International Journal of Web Information Systems

Distributed mashups: a collaborative approach to data integration

Tuan-Dat Trinh Peter Wetz Ba-Lam Do Elmar Kiesling A Min Tjoa

Received 23 April 2015
Revised 20 May 2015
Accepted 20 May 2015

Distributed mashups: a collaborative approach to data integration

Tuan-Dat Trinh, Peter Wetz and Ba-Lam Do
Vienna University of Technology, Vienna, Austria
Elmar Kiesling
Information and Software Engineering Group, Vienna University of Technology, Vienna, Austria, and
A. Min Tjoa
Vienna University of Technology, Vienna, Austria

Abstract

Purpose – This paper aims to present a collaborative mashup platform for dynamic integration of heterogeneous data sources. The platform encourages sharing and connects data publishers, integrators, developers and end users.

Design/methodology/approach – This approach is based on a visual programming paradigm and follows three fundamental principles: openness, connectedness and reusability. The platform is based on semantic Web technologies and the concept of linked widgets, i.e. semantic modules that allow users to access, integrate and visualize data in a creative and collaborative manner.

Findings – The platform can effectively tackle data integration challenges by allowing users to explore relevant data sources for different contexts, tackling the data heterogeneity problem and facilitating automatic data integration, easing data integration via simple operations and fostering reusability of data processing tasks.

Research limitations/implications – This research has focused exclusively on conceptual and technical aspects so far; a comprehensive user study, extensive performance and scalability testing is left for future work.

Originality/value – A key contribution of this paper is the concept of distributed mashups. These *ad hoc* data integration applications allow users to perform data processing tasks in a collaborative and distributed manner simultaneously on multiple devices. This approach requires no server infrastructure to upload data, but rather allows each user to keep control over their data and expose only relevant subsets. Distributed mashups can run persistently in the background and are hence ideal for real-time data monitoring or data streaming use cases. Furthermore, we introduce automatic mashup composition as an innovative approach based on an explicit semantic widget model.

Keywords Advanced web applications, Web data integration, Web semantics architectures, Applications and standards, Workflow architectures in support of collaboration process

Paper type Research paper

International Journal of Web Information Systems
Vol. 11 No. 3, 2015
pp. 370-396

1744-0084
DOI 10.1108/IJWIS-04-2015-0018

1. Introduction

Data, due to their abundant availability, play an increasingly important role in everyday life. Collecting relevant data from various sources and extracting useful information from it, however, has not become easier, as the quantity of data made available by

organizations and governments has grown. Open data – which cover a wide range of topics, e.g. economy, currency, geography, entertainment, weather, transportation, etc. – can be used for various purposes.

For users to benefit from the available data's value, however, we need to deal with a number of data integration challenges:

- *Data heterogeneity* makes it difficult to integrate different kinds of data in various formats (e.g. CSV, XML, JSON, RDF, JSON-LD) spread among various storage infrastructures (e.g. databases, files, cloud, personal computers, mobile phones).
- Tedious manual data integration processes that users perform to collect, clean, enrich, integrate and visualize data are typically neither *reproducible* nor *reusable*.
- 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.
- Lack of means for the identification of relevant data sources and meaningful ways to integrate them.

Furthermore, many data integration and analysis tasks are collaborative by nature. They often require the sharing of data held privately by various stakeholders, a – often geographically dispersed – team with a broad skill set, and the agreement on a common interpretation to arrive at relevant insights. Conceiving data exploration, integration, and analysis as a collaborative process hence creates strong potential for both simple *ad hoc* data sharing and sophisticated data-driven decision support.

In this paper, we introduce a mashup-based *collaborative* data integration platform that provides a technical infrastructure for such data-centric cooperative work. The paper is an extended version of work published by Trinh *et al.* (2014). We extend our previous work by introducing the concepts of collaborative and distributed mashups. Following an end-user visual programming paradigm, we develop a *widget-based mashup* approach that lets users create data-centric applications easily out of simple building blocks. With a limited upfront learning time investment (Nardi, 1993), users can collaboratively combine available widgets and build *ad hoc* applications that integrate data from various sources.

Another innovative aspect of our work is that widgets can be executed in *distributed* environments. Mashups can be composed of both *client* and *server widgets*. *Client widgets* are executed in the local context of a Web browser environment. *Server widgets* can be written in various programming languages such as C++, Objective-C, C#, PHP, Python, JavaScript and Java. They 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 or to use the computing resources of the node to persistently process data in the background. This architecture allows stakeholders to expose their private data selectively by contributing *server widgets* as functional black boxes. Only the owner of each *server widget* can control its parameters and hence the output it provides. This approach facilitates efficient *ad hoc* data integration involving multiple stakeholders that contribute data and computing resources.

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.

The platform implementation follows three fundamental principles: *openness*, *connectedness* and *reusability*. *Openness* means that developers can implement and directly add their widgets to the platform. *Connectedness* implies that users can integrate widgets from various developers and users by connecting them in a mashup. Finally, we foster *reusability* by allowing users to share and adapt widgets and mashups.

A prototype implementation of the platform is available at http://linkedwidgets.org. It provides:

- a tool that supports developers in creating and annotating widgets;
- a tool for users to locate relevant widgets; and
- a drag-and-drop collaborative mashup editor.

We make use of semantic Web technologies to model-linked widgets. This allows us to leverage their internal model to automatically identify opportunities for the integration of data sources. Furthermore, from a specified list of data sources and related available linked widgets, we can automatically integrate the data.

The remainder of this paper is organized as follows. Section 2 provides an overview of the platform architecture. Section 3 introduces linked widgets and their semantic model, which constitute the basic elements of the platform; Section 4 outlines how server and client widgets are linked and interact; Section 5 illustrates the applicability by means of simple use cases and Section 6 discusses our prototype implementation. Finally, we discuss related work, draw conclusions and outline future research directions in Sections 7 and 8, respectively.

2. Architecture

Figure 1 provides an overview of the platform architecture. It serves three major groups of stakeholders: *developers*, *mashup creators* and *mashup users*. The platform can integrate raw data in CSV, XML, JSON or HTML formats; furthermore, data can be collected from databases, cloud/API services or the linked open data (LOD) cloud.

Linked widgets constitute the basic elements of the platform; they extend the concept of standard Web widgets[1] with a semantic model. This semantic model, which follows the linked data principles (Bizer *et al.*, 2009), is used to annotate the input and output of widgets as well as their relations among each other.

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[2] for data transmission between widgets. *Linked widgets* are our key concept to tackle the challenge of data heterogeneity. They standardize data and lift arbitrary data sources to a semantic level.

Depending on their execution mode, widgets may be classified as *client* or *server widgets*. From a functional point of view, we can furthermore categorize widgets as follows:

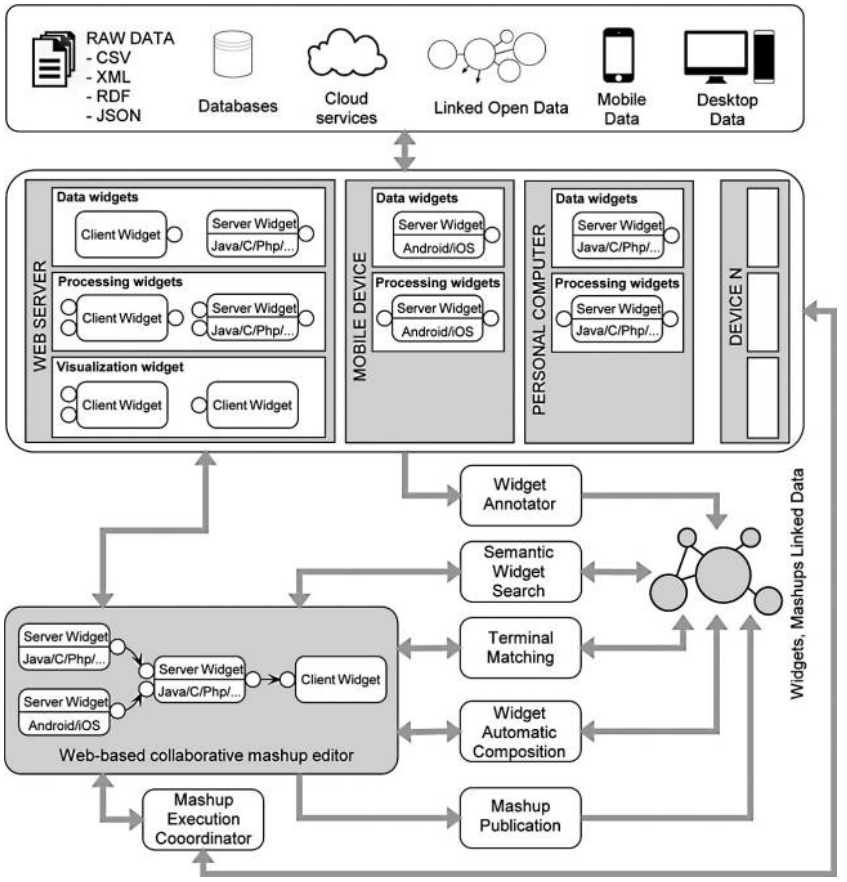

Figure 1.
Mashup platform architecture

- *data widgets* collect data from one or multiple data sources;
- *processing widgets* process and combine data in different ways through enrichment, transformation and aggregation;
- *visualization widgets* finally display results.

Accordingly, widgets are organized into three layers, i.e. a data layer, a business logic layer and a presentation layer. Each mashup consists of at least one *data widget* and a single or multiple *visualization widgets*; *processing widgets* are optional. Widgets are reusable and can be parametrized per mashup.

Developers can contribute *client widgets* to the platform; 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* tool 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 platform.

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:

(1) *local mashups* that consist exclusively of client widgets;

(2) *hybrid mashups* that make use of both client and server widgets; and

(3) *distributed mashups* that consist entirely of server widgets, except for the final visualization widget(s).

A *local mashup* does not use any resources of the platform 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 execute 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 mashup, the coordination mechanisms differ to conserve computing resources. Details of the protocols and mechanisms are discussed in Section 4.

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; the module validates connections using the semantic model. It helps creators to speed up the mashup creation process.

The *automatic mashup composition* module is a more advanced approach in that it can automatically compose a complete mashup from a widget or a complete branch that consumes/provides data for a specific output/input terminal. "*Complete*" in this context means that all terminals must be wired, i.e. have a valid connection.

Mashup users can save and publish mashups on their Web site 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. Linked widgets

A linked widget is similar to a Web service, in that it has multiple inputs and a single output. Important distinctions, however, include that:

- widgets are intended to be used by end users, whereas Web services are intended for developers;
- widgets have a user interface and are hence easier to handle than Web services;
- widgets are more versatile than Web services, e.g. they can visualize data;
- a valid combination of widgets can collaborate automatically by default, whereas Web service composition requires more technical work, e.g. conformation of parameters;
- connected widget can interact with each other both ways, Web service communication is always sequential and unidirectional;
- widgets can be deployed on various devices and in various environments, whereas Web services typically run on a server; and finally
- linked widgets are always associated with a semantic model.

Developers can implement linked widgets as either *client* or *server widgets*. The following two sub-sections provide technical details on each of those types.

3.1 Client widgets

Most of the widgets currently implemented are *client widgets*. They are executed on the client side, i.e. they use client memory and processor resources; data are collected and processed on-the-fly in the browser. The server hosting the *client widgets* is not necessarily the platform server, i.e. a mashup may combine widgets hosted on various servers. This makes the platform flexible, allowing external parties to host widgets on their own infrastructure and using their technological basis.

Each *client widget* consists of:

- a semantic model;
- an execution function;
- input and/or output terminals;
- 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 2); and
- its own 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:

(1) inject a JavaScript file[3] to facilitate cooperation with other widgets;

(2) define the input and/or output configuration; and

(3) 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, we collect output data from all relevant widgets to build the *data* object and pass it to the *run*.

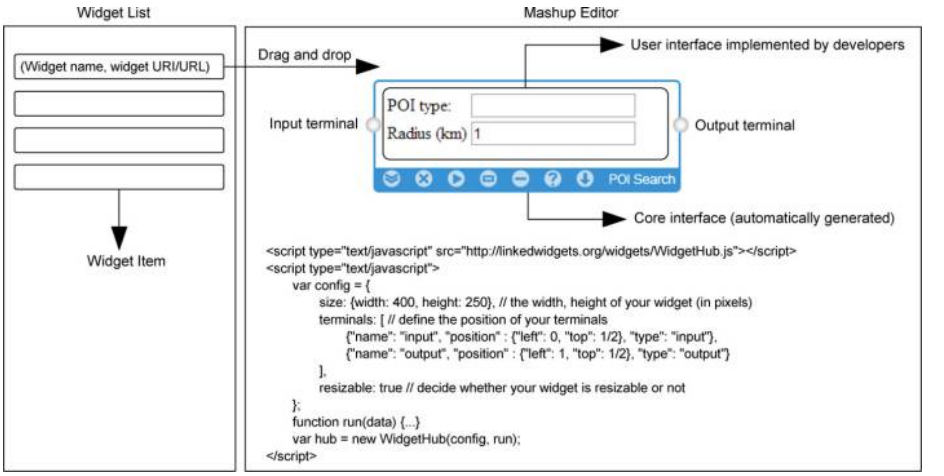

Figure 2.
Client widget components

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 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 URL; and later, to execute advanced functions, such as terminal matching and automatic mashup composition, we use the URI to load the widgets' model and other metadata. On the other hand, if the widget is still in the developing/testing stage, we simply 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 platform displays a complete widget interface as shown in Figure 2. 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.

3.2 Server widgets

Client widgets are easy to develop and necessary for a lightweight and scalable mashup platform. 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 in the web. Furthermore, it also cannot deal with 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, we designed *server widgets* to shift the execution function from the browser environment to standalone application environments. These server widgets consist of two main parts:

(1) a *user Web interface,* which is the same as for *client widgets*, but without the *run* function; and

(2) a *remote executor, is the client interface* 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. At the same time, we set up a connection channel between this *client interface* and the *remote executor* of the *server widget*. 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 4.

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 platform from previous approaches. *Server widgets* hence provide various benefits:

- They can act as a data connector to obtain and provide data on different services, devices, or systems for a mashup.
- They introduces collaborative use cases where each participant contributes *data* or *processing widgets* to a shared mashup.
- Because their computing tasks are performed within the hosting devices, similar to *client widgets*, they reduce the platform server load.
- They can run persistently in the background to collect or process data for data monitoring or data streaming applications.
- *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*; *visualization widgets* are always *client widgets*. 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, are ideal environments 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.

To create *server widgets*, developers first define the *client interface* in a similar manner to the *client widgets* implementation. Furthermore, they need to build the *remote executor* component. To this end, they download the widget library utilities for their preferred programming language, and implement an abstract method to define the widget job, i.e. to transform input data into output data. The widget cooperation protocol is already implemented and included in the utilities.

3.3 Lined widget model

Client- and server-linked widgets have different execution environments, but they share the same model. Due to similarities with Web services, we initially considered

describing linked widgets semantically using SAWSDL Kopecky *et al.* (2007), OWL-S[4] or WSMO[5]. These languages are well-suited for the formal specification of interfaces, such as our linked widgets' input and output terminals. They do not, however, allow establishing a well-defined semantic relation between input and output data. As a precondition for advanced widget exploration and automatic mashup composition algorithms, an explicit semantic link between input and output terminals is highly beneficial. To this end, several formal annotation methods using a graph-based model have been developed in the literature (Taheriyan *et al.*, 2012, 2013; Verborgh *et al.*, 2011). To specify the internal and external semantic model of linked widgets, we adopt and adapt the Karma model (Taheriyan *et al.*, 2013). In addition to the input and output specification, as well as their relation, the resulting linked widget model also includes provenance and license information. 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

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. To present relations between two arbitrary nodes in the input and output trees, we reuse the SWRL[6] vocabulary. We associate the input and output terminals with descriptions that represent their full tree structures in JSON.

Figure 3 illustrates the model of the *POI Search* widget that will be used in our local mashup example (Section 5.1). The widget takes an array of arbitrary objects containing the *wgs84:location* property as input. Its domain is the *Point* class with two literal properties, i.e. *lat* and *long*. The widget output is an array of GeoNames[7] features satisfying the distance filter specified in the *POI Search* widget.

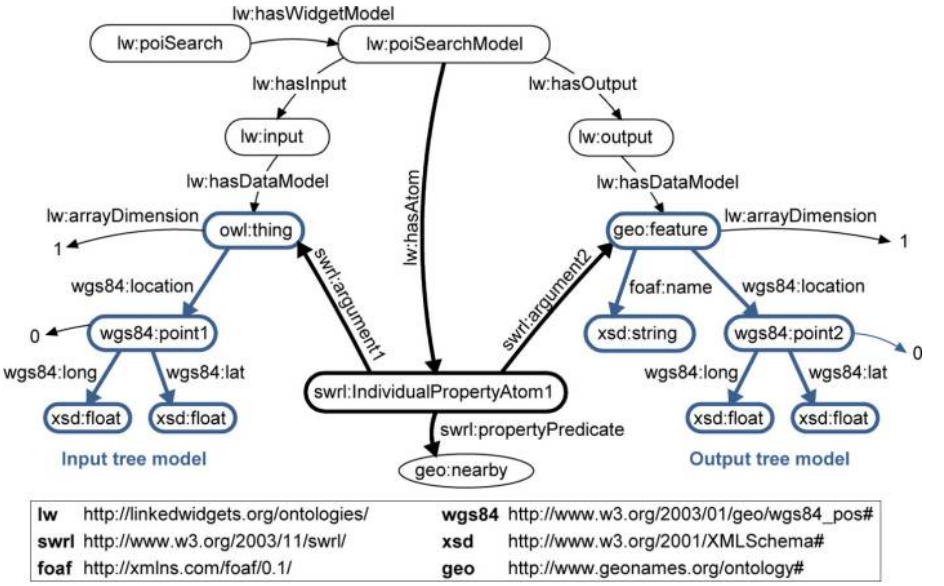

Figure 3.
Semantic model of the POI Search widget

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.

The *point*, *location*, *lat* and *long* terms are available in different vocabularies. However, due to its wide distribution we chose *wgs84*. The *widget annotator* module should 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* (Figure 3), 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.

4. Mashup communication protocol

This section discusses the communication protocol between widgets for the three types of mashups, i.e. *local*, *hybrid* and *distributed* mashups. The protocol is designed to facilitate efficient communication between independently developed widgets executed on various devices while minimizing platform server load.

4.1 Local protocol

Local mashups consist entirely of locally executed *client widgets* that communicate entirely within the client's Web browser. Technically, each *client widget* is an HTML iframe that can trigger events. Each event is associated with messages that are consumed by listeners registered by other iframes. As an example for how the protocol facilitates communication at runtime, consider a mashup with three widgets $A \rightarrow B \rightarrow C$. Typically, when a user triggers an action to run a widget, e.g. widget C, this action requires all preceding widgets to run first. Because widget C requires output from widget B, which in turn, requires output from widget A, widgets A and B need to run first. Figure 4 shows the messages transferred between the coordinator and the widgets. This involves the following types of messages:

- *requestToRun* messages inform the coordinator that a widget has to be executed in order to deliver output;
- *requestOutput* messages are passed from the coordinator to widgets to request their output; and

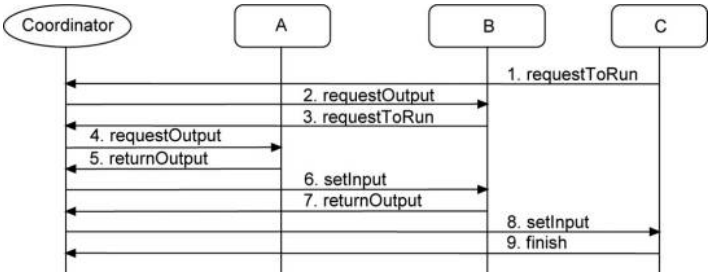

Figure 4.
Local mashup communication protocol

- *setInput* messages supply the input for a widget, and (iv) *returnOutput* messages submit a widget's output to the coordinator.

4.2 Remote protocol

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.

For the remote mashup communication protocol, we use the publish/sub-scribe model and a coordination server. To explain the protocol, consider a sample mashup with four widgets (cf. Figure 5):

(1) S_1, a server data widget, which runs on a personal computer to get data from a file.

(2) S_2, another server data widget, which runs on an Android phone to obtain its data, e.g. call logs.

(3) S_3, a server processing widget, which runs on a Web server.

(4) C_1, a client visualization widget.

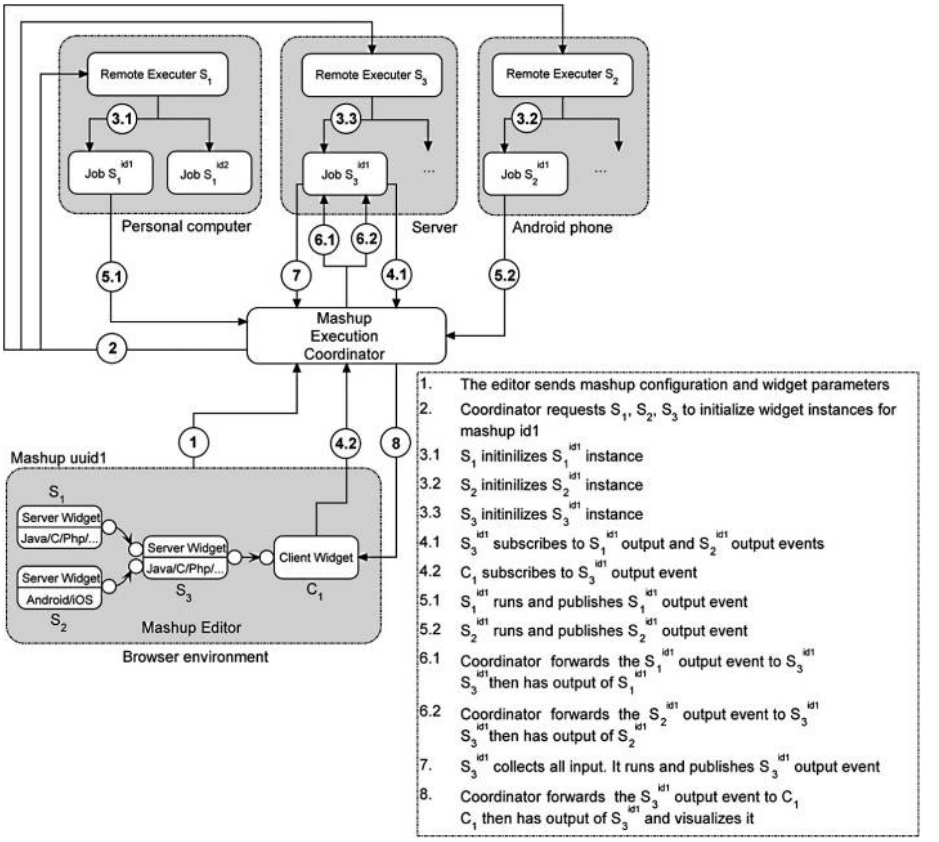

Figure 5.
Remote mashup communication protocol

To differentiate multiple instances of a server widget used in multiple mashups, we associate each mashup with a universally unique identifier (UUID), e.g. *id*1 in our example. As soon as a user triggers the execution of a mashup, the platform collects the parameters for all server widgets and forwards them to the coordinator (Step 1 in Figure 5). These parameters are:

- the list of URIs of widgets used;
- the configuration of the mashup, i.e. all connections between an output terminal of a widget and an input terminal of another widget; and
- 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 3.2. 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 *id*1. 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). 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 arrive.

4.3 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.

5. Sample data integration use cases

In the following, we illustrate the use of local and hybrid mashups by means of two simple example use cases.

5.1 Local mashups

The example depicted in Figure 6 illustrates how a mashup can inform everyday decision-making. All widgets used are *client widgets*. Our goal is to find suitable locations for running in a park under the condition that it should also possible to have a swim nearby. In particular, in this example, we want to find a park in Vienna that is near a swimming pool and provides good air quality and comfortable air temperature.

We start from a list of public swimming pools in Vienna – which can be retrieved from Vienna Open Government Data[8] – and use the *POI Search* widget to retrieve nearby parks via a SPARQL query to the Linked Geo Data server. At this point, we have all parks that satisfy the condition of being near a swimming pool. Next, we add weather forecasts from Wunderground[9], using the *Weather Forecast* and *Weather Condition* widgets. We specify our preferred time period in the *Weather Forecast* widget and select the weather conditions we prefer (e.g. temperature or air pressure) in the *Weather Condition* widget. For each park in the input, this widget obtains measurements of the nearest station (out of 30,000 Wunderground stations all over the world). We use the *Air Quality Filter* widget[10] to get parks with good air quality only. We then add Flickr images of the parks based on their location using the *Flickr Geo Image* widget.

Finally, we present the collected information in the *Map Viewer* visualization widget. This widget 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 cube http://purl.org/linked-data/cube#DataSet class, then the *Map Viewer* will automatically parse the data and visualize it in a chart. In the mashup, we display charts of the weather forecast for each park. The validity of all connections between widgets is enforced by the platform 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.

5.2 Hybrid mashups

For the second example, situated in an enterprise context, consider the need to integrate data from several Excel and Google spreadsheets. 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 repeated whenever the source data changes.

A mashup example that accomplishes such a task collaboratively is illustrated in Figure 7. 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:

(1) a point of sale (POS) spreadsheet that contains their respective ID, name, latitude, longitude, city, country; and

(2) 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 manager, who updates the data every day by adding new rows into the spreadsheet.

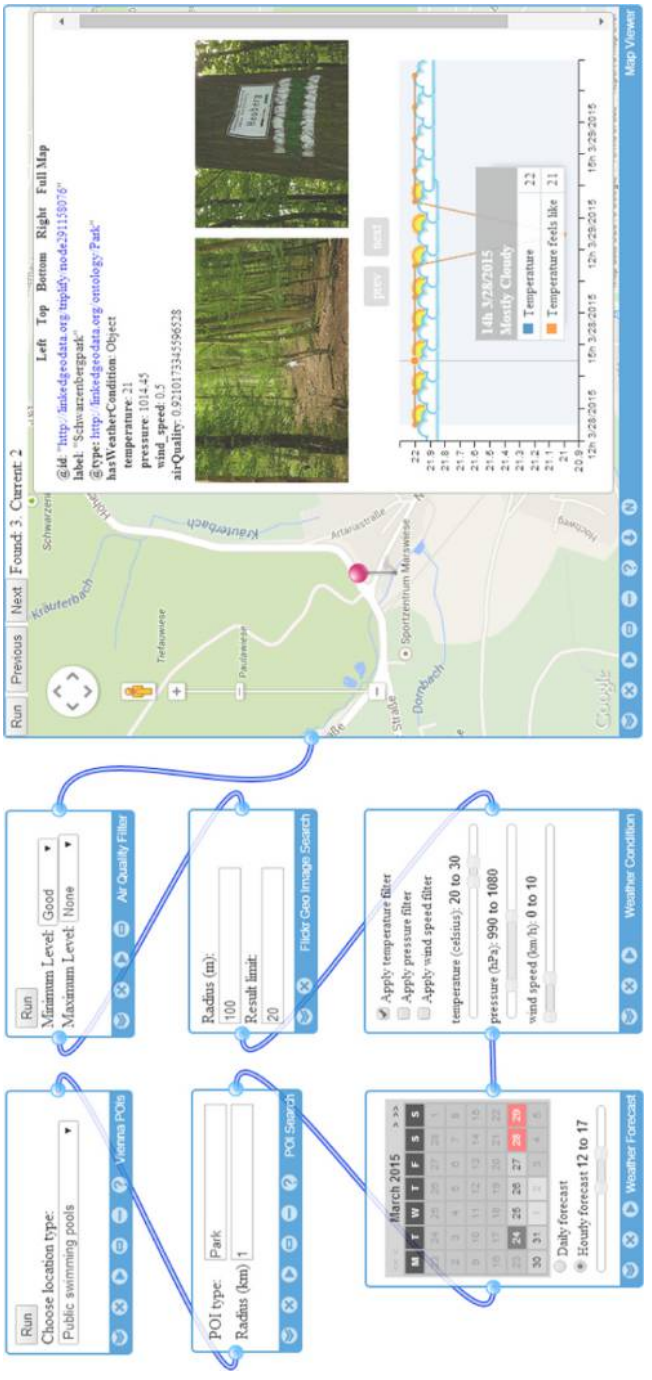

Figure 6.
Local mashup example for locating a park

Figure 7.
A collaborative mashup example for sales data integration

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[11]. Next, each branch manager contributes their respective *Excel Sheet* widget (which is a *server widget*) to load and convert his sale data into a cube dataset. To this end, they install 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 7 is the mashup as seen by point of sale manager *A*; he has to enter the token of his *Excel Sheet* server widget's deployment into his *client interface* of the widget. Meanwhile, the tokens of other shops are filled out by the respective branch managers. To prepare 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 7.

This is a *hybrid 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. POS can easily be added to or removed from the mashup.

6. Prototype implementation

6.1 Communication protocols

Local widget interaction: the client mashup coordinator is written in JavaScript and executed locally. This implies that platform 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 favors performance and scalability.

Remote widget interaction: we have identified two major architectural options for implementing the remote mashup protocol. The first is using Web services; the second is to implement it on top of a WebSocket Vanessa *et al.* (2013) infrastructure. We will discuss each of these options below.

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 bidirectional 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, are transferred between and processed inside the *remote executors* themselves. This reduces the server load because the coordinator just calls the *execution services*. 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 cannot deploy Web services on such devices. This would result in unidirectional communication channels. To simulate bidirectional connections and the publish/subscribe model, we would have to use polling or long-polling solutions. However, this approach has unfavorable scaling characteristics and generate 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. WebSockets provide full-duplex communication channels over a single TCP 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 set up the protocol.

Following the WebSocket approach, the *remote executors* are essentially WebSocket clients, and the coordinator is a WebSocket server. We use WAMP[12] to implement these components. 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 server widgets for ubiquitous computing environments.

A potential disadvantage of this architecture is that *remote executors* cannot directly transmit data to each other on their own. All data need to be passed through the WAMP server. The coordinator therefore becomes the performance bottleneck of the platform. To tackle this problem, we can deploy multiple WAMP servers and make use of load-balancing methods (Bourke and Server Load Balancing, 2001). Furthermore, server widgets that run on a Web server may return the token rather than the complete data to the coordinator, which can then forward the token 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:

- lower latency compared to traditional HTTP connections;
- lower amount of data transferred; and
- the wide range of supported languages, which provide the basis for ubiquitous computing environments for server widgets.

6.2 Collaborative mashup editor

The mashup editor is a key element of the platform; 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.

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 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 might 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 are no longer shared.

6.3 Automatic data exploration and integration facilities

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 at 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. Conditions that have to be satisfied for the terminals are:

- matching class and array dimension; and
- matching attributes, i.e. the set of attributes required by the input terminal must be a subset of the attributes provided by the output terminal.

In the example presented in Figure 6, for instance, we can thereby determine that we can 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 Wunderground.

A mashup essentially integrates 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* algorithm, we know exactly which widgets can be connected. Based on that, we have designed a graph search algorithm to determine all possible ways to integrate data based on a specified list of data sources and related available linked widgets. The *automatic mashup composition* module is included in the editor.

6.4 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 8 is an example that illustrates the definition of a semantic model for the *POI Search* widget (Figure 3). 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.

Finally, as soon as they have deployed their widgets, developers can submit their work to the platform 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[13].

6.5 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

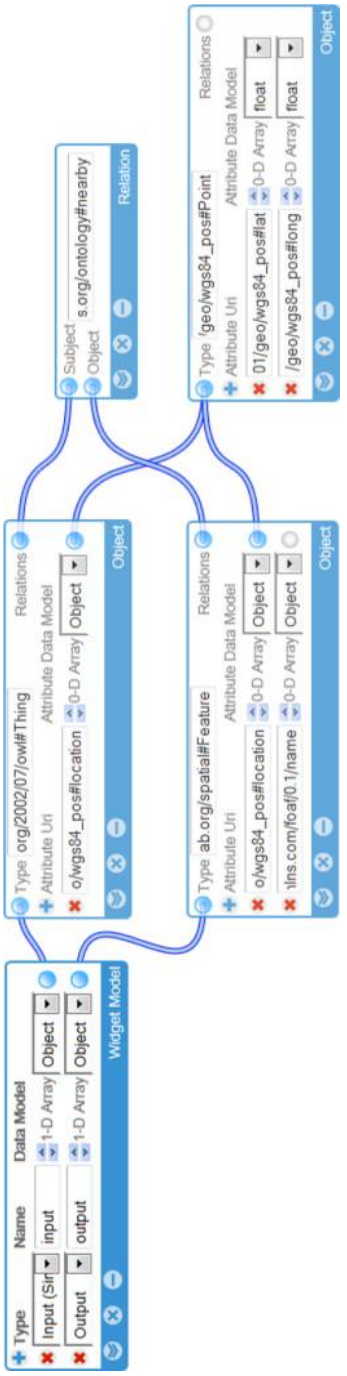

Figure 8.
Visual model defined for the POI Search widget

platform, 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 9. Users can use short names for classes and properties in their model. We also provide *tolerant search*. Tolerant search uses ontology alignment methods Euzenat and Shvaiko (2013) to return widgets that have models similar to their specification.

7. Related work

Researchers have been developing mashup-based tools for years. Many of them are geared toward end-users and aim to allow them to efficiently create applications by connecting simple and lightweight entities.

Engard (2009) introduces definitions, summaries and practical uses of mashup. Aghaee and Pautasso (2012a) provide a detailed overview of mashup approaches. They discuss open research challenges which we partly address with our platform. For instance, we address the *Simplicity* and *Expressive Power Tradeoff* challenges through a semantic model. They also evaluate Yahoo! Pipes Pruett (2007), IBM Mashup Center[14], Presto Cloud[15] and ServFace (Nestler *et al.*, 2010). A common limitation they identify for all these platforms is that the wiring paradigm is hard to grasp for non-expert end users. We aim to overcome this barrier by recommending valid wiring options to the user.

Other surveys in mashup literature (Aghaee and Pautasso, 2012b) have developed a number of evaluation criteria and identified shortcomings of existing approaches. Among these shortcomings are lacks of support for:

- event-based behavior;
- component discovery features; and
- language-dependent mashup components.

Computer scientists addressed some of these shortcomings in more recent contributions, but some still remain an open challenge.

Grammel and Storey (2010) review six different approaches and identify potential areas of improvement and future research. For instance, they argue 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. This feature is also provided by the platform presented in this paper. However, detecting invalid mashups still remains a challenge that requires appropriate debugging mechanisms for non-programmers.

Di Lorenzo *et al.* (2009) analyzed the strengths and weaknesses of popular mashup tools, i.e. Damia (Simmen *et al.*, 2008), Yahoo! Pipes, Microsoft Popfly (Griffin, 2008), Google Mashup Editor (Tony, 2008), Exhibit (Huynh *et al.*, 2007), Apatar[16], MashMaker (Ennals and Garofalakis, 2007), with respect to data integration. All of them are server side applications, meaning that mashups and the data involved both are

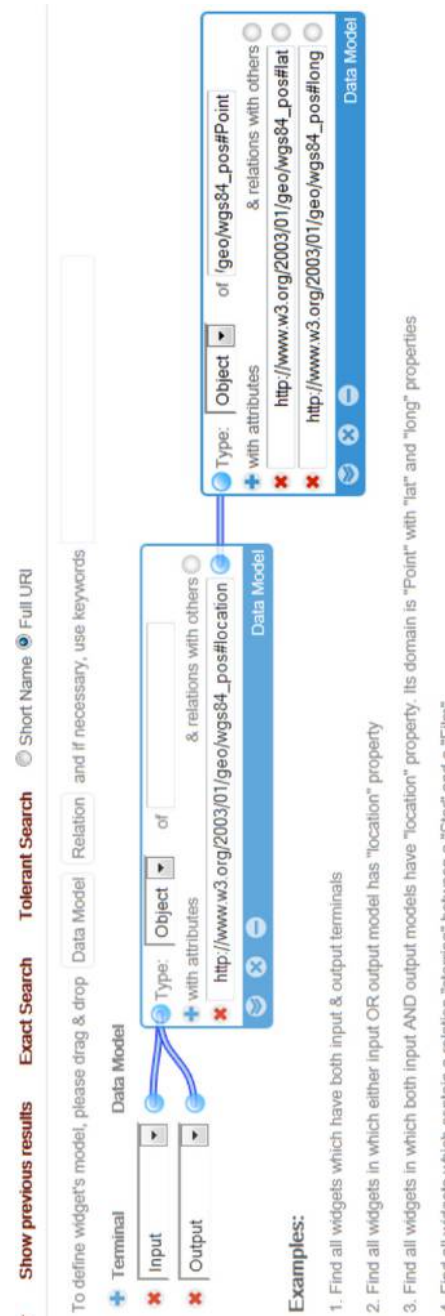

Figure 9.
A widget search that defines the input and output semantic models

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 they are supposed to target "non-expert" users.

Daniel *et al.* (2010) discuss the concept of *process mashups* by introducing three dimensions, i.e. multi-user support, multi-page navigation and workflow support. From that, they classify mashups into eight classes, based on different combinations of these dimensions. They are:

(1) *simple mashups*, e.g. mashArt (Daniel *et al.,* 2009), Yahoo! Pipes, MashMaker;

(2) *multi-page mashups*, e.g. EzWeb (Lizcano *et al*., 2008);

(3) *guided mashups* – no tool exists;

(4) *page ow mashups*, e.g. ServFace Builder (Nestler *et al.,* 2010);

(5) *shared page mashups* – no tool exists;

(6) *shared space mashups*, e.g. IBM Mashup Center;

(7) *cooperative mashups*, e.g. Gravity[17]; and

(8) *process mashups*, e.g. MarcoFlow (Daniel *et al.,*2010).

According to their classification, the linked widgets platform is a *process mashups* tool.

Blichmann *et al.* (2013) present their vision of collaborative mashups by specifying three challenges. They are:

(1) to develop a mechanism for unified handling of (non) collaborative components;

(2) to synchronize differently implemented components with identical functionality; and

(3) to support fine-grained sharing of mashup composition parts.

They then present their preliminary solutions with their CRUISe platform.

PEUDOM (Picozzi, 2013) claims to be a platform for multiple devices and collaborative mashups. It allows users to create components on top of REST services and to combine these components to build mashups. Similar to the linked widgets platform, it supports 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.

Salminen and Mikkonen (2013) 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. This is impossible in PEUDEM.

Super Stream Collider (SSC) (Hoan *et al.,* 2012), MashQL (Jarrar and Dikaiakos, 2008) and DERI Pipes (Le-Phuoc *et al.,* 2009) are three platforms aimed at semantic data processing. Whereas SSC consumes live stream data only, MashQL allows users to easily create a SPARQL query, using 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. There are multiple other platforms which we only want to point at, such as Vegemite (Lin *et al.,* 2009), Paggr (Nowack, 2009) or Marmite (Wong and Hong, 2007). They all follow a mashup-based approach to ease users' access to data sources, but unlike our approach, they do not make use of semantic models.

To sum up, the most apparent differences between the linked widget platform and similar approaches are as follows:

- We present a high-level and problem-oriented data processing platform. 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.
- We allow and encourage developers to contribute their widgets to the platform to extend the number of data sources it can process.
- 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 platform to deal with the issue of data heterogeneity.
- We provide an environment for interactive collaborative mashup creation and analysis.
- We introduce the concept of *distributed mashups*.

8. Conclusion and future work

In this paper, we present a collaborative mashup platform for dynamic integration of heterogeneous data sources. The platform is designed based on the principle of openness; it is a hub that connects developers, data publishers, data integrators and end users.

To foster reusability and creativity, we modularize functionality into linked widgets that users can recombine to create new applications. We illustrate the value of this approach by means of sample use cases that integrate data from various sources. We also design an architecture that allows a group of people to collaboratively build such *ad hoc* applications. We make use of both client and server computing resources to create a powerful, extensible and scalable data integration platform. 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. Semantic annotation of linked widgets and the utilization of their metadata allows users to explore widgets and locate relevant data sources. It is also possible to automatically suggest all possible widget combinations based on a list of available widgets.

A prototype Web-based mashup editor[18] 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.

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 platform assigns each resource of the output data a dereferencable URI that can be looked up by people and user agents.

Future research will also aim to improve our automatic mashup composition algorithm. 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 platform as plugins.

Notes

1. www.w3.org/TR/widgets/ (accessed 10 April 2015).
2. www.w3.org/TR/json-ld/ (accessed 10 April 2015).
3. http://linkedwidgets.org/widgets/WidgetHub.js (accessed 10 April 2015).
4. www.w3.org/Submission/OWL-S/ (accessed 10 April 2015).
5. www.w3.org/Submission/WSMO/ (accessed 10 April 2015).
6. www.w3.org/Submission/SWRL/ (accessed 10 April 2015).
7. www.geonames.org/ (accessed 10 April 2015).
8. https://open.wien.gv.at/site/open-data/ (accessed 10 April 2015).
9. www.wunderground.com/ (accessed 10 April 2015).
10. www.airqualitynow.eu/index.php (accessed 10 April 2015).
11. www.w3.org/TR/vocab-data-cube/ (accessed 10 April 2015).
12. http://wamp.ws/ (accessed 10 April 2015).
13. http://ogd.ifs.tuwien.ac.at/sparql (accessed 10 April 2015).
14. http://pic.dhe.ibm.com/infocenter/mashhelp/v3/index.jsp (accessed 10 April 2015).
15. http://mdc.jackbe.com/enterprise-mashup (accessed 10 April 2015).
16. www.apatar.com/ (accessed 10 April 2015).
17. www.sdn.sap.com/irj/scn/weblogs?blog=/pub/wlg/17826 (accessed 10 April 2015).
18. http://linkedwidgets.org (accessed 10 April 2015).

References

Aghaee, S. and Pautasso, C. (2012), "End-user programming for web mashups: open research challenges", *ICWE '11 Proceedings of the 11th International Conference on Current Trends in Web Engineering, Springer Berlin Heidelberg*, pp. 347-351, available at: http://dx.doi.org/10.1007/978-3-642-27997-3_38

Aghaee, S. and Pautasso, C. (2012), "An evaluation of mashup tools based on support for heterogeneous mashup components", *ICWE '11 Proceedings of the 11th International Conference on Current Trends in Web Engineering, Springer Berlin Heidelberg*, pp. 1-12, available at: http://dx.doi.org/10.1007/978-3-642-27997-3_1

Bizer, C., Heath, T. and Berners-Lee, T. (2009), "Linked data the story so far", *International Journal on Semantic Web*, Vol. 5 No. 3, pp. 1-22.

Blichmann, G., Radeck, C. and Mei\s sner, K. (2013), "Enabling end users to build situational collaborative Mashups at runtime", *ICIW The Eighth International Conference on Internet and Web Applications and Services*, pp. 120-123, available at: www.thinkmind.org/index.php?view=article&articleid=iciw_2013_5_40_20132

Bourke, T. and Server Load Balancing (2001), "Help for network administrators", O'Reilly Media, Incorporated, available at: http://books.google.at/books?id=l9uD3smAC

Daniel, F., Casati, F., Benatallah, B. and Shan, M.C. (2009), "Hosted universal composition: models, languages and infrastructure in mashart", *Conceptual Modeling- ER, Springer*, pp. 428-443, available at: http://link.springer.com/chapter/10.1007/978-3-642-04840-1_32

Daniel, F., Koschmider, A., Nestler, T., Roy, M. and Namoun, A. (2010), "Toward process mashups: key ingredients and open research challenges", *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups, ACM*, p. 9, available at: http://dl.acm.org/citation.cfm?id=1945008

Daniel, F., Soi, S., Tranquillini, S., Casati, F., Heng, C. and Yan, L. (2010), "From People to services to UI: distributed orchestration of user interfaces", in Hull, R., Mendling, J. and Tai, S. (Eds), *Business Process Management, Lecture Notes in Computer Science*, Springer Berlin Heidelberg, Vol. 6336, pp. 310-326, available at: http://dx.doi.org/10.1007/978-3-642-15618-2_22

Di Lorenzo, G., Hacid, H., Paik, H.Y. and Benatallah, B. (2009), "Data integration in mashups", *SIGMOD Record*, Vol. 38 No. 1, pp. 59-66, available at: http://doi.acm.org/10.1145/1558334.1558343

Engard, NC. (2009), "Library mashups: exploring new ways to deliver library data", *Information Today*, Medford.

Ennals, R.J. and Garofalakis, M.N. (2007), "Mashmaker: mashups for the masses", *SIGMOD '07 Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY*, pp. 1116-1118, available at: http://doi.acm.org/10.1145/1247480.1247626

Euzenat, J. and Shvaiko, P. (2013), "Ontology matching", 2nd ed., Springer-Verlag Berlin Heidelberg, Heidelberg.

Grammel, L. and Storey, M.A. (2010), "A survey of mashup development environments", in Chignell, M., Cordy, J., Ng, J. and Yesha, Y. (Eds), *The Smart Internet, Lecture Notes in Computer Science*, Springer Berlin Heidelberg, Vol. 6400, pp. 137-151, available at: http://dx.doi.org/10.1007/978-3-642-16599-3_10

Griffin, E. (2008), "Foundations of Popfly: rapid mashup development", *Books for Professionals by Professionals*, Apress, available at: https://books.google.at/books?id=3Hd5htUzY9gC

Hoan, N.M.Q., Martin Serrano, D.L.P. and Hauswirth, M. (2012), "Super stream collider-linked stream mashups for everyone", *Proceedings of the Semantic Web Challenge co-located with ISWC, Boston, MA*.

Huynh, D.F., Karger, D.R. and Miller, R.C. (2007), "Exhibit: lightweight structured data publishing", *WWW '07 Proceedings of the 16th International Conference on World Wide Web, ACM, New York, NY*, pp. 737-746, available at: http://doi. acm.org/10.1145/1242572.1242672

Jarrar, M. and Dikaiakos, M.D. (2008), "Mashql: a query-by-diagram topping sparql", *ONISW '08 Proceedings of the 2nd International Workshop on Ontologies and Information Systems for the Semantic Web, ACM, New York, NY*, pp. 89-96, available at: http://doi.acm.org/10.1145/1458484.1458499

Kopecky, J., Vitvar, T., Bournez, C. and Farrell, J. (2007), "Sawsdl: semantic annotations for WSDL and XML schema", Internet Computing, *IEEE*, Vol. 11 No. 6, pp. 60-67, available at: http://dx.doi.org/10.1109/MIC.2007.134

Le-Phuoc, D., Polleres, A., Hauswirth, M., Tummarello, G. and Morbidoni, C. (2009), "Rapid prototyping of semantic mashups through semantic web pipes", *WWW '09 Proceedings of the 18th International Conference on World Wide Web, ACM, New York, NY*, pp. 581-590, available at: http://doi.acm.org/10.1145/1526709.1526788

Lin, J., Wong, J., Nichols, J., Cypher, A. and Lau, T.A. (2009), "End-user programming of mashups with vegemite", *IUI '09 Proceedings of the 14th International Conference on Intelligent User Interfaces, ACM, New York, NY*, pp. 97-106, available at: http://doi.acm.org/10.1145/1502650.1502667

Lizcano, D., Soriano, J., Reyes, M. and Hierro, J. (2008), "EzWeb/FAST: reporting on a Successful mashup-based solution for developing and deploying composite applications in the upcoming, 'Ubiquitous SOA'", *Mobile Ubiquitous Computing, Systems, Services and Technologies*, *The Second International Conference on UBICOMM '08*, Valencia, pp. 488-495.

Nardi, B.A. (1993), *A Small Matter of Programming: Perspectives on End User Computing*, MIT Press, Cambridge, MA.

Nestler, T., Feldmann, M., Hbsch, G., Preuner, A. and Jugel, U. (2010), "The servface builder - a wysiwyg approach for building service-based applications", in Benatallah, B., Casati, F., Kappel, G. and Rossi, G. (Eds), Web Engineering, Lecture Notes in Computer Science, Springer Berlin Heidelberg, Vol. 6189, pp. 498-501, available at: http://dx.doi.org/10.1007/978-3-642-13911-6_37

Nestler, T., Feldmann, M., Hübsch, G., Preuβner, A. and Jugel, U. (2010), "The servface builder - a wysiwyg approach for building service-based applications", *ICWE '10 Proceedings of the 10th International Conference on Web Engineering, Springer-Verlag, Heidelberg*, pp. 498-501, available at: http://dl.acm.org/citation.cfm?id=1884110.1884155

Nowack, B. (2009), "Paggr: linked data widgets and dashboards", *Web Semant*, Vol. 7 No. 4, pp. 272-277, available at: http://dx.doi.org/10.1016/j.websem.2009.09.005

Picozzi, M. (2013), "End user development of multidevice and collaborative mashups", CHItaly (Doctoral Consortium), Citeseer, pp. 55-65.

Pruett, M. (2007), *Yahoo! Pipes*, 1st edn., O'Reilly Media.

Salminen, A. and Mikkonen, T. (2013), "Towards pervasive mashups in embedded devices: comparing procedural and declarative approach", *IJCNDS*, Vol. 10 No. 3, pp. 195-215, available at: http://dx.doi.org/10.1504/IJCNDS.2013.053077

Simmen, D.E., Altinel, M., Markl, V., Padmanabhan, S. and Singh, A. (2008), "Damia: data mashups for intranet applications", *SIGMOD '08 Proceedings of the 2008 ACM SIG-MOD International Conference on Management of Data, ACM, New York, NY*, pp. 1171-1182, available at: http://doi.acm.org/10.1145/1376616.1376734

Taheriyan, M., Knoblock, C.A., Szekely, P. and Ambite, J.L. (2012), "Rapidly integrating services into the linked data cloud", *Proceedings of the 11th International Conference on The Semantic Web - Volume Part I, Lecture Notes in Computer Science, Springer Berlin Heidelberg*, pp. 559-574.

Taheriyan, M., Knoblock, C., Szekely, P. and Ambite, J. (2013), "A graph-based approach to learn semantic descriptions of data sources", *The Semantic Web ISWC*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, Vol. 8218, pp. 607-623.

Tony, L. (2008), "Creating Google Mashups with the Google Mashup Editor", Lotontech Limited, available at: https://books.google.at/books?id=6z_BLopvZ70C

Trinh, T.D., Wetz, P., Do, B.L., Anjomshoaa, A., Kiesling, E. and Tjoa, A.M. (2014), "A web-based platform for dynamic integration of heterogeneous data", *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services, Hanoi, 4-6 December*, pp. 253-261, available at: http://doi.acm.org/10.1145/2684200.2684291

Vanessa, W., Peter, M. and Frank, S. (2013), *The Definitive Guide to HTML5 WebSocket*, Apress, New York, NY.

Verborgh, R., Steiner, T., Van Deursen, D., Van de Walle, R. and Gabarró Vallés, J. (2011), "Efficient runtime service discovery and consumption with hyperlinked RESTdesc", *Proceedings of the 7th International Conference on Next Generation Web Services Practices*, *Salamanca*, pp. 373-379.

Wong, J. and Hong, J.I. (2007), "Making mashups with marmite: Towards end-user programming for the web", *CHI '07 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, New York, NY*, pp. 1435-1444, available at: http://doi.acm.org/10.1145/1240624.1240842

Corresponding author
Tuan-Dat Trinh can be contacted at: tuan.trinh@tuwien.ac.at