDL), CRUISe can automatically select and integrate components at runtime, using a semantic discovery and ranking algorithm. The algorithm is based on the semantic compatibility between components; it does not consider the compliance between component templates and implementations. Therefore, CRUISe uses a semantic mediation to tackle the heterogeneity between dynamically integrated components and composition models. When necessary, the mediation can (i) rename operations, events, parameters and properties of components, (ii) rearrange parameters of outgoing and incoming events (iii) or perform a transformation to enforce data compatibility.

To sum up and to contrast the related work with the work discussed in this thesis we would like to highlight that our contribution consists in (i) an automatic mashup

[30] http://www.eclipse.org/modeling/emf/ (accessed Nov. 01, 2015)

composition algorithm to simplify mashup development, and (ii) the interactive tagging module to compose mashups from text. The foundation of both approaches is the semantic model of widgets. Whereas related work heavily relies on composition knowledge to recommend widget connections, we can proactively discover all possible connections by means of terminal matching, and compose all mashups from a set of widgets.

6.6 Natural Language-supported Mashups

There are several approaches to enable users to compose mashups by specifying them in text:

1. Users type arbitrary sentences or questions in natural language. The text is then processed to compose mashups for them. This is very simple and easy to use; however, because of the variety in our language, the input text is likely to be ambiguous. Thus it is difficult to build up precise mashups for users. An example of a similar approach can be found in AutoSPARQL [154]. Using active supervised machine learning to generate SPARQL queries, it can answer natural language queries over RDF knowledge bases.
2. To address ambiguity, a subset of natural language can be used. This means the syntax and vocabulary are restricted to specific patterns.
3. Some authors design and use a dedicated mashup language that can be based on natural language (e.g., [155]). This creates, however, another barrier to end users, as they have to learn the syntax of the new language.
4. Finally, dialog-based approaches (cf. [148]) are possible. We ask users questions; based on the answers, we can ask further questions to narrow the context step-by-step.

In the following, we review a number of frameworks that support automatic mashup composition based on natural language.

6.6.1 NaturalMash

NaturalMash [156] is an end-user mashup development tool. It is the combination of different techniques such as *live programming*, *programming by demonstration*, *natural language programming*, and *What You See Is What You Get*.

From available web APIs such as LinkedIn, eBay, Last.fm, Google search, Google map, BBC, Facebook, Youtube, NaturalMash enables users to enter a simple description of their desired mashup and returns an automatically composed mashup to them. Its interface contains a *text field* for describing the mashup integration logic, and a *visual field* that is a WYSWYG interface to design and preview the mashup being created.

As an example (cf. [156]), users can type "*Find songs titled **mashup**. When an item is selected, search YouTube videos about **title**. When a video item is selected, post **video link** to my wall*" in the *text field.* The language used is a subset of English with restricted vocabularies and grammars. Alternatively, users can drag and drop available APIs to the *visual field* and design their mashup.

To this end, for each web API, NaturalMash creates an abstract component that describes the functionality of the API by a *Task* and an *Event*. A *Task* is defined as a "*passive atomic operation*" that takes input data, processes, and returns output. An *Event* is defined as an "*active source of control*" that describes a condition; if the condition is satisfied, it issues a message associated with parameters. In the example description above, NaturalMash uses the *Task* behavior of Last.fm API to search for songs titled "mashup" and returns a table of songs. An *Event* then can be the *click* action on a song.

In the next step, NaturalMash creates a natural language representation of the API; All *Tasks* and *Events* are associated with particular natural language descriptions. Two respective types of descriptions are "*Imperative sentences*" and "*Causal sentences*". To reduce sources of error, the *text field* supports advanced features such as autocompelete text suggestion or restricting the possible input characters.

NaturalMash is a live mashup tool; it can compile and run textual mashup descriptions. The descriptions are sent to a *compilation pipeline* and transformed into web service compositions, which are executed by JOpera Engine [129] (i.e., a service composition tool that provides a visual language to define control flows and data flows of service processes). The compilation consists of six steps: (i) *Natural Language Parsing*, (ii) *Constrained Natural Language Parsing*, (iii) *API Binding*, (iv) *Data Flow Resolution*, (v) *Intermediate Model*, and (vi) *Emitter*.

6.6.2 Natural Mashups

Natural Mashups [157] is a web-based service composition tool that enables users to create on-the-fly *composite services* from a descriptive request expressed in a restricted form of natural language. The request is based on the vocabulary and syntax patterns of all components in the users' service catalog. This means users need to be aware of available services, or their requests cannot be understood. Natural Mashups supports two languages (i.e., English and French) and combines the usage of text and speech to specify a request.

Similar to NaturalMash, for each supported web service, Natural Mashups creates an abstract service, which is associated with *natural language annotation*. The annotation consists of two elements, i.e., *Activation terms* and the *Rules part*. The former specifies the possible terms that if matched, the natural language parser can activate the respective service. The latter defines a set of patterns that can be recognized for the service and the mechanism to obtain the parameters from the request.

To compile a textual request into a composite mashup, Natural Mashups analyzes the text, using the annotation of abstract services; the result is a list of matched abstract services and resolved arguments. From that, it creates a sequence of service calls in the orchestration script. The script finally is translated into different executable code, depending on the target framework. The default framework is SPATEL ENGINE [158], which is a SOA oriented framework using UML interfaces and state machines to represent services' operations and behaviors. Alternatively, the code can be sent to the Open Mashups tool [159] to build up a mashup, which can be automatically transformed into an iPhone application.

For the example request "*Send to mum Paris weather translated in Spanish*" (cf. [157]), Natural Mashups creates an orchestration of four services as follows: (i) resolving the nick name (i.e., "*mum*") in the user's address book, (ii) accessing the weather forecast in Paris, (iii) translating the given text into Spanish, and (iv) sending a SMS or an email. To support users in being aware of good textual description that is understandable by Natural Mashups, it makes use of both *text auto-completion* and *contextual help menu* techniques.

6.6.3 EnglishMash

Aghaee and Pautasso [155] design EnglishMash, which is a natural mashup composition environment based on their *controlled natural language* that supports users in developing mashups. The typical drawback of this approach is that users are required to acquire special skills, i.e., component capabilities, algorithmic thinking, and the language syntax. To address this issue, the environment interactively responses to users' acts such as correcting their mistakes, or displaying the live preview of their textual mashup description.

EnglishMash associates its mashup components with natural language patterns. For example, the pattern for the *Twitter search component* is "*search tweets at [coordinate: longitude, latitude]*". It hence allows users to express in natural language a number of programming techniques such as *conditional branches*, *event handling* and *iteration.* An example expression is "*When the map is clicked, do as follows. Display a marker at the location, and search for tweets at the location. Finally, show the tweets on the table*" (cf. [155])

EnglishMash requires users to have several basic skills and knowledge. (i) To be able to build up mashups, users need to be aware of all available components as well as their functionalities. (ii) Users should have problem-solving skills; they can think algorithmically to orchestrate available services and create an added value. (iii) Users should be able to restrict their English sentence so that it is understandable and executable by EnglishMash.

The typical work flow to play with EnglishMash is: (i) having in mind the goal of the mashup, (ii) identifying and adding necessary mashup components to the being constructed mashup, (iii) developing the logic of the mashup in natural language, (iv) and previewing the live execution of the mashup. During this process, users interactively get feedback of their syntax and runtime error; they also receive the auto-completion support by means of a drop-down menu containing suggestions.

6.7 Programming-by-demonstration Mashups

Since the Linked Widget framework does not follow the programming-by-demonstration approach, in this section, we only presents Vegemite as an exemplification.

Vegemite [45] is the extension of the CoScripter web automation tool (which is an end-user programming system presented in [160]) by a spreadsheet environment. It uses programming-by-demonstration techniques to enable users to collect and process information from various web sites and populate a table.

Starting with an empty table (which is the so-called *Vegetable*), users can manually add new data or copy from other spreadsheet sources. Moreover, they can extract data from a web page and add it to the table, using the *direct-manipulation* tool. By performing a series of actions on the table and the web page, users can demonstrate and instruct the *manipulation* tool to fill additional rows and columns to the *Vegetable*. To this end, the user actions are recorded as scripts, which can be re-executed at any time to complete or refresh data on the *Vegetable*.

Vegemite scripts cannot only collect data but also allow for low-level web actions such as filling the values into HTML forms, or clicking on HTML links. Based on the values collected in the *Vegetable*, users, in advanced, can perform further web actions, e.g., sending an email that contains data in the table.

Vegemite is implemented as a Mozilla Firefox add-on. It consists of two main components, i.e., (i) Vegetables, (ii) and CoScripter engine. The former is implemented as a table editor whereas the latter is a sidebar borrowed from CoScripter to create, edit, and save Vegemite scripts.

When editing scripts, users can copy and paste operations. The scripts resemble English and are understandable to both humans and computers. A *Vegetable* is composed of tabular data and Vegemite scripts. Both are permanently stored in a central server so that users can access to their data on different machine with Firefox and the add-on already installed.

Summary of Mashup Tools To sum up, the most apparent differences between the Linked Widget framework and similar approaches are as follows: (i) We present a high-level and problem-oriented data processing framework. Users do not have to be familiar with special technological and programming concepts, e.g., conditional statements or loops, to perform data integration tasks. They first define a goal, e.g., search for POIs near a place, then can discover appropriate widgets, and finally arrange them in a mashup. To ease this process, we organize widgets in domain-specific collections. Additionally, we provide keyword and semantic search features based on the widget model. (ii) We allow and encourage developers to contribute their widgets to the framework to extend the number of data sources it can process. (iii) We use semantic web technologies to model widgets and facilitate automatic data exploration and data integration via widgets. This imposes the semantic format on widget output and helps the framework to address data heterogeneity. (iv) We provide an environment for interactive collaborative mashup creation and analysis. (v) We introduce and implement the concept of *distributed mashups*.

CHAPTER 7

Conclusions and Future Work

7.1 Conclusions

The web of data is growing at a staggering pace; a large number of data sources, APIs, services, and data visualizations are publicly available. In order to satisfy their information needs, users increasingly have the opportunity to interact with and integrate information from various sources.

However, end users are not able to directly access, explore, and combine different sources due to a number of technological barriers: (i) Data is stored in heterogeneous formats, e.g., XML, CSV, JSON, RDF, HTML, and Portable Document Format (PDF); (ii) users do not know where to find required data sources; (iii) provided that they are aware of appropriate sources, they frequently do not have the means and skills to access them; and (iv) if users are capable of collecting raw data from various sources, they typically cannot perform necessary data processing and data integration tasks.

In this thesis, we present our approach for dynamic and automatic exploration and integration of heterogeneous data sources for non-expert users. We provide the answers to the three research questions as explained in Table 7.1.

To foster reusability and creativity, we modularize functionality into Linked Widgets that users can recombine in order to create new applications. We make use of both client and server computing resources to create a powerful, extensible and scalable data integration model. With server Linked Widgets, data processing tasks can be run persistently and be distributed among various devices. This is particularly useful for data streaming or data monitoring use cases.

We divide Linked Widgets into three categories, i.e., *data*, *processing*, and *visualization* widgets which perform the data retrieval, data processing/data integration, and data presentation tasks, respectively. The internal mechanics of these complicated tasks are not visible to end users because they are encapsulated inside the widgets. Widgets have input and output terminals and can be easily linked to each other by end users. When combined, each widget can receive data and return its processed output data to another

Research Question	**Answer**
RQ 1 Data Heterogeneity	We make use of Linked Widgets, which are web widgets backed by a semantic model, to lift data in different formats to a semantic level (cf. Sections 3.3 and 3.4.3).
RQ 2 Collaborative Data Integration	We categorize Linked Widgets into client and server widgets. On the one hand, client widgets can collect data on the web and process it in a web browser environment. On the other hand, with server widgets, mashups can be assembled from components that are distributed among various devices. This facilitates collaborative data integration in which each stakeholder can contribute their data and computing resources to the shared data processing flow. To this end, we design local, remote, and hybrid protocols (cf. Section 3.6) for widget interaction and present mashup patterns for collaborative, persistent, distributed, and streaming use cases (cf. Section 3.7).
RQ 3 Automatic Data Integration	We present our mechanisms to (i) validate the links between widgets and discover all widgets that can provide data to or consume data from input and output terminals of a widget (cf. Section 3.9.1), (ii) automatically compose mashups from a given set of widgets (cf. Section 3.9.2), and (iii) create auto-composed mashups from structured text (cf. Section 3.10).

Table 7.1: Answers to research questions

widget. This mechanism allows users to compose data-centric applications out of available Linked Widgets. These ad-hoc mashup applications, in turn, are re-edited, used, and shared among end user communities.

Based on the concept of Linked Widgets, we design a conceptual framework that aims to (i) connect developers, data publishers, data integrators, and end users, (ii) provide universal and practical advantages without restrictions on domain or data sources, (iii) allow users to combine/integrate multiple (linked) open data sources and leverage their joint value, (iv) allow novice users to easily analyze, integrate and visualize data, and (v) allow users to perform data processing tasks in a collaborative and distributed manner simultaneously on multiple devices.

The framework is built upon semantic web technologies and its design follows three guiding principles: *openness*, *connectedness*, and *reusability*. *Openness* distinguishes our approach from similar ones and is the key to tackle disparate data sources. It means developers can implement and directly add their new widgets to the framework. *Connectedness* means users can combine data from different sources by connecting widgets of distinct developers. Finally, we foster *reusability* by allowing users to make use of the same widget in a dynamic and creative manner to compose various applications.

We leverage the power of semantic mashups to help novice users to easily explore and integrate available web resources by means of widgets. We encapsulate semantic, graph-based models inside Linked Widgets and provide mechanisms to annotate their input and output. Widgets can obtain and process both semantic and non-semantic data. They lift non-semantic data to a semantic level and produce semantic output data according to its predefined model.

Based on the widget models, we design novel mechanisms to ease the composition process. *Terminal matching* and *semantic widget search* are aimed at users who are already familiar with the composition environment, but need support in finding appropriate widgets. *Auto-composition* targets novices who have no or little experience in mashup development and need to be guided through the composition process. Finally, the *tag-based composition* mechanism allows users to compose mashups from structured text and hence considerably simplifies the mashup development process.

A prototype implementation[1] of the framework is already available. We illustrate the feasibility and effectiveness of the framework through location-based use cases combining data from five LOD datasets and three other open data sources. These examples can easily be extended by adding additional blocks for new datasets. Users do not have to manually browse different URIs or write SPARQL queries to retrieve and aggregate data; our approach provides a common framework to integrate and visualize the desired information. It could also serve well as a rapid prototyping environment for data integration projects.

7.2 Future Work

7.2.1 Mobile Mashup Editor

A prototype web-based mashup editor[2] is already available online. Even though users can use this editor on a mobile browser to combine widgets, we plan to implement a separate editor that is designed especially for mobile devices. Together with server widgets, the mobile mashup editor completes our mobile mashup environment.

7.2.2 Publish Mashup Result as Linked Data

Linked Widgets lift data in arbitrary formats to semantic data. Because distributed mashups can run persistently, we can access their semantic output data at any time. Publishing a distributed mashup thereby means publishing a data source which can again be consumed by other entities. To foster automatic data integration, we plan to transform output data into Linked Data. To this end, the framework assigns each resource of the output data a dereferenceable URI that can be looked up by people and user agents.

[1] `http://linkedwidgets.org/` (accessed Nov. 01, 2015)

[2] `http://linkedwidgets.org` (accessed Nov. 01, 2015)

7.2.3 Automatic Mashup Composition

Future research will also aim to improve our automatic mashup composition algorithm by adding heuristics for the search and backtracking steps. Furthermore, we aim to enable developers (who can already access widget annotations as Linked Open Data) to implement custom composition algorithms for particular data domains and integrate them into the framework as plugins.

7.2.4 Linked Widgets Semantic Model

At present, we annotate input and output models of widgets only; to also annotate parameters in the user interface of widgets has not been considered, yet. This helps us to have a light-weight semantic model, but the detailed functionality of widgets is not well described. The Tag-based Composition Module hence cannot automatically set up appropriate parameters for member widgets of composed mashups.

Another issue when modeling widgets is that some of them cannot be semantically annotated in a meaningful manner. Generic widgets that perform filter or aggregation tasks do not have fixed models; their interfaces and output depend on the input data which the widget is only aware of at run time. Similarly, a data widget can output different data models, depending on different parameters defined in its interface. We intend to design a dynamic run time model for such widgets. As soon as users drag the widget into a mashup and parameterize it, its semantic model is propagated to facilitate automatic composition including widgets whose models may vary during run time.

7.2.5 Widget Change Management

Another research challenge is to tackle widgets that evolve over time. At present, developers can change the widget source code at any time because it is decoupled from the framework. Changing the widget model, however, would break existing mashups and is hence problematic. To address this issue, we already store widget versions and use our widget provenance ontology to keep track of them. Future research will focus on leveraging such annotation to effectively manage and maintain the framework resources.

7.2.6 Natural Language-supported Mashups

We currently do not process un-tagged words input by users in the Tag-based Composition Module; these strings can contain information to further clarify the user's context and help to make the composed mashup more precise. We plan to improve our approach and analyze the whole input sentence – at least for sentences in frequent and useful patterns.

Moreover, if the mashup context given by users is not clear enough or there are no mashups that can be automatically composed, we can follow the dialog-based approach. We ask the user, receive their answer, and adapt the context step-by-step.

7.2.7 Widget Collections for Various Domains

Finally, as we are still in a preliminary stage, we plan to create more widgets for different domains. This will allow users to work with different kinds of data sources (e.g., governmental, financial, environmental data etc.) and types of data (e.g., open, linked, tabular data etc.), without technical barriers. From a data perspective, the vision is to support users in their everyday decision-making. Another interesting direction for future research is the automatic creation of new widgets able to handle dynamic web data as an input source.

Appendix

A Running Example of the Automatic Mashup Composition Algorithm

In the Table A.1, we present the detailed steps of the automatic mashup composition algorithm to compose complete mashups of w_2 in the graph of seven widgets illustrated in Figure 3.19.

Step	**uiVertices**[1]	**sfEdges**[2]	**sbEdges**[3]	**mashup**	**isDead**
Start	[ ]	[ ]	[ ]	[ ]	false
Initialize	[i_1^2]	[(i_2^3, o^2)]	[ ]	[ ]	false
goForward(i_2^3)	[i_1^2, i_1^3, i_2^3]	[ ]	[ ]	[(i_2^3, o^2), (o^3, i_1^3), (o^3, i_2^3)]	false
goForward(o^3)	[i_1^2, i_1^3, i_2^3]	[(i_1^6, o^3), (i_1^5, o^3)]	[ ]	[(i_2^3, o^2), (o^3, i_1^3), (o^3, i_2^3)]	false
	[i_1^2, i_1^3, i_2^3]	[(i_1^6, o^3)]	[ ]	[(i_2^3, o^2), (o^3, i_1^3), (o^3, i_2^3), (i_1^5, o^3)]	false
goForward(i_1^5)	[i_1^2, i_1^3, i_2^3]	[(i_1^6, o^3)]	[ ]	[(i_2^3, o^2), (o^3, i_1^3), (o^3, i_2^3), (i_1^5, o^3)]	false
goBackward(i_1^3)*	[i_1^2, i_1^3, i_2^3]	[(i_1^6, o^3)]	[(i_1^3, o^4), (i_1^3, o^7)]	[(i_2^3, o^2), (o^3, i_1^3), (o^3, i_2^3), (i_1^5, o^3)]	false
	[i_1^2, i_1^3, i_2^3]	[(i_1^6, o^3)]	[(i_1^3, o^4)]	[(i_2^3, o^2), (o^3, i_1^3), (o^3, i_2^3), (i_1^5, o^3), (i_1^3, o^7)]	false

Step	uiVertices[1]	sfEdges[2]	sbEdges[3]	mashup	isDead
goBackward(o^7)	[i^2_1, i^3_1, i^3_2, i^7_1]	[(i^6_1, o^3)]	[(i^3_1, o^4)]	[(i^3_2, o^2), (o^3, i^3_1), (o^3, i^3_2), (i^5_1, o^3), (i^3_1, o^7), (o^7, i^7_1)]	false
goBackward(i^7_1)*	DEAD				true
Backtrack	[i^2_1, i^3_1, i^3_2]	[(i^6_1, o^3)]	[(i^3_1, o^4)]	[(i^3_2, o^2), (o^3, i^3_1), (o^3, i^3_2), (i^5_1, o^3)]	false
Try next edge of sbEdges	[i^2_1, i^3_1, i^3_2]	[(i^6_1, o^3)]	[ ]	[(i^3_2, o^2), (o^3, i^3_1), (o^3, i^3_2), (i^5_1, o^3), (i^3_1, o^4)]	false
goBackward(o^4)	[i^2_1, i^3_1, i^3_2]	[(i^6_1, o^3)]	[ ]	[(i^3_2, o^2), (o^3, i^3_1), (o^3, i^3_2), (i^5_1, o^3), (i^3_1, o^4)]	false
goBackward(i^2_1)*	[i^2_1, i^3_1, i^3_2]	[(i^6_1, o^3)]	[(i^2_1, o^1)]	[(i^3_2, o^2), (o^3, i^3_1), (o^3, i^3_2), (i^5_1, o^3), (i^3_1, o^4)]	false
	[i^2_1, i^3_1, i^3_2]	[(i^6_1, o^3)]	[ ]	[(i^3_2, o^2), (o^3, i^3_1), (o^3, i^3_2), (i^5_1, o^3), (i^3_1, o^4), (i^2_1, o^1)]	false
goBackward(o^1)	[i^2_1, i^3_1, i^3_2]	[(i^6_1, o^3)]	[ ]	[(i^3_2, o^2), (o^3, i^3_1), (o^3, i^3_2), (i^5_1, o^3), (i^3_1, o^4), (i^2_1, o^1)]	false
A complete mashup is found (All vertices of the *uiVertices* are visited)					false
Backtrack	[i^2_1, i^3_1, i^3_2]	[(i^6_1, o^3)]	[ ]	[(i^3_2, o^2), (o^3, i^3_1), (o^3, i^3_2)]	false

Step	uiVertices[1]	sfEdges[2]	sbEdges[3]	mashup	isDead
Try next edge of sfEdges	$[i_1^2, i_1^3, i_2^3]$	[ ]	[ ]	$[(i_2^3, o^2), (o^3, i_1^3), (o^3, i_2^3), (i_1^5, o^3), (i_1^3, o^4), (i_1^2, o^1)]$	false
goForward(i_1^6)	$[i_1^2, i_1^3, i_2^3]$	$[(o^6, i_1^6)]$	[ ]	$[(i_2^3, o^2), (o^3, i_1^3), (o^3, i_2^3), (i_1^5, o^3), (i_1^3, o^4), (i_1^2, o^1)]$	false
	$[i_1^2, i_1^3, i_2^3]$	[ ]	[ ]	$[(i_2^3, o^2), (o^3, i_1^3), (o^3, i_2^3), (i_1^5, o^3), (i_1^3, o^4), (i_1^2, o^1), (o^6, i_1^6)]$	false
goForward(o^6)	DEAD				true
Algorithm ends because both *sbEdges* and *sfEdges* are now empty.					

*We need to visit all vertices of *uiVertices*

[1] Un-visited input vertices

[2] Saved forward edges

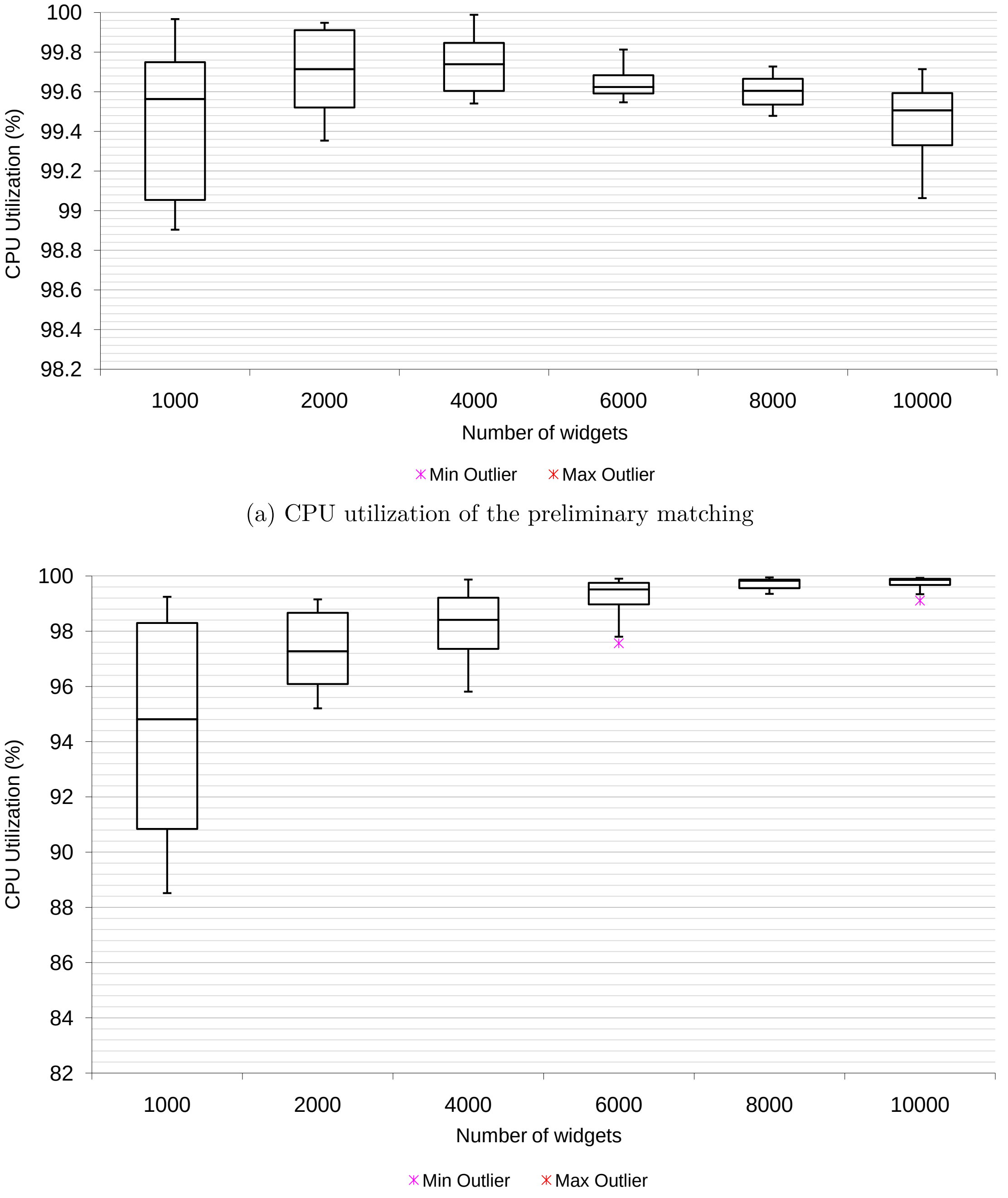
[3] Saved backward edges

Table A.1: Running example of the automatic mashup composition algorithm

B Box Plots of the Terminal Matching Experiment

In the following, we present the box plots of data from the terminal matching experiment.

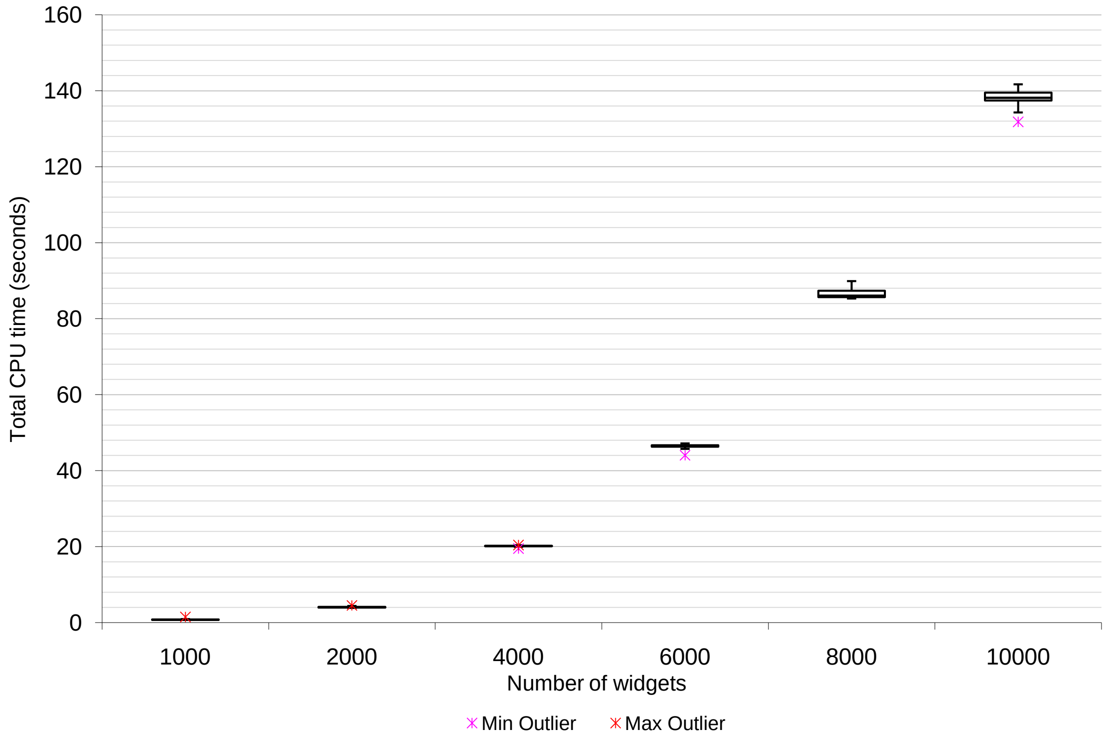
1. N**-experiment** ($X = 10$ and $R = 0.2$): Figures B.1 and B.2, respectively, present the CPU utilization and the total CPU time of the *preliminary matching* and *full matching* for each $N \in \{1000, 2000, 4000, 6000, 8000, 10000\}$.
2. X**-experiment** ($N = 8000$ and $R = 0.2$): Figures B.3 and B.4, respectively, present the CPU utilization and the total CPU time of the *preliminary matching* and *full matching* for each $X \in \{1, 5, 10, 15, 20\}$.
3. R**-experiment** ($N = 8000$ and $X = 10$): Figures B.5 and B.6, respectively, present the CPU utilization and the total CPU time of the *preliminary matching* and *full matching* for each $R \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$.



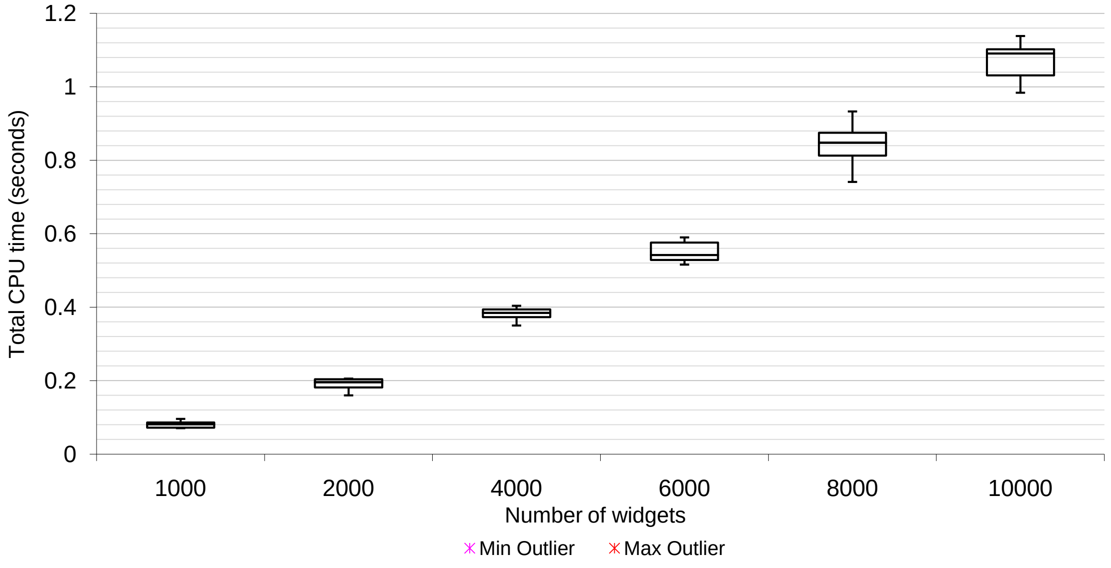
(a) CPU utilization of the preliminary matching

(b) CPU utilization of the full matching

Figure B.1: Terminal matching: CPU utilization as a function of number of widgets (Matching rate $R = 0.2$ and model complexity $X = 10$)

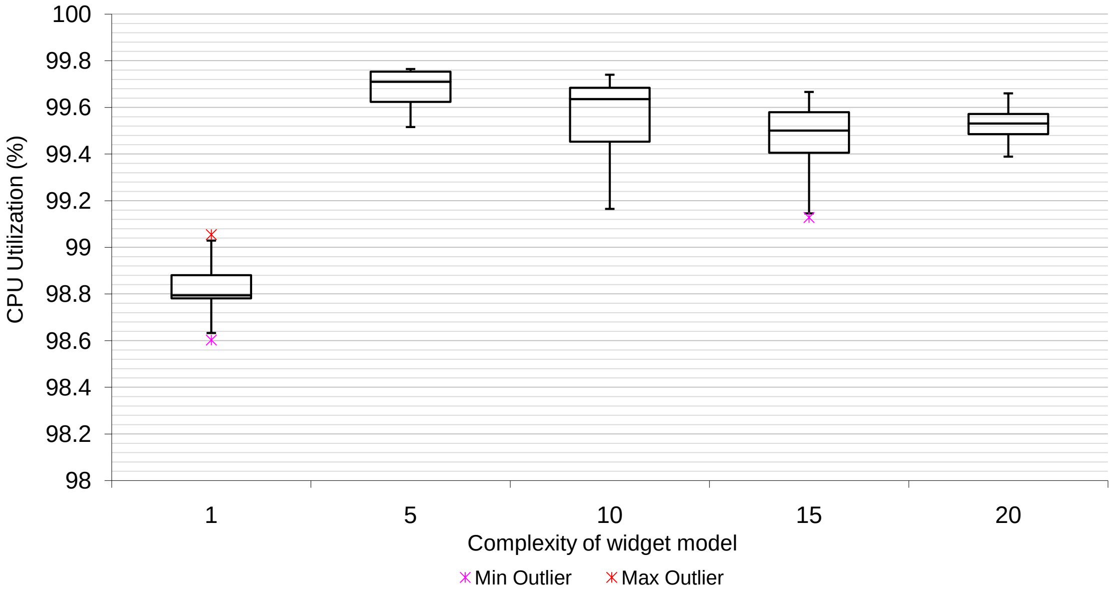


(a) Total CPU time of the preliminary matching

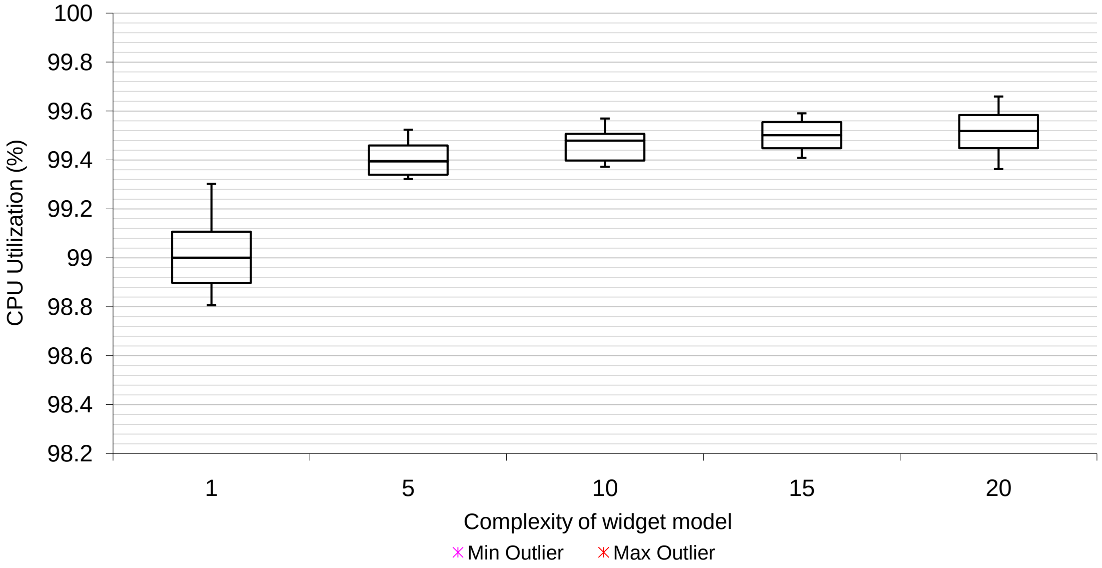


(b) Total CPU time of the full matching

Figure B.2: Terminal matching: total CPU time as a function of number of widgets (Matching rate $R = 0.2$ and model complexity $X = 10$)

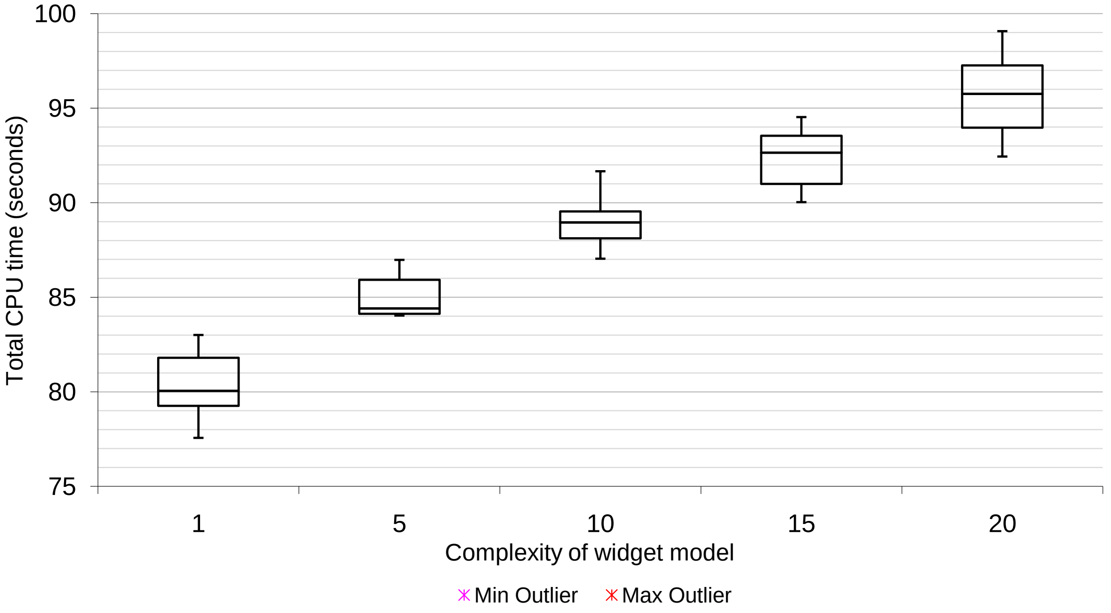


(a) CPU utilization of the preliminary matching

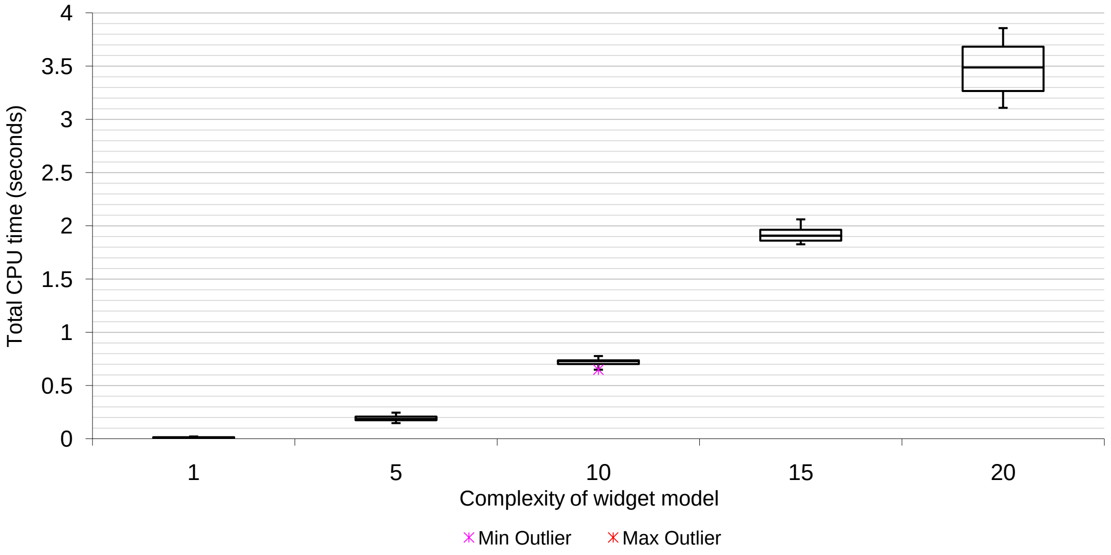


(b) CPU utilization of the full matching

Figure B.3: Terminal matching: CPU utilization as a function of complexity of widget model (Number of widget $N = 8000$ and matching rate $R = 0.2$)

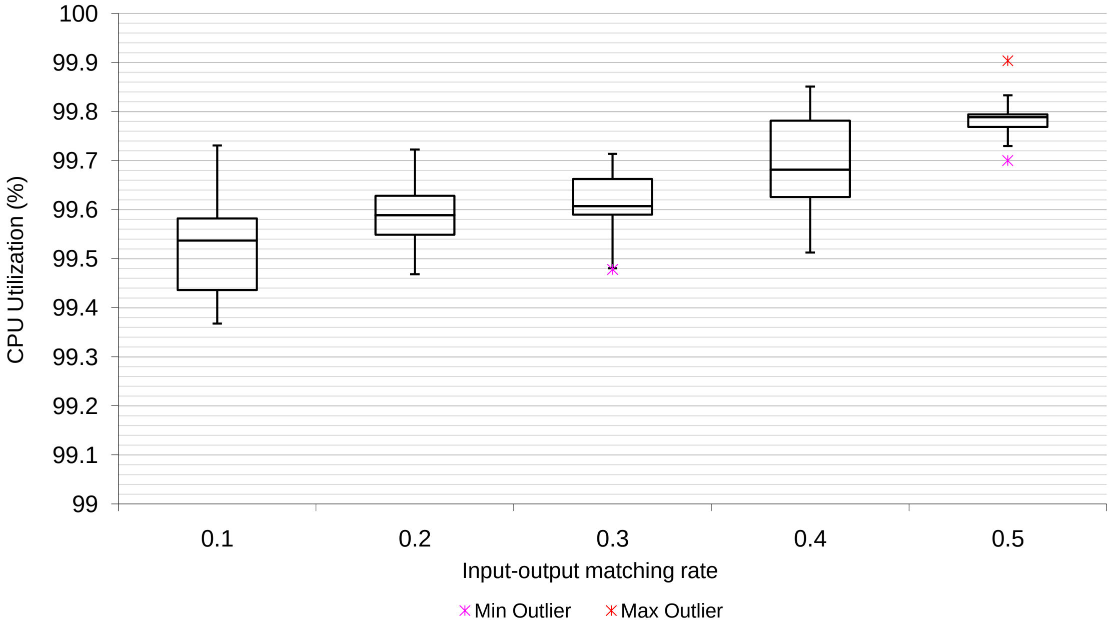


(a) Total CPU time of the preliminary matching

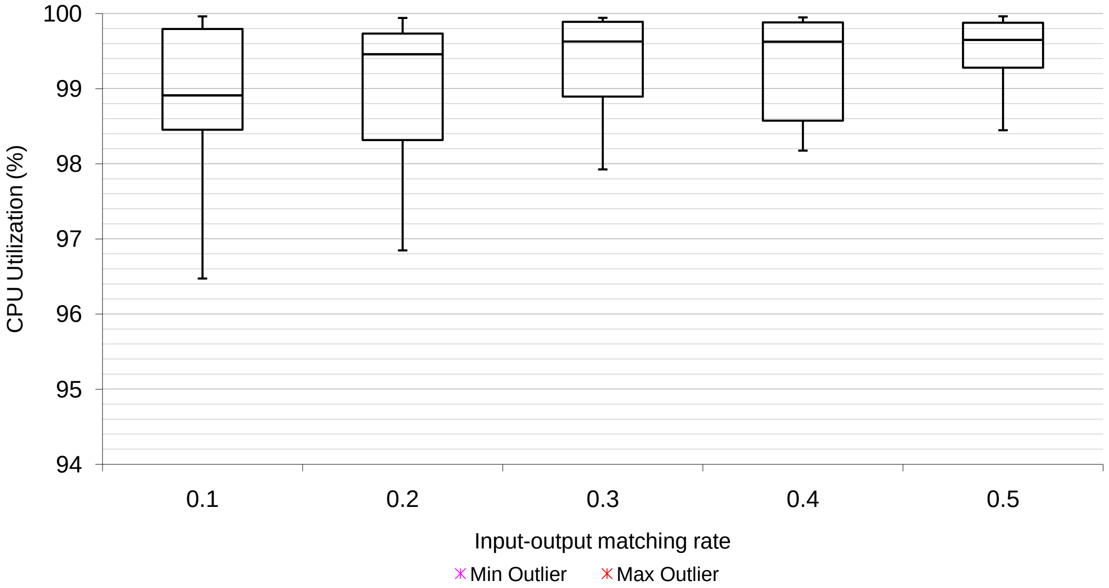


(b) Total CPU time of the full matching

Figure B.4: Terminal matching: total CPU time as a function of complexity of widget model (Number of widget $N = 8000$ and matching rate $R = 0.2$)

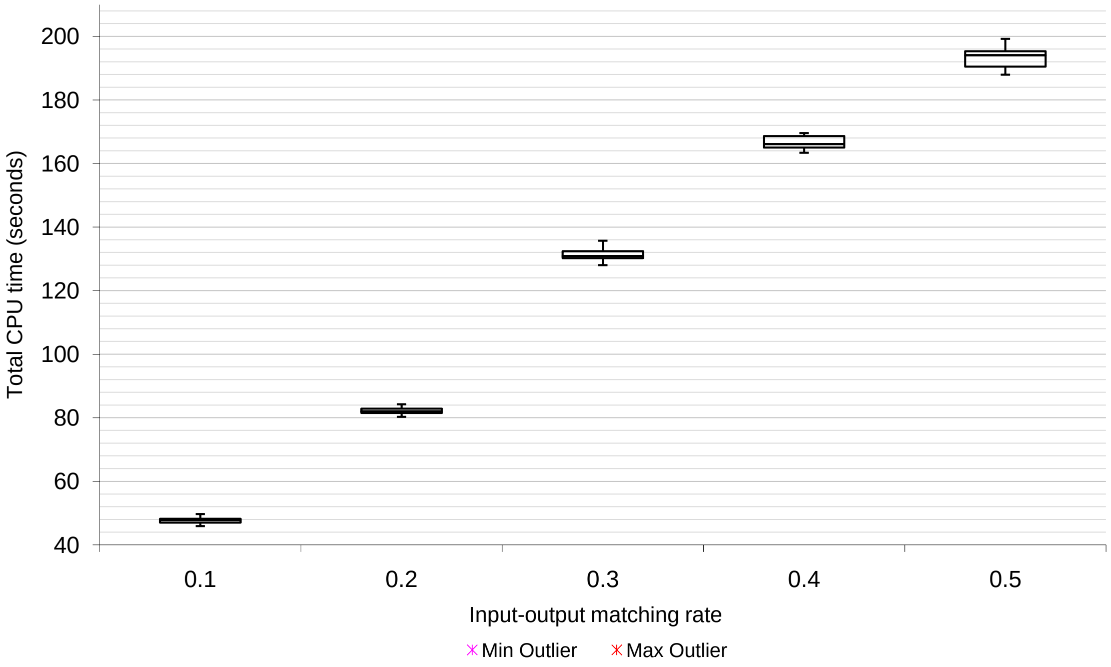


(a) CPU utilization of the preliminary matching

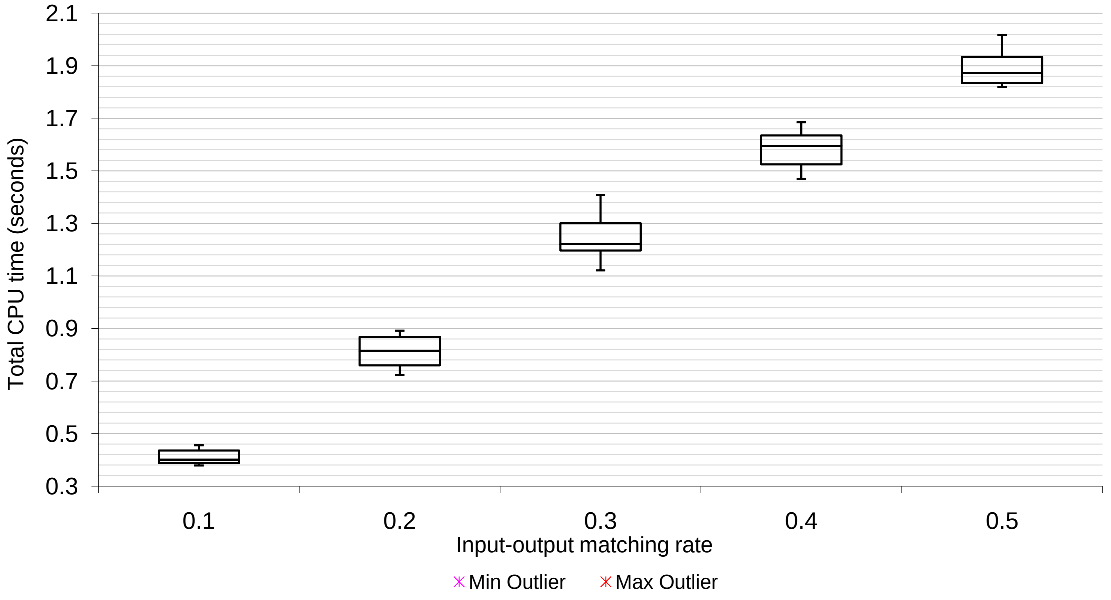


(b) CPU utilization of the full matching

Figure B.5: Terminal matching: CPU utilization as a function of input-output matching rate (Number of widget $N = 8000$ and model complexity $X = 10$)



(a) Total CPU time of the preliminary matching



(b) Total CPU time of the full matching

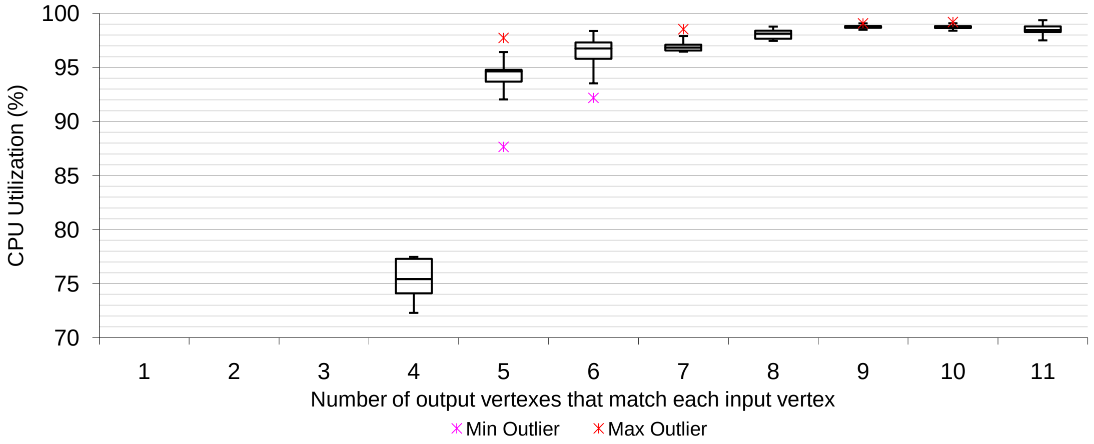
Figure B.6: Terminal matching: total CPU time as a function of input-output matching rate (Number of widget $N = 8000$ and model complexity $X = 10$)

C Box Plots of the Automatic Mashup Composition Experiment

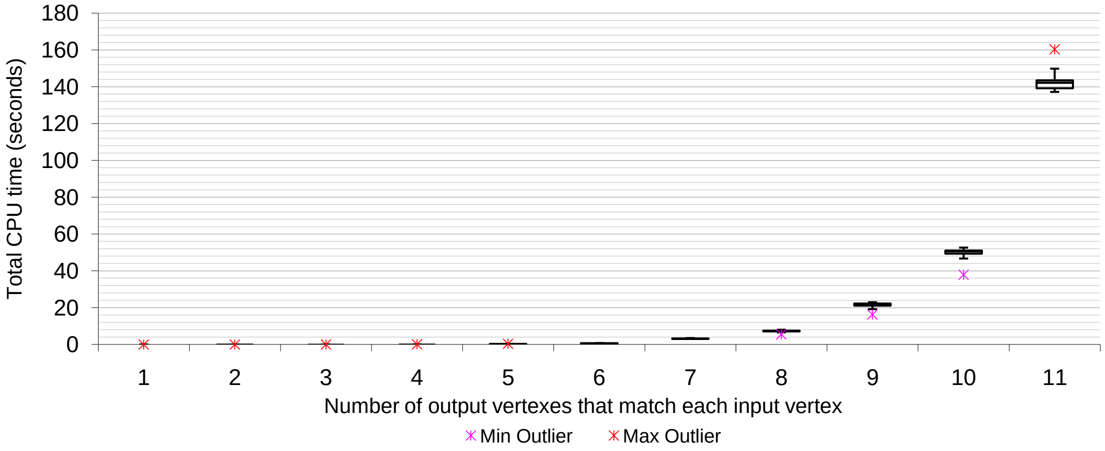
In the following, we present the box plots of data from the automatic mashup composition experiment.

***R*-experiment**: Figure C.1 shows the CPU utilization and the total CPU time of the algorithm over increasing number of matched output vertices for each input vertex.

N^d, N^p, and N^v experiments: Figures C.2, C.3, and C.4 illustrate the CPU utilization and the total CPU time of the algorithm over increasing number of data, processing, and visualization widgets, respectively. The total CPU time is measured until 20,000 mashups have been composed.

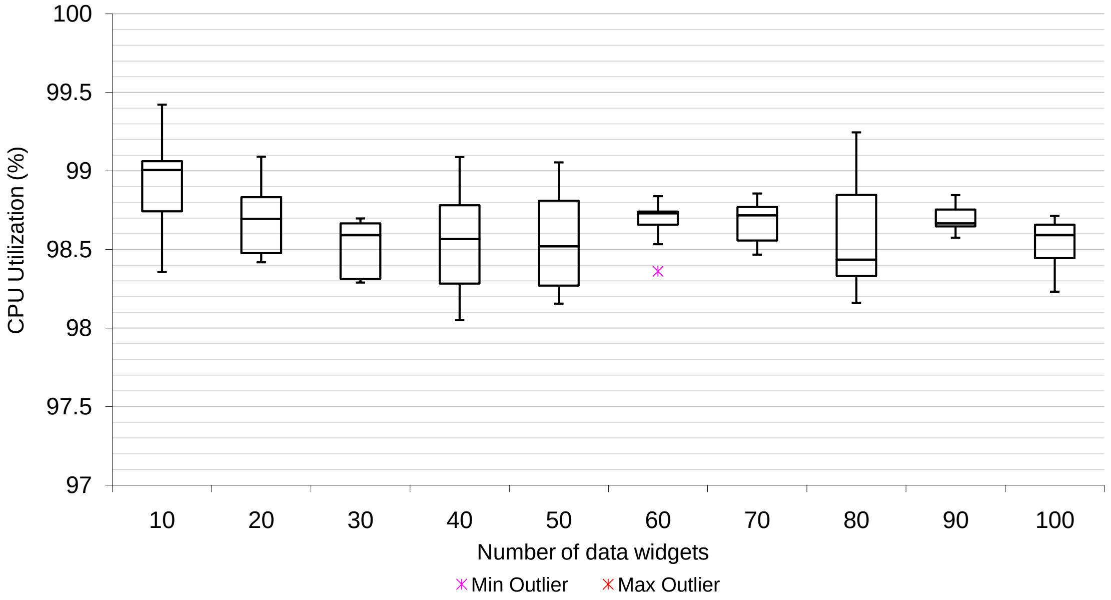


(a) CPU utilization

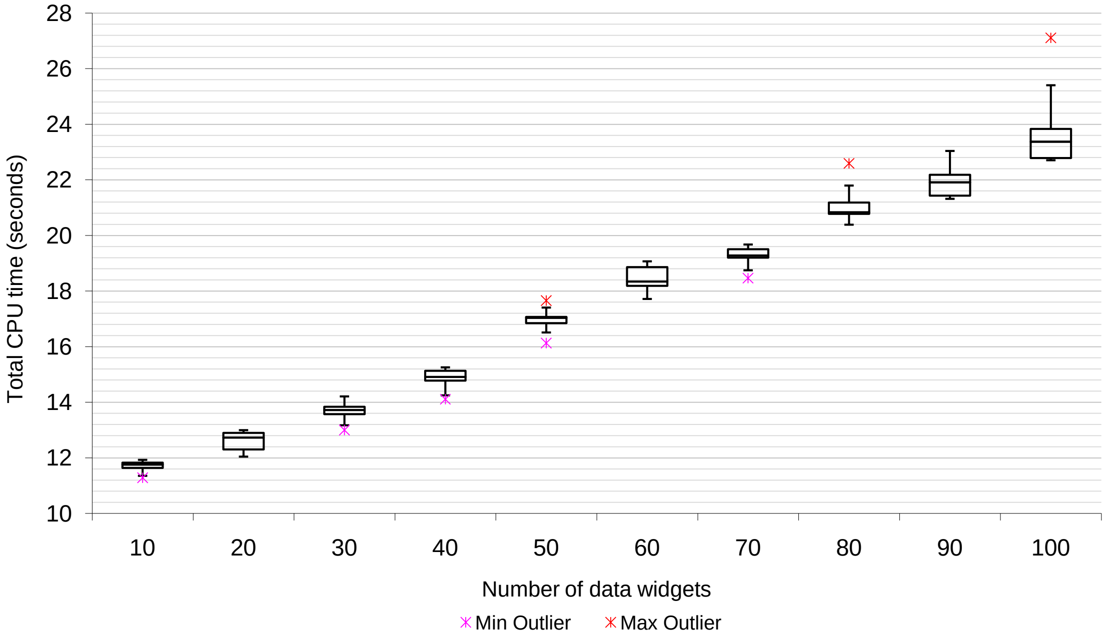


(b) Total CPU time

Figure C.1: Automatic mashup composition: CPU utilization and total CPU time for increasing input-output matching rate ($N^d = 5$, $N^p = 10$, $N^v = 5$)

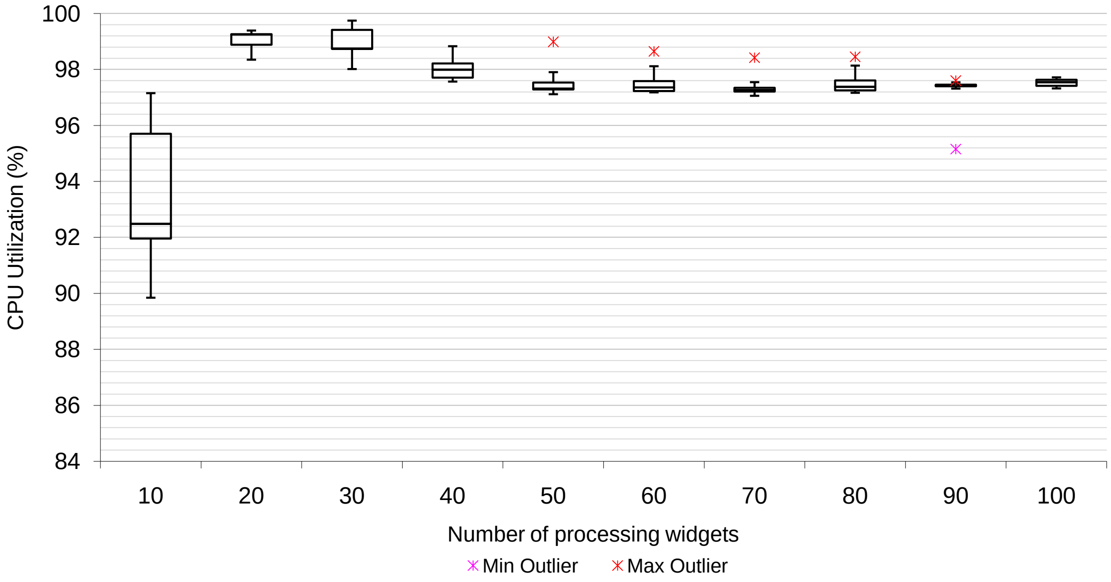


(a) CPU utilization

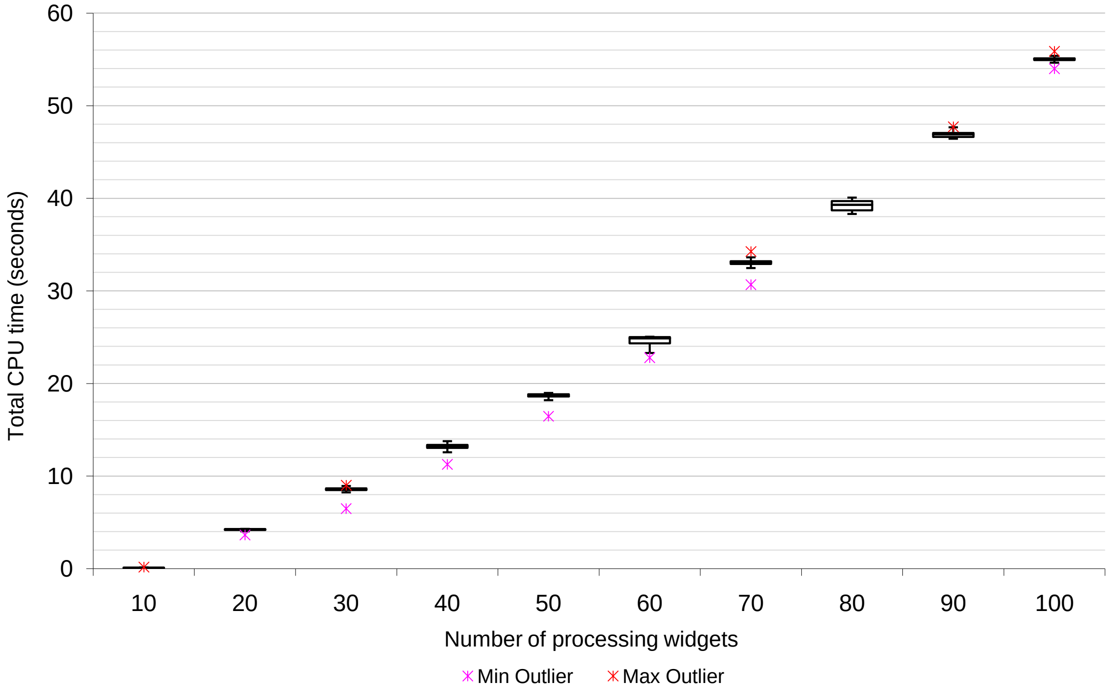


(b) Total CPU time

Figure C.2: Automatic mashup composition: CPU utilization and total CPU time (which are observed until *20,000 mashups* have been composed) for increasing number of data widgets ($N^p = 100$, $N^v = 50$, $R = 0.2$)

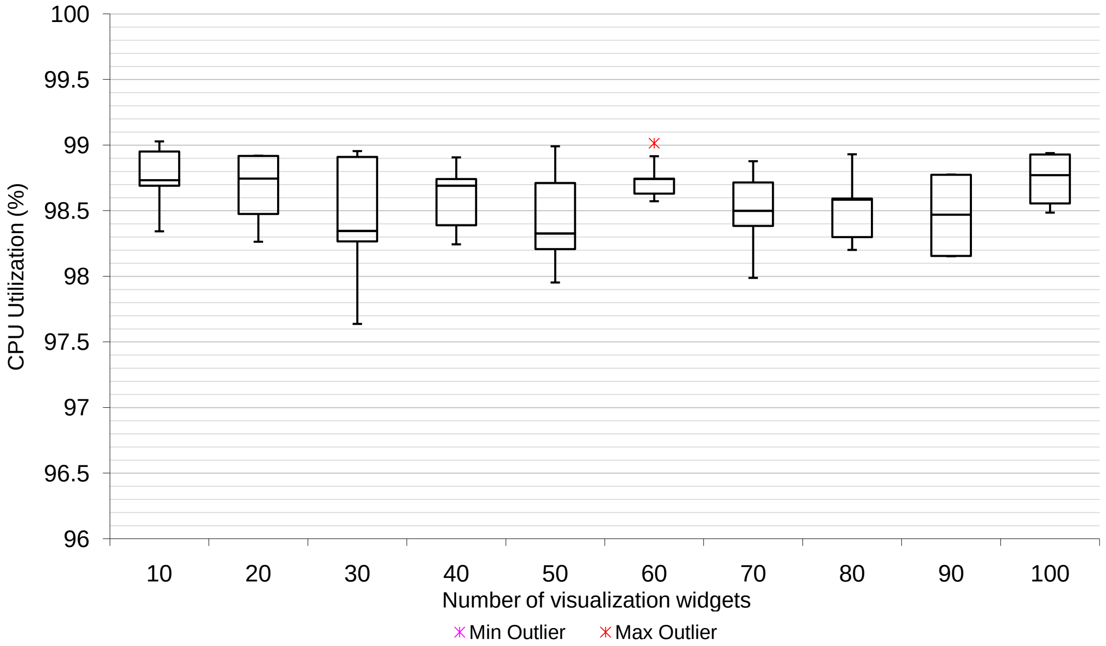


(a) CPU utilization

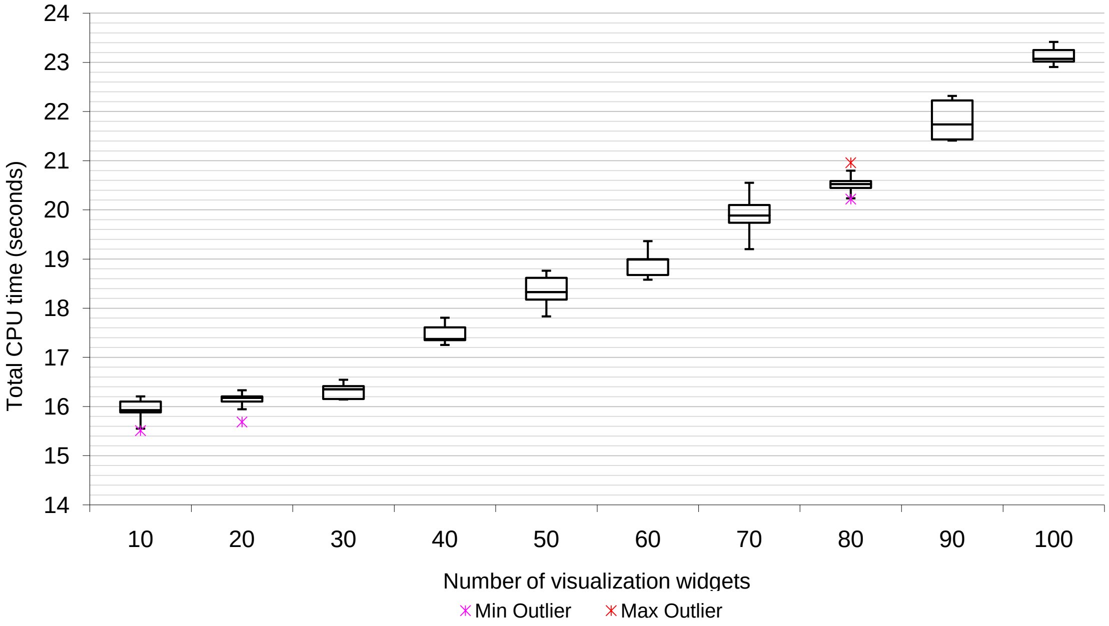


(b) Total CPU time

Figure C.3: Automatic mashup composition: CPU utilization and total CPU time (which are observed until *20,000 mashups* have been composed) for increasing number of processing widgets ($N^d = 50$, $N^v = 50$, $R = 0.2$)



(a) CPU utilization



(b) Total CPU time

Figure C.4: Automatic mashup composition: CPU utilization and total CPU time (which are observed until *20,000 mashups* have been composed) for increasing number of visualization widgets ($N^d = 50$, $N^p = 100$, $R = 0.2$)

Bibliography

[1] Tuan-Dat Trinh, Ba-Lam Do, Peter Wetz, Amin Anjomshoaa, Elmar Kiesling, and A Min Tjoa. A Drag-and-block Approach for Linked Open Data Exploration. In *Proceedings of the 5th International Workshop on Consuming Linked Data (COLD 2014) co-located with the 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Italy, October 20, 2014.*, 2014. URL http://ceur-ws.org/Vol-1264/cold2014_TrinhDWAKT.pdf.

[2] Tuan-Dat Trinh, Peter Wetz, Ba-Lam Do, Amin Anjomshoaa, Elmar Kiesling, and A Min Tjoa. A Web-based Platform for Dynamic Integration of Heterogeneous Data. In *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services, Hanoi, Vietnam, December 4-6, 2014*, pages 253–261, 2014. doi: 10.1145/2684200.2684291. URL http://doi.acm.org/10.1145/2684200.2684291.

[3] Tuan-Dat Trinh, Peter Wetz, Ba-Lam Do, Elmar Kiesling, and A Min Tjoa. Distributed Mashups: A Collaborative Approach to Data Integration. *IJWIS*, 11(3):370–396, 2015. doi: 10.1108/IJWIS-04-2015-0018. URL http://dx.doi.org/10.1108/IJWIS-04-2015-0018.

[4] Tuan-Dat Trinh, Peter Wetz, Ba-Lam Do, Amin Anjomshoaa, Elmar Kiesling, and A Min Tjoa. Implementing Linked Widgets: Lessons Learned for Linked Data Developers. In *Proceedings of the ISWC Developers Workshop 2014, co-located with the 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Italy, October 19, 2014.*, pages 25–30, 2014. URL http://ceur-ws.org/Vol-1268/paper5.pdf.

[5] Tuan-Dat Trinh, Ba-Lam Do, Peter Wetz, Amin Anjomshoaa, and A Min Tjoa. Linked Widgets: An Approach to Exploit Open Government Data. In *The 15th International Conference on Information Integration and Web-based Applications & Services, IIWAS '13, Vienna, Austria, December 2-4, 2013*, page 438, 2013. doi: 10.1145/2539150.2539252. URL http://doi.acm.org/10.1145/2539150.2539252.

[6] Tuan-Dat Trinh, Peter Wetz, Ba-Lam Do, Amin Anjomshoaa, Elmar Kiesling, and A Min Tjoa. Linked Widgets Platform: Lowering the Barrier for Open Data

Exploration. In *The Semantic Web: ESWC 2014 Satellite Events - ESWC 2014 Satellite Events, Anissaras, Crete, Greece, May 25-29, 2014, Revised Selected Papers*, pages 171–182, 2014. doi: 10.1007/978-3-319-11955-7_14. URL http://dx.doi.org/10.1007/978-3-319-11955-7_14.

[7] Tuan-Dat Trinh, Ba-Lam Do, Peter Wetz, Amin Anjomshoaa, Elmar Kiesling, and A Min Tjoa. Open Mashup Platform - A Smart Data Exploration Environment. In *Proceedings of the ISWC 2014 Posters & Demonstrations Track a track within the 13th International Semantic Web Conference, ISWC 2014, Riva del Garda, Italy, October 21, 2014.*, pages 53–56, 2014. URL http://ceur-ws.org/Vol-1272/paper_45.pdf.

[8] Tuan-Dat Trinh, Peter Wetz, Ba-Lam Do, Amin Anjomshoaa, Elmar Kiesling, and A Min Tjoa. Open Linked Widgets Mashup Platform. In *Proceedings of the AI Mashup Challenge 2014 co-located with 11th Extended Semantic Web Conference (ESWC 2014), Crete, Greece, May 27, 2014.*, 2014. URL http://ceur-ws.org/Vol-1200/paper2.pdf.

[9] Joel Gurin. *Open Data Now: The Secret to Hot Startups, Smart Investing, Savvy Marketing, and Fast Innovation.* McGraw-Hill Education, 2014. ISBN 978-0-07-182977-9.

[10] Marijn Janssen, Yannis Charalabidis, and Anneke Zuiderwijk. Benefits, Adoption Barriers and Myths of Open Data and Open Government. *Information Systems Management*, 29(4):258–268, 2012. doi: 10.1080/10580530.2012.716740. URL http://dx.doi.org/10.1080/10580530.2012.716740.

[11] Katrin Braunschweig, Julian Eberius, Maik Thiele, and Wolfgang Lehner. The State of Open Data - Limits of Current Open Data Platforms. In *Proceedings of the 21st World Wide Web Conference 2012, Web Science Track at WWW'12, Lyon, France, April 16-20, 2012.* ACM, 2012. ISBN 978-1-4503-1229-5.

[12] Hannu Jaakkola, Timo Mäkinen, and Anna Eteläaho. Open Data: Opportunities and Challenges. In *Proceedings of the 15th International Conference on Computer Systems and Technologies*, CompSysTech '14, pages 25–39, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2753-4. doi: 10.1145/2659532.2659594. URL http://doi.acm.org/10.1145/2659532.2659594.

[13] Norman W. Paton, Klitos Christodoulou, Alvaro A. A. Fernandes, Bijan Parsia, and Cornelia Hedeler. Pay-as-you-go Data Integration for Linked Data: Opportunities, Challenges and Architectures. In *Proceedings of the 4th International Workshop on Semantic Web Information Management*, SWIM '12, pages 3:1–3:8, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1446-6. doi: 10.1145/2237867.2237870. URL http://doi.acm.org/10.1145/2237867.2237870.

[14] An-Hai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration.* Elsevier Science, 2012. ISBN 978-0-12-391479-8.

[15] John Hebeler. *Semantic Web Programming.* Timely. Practical. Reliable. Wiley, Indianapolis, Ind, 2009. ISBN 978-0-470-41801-7.

[16] Grigoris Antoniou and Frank Van Harmelen. *A Semantic Web Primer.* Cooperative information systems. MIT Press, Cambridge, Mass, 2004. ISBN 978-0-262-01210-2.

[17] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data – the Story so Far. *International journal on semantic web and information systems*, 5(3):1–22, 2009. URL `http://eprints.soton.ac.uk/271285/`.

[18] Sören Auer, Volha Bryl, and Sebastian Tramp. *Linked Open Data – Creating Knowledge Out of Interlinked Data: Results of the LOD2 Project.* Lecture Notes in Computer Science. Springer International Publishing, 2014. ISBN 978-3-319-09846-3.

[19] Ian Millard, Hugh Glaser, Manuel Salvadores, and Nigel Shadbolt. Consuming Multiple Linked Data Sources: Challenges and Experiences. In *Proceedings of the First International Workshop on Consuming Linked Data, Shanghai, China, November 8, 2010*, 2010. URL `http://ceur-ws.org/Vol-665/MillardEtAl_COLD2010.pdf`.

[20] Andreas Harth, Katja Hose, and Ralf Schenkel. *Linked Data Management.* Emerging directions in database systems and applications. CRC Press, 2014. ISBN 978-1-4665-8241-5.

[21] Robert Lee McCann and University of Illinois at Urbana-Champaign. *Efficient Data Integration: Automation, Collaboration, and Relaxation.* University of Illinois at Urbana-Champaign, 2007. ISBN 978-0-549-34103-1.

[22] Rosen Publishing Group. *The Future of the Web.* Scientific American cutting-edge science. Rosen Pub., 2007. ISBN 978-1-4042-0989-3.

[23] Tim Berners-Lee and Mark Fischetti. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor.* Paw Prints, 2008. ISBN 978-1-4395-0036-1.

[24] Keith DeWeese and Dan Segal. *Libraries and the Semantic Web:An Introduction to Its Applications and Opportunities for Libraries.* Morgan & Claypool, 2014. ISBN 978-1-62705-196-5. URL `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7007876`.

[25] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific american*, 284(5):28–37, 2001.

[26] Bob DuCharme. Integrate Disparate Data Sources with Semantic Web Technology. Technical report, TopQuadrant, September 2010. URL `http://www.ibm.com/developerworks/library/x-disprdf/`.

[27] San Murugesan. Understanding Web 2.0. *IT Professional*, 9(4):34–41, July 2007. ISSN 1520-9202. doi: 10.1109/MITP.2007.78. URL `http://dx.doi.org/10.1109/MITP.2007.78`.

[28] Darlene Fichter. What is a Mashup. In *Library Mashups: Exploring New Ways to Deliver Library Data*. Information Today, Incorporated, 2010.

[29] Florian Daniel and Maristella Matera. *Mashups: Concepts, Models and Architectures*. Springer, Heidelberg, 2014. ISBN 978-3-642-55048-5.

[30] Raymond Yee. *Pro Web 2.0 Mashups: Remixing Data and Web Services*. Books for professionals by professionals. Apress, 2008. ISBN 978-1-4302-0286-8.

[31] Anant Jhingran. Enterprise Information Mashups: Integrating Information, Simply. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 3–4, Seoul, Korea, 2006. VLDB Endowment. URL `http://dl.acm.org/citation.cfm?id=1182635.1164128`.

[32] David E. Simmen, Mehmet Altinel, Volker Markl, Sriram Padmanabhan, and Ashutosh Singh. Damia: Data Mashups for Intranet Applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1171–1182. ACM, 2008. URL `http://dl.acm.org/citation.cfm?id=1376734`.

[33] Mark Pruett. *Yahoo! Pipes*. O'Reilly, first edition, 2007. ISBN 978-0-596-51453-2.

[34] Eric Griffin. *Foundations of Popfly: Rapid Mashup Development*. Apress, 2008. ISBN 978-1-4302-0568-5.

[35] Loton Tony. *Creating Google Mashups with the Google Mashup Editor*. Lotontech Limited, 2008. ISBN 1-4404-5984-3.

[36] David F. Huynh, David R. Karger, and Robert C. Miller. Exhibit: Lightweight Structured Data Publishing. In *Proceedings of the 16th international conference on World Wide Web*, pages 737–746. ACM, 2007. URL `http://dl.acm.org/citation.cfm?id=1242672`.

[37] Robert J. Ennals and Minos N. Garofalakis. MashMaker: Mashups for the Masses. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1116–1118. ACM, 2007. URL `http://dl.acm.org/citation.cfm?id=1247626`.

[38] Jeffrey Wong and Jason I. Hong. Making Mashups with Marmite: Towards End-User Programming for the Web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1435–1444. ACM, 2007. URL `http://dl.acm.org/citation.cfm?id=1240842`.

[39] Mau Quoc Hoan Nguyen, Martin Serrano, Danh Le-Phuoc, and Manfred Hauswirth. Super Stream Collider–Linked Stream Mashups for Everyone. In *Proceedings of the Semantic Web Challenge co-located with ISWC 2012*, Boston, US, 2012.

[40] Danh Le-Phuoc, Axel Polleres, Manfred Hauswirth, Giovanni Tummarello, and Christian Morbidoni. Rapid Prototyping of Semantic Mash-Ups Through Semantic Web Pipes. In *Proceedings of the 18th international conference on World wide web*, pages 581–590. ACM, 2009. URL `http://dl.acm.org/citation.cfm?id=1526788`.

[41] Mustafa Jarrar and Marios 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*, pages 89–96. ACM, 2008. URL `http://dl.acm.org/citation.cfm?id=1458499`.

[42] Muhammad Imran, Stefano Soi, Felix Kling, Florian Daniel, Fabio Casati, and Maurizio Marchese. On the Systematic Development of Domain-specific Mashup Tools for End Users. In *Web Engineering*, pages 291–298. Springer, 2012. URL `http://link.springer.com/chapter/10.1007/978-3-642-31753-8_22`.

[43] Guiling Wang, Shaohua Yang, and Yanbo Han. Mashroom: End-user Mashup Programming Using Nested Tables. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 861–870, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-487-4. doi: 10.1145/1526709.1526825. URL `http://doi.acm.org/10.1145/1526709.1526825`.

[44] Rob Ennals, Eric Brewer, Minos Garofalakis, Michael Shadle, and Prashant Gandhi. Intel Mash Maker: Join the Web. *ACM SIGMOD Record*, 36(4):27–33, 2007. URL `http://dl.acm.org/citation.cfm?id=1361355`.

[45] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. End-user Programming of Mashups with Vegemite. In *Proceedings of the 14th international conference on Intelligent user interfaces*, pages 97–106. ACM, 2009. URL `http://dl.acm.org/citation.cfm?id=1502667`.

[46] Florian Daniel, Fabio Casati, Boualem Benatallah, and Ming-Chien Shan. Hosted Universal Composition: Models, Languages and Infrastructure in Mashart. In *Conceptual Modeling-ER 2009*, pages 428–443. Springer, 2009. URL `http://link.springer.com/chapter/10.1007/978-3-642-04840-1_32`.

[47] J. Jeffrey Hanson. *Mashups: Strategies for the Modern Enterprise*. Addison-Wesley Professional, 1st edition, 2009. ISBN 0-321-59181-X 978-0-321-59181-4.

[48] Wolfgang Pree. *Design Patterns for Object-oriented Software Development*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995. ISBN 0-201-42294-8.

[49] Vincent Balat. Client-server Web Applications Widgets. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13 Companion, pages 19–22, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-2038-2. URL `http://dl.acm.org/citation.cfm?id=2487788.2487795`.

[50] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web Into a Global Data Space*. Synthesis Lectures on Web Engineering Series. Morgan & Claypool, 2011. ISBN 978-1-60845-430-3.

[51] Anupriya Ankolekar, Markus Krötzsch, Thanh Tran, and Denny Vrandecic. The Two Cultures: Mashing up Web 2.0 and the Semantic Web. In *Proceedings of the 16th international conference on World Wide Web. 2007 may 7-8*. ACM Press, 2007.

[52] Amit Sheth, Kunal Verma, and Karthik Gomadam. Semantics to Energize the Full Services Spectrum. *Communications of the ACM*, 49(7):55–61, 2006. URL `http://dl.acm.org/citation.cfm?id=1139949`.

[53] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic Web Services. *IEEE intelligent systems*, 16(2):46–53, 2001. URL `http://www.computer.org/csdl/mags/ex/2001/02/x2046.pdf`.

[54] Tim O'Reilly. What Is Web 2.0 - Design Patterns and Business Models for the Next Generation of Software. Technical report, University Library of Munich, Germany, 2005. URL `http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1008839`.

[55] Luciano Floridi. Web 2.0 vs. the Semantic Web: A Philosophical Assessment. *Episteme*, 6(01):25–37, 2009. ISSN 1750-0117. doi: 10.3366/E174236000800052X. URL `http://journals.cambridge.org/article_S1742360000001180`.

[56] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill higher education. McGraw-Hill Education, 2010. ISBN 978-0-07-337597-7.

[57] Henry Lieberman, Fabio Paternò, and Volker Wulf. *End User Development*. Human–Computer Interaction Series. Springer Netherlands, 2006. ISBN 978-1-4020-5386-3.

[58] Bonnie A. Nardi. *A Small Matter of Programming Perspectives on End User Computing*. MIT Press, Cambridge, MA, 1993. ISBN 0-585-32629-0 978-0-585-32629-0 0-262-28040-X 978-0-262-28040-2.

[59] Claude Ghaoui. *Encyclopedia of Human Computer Interaction*. ITPro collection. Idea Group Reference, 2005. ISBN 978-1-59140-798-0.

[60] Lars Grammel and Margaret-Anne Storey. An End User Perspective on Mashup Makers. *University of Victoria Technical Report DCS-324-IR*, 2008.

[61] Cinzia Cappiello, Florian Daniel, Maristella Matera, Matteo Picozzi, and Michael Weiss. Enabling End User Development Through Mashups: Requirements, Abstractions and Innovation Toolkits. In *End-User Development*, pages 9–24. Springer, 2011. URL http://link.springer.com/chapter/10.1007/978-3-642-21530-8_3.

[62] Thomas Fischer, Fedor Bakalov, and Andreas Nauerz. An Overview of Current Approaches to Mashup Generation. In *Proceedings of the International Workshop on Knowledge Services and Mashups (KSM09)*, pages 254–259. Citeseer, 2009. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.416.7684&rep=rep1&type=pdf#page=255.

[63] Marwan Sabbouh, Jeff Higginson, Salim Semy, and Danny Gagne. Web Mashup Scripting Language. In *Proceedings of the 16th international conference on World Wide Web*, pages 1305–1306. ACM, 2007. URL http://dl.acm.org/citation.cfm?id=1242821.

[64] Erhard Rahm, Andreas Thor, and David Aumueller. Dynamic Fusion of Web Data. In *Database and XMLTechnologies*, volume 4704 of *Lecture Notes in Computer Science*, pages 14–16. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-75287-5. URL http://dx.doi.org/10.1007/978-3-540-75288-2_2.

[65] Rattapoom Tuchinda, Pedro Szekely, and Craig A. Knoblock. Building Mashups by Example. In *Proceedings of the 13th international conference on Intelligent user interfaces*, pages 139–148. ACM, 2008. URL http://dl.acm.org/citation.cfm?id=1378792.

[66] David F. Huynh, Robert C. Miller, and David R. Karger. Potluck: Data Mash-Up Tool for Casual Users. In *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 239–252. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-76297-3. URL http://dx.doi.org/10.1007/978-3-540-76298-0_18.

[67] Michael Pierre Carlson, Anne H.H. Ngu, Rodion Podorozhny, and Liangzhao Zeng. Automatic Mash up of Composite Applications. In *Service-Oriented Computing–ICSOC 2008*, pages 317–330. Springer, 2008. URL http://link.springer.com/chapter/10.1007/978-3-540-89652-4_25.

[68] Thomas Fischer, Fedor Bakalov, and Andreas Nauerz. Towards an Automatic Service Composition for Generation of User-Sensitive Mashups. In *LWA*, pages 14–16. Citeseer, 2008. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.9191&rep=rep1&type=pdf#page=20.

[69] Margaret Burnett, Curtis Cook, and Gregg Rothermel. End-User Software Engineering. *Communications of the ACM*, 47(9):53–58, 2004. URL http://dl.acm.org/citation.cfm?id=1015889.

[70] Hendrik, Amin Anjomshoaa, and A Min Tjoa. Towards Semantic Mashup Tools for Big Data Analysis. In *Information and Communication Technology*, volume 8407 of *Lecture Notes in Computer Science*, pages 129–138. Springer Berlin Heidelberg, 2014. ISBN 978-3-642-55031-7. URL `http://dx.doi.org/10.1007/978-3-642-55032-4_13`.

[71] Johannes Lorey, Felix Naumann, Benedikt Forchhammer, Andrina Mascher, Peter Retzlaff, and Armin ZamaniFarahani. Black Swan: Augmenting Statistics with Event Data. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management (CIKM 2011)*, pages 2517–2520, Glasgow, United Kingdom, 2011. doi: 10.1145/2063576.2064007. URL `http://doi.acm.org/10.1145/2063576.2064007`.

[72] Saeed Aghaee and Cesare Pautasso. End-User Programming for Web Mashups: Open Research Challenges. In *Proceedings of the 11th International Conference on Current Trends in Web Engineering*, ICWE '11, pages 347–351, Paphos, Cyprus, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-27996-6. doi: 10.1007/978-3-642-27997-3_38. URL `http://dx.doi.org/10.1007/978-3-642-27997-3_38`.

[73] E.Michael Maximilien, Hernan Wilkinson, Nirmit Desai, and Stefan Tai. A Domain-Specific Language for Web APIs and Services Mashups. In *Service-Oriented Computing – ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 13–26. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-74973-8. URL `http://dx.doi.org/10.1007/978-3-540-74974-5_2`.

[74] Fedor Bakalov, Birgitta König-Ries, Andreas Nauerz, and Martin Welsch. Ontology-Based Multidimensional Personalization Modeling for the Automatic Generation of Mashups in Next-Generation Portals. In *First International Workshop on Ontologies in Interactive Systems ONTORACT '08*, pages 75–82. IEEE, September 2008. ISBN 978-0-7695-3542-5 978-1-4244-3459-6. doi: 10.1109/ONTORACT.2008.13. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4756198`.

[75] Saeed Aghaee and Cesare Pautasso. An Evaluation of Mashup Tools Based on Support for Heterogeneous Mashup Components. In *Current Trends in Web Engineering*, pages 1–12. Springer, 2012. URL `http://link.springer.com/chapter/10.1007/978-3-642-27997-3_1`.

[76] Lars Grammel and Margaret-Anne Storey. A Survey of Mashup Development Environments. In *The smart internet*, pages 137–151. Springer, 2010. URL `http://link.springer.com/chapter/10.1007/978-3-642-16599-3_10`.

[77] Andrew Ko, Brad Myers, and Htet Aung. Six Learning Barriers in End-User Programming Systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, pages 199–206, Washington,

DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8696-5. doi: 10.1109/VLHCC.2004.47. URL http://dx.doi.org/10.1109/VLHCC.2004.47.

[78] Giusy Di Lorenzo, Hakim Hacid, Hye-young Paik, and Boualem Benatallah. Data Integration in Mashups. *ACM Sigmod Record*, 38(1):59–66, 2009. URL http://dl.acm.org/citation.cfm?id=1558343.

[79] Florian Daniel, Agnes Koschmider, Tobias Nestler, Marcus Roy, and Abdallah Namoun. Toward Process Mashups: Key Ingredients and Open Research Challenges. In *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, page 9. ACM, 2010. URL http://dl.acm.org/citation.cfm?id=1945008.

[80] David Lizcano, Javier Soriano, Marcos Reyes, and Juan J. Hierro. EzWeb/FAST: Reporting on a Successful Mashup-Based Solution for Developing and Deploying Composite Applications in the Upcoming "Ubiquitous SOA". In *Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008. UBICOMM '08.*, pages 488–495, Valencia, 2008. doi: 10.1109/UBICOMM.2008.61.

[81] Tobias Nestler, Marius Feldmann, Gerald Hübsch, André Preußner, and Uwe Jugel. The ServFace Builder - A WYSIWYG Approach for Building Service-Based Applications. In *Web Engineering*, volume 6189 of *Lecture Notes in Computer Science*, pages 498–501. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-13910-9. URL http://dx.doi.org/10.1007/978-3-642-13911-6_37.

[82] Florian Daniel, Stefano Soi, Stefano Tranquillini, Fabio Casati, Chang Heng, and Li Yan. From People to Services to UI: Distributed Orchestration of User Interfaces. In *Business Process Management*, volume 6336 of *Lecture Notes in Computer Science*, pages 310–326. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15617-5. URL http://dx.doi.org/10.1007/978-3-642-15618-2_22.

[83] Gregor Blichmann, Carsten Radeck, and Klaus Meißner. Enabling End Users to Build Situational Collaborative Mashups at Runtime. In *ICIW 2013, The Eighth International Conference on Internet and Web Applications and Services*, pages 120–123, 2013. URL http://www.thinkmind.org/index.php?view=article&articleid=iciw_2013_5_40_20132.

[84] Stefan Pietschmann, Carsten Radeck, and Klaus Meißner. Semantics-Based Discovery, Selection and Mediation for Presentation-Oriented Mashups. In *Proceedings of the 5th International Workshop on Web APIs and Service Mashups*, page 7. ACM, 2011. URL http://dl.acm.org/citation.cfm?id=2076014.

[85] Brigitte Endres-Niggemeyer. *Semantic Mashups.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-36402-0 978-3-642-36403-7. URL http://link.springer.com/10.1007/978-3-642-36403-7.

[86] Aikaterini K. Kalou, Tzanetos Pomonis, Dimitrios A. Koutsomitropoulos, and Theodore S. Papatheodorou. Intelligent Book Mashup: Using Semantic Web Ontologies and Rules for User Personalisation. In *Proceedings of the 4th IEEE International Conference on Semantic Computing (ICSC 2010), September 22-24, 2010, Carnegie Mellon University, Pittsburgh, PA, USA*, pages 536–541, 2010. doi: 10.1109/ICSC.2010.78. URL `http://dx.doi.org/10.1109/ICSC.2010.78`.

[87] Christian Bizer, Richard Cyganiak, and Tobias Gauss. The RDF Book Mashup: From Web APIs to a Web of Data. In *Proceedings of the ESWC'07 Workshop on Scripting for the Semantic Web, SFSW 2007, Innsbruck, Austria, May 30, 2007*, 2007. URL `http://ceur-ws.org/Vol-248/paper4.pdf`.

[88] D. Gagne, M. Sabbouh, S. Bennett, and S. Powers. Using Data Semantics to Enable Automatic Composition of Web Services. In *Services Computing, 2006. SCC '06. IEEE International Conference on*, pages 438–444, September 2006. doi: 10.1109/SCC.2006.112.

[89] Stefan Pietschmann, Vincent Tietz, Jan Reimann, Christian Liebing, Michèl Pohle, and Klaus Meißner. A Metamodel for Context-Aware Component-Based Mashup Applications. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services*, pages 413–420. ACM, 2010. URL `http://dl.acm.org/citation.cfm?id=1967551`.

[90] Muhammad Imran, Felix Kling, Stefano Soi, Florian Daniel, Fabio Casati, and Maurizio Marchese. ResEval Mash: A Mashup Tool for Advanced Research Evaluation. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 361–364. ACM, 2012. URL `http://dl.acm.org/citation.cfm?id=2188049`.

[91] Patrick Gaubatz and Uwe Zdun. UML2 Profile and Model-Driven Approach for Supporting System Integration and Adaptation of Web Data Mashups. In *Current Trends in Web Engineering*, pages 81–92. Springer, 2012. URL `http://link.springer.com/chapter/10.1007/978-3-642-35623-0_9`.

[92] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004. ISBN 0-13-146575-9. URL `http://dl.acm.org/citation.cfm?id=1096077`.

[93] Natalya F. Noy and Deborah L. Mcguinness. Ontology Development 101: A Guide to Creating Your First Ontology. Technical report, Stanford University, 2001.

[94] Thomas R. Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal of Human-Computer Studies*, 43(5-6): 907–928, December 1995. ISSN 1071-5819. doi: 10.1006/ijhc.1995.1081. URL `http://dx.doi.org/10.1006/ijhc.1995.1081`.

[95] Tom Gruber. Ontology. In *Encyclopedia of Database Systems*, pages 1963–1965. Springer US, 2009. ISBN 978-0-387-35544-3. URL http://dx.doi.org/10.1007/978-0-387-39940-9_1318.

[96] Jacek Kopecky, Tomas Vitvar, Carine Bournez, and Joel Farrell. SAWSDL: Semantic Annotations for WSDL and XML Schema. *Internet Computing, IEEE*, 11 (6):60–67, 2007. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4376229.

[97] Mohsen Taheriyan, Craig A. Knoblock, Pedro Szekely, and José Luis Ambite. Rapidly Integrating Services into the Linked Data Cloud. In *The Semantic Web–ISWC 2012*, pages 559–574. Springer, 2012. URL http://link.springer.com/chapter/10.1007/978-3-642-35176-1_35.

[98] Mohsen Taheriyan, Craig A. Knoblock, Pedro Szekely, and José Luis Ambite. A Graph-Based Approach to Learn Semantic Descriptions of Data Sources. In *The Semantic Web–ISWC 2013*, pages 607–623. Springer, 2013. URL http://link.springer.com/chapter/10.1007/978-3-642-41335-3_38.

[99] Ruben Verborgh, Thomas Steiner, Davy Van Deursen, Rik Van de Walle, and Joaquim Gabarró Vallés. Efficient Runtime Service Discovery and Consumption with Hyperlinked RESTdesc. In *Proceedings of the 7th International Conference on Next Generation Web Services Practices (2011)*, pages 373–379, October 2011. doi: 10.1109/NWeSP.2011.6088208.

[100] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. ISBN 0-7167-1045-5.

[101] Rakesh M. Verma and Steven W. Reyner. An Analysis of a Good Algorithm for the Subtree Problem, Correlated. *SIAM Journal on Computing*, 18(5):906–908, October 1989. ISSN 0097-5397. doi: 10.1137/0218062. URL http://dx.doi.org/10.1137/0218062.

[102] Wang Vanessa, Moskovits Peter, and Salim Frank. *The Definitive Guide to HTML5 WebSocket*. Apress, New York, NY, USA, 2013. ISBN 978-1-4302-4741-8.

[103] Tony Bourke. *Server Load Balancing*. Help for network administrators. O'Reilly Media, Incorporated, 2001. ISBN 978-0-596-00050-9.

[104] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. DBpedia – A Large-Scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web*, 6(2):167–195, 2015. doi: 10.3233/SW-140134. URL http://dx.doi.org/10.3233/SW-140134.

[105] Pablo N. Mendes, Max Jakob, Andrés García-Silva, and Christian Bizer. DBpedia Spotlight: Shedding Light on the Web of Documents. In *Proceedings of the 7th International Conference on Semantic Systems*, pages 1–8. ACM, 2011. URL http://dl.acm.org/citation.cfm?id=2063519.

[106] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2 edition, September 2000. ISBN 0-13-014400-2.

[107] Jan Karel Lenstra, A.H.G Rinnooy Kan, and Alexander Schrijver. *History of Mathematical Programming: A Collection of Personal Reminiscences*. CWI, 1991. ISBN 978-0-444-88818-1.

[108] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009. ISSN 0362-5915. doi: 10.1145/1567274.1567278. URL http://doi.acm.org/10.1145/1567274.1567278.

[109] Ruben Verborgh, Miel Vander Sande, Pieter Colpaert, Sam Coppens, Erik Mannens, and Rik Van de Walle. Web-Scale Querying through Linked Data Fragments. In *Proceedings of the 7th Workshop on Linked Data on the Web*, April 2014. URL http://events.linkeddata.org/ldow2014/papers/ldow2014_paper_04.pdf.

[110] Jérôme Euzenat and Pavel Shvaiko. *Ontology Matching, Second Edition*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-38720-3.

[111] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. JSON-LD 1.0, 2014. URL http://www.w3.org/TR/json-ld/.

[112] Glen Hart and Catherine Dolbear. *Linked Data: A Geographic Perspective*. CRC Press, 2013.

[113] Werner Kuhn, Tomi Kauppinen, and Krzysztof Janowicz. Linked Data – A Paradigm Shift for Geographic Information Science. In *Geographic Information Science*, volume 8728 of *Lecture Notes in Computer Science*, pages 173–186. Springer International Publishing, 2014. ISBN 978-3-319-11592-4. URL http://dx.doi.org/10.1007/978-3-319-11593-1_12.

[114] Kwok-Bun Yue. Experience on Mashup Development with End User Programming Environment. *Journal of Information Systems Education*, 21(1):111, 2010. URL http://prtl.uhcl.edu/portal/page/portal/SCE/COMPUTING_MATHMATICS_DIV/CM_Documents/YahooPipe_JISE_Paper_Published.pdf.

[115] Andreas Schultz, Andrea Matteini, Robert Isele, Christian Bizer, and Christian Becker. LDIF - Linked Data Integration Framework. In *Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn,*

Germany, October 23, 2011, 2011. URL http://ceur-ws.org/Vol-782/SchultzEtAl_COLD2011.pdf.

[116] Devis Bianchini and Valeria De Antonellis. Linked Data Services and Semantics-Enabled Mashup. In *Semantic Search over the Web*, Data-Centric Systems and Applications, pages 283–307. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-25007-1. URL http://dx.doi.org/10.1007/978-3-642-25008-8_11.

[117] Andreas Harth, Craig A. Knoblock, Steffen Stadtmüller, Rudi Studer, and Pedro Szekely. On-the-fly Integration of Static and Dynamic Linked Data. In *Proceedings of the Fourth International Workshop on Consuming Linked Data co-located with the 12th International Semantic Web Conference*, pages 1613–0073, 2013. URL http://planet-data.org/sites/default/files/publications/HarthEtAl_COLD2013.pdf.

[118] Tuan-Nhat Tran, Duy-Khanh Truong, Hanh-Huu Hoang, and Thanh-Manh Le. Linked Data Mashups: A Review on Technologies, Applications and Challenges. In *Intelligent Information and Database Systems*, volume 8398 of *Lecture Notes in Computer Science*, pages 253–262. Springer International Publishing, 2014. ISBN 978-3-319-05457-5. URL http://dx.doi.org/10.1007/978-3-319-05458-2_27.

[119] Fuyuko Matsumura, Iwao Kobayashi, Fumihiro Kato, Tetsuro Kamura, Ikki Ohmukai, and Hideaki Takeda. Producing and Consuming Linked Open Data on Art with a Local Community. In *COLD*, 2012. URL http://ceur-ws.org/Vol-905/MatsumuraEtAl_COLD2012.pdf.

[120] Fahad Alahmari, James A. Thom, Liam Magee, and Wilson Wong. Evaluating Semantic Browsers for Consuming Linked Data. In *Proceedings of the Twenty-Third Australasian Database Conference - Volume 124*, ADC '12, pages 89–98, Darlinghurst, Australia, Australia, 2012. Australian Computer Society, Inc. ISBN 978-1-921770-05-0. URL http://dl.acm.org/citation.cfm?id=2483739.2483751.

[121] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. An Efficient SQL-based RDF Querying Scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1216–1227, Trondheim, Norway, 2005. VLDB Endowment. ISBN 1-59593-154-6. URL http://dl.acm.org/citation.cfm?id=1083592.1083734.

[122] Arto Salminen and Tommi Mikkonen. Towards Pervasive Mashups in Embedded Devices: Comparing Procedural and Declarative Approach. *IJCNDS*, 10(3):195–215, 2013. doi: 10.1504/IJCNDS.2013.053077. URL http://dx.doi.org/10.1504/IJCNDS.2013.053077.

[123] Tommi Mikkonen and Arto Salminen. Towards Pervasive Mashups in Embedded Devices. In *IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 35–42. IEEE, August 2010. ISBN 978-1-4244-8480-5. doi: 10.1109/RTCSA.2010.16. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5591281`.

[124] Ray Rischpater and Daniel Zucker. Working with the Nokia Qt SDK. In *Beginning Nokia Apps Development*, pages 39–57. Apress, 2010. ISBN 978-1-4302-3177-6. URL `http://dx.doi.org/10.1007/978-1-4302-3178-3_3`.

[125] Yoon-Seop Chang, Seong-Ho Lee, Jae-Chul Kim, and Young-Jae Lim. Study on Mobile Mashup Webapp Development Tools for Different Devices and User Groups. In *Information Networking (ICOIN), 2014 International Conference on*, pages 433–438, February 2014. doi: 10.1109/ICOIN.2014.6799719.

[126] Fabio Corvetta, Maristella Matera, Riccardo Medana, Elisa Quintarelli, Vincenzo Rizzo, and Letizia Tanca. Designing and Developing Context-Aware Mobile Mashups: The CAMUS Approach. In *Engineering the Web in the Big Data Era*, volume 9114 of *Lecture Notes in Computer Science*, pages 651–654. Springer International Publishing, 2015. ISBN 978-3-319-19889-7. URL `http://dx.doi.org/10.1007/978-3-319-19890-3_49`.

[127] Carmelo Ardito, Maria Francesca Costabile, Giuseppe Desolda, Rosa Lanzilotti, Maristella Matera, Antonio Piccinno, and Matteo Picozzi. User-Driven Visual Composition of Service-Based Interactive Spaces. *Journal of Visual Languages & Computing*, 25(4):278 – 296, 2014. ISSN 1045-926X. doi: http://dx.doi.org/10.1016/j.jvlc.2014.01.003. URL `http://www.sciencedirect.com/science/article/pii/S1045926X14000299`.

[128] Shang-Pin Ma, Yang-Sheng Ma, and Wen-Tin Lee. State-Driven and Brick-Based Mobile Mashup. In *Mobile Services (MS), 2015 IEEE International Conference on*, pages 190–196, June 2015. doi: 10.1109/MobServ.2015.35.

[129] Cesare Pautasso and Gustavo Alonso. The JOpera Visual Composition Language. *J. Vis. Lang. Comput.*, 16(1-2):119–152, February 2005. ISSN 1045-926X. doi: 10.1016/j.jvlc.2004.08.004. URL `http://dx.doi.org/10.1016/j.jvlc.2004.08.004`.

[130] Matteo Picozzi. End User Development of Multidevice and Collaborative Mashups. In *CHItaly 2013 Doctoral Consortium*, pages 55–65, Trento, 2013. Citeseer.

[131] Nelly Schuster, Raffael Stein, and Christian Zirpins. A Mashup Tool for Collaborative Engineering of Service-Oriented Enterprise Documents. In *Information Systems Evolution*, volume 72 of *Lecture Notes in Business Information Processing*, pages 166–173. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-17721-7. URL `http://dx.doi.org/10.1007/978-3-642-17722-4_12`.

[132] Jason J. Jung. ContextGrid: A Contextual Mashup-Based Collaborative Browsing System. *Information Systems Frontiers*, 14(4):953–961, 2012. ISSN 1387-3326. doi: 10.1007/s10796-011-9315-z. URL http://dx.doi.org/10.1007/s10796-011-9315-z.

[133] Michael Hertel, Alexey Tschudnowsky, and Martin Gaedke. Conflict Resolution in Collaborative User Interface Mashups. In *Engineering the Web in the Big Data Era*, volume 9114 of *Lecture Notes in Computer Science*, pages 659–662. Springer International Publishing, 2015. ISBN 978-3-319-19889-7. URL http://dx.doi.org/10.1007/978-3-319-19890-3_51.

[134] Yi Xu, Chengzheng Sun, and Mo Li. Achieving Convergence in Operational Transformation: Conditions, Mechanisms and Systems. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work &#38; Social Computing*, CSCW '14, pages 505–518, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2540-0. doi: 10.1145/2531602.2531629. URL http://doi.acm.org/10.1145/2531602.2531629.

[135] Jinghai Rao and Xiaomeng Su. A Survey of Automated Web Service Composition Methods. In *Semantic Web Services and Web Process Composition*, pages 43–54. Springer, 2005. URL http://link.springer.com/chapter/10.1007/978-3-540-30581-1_5.

[136] Schahram Dustdar and Wolfgang Schreiner. A Survey on Web Services Composition. *International journal of web and grid services*, 1(1):1–30, 2005. URL http://inderscience.metapress.com/index/473epwhagp6hbeba.pdf.

[137] Thomas Fischer, Fedor Bakalov, Birgitta König-Ries, Andreas Nauerz, and Martin Welsch. An Evolutionary Algorithm for Automatic Composition of Information-gathering Web Services in Mashups. In *Seventh IEEE European Conference on Web Services ECOWS '09*, pages 39–48. IEEE, November 2009. ISBN 978-0-7695-3854-9. doi: 10.1109/ECOWS.2009.9. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5341673.

[138] Yuzhang Feng, Anitha Veeramani, and Rajaraman Kanagasabai. Enabling On-Demand Mashups of Open Data with Semantic Services. In *2013 International Conference on Parallel and Distributed Systems*, volume 0, pages 755–759, 2012. doi: http://doi.ieeecomputersociety.org/10.1109/ICPADS.2012.122.

[139] Rima Kilany Chamoun. Smart: Semantically Mashup Rest Web Services. *CoRR*, abs/1311.3078, 2013. URL http://arxiv.org/abs/1311.3078.

[140] Anton V. Riabov, Eric Boillet, Mark D. Feblowitz, Zhen Liu, and Anand Ranganathan. Wishful Search: Interactive Composition of Data Mashups. In *Proceedings of the 17th international conference on World Wide Web*, pages 775–784. ACM, 2008. URL http://dl.acm.org/citation.cfm?id=1367602.

[141] Anne H.H. Ngu, Michael P. Carlson, Quan Z. Sheng, and Hye-young Paik. Semantic-Based Mashup of Composite Applications. *IEEE Transactions on Services Computing*, 3(1):2–15, January 2010. ISSN 1939-1374. doi: 10.1109/TSC.2010.8. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5432153`.

[142] Hazem Elmeleegy, Anca Ivan, Rama Akkiraju, and Richard Goodwin. Mashup Advisor: A Recommendation Tool for Mashup Development. In *Web Services, 2008. ICWS '08*, pages 337–344, Beijing, September 2008. IEEE. doi: 10.1109/ICWS.2008.128. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4670193`.

[143] Daniel Deutch, Ohad Greenshpan, and Tova Milo. Navigating in Complex Mashed-up Applications. *Proceedings of the VLDB Endowment*, 3(1-2):320–329, 2010. URL `http://dl.acm.org/citation.cfm?id=1920885`.

[144] Lin Bai, Dan Ye, and Jun Wei. A Goal Decomposition Approach for Automatic Mashup Development. In *Enterprise Interoperability*, pages 20–33. Springer, 2012. URL `http://link.springer.com/chapter/10.1007/978-3-642-33068-1_4`.

[145] Carlos Rodríguez, Soudip Roy Chowdhury, Florian Daniel, Hamid R. Motahari Nezhad, and Fabio Casati. Assisted Mashup Development: On the Discovery and Recommendation of Mashup Composition Knowledge. In *Web Services Foundations*, pages 683–708. Springer New York, New York, NY, 2014. ISBN 978-1-4614-7517-0 978-1-4614-7518-7. URL `http://link.springer.com/10.1007/978-1-4614-7518-7_27`.

[146] Serge Abiteboul, Ohad Greenshpan, Tova Milo, and Neoklis Polyzotis. MatchUp: Autocompletion for Mashups. In *Data Engineering, 2009. ICDE '09*, pages 1479–1482, Shanghai, March 2009. IEEE. ISBN 978-1-4244-3422-0. doi: 10.1109/ICDE.2009.47. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4812552`.

[147] Carsten Radeck, Alexander Lorz, Gregor Blichmann, and Klaus Meißner. Hybrid Recommendation of Composition Knowledge for End User Development of Mashups. In *ICIW 2012, The Seventh International Conference on Internet and Web Applications and Services*, pages 30–33, 2012. URL `http://www.thinkmind.org/index.php?view=article&articleid=iciw_2012_2_10_20180`.

[148] Soudip Roy Chowdhury, Olexiy Chudnovskyy, Matthias Niederhausen, Stefan Pietschmann, Paul Sharples, Florian Daniel, and Martin Gaedke. Complementary Assistance Mechanisms for End User Mashup Composition. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 269–272. International World Wide Web Conferences Steering Committee, 2013. URL `http://dl.acm.org/citation.cfm?id=2487919`.

[149] Matthias Henneberger, Bernd Heinrich, Florian Lautenbacher, and Bernhard Bauer. Semantic-Based Planning of Process Models. *Proceedings of the Multikonferenz Wirtschaftsinformatik (MKWI) 2008*, pages 1677–1689, 2008.

[150] Vincent Tietz, Gregor Blichmann, Stefan Pietschmann, and Klaus Meißner. Task-Based Recommendation of Mashup Components. In *Current Trends in Web Engineering*, pages 25–36. Springer, 2012. URL `http://link.springer.com/chapter/10.1007/978-3-642-27997-3_3`.

[151] Ohad Greenshpan, Tova Milo, and Neoklis Polyzotis. Autocompletion for Mashups. *Proc. VLDB Endow.*, 2(1):538–549, August 2009. ISSN 2150-8097. doi: 10.14778/1687627.1687689. URL `http://dx.doi.org/10.14778/1687627.1687689`.

[152] Stefan Pietschmann. A Model-Driven Development Process and Runtime Platform for Adaptive Composite Web Applications. *International Journal on Advances in Internet Technology*, 2(4):277–288, 2010. URL `http://www.thinkmind.org/index.php?view=article&articleid=inttech_v2_n4_2009_2`.

[153] Tomas Vitvar, Jacek Kopecky, Maciej Zaremba, and Dieter Fensel. WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web. In *Web Services, 2007. ECOWS '07. Fifth European Conference on*, pages 77–86, November 2007. doi: 10.1109/ECOWS.2007.30.

[154] Jens Lehmann and Lorenz Bühmann. AutoSPARQL: Let Users Query Your Knowledge Base. In *The Semantic Web: Research and Applications*, pages 63–79. Springer, 2011. URL `http://link.springer.com/chapter/10.1007/978-3-642-21034-1_5`.

[155] Saeed Aghaee and Cesare Pautasso. EnglishMash: Usability Design for a Natural Mashup Composition Environment. In *Current Trends in Web Engineering*, pages 109–120. Springer, 2012. URL `http://link.springer.com/chapter/10.1007/978-3-642-35623-0_12`.

[156] Saeed Aghaee, Cesare Pautasso, and Antonella De Angeli. Natural End-User Development of Web Mashups. In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, pages 111–118. IEEE, 2013. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6645253`.

[157] Mariano Belaunde and Slim Ben Hassen. Service Mashups Using Natural Language and Context Awareness: A Pragmatic Architectural Design. In *Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011 15th IEEE International*, pages 404–411, Helsinki, August 2011.

[158] Mariano Belaunde and Paolo Falcarin. Realizing an MDA and SOA Marriage for the Development of Mobile Services. In *Model Driven Architecture – Foundations*

and Applications, volume 5095 of *Lecture Notes in Computer Science*, pages 393–405. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-69095-5. URL `http://dx.doi.org/10.1007/978-3-540-69100-6_28`.

[159] Fabrice Desre. Open Mashups: User Generated Applications for the Masses. Technical report, Orange Labs, 2009. URL `http://www.archive.org/details/Cubicgarden-SANY0010a666-2`.

[160] Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 943–946, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-593-9. doi: 10.1145/1240624.1240767. URL `http://doi.acm.org/10.1145/1240624.1240767`.