An Environment for End-User Development of Web Mashups[1]

GIUSEPPE GHIANI[2], FABIO PATERNÒ[2], LUCIO DAVIDE SPANO[3], GIULIANO PINTORI[2],

ABSTRACT

End-User Development aims to find novel ways that are suitable and intuitive for end users to create their own applications. We present a graphical environment in which users create new mashups by directly selecting interaction elements, content and functionalities from existing Web applications without requiring the intervention of expert developers. Then, users just need to exploit a copy-paste metaphor to indicate how to compose the selected interactive content and functionalities in the new mashup. The environment is enabled by a Web-based platform accessible from any browser, and is suitable for users without particular programming skills. We describe the architecture of our platform and how it works, including its intelligent support, show example applications, and report the results of first user studies.

Keywords: End User Development; Web Mashups; User Interface Development Tools

1 INTRODUCTION

In recent years, the penetration of Internet applications in all work and leisure activities has made it possible for people to use computers in an increasing number of possible contexts, and for an increasingly wide range of tasks. However, end users have a wide variety of interests and requirements, and existing applications often do not support directly the wide dynamic set of tasks they want to accomplish. In order to fill this gap, it is useful to provide environments that allow them to obtain applications that better fit their needs. Indeed, even if they are not professional developers, they are becoming more and more familiar with software technologies and this sets the ground for creating tools for crafting their own solutions. End-User Development (EUD) is a recent discipline that refers to the approaches allowing people without experience in programming to develop their applications, or at least modify them, in order to better support their specific tasks (Lieberman et al., 2006). In this area various research works has started to investigate novel solutions that are suitable and pleasant for end users to create new applications. In (Ko et al.,

[1] Contact Information: Fabio Paternò, CNR-ISTI, tel. +390503153066, fax +390503152810, fabio.paterno@isti.cnr.it
[2] CNR-ISTI
[3] University of Cagliari

2011) several approaches to end-user software engineering have been reviewed, but the authors dedicated limited attention to emerging EUD approaches in Web environments, which have started to be addressed by recent work (Cypher et al., 2010) and are discussed in the related work section.

In this paper we focus on the Web, since it allows easy access to large amounts of data and applications (e.g. e-commerce sites, social networks, e-mail, etc.). In addition, the Web is a flexible platform that, differently from native applications, implicitly allows customization. Web pages are defined in HTML, and are represented in the browser by a *Document Object Model (DOM)*. The DOM is inspectable through the browser, from where it can be manipulated in order to modify the content as well as the behaviour of the page (i.e. make the page react in a different way when some events occur). It is worth noting that, differently from native applications, such customizations can be made directly by the user, without involving the original page developers.

We moreover focus on end user developers aiming to create new applications starting with components of existing interactive Web applications, i.e. a mashup editor. From the Human-Computer Interaction perspective, mashup refers to a composition of contents and/or features from several sources that determine new client-side interactive applications. In general, Web mashups can combine data, presentations and functionalities from different Web sites into a single, novel, Web application. For example, most of the first mashups were created to combine geographical maps or to better manage photos. Usually, Mashup applications are created by developers exploiting Web APIs or programmatically gathering content from existing Web pages. Therefore, creating mashups has required some technical background (such as programming skills) that most Web end users do not have. Some initial ideas for creating Web mashups that we consider in this work were introduced in (Ghiani et al., 2011). However, that solution still required manual intervention by people with good technical knowledge in order to create connections for enabling communication among the components of different Web applications, it was limited in terms of components types that could be combined, and did not provide any support for sharing mashups.

In this paper, we present the MashupEditor, a novel environment based on an intelligent environment for end-user development of Web mashups. The MashupEditor's main goals are: to allow end users to create Web mashups by reusing existing components from different applications, regardless of their technical skills; to create novel ways to combine such components; to support sharing of the results' compositions with other people, possibly via social networks. The environment does not require knowledge of JavaScript, which is often the most problematic part in Web applications development for non-professional end users. In the composition process, end users exploit an intuitive copy-paste metaphor, which has been inspired by the *programming by example* paradigm. In particular, copy-paste examples provided by the users are used to infer how to compose the components of the existing Web applications.

More precisely, the main contributions of the proposed environment are:

- An editor to create new mashup widgets from components of existing Web applications simply using a Web browser, without specific extensions, and a proxy-server that includes scripts that allow users to select the desired components by direct manipulation;
- A method and a supporting tool for composing Web components from different applications through an intuitive and familiar copy-paste metaphor for creating novel Web applications;
- A solution for preserving users' selected preferences regarding the application output when new queries to remote services are submitted.

In the paper, after discussing related work, we describe an example application of our environment. We detail the underlying architecture and functionalities of the environment in order to explain how the intelligent support is provided to users. Then, we report on evaluation of the MashupEditor functionalities, which has been carried out through two user tests. Lastly, we draw some conclusions and provide indications for further work.

2 RELATED WORK

In this section, we discuss various contributions relevant to the approach for the creation of Web mashup applications that is presented in this paper. We briefly discuss a representative set of

various types of mashup environments for Web applications, already proposed by academic and industrial research groups. We then discuss some of them focusing on criteria relevant from the perspective of end user composition and development.

2.1 Approaches to Mashup Applications

The term mashup has been used in both research and industrial settings for defining a broad set of environments able to create new applications by composing information from different sources. Generally speaking, different types of mashups are possible depending on the aspects that they are able to compose (data, functionalities, user interfaces). Such an approach is better supported by the Web architecture, and hence has found its main applications in Web settings.

A first distinction among mashups can be made according to their targets: enterprises and consumers. Enterprise mashups are tools for combining resources, applications and heterogeneous data from various sources in order to solve enterprise-related problems. For example, EzWeb Enterprise Mashup (Soriano et al., 2007) is a mashup platform developed by Telefónica[4] targeted to enterprises. A catalogue is available on the platform containing a set of predefined gadgets, which can be composed through a piping metaphor defining the execution flow. It is possible to have several pipes running at the same time. Other examples of such kind of mashup tools are IBM Damia and SAP Research Rooftop (Hoyer et al., 2009). Consumer mashups, aimed at Internet users, exemplify the capabilities of Web 2.0 by combining diverse kinds of data from several public sources into information that is then displayed on a Web browser. One example in this area is Yahoo! Pipes[5], a Web environment to make mashups that exploits data from sources such as RSS feeds or Web services. The created mashups can be saved and made publicly available. Both enterprise and consumer mashups are devoted to providing quick solutions to narrow scope problems. In this work, we aim to provide an environment for consumer mashups, exploiting existing Web applications as the information source, rather than providing a set of predefined gadgets created by professional developers. In addition, we avoid the usage of

[4] http:// www.telefonica.com
[5] http://pipes.yahoo.com/pipes/

programming languages (based e.g. on imperative or data-flow constructs). In order to define the mashup's behaviour, we utilize an intelligent backend for tracking the user's actions and rely on well-known UI interactions (such as copy-and-paste).

According to their architecture, mashup systems can also be classified as client-based or server-based. The client-based ones rely on a Web browser to combine and show data, while server-based mashup systems instead perform analyses and combinations within a Web server, and subsequently forward data to the Web browser for visualization. In this respect, we have a mixed approach, since with our tool the composition is performed on the client side, and it also exploits functionalities provided by a proxy-server.

In general, interest in how to support EUD through mashups has recently increased. Some authors (Soi, Daniel, and Casati, 2014) have even considered adopting a domain specific approach, which prioritises intuitiveness over expressive power, even if they mention that developing mashup platforms—domain-specific or not—is complex and time consuming. In the following we discuss some work carried out in this area and indicate how we contribute to the state of art.

2.2 Page Customization

Different tools allow the user to change the layout and/or the behaviour of existing Web pages in order to better tailor them to different needs, with different relevant techniques.

A simple but effective way to customize a Web page is to provide an entry point (e.g. a browser extension) for injecting JavaScript code into it. For instance, Greasemonkey[6] is an extension for the Mozilla Firefox browser that allows choosing from among a set of custom scripts when a Web page is accessed. The tool not only lets the user automatically add new contents and combine them with data from other pages, but also hides texts or images (e.g. unwanted advertisements), adds shortcuts to external pages, fills in forms and compares data from several Web sites (e.g. prices from online stores). In order to exploit Greasemonkey, users typically write JavaScript code for

[6] https://addons.mozilla.org/it/firefox/addon/greasemonkey

accessing and modifying the page Document Object Model (DOM). Scripts are stored in text files that include some meta-information defining author, version and the set of suitable Web sites.

An enhanced strategy for managing Greasemonkey scripts that refer to pages whose layout is updated often was presented in (Díaz et al., 2010). The main drawback to such an approach is that it requires knowledge of HTML and JavaScript, thus preventing most potential users from exploiting it. Furthermore, its usage can lead to some problems, e.g. layout variations of a page might turn the associated script into a useless or even break the page functionalities. In our work, we combine intelligent support with the copy-paste metaphor in order to allow end users that are not able to write code to create new applications from existing Web components by indicating how they should be composed. An approach exploiting crowdsourcing techniques for customizing Web applications is reported in (Nebeling et al., 2012). Its targets range from developers to end users, and the platform is devoted to facilitating the development of Web interfaces. This is done by combining components from several Web sites, which are refined through users' contributions. For each component, the environment shows the solutions developed by various users, so that the developer can choose the preferred one. We support the sharing of mashups created by end users through social networks as well, in order to enhance user involvement and motivation and facilitate sharing of their results with friends and colleagues.

2.3 Task Automation

Different contributions in the literature provide the support for automatically performing a set of actions on a given Web page, e.g. for speeding-up tasks that are repeated over time. Chickenfoot (Miller et al., 2010) is another Mozilla Firefox extension aiming to provide a set of high level commands, such as "click", "enter", "pick", "keypress", "go" that can be performed automatically. Users can then create custom sequences of automatic actions, and even automatically access pages external to the visited ones, thus creating information mashups. Even if the statements exploited in this approach have a higher level of abstraction with respect to e.g. JavaScript, this approach still envisions some programming effort by the user. We aim to avoid this by providing an intuitive metaphor to create new applications.

CoScripter (Leshed et al., 2008) allows the user to define the actions to automate by recording the user's interaction. A set of heuristics is used for generating scripts from user interactions. The scripts are understandable, as they are written in terms of pseudo-natural-language instructions. It is possible to share such interaction sequences with other users through a wiki allowing access to a repository of scripts. Novices can thus start by selecting a previously created script from the repository and installing it in the CoScripter sidebar. We share with this approach the idea of automatically deriving the definition of the mashup behavior by recording the user's actions when performing copy-paste based interactions. However, we avoid the code-like presentation of the final results for sharing and debugging, and support composition among components of various types of Web applications.

Vegemite (Lin et al., 2009) extends CoScripter by introducing a spreadsheet-oriented environment that lets users organize data into the so-called VegeTables. The user can explicitly provide spreadsheet content or may instruct the system to automatically extract it from a Web page (e.g. a search results page). VegeTables content can be used by Vegemite to fill in forms and query Web applications. Query results are managed and can be in turn merged with existing VegeTables. Vegemite is released as a Mozilla browser extension. This system shares with our approach the user action recording metaphor, and the possibility to reproduce them. Differently from the MashupEditor, the data is scraped and collected in a separate spreadsheet, where the user performs the manipulation actions. Instead, our tool keeps the original representation of the Web components, without introducing additional panels. In addition, Vegemite has been designed in order to support ad hoc mashups: data aggregations that are used once for a specific purpose and then they are thrown away. The environment that we propose allows delivering an end-user crafted application, which can be directly used by other people, without the need of analysing how it was created.

Another technique that has been investigated for supporting the end-user development is the Programming by example (also known as programming by demonstration), which lets the system

be “instructed” by the user performing examples of tasks she wants to automate. Such a strategy is twofold, as it relies on recognition (i.e. interpreting user actions) as well as on generalization (i.e. abstracting action semantics in order to reapply them to different situations). (Faaborg et al., 2006) have implemented Creo/Miro, a prototype that relies on the programming by example paradigm by “recording” the user actions in an example interaction, a technique also used in this work. The recorded actions are then generalized: Creo replays them in different situations while Miro highlights the most relevant parts of the output (i.e. it enhances interesting terms with hyperlinks). Our approach differs from it since Creo/Miro exploits the recorded actions to obtain a generalized procedure for executing a set of user actions on Web sites different from the one used as example. Instead, we use the recorded actions to identify the particular set of Web site components to connect in order to provide support for a precise task in a particular context, which is useful for the user.

NaturalMash (Aghaee and Pautasso, 2014) is also a Web-based environment that allows non-programmers to exploit existing Web resources by combining their input/output. The resulting artefact is a novel application that can provide the user in real time with output in a customized way. For instance, a single user input can be used to query a search engine and then the resulting items can be used at the same time as input for an automatic search through another service. Although some of the use cases are similar to the ones we consider, there are several differences in the way the use cases are supported. NaturalMash users start defining a mashup by picking ingredients from a toolbar that includes services/contents available through Web APIs. Our tool has a broader scope and allows users to potentially import existing functionalities from any standard Web site. This is enabled in our MashupEditor by a procedure that monitors how the users interact with online functionalities such as search engines, and what output parts they select. In NaturalMash, users specify how to bind components together through a natural language subset. However, the mashup components associated to textual expressions are predefined and require preprocessing by expert programmers. PEUDOM (Matera et al., 2013) is another Web-based platform that allows end users to compose components associated with registered Web services

into a mashup. Components are defined by professional developers, and can subsequently be connected by means of drag-and-drop actions and by selecting the binding properties from some dropdown menus. We aim to provide a solution that does not require the preparation of the possible components by developers and can immediately support creation of mashups components through the use of a proxy server that includes scripts for this purpose in a transparent way for end users.

Similar differences can also be found with solutions that involve some choreographic approach, which are able to automatically compose elements based on their messaging capabilities. Such approaches require that the involved widgets and their communication capabilities be designed and implemented according to specific rules in order to enable their ability to compose themselves autonomously. One issue in this case is that users may have difficulties understanding the resulting mashup behaviour. (Tschudnowsky et al., 2014) have addressed this type of issue by introducing some awareness and control mechanisms on top of such intelligent infrastructure. For example, they consist in visualizing data exchanged across widgets and allowing users to disable some inter-widget communication (IWC). In our case, we allow end users to create mashups from existing Web components without requiring that they be defined according to any standard (e.g. W3C widgets), and leave it up to the users over to define the connections amongst the components that can be manipulated through copy-paste operations.

2.4 Selection of Web Components

Different techniques for extracting Web components from existing Web pages have been proposed. The selection may rely on the structure of the Web page code or, alternatively, on its visualization. In the proposed mashup environment, the selection is based on the HTML structure. However, as demonstrated in (Cai et al., 2003), it is possible to automatically derive from the page rendering a structure that is closer to the user's perception, and may be employed for the cases where the code structure differs significantly from the visual result. The same page structure approach is exploited in (Dontcheva et al., 2006), where the user can select different Web page elements and specify tags in order to categorize the selected contents. After providing some

selection examples, the system is able to generalize patterns on similar Web templates (e.g. different pages of the same Website or other Websites with similar structure or contents). The collected data is exploited to create automatic summaries for retrieving or sharing information. The approach has been extended in (Dontcheva et al., 2007), where the authors added the support for extracting interactive elements (e.g. input forms) in addition to the content. This has allowed the creation of a system able to merge contents or perform parallel searches on different Web sites. Differently from the approach presented in this paper, that proposal exploits the Google search API in order to find the content related to a specific query on every Web site. Therefore, the search results are those provided by a Google search on the specified Web site domain. Our approach exploits a proxy server that is able to request data directly from the original Web site, thereby enhancing the reliability of the presented results.

A prototype for filtering navigation across well-structured Web pages, named Sifter, was introduced in (Huynh et al., 2006). Sifter is a browser extension that lets the users specify which information they are interested in and how they would like to display it. A relevant difference with our environment is in the system architecture: Sifter works on a plugin, i.e. client side, while our environment has a client-server structure and relies on a proxy. The differences in the underlying algorithms are discussed in the Appendix (see “Intelligent Filtering when Executing the Mashup”).

In (Nichols et al., 2007) and (Nichols et al., 2008) a proxy-based approach for automatically creating mobile versions of desktop Web sites is proposed. The solution is based on a remote control metaphor, with the user controlling real desktop pages that pass through the proxy server, which splits the original pages into different parts. The proxy server handles the interaction with the original page, therefore the contents proposed to the user are those coming from the original Web site. Similarly to our solution, the support is able to identify parts of the pages that are repeated according to the data from a given source and is able to extract contents for different query results, by using similarity heuristics based on the HTML code structure. However, in that work such techniques are exploited for adaptation to mobile interaction, thus focusing on only one

Web site at a time. We enhance the approach by connecting different parts coming from different Web sites for creating new composite applications.

d.mix (Hartmann et al., 2007) aimed to create applications based on Web services through a site-to-site service map. With d.mix, users navigate Web sites (that have been previously annotated) and select the elements of interest. The platform is able to generate all the code needed to exploit the original application Web services that manage the selected elements. The platform also provides a set of examples that can be modified and composed to create applications. However, users are required to have some programming knowledge for modifying the code.

2.5 Summary on Approaches to End User Development in Web Environments

In order to better position our contribution in the relevant literature, we have identified the following six dimensions to analyse the main features of a set of representative tools for end user development of Web applications:

- *Deployment type*: how the tool is deployed, e.g. as Web browser extension or through some type of server. This dimension is related to the way the tool is installed/accessed and to the cross-platform compatibility (e.g., type of operating system, browser, etc.), thus having an impact on usability.
- *Programming knowledge*: the level of familiarity with programming languages that is required to properly use the tool. Our goal is to allow end users without programming experience to create new applications starting from existing ones in order to better support their needs. A good trade-off between intuitiveness and flexibility of the development environment is thus desirable.
- *Reuse of existing resources*: to what extent the tool allows reutilization of existing Web resources (e.g., content, functionality). We consider this dimension to be of relevance for end user development. Existing resources can indeed be used as a starting point for creating novel artefacts rather than starting their development from scratch.

- *Components selection*: whether/how the user can choose components from existing Web pages in order to develop new applications. This deals with the way existing contents/functionalities are selected (e.g. by direct selection or via custom scripts).
- *Components composition*: whether/how it is possible to create new connections among selected components. This dimension deals with the way selected components are functionally linked in order to furnish, as a whole, novel functionalities, which constitute the added value of the mashup.

Table 1 summarizes how a representative set of relevant tools for EUD of Web applications can be characterized with respect to the above mentioned dimensions. Another design space is introduced in (Cao et al., 2011), which shows some overlap with our space, such as the Programming knowledge dimension. However, in that work, the discussion is mainly on how to stimulate ideas in end user developers. In our case the focus is more on technical aspects, in order to compare several tools in terms of the support provided to the user for actually creating and modifying their artefacts.

Most of the tools considered have been released as extensions for a specific browser (i.e. Firefox Mozilla), with the exception of Greasemonkey and d.mix. The former is available as a plugin for three different browsers, the latter is a Web application. Our MashupEditor is a Web application as well, providing all its functionalities through any standard Web browser without the need for any extension, and exploits the support of a proxy server. The advantages of such a design decision are twofold. On the one hand, the proposed solution allows creating mashups that have the same availability on the Web as well as the applications that provide their content: they can be accessed remotely through different devices and by different people. On the other, the proxy solution supports DOM manipulations and analysis that allows supporting a wide variety of existing Web technologies, thereby limiting performance issues that affect many browser extensions that provide similar features.

Tool	Deployment type	Programming knowledge	Reuse of existing resources	Components selection	Components composition
d.mix (Hartmann et al., 2007)	Web application	HTML and (Ruby) scripting	Components visible on the browser to obtain new applications	Direct picking from Web UIs	Editing of existing HTML code, script insertion.
Greasemonkey (mozilla.org)	Browser extension (Firefox, Flock, Web / Epiphany)	HTML and JavaScript	Single Web page per time	Implicit: indirect interaction with UI via JavaScript	No
Chickenfoot (Miller et al., 2010)	Browser extension (Firefox)	Ability to understand high-level commands	Automation of simple actions on existing websites	Implicit: indirect interaction with UI via JavaScript	No
CoScripter (Leshed et al., 2008)	Browser extension (Firefox)	Ability to understand code-like descriptions	Automation of simple actions on existing websites	Implicit in recorded user interaction	No
Vegemite (Lin et al., 2009)	Browser extension (Firefox)	Ability to understand code-like descriptions	Automation of queries on existing websites	Direct picking from Web UIs	Copy-Paste metaphor between Web and VegeTables
Sifter (Huynh et al., 2006)	Browser extension (Firefox)	None	Components visible on the browser composed to enhance the application	By direct picking from Web UIs	Full automatic or user-adjusted
NaturalMash	Web application	Programming only for creating the possible ingredients	Automation of access to existing Web services mapped into "ingredients".	Development of "ingredients" from existing Web services	Natural language and drag-and-drop metaphor.
PEUDOM	Web application	Programming only for creating the possible PEUDOM components	Automation of access to existing Web services mapped into PEUDOM components.	From PEUDOM Component Editor	Drag-and-drop metaphor and other settings.
MashupEditor	Accessible through standard browsers with the support of a proxy server	None	Components visible on the browser to obtain new applications	Direct selection from Web UIs	Copy-Paste metaphor between existing Web components

Table 1. Features of EUD Web tools.

All the indicated tools but Sifter require that the user has some programming knowledge. The required skills range from the ability to understand logical scripts and their relations with user interface elements, to familiarity with HTML and JavaScript. Our MashupEditor, similarly to Sifter, does not require any particular programming skill.

All the considered tools allow some degree of reuse of existing resources. With CoScripter, Chickenfoot and Vegemite it is possible to automate actions/queries that have been performed on Web pages. Greasemonkey is devoted to customizing single pages, while Sifter and d.mix support reutilization of components from more than one page at a time, as our MashupEditor.

The tools that allow selection of existing resources provide more or less explicit mechanisms. D.mix, Vegemite and Sifter, as well as our MashupEditor, rely on an explicit selection strategy that lets the user directly pick components from the original UI. In Greasemonkey and Chickenfoot the selection is implicitly defined in the JavaScript created by the user, which specifies which components are involved in the page customization/enhancement. In CoScripter selection is implicit in the sequence of user actions detected by the tool at recording time.

Composition of selected components is possible in d.mix, Vegemite, NaturalMash, Sifter, and PEUDOM. In d.mix this is done by editing HTML and adding JavaScript. Sifter relies on a semi-automatic mechanism that consists of the system creating a preview composition that the user may manually refine. Vegemite and our MashupEditor exploit instead a high level copy-paste metaphor for connecting selected components. Vegemite involves the creation of tables for organizing copied values that can then be pasted into target components. In the MashupEditor copy-paste is performed directly between the source component (e.g. a search result item) and the target one (e.g. the input field of another search form) in order to create a new connection. Different metaphors are explored in NaturalMash and PEUDOM: NaturalMash supports the use of natural language expressions to indicate the required compositions, while PEUDOM supports drag-and-drop. We have preferred to focus on a copy-paste metaphor, which is familiar for end users and

allows the environment to gather the information necessary to compose the mashup exploiting various types of existing Web applications.

3 THE PROPOSED APPROACH FOR CREATING WEB MASHUPS

This section presents the MashupEditor, our intelligent graphical platform for End-User Development (EUD) of Web mashups. The platform supports direct manipulation of interactive components that the user is interested in reusing for creating new applications. Components composition is facilitated by the intelligent support that exploits examples provided by the users for this purpose (see a demo in the video at http://youtu.be/Yb03lIHfyk4). The created mashups can be saved in a local repository and subsequently reused. An additional functionality is mashup sharing through social networks (e.g. Facebook) in order to allow other users benefit from such customizations.

3.1 Example scenario

In order to introduce the features of the Mashup Platform, we describe an example scenario.

Bob is passionate about motorcycling and never misses the opportunity for a trip on the road. The summer is coming and, as usual, he starts planning the route for his next vacation. In order to define how to break up the journey and the duration of the stops, he needs information about the weather and a list of points of interest in a specific country or region. In addition, he usually searches through the news Web sites for local events and political situation in the transit areas. Since this information is available on the Web by accessing different sources, Bob considers the creation of a mashup to facilitate the search.

The initial step for creating a new mashup is choosing a name and providing a short description for it. He names his project “Sidecar” and adds “a travel companion for bikers” as description. At that point, one or more mashup widgets (in the following referred to as widgets) can be iteratively added to the application. In this paper, we consider widgets one or more interactive components of existing Web applications that can be added to a mashup and thus exploited at run time. Bob can

add widgets to his mashup in two different ways: the first one is by directly selecting components from existing Web applications. This allows him to create new widgets that are immediately added to the mashup. The other one is by loading existing widgets from a personal library or from a public repository.

As Bob is leaving Poland heading South-East, he starts by visiting Wikipedia for getting a general description about Ukraine. He accesses the Web site through the Mashup environment, inserting the URL from the editor interface. After that, he can navigate the Web site as usual. When Bob performs a search, the Mashup Support is able to identify which input elements of the Wikipedia forms he has filled in. Then, Bob can select the components for his mashup in the resulting page, by left-clicking on them. Their actual dispatch to the MashupEditor is triggered by clicking on a small icon appearing on top of the selected components (see Figure 1). In this case, Bob selects the table containing the biggest cities in the country. Once the selection is received, the environment adds the related components to a new widget in the mashup working area, including both the filled input elements and the selected search results. Bob repeats this procedure for creating a widget including the information from WeatherOnline, a weather forecast Web site.

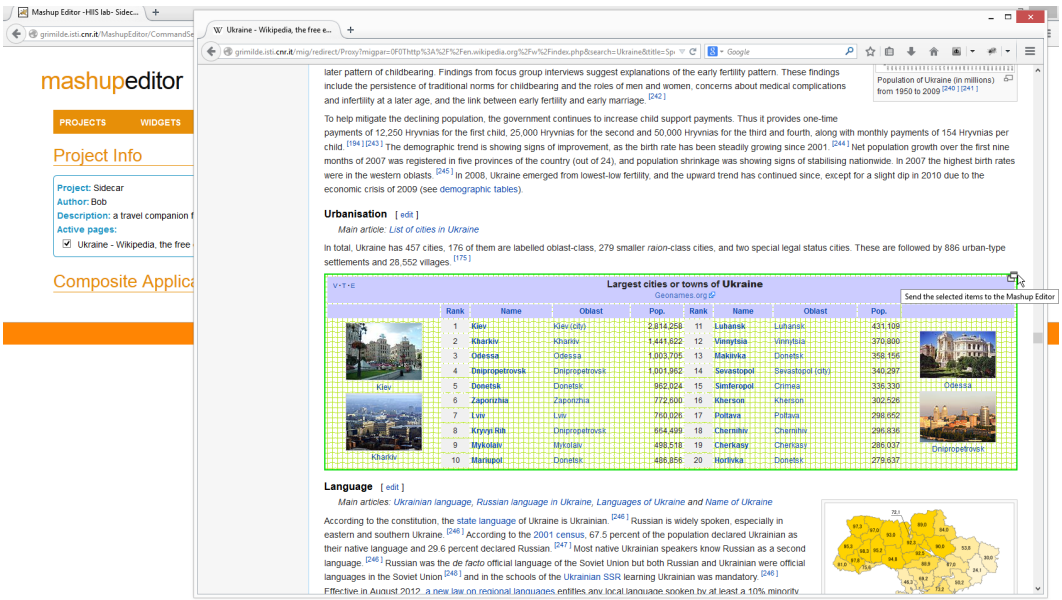

Fig. 1. Dispatching a table from Wikipedia to the MashupEditor.

In order to geographically locate the places he is interested in, Bob enables the Google Maps widget in the mashup working area. Bob also needs a widget for searching the latest news about a specific place. To this end, he exploits the mashup repository and finds one coming from the BBC Web site. Once he selects the widget, it is automatically added to the environment. At this point, the mashup environment shows three different widgets, one for each information source Bob needs.

In order to coordinate the research on the various widgets in the mashup, it is necessary to connect them. Connections are defined during a "recording" phase, explicitly activated by the user. During this phase, user actions are interpreted by the system. For example, copying "Kiev" from the Wikipedia table and pasting it into the WeatherOnline user interface input element causes the platform to bind the two widgets. In detail, when this copy–paste occurs, the intelligent support creates and saves a connection between the Wikipedia and the WeatherOnline widgets. By using this metaphor, Bob connects the name of a city in the Wikipedia table with the Google Maps, the BBC, and the WeatherOnline widget.

Once the action registration is terminated, the resulting mashup interface changes. The intelligent support is able to understand that the Wikipedia table contains different data entries similar to that Bob selected. Therefore, it adds the support for selecting a different city by clicking on its name. This action is suggested to the user through a small icon nearby the city name. In addition, the support removes the input forms where Bob pasted the city name, since they are now directly connected to the Wikipedia table entries, from where they receive their input. The resulting mashup is shown in Figure 2.

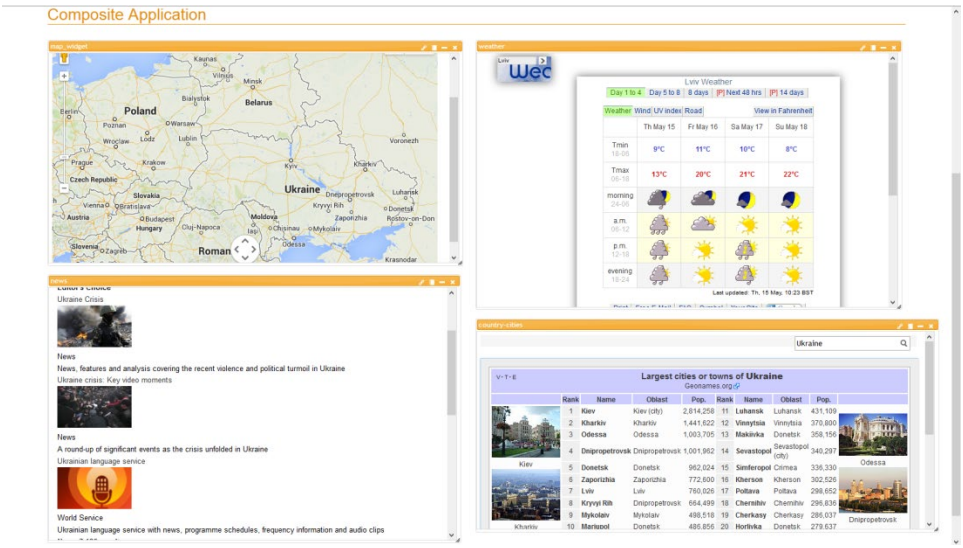

Fig. 2. The resulting mashup that includes four widgets from (clockwise from top-left): Google Maps, WeatherOnLine, BBC, Wikipedia.

When Bob clicks the Kharkiv entry in the Wikipedia table, the content inside the different widgets is updated: Google Maps pinpoints the position of the selected city, the WeatherOnLine widget shows Kharkiv weather forecast, while the BCC widget shows the search results for the “Kharkiv” keyword (see Figure 3).

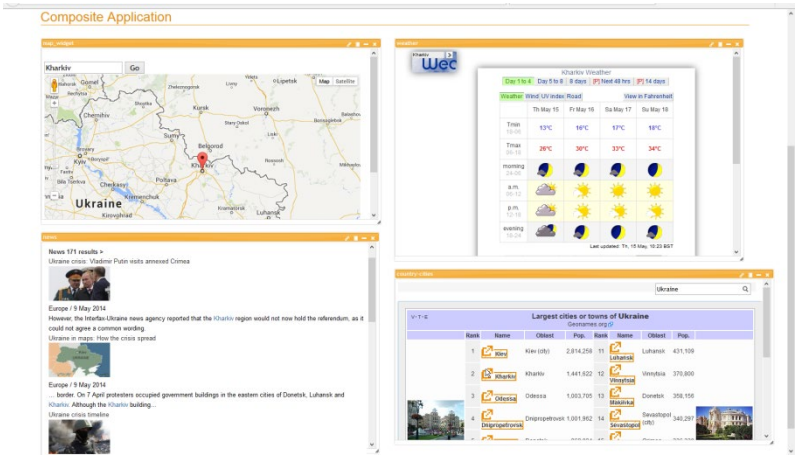

Fig. 3. The updated mashup resulting from clicking on “Kharkiv” in the bottom-right widget.

Bob can exploit the mashup for receiving information on another country, for instance Romania, simply writing its name on the Wikipedia widget search bar. Its content is updated accordingly

and Bob can select among the biggest cities in Romania for visualizing their position, the latest news and the weather forecast.

4 THE INTELLIGENT EUD PLATFORM FOR WEB MASHUP

4.1 Overall Platform Architecture

The authoring platform is made up of a server-side part, the *Proxy/Mashup Server*, and a client-side part, the *EUD Environment* (see Figure 4).

The *Proxy/Mashup Server* is the part acting as the annotation proxy when the source Web applications are initially accessed. The annotation phase consists of injecting JavaScript excerpts that will allow users to interactively select the components to be included in the mashup.

The Proxy/Mashup Server part is implemented as a Java Servlet and JSPs. The *Mashup Support* is the enabling core of the environment and furnishes all the mechanisms to manage mashups. It resides on the server side and is implemented as a Java Servlet, which exploits a number of other Java classes, each one providing certain functionalities or describing some entity.

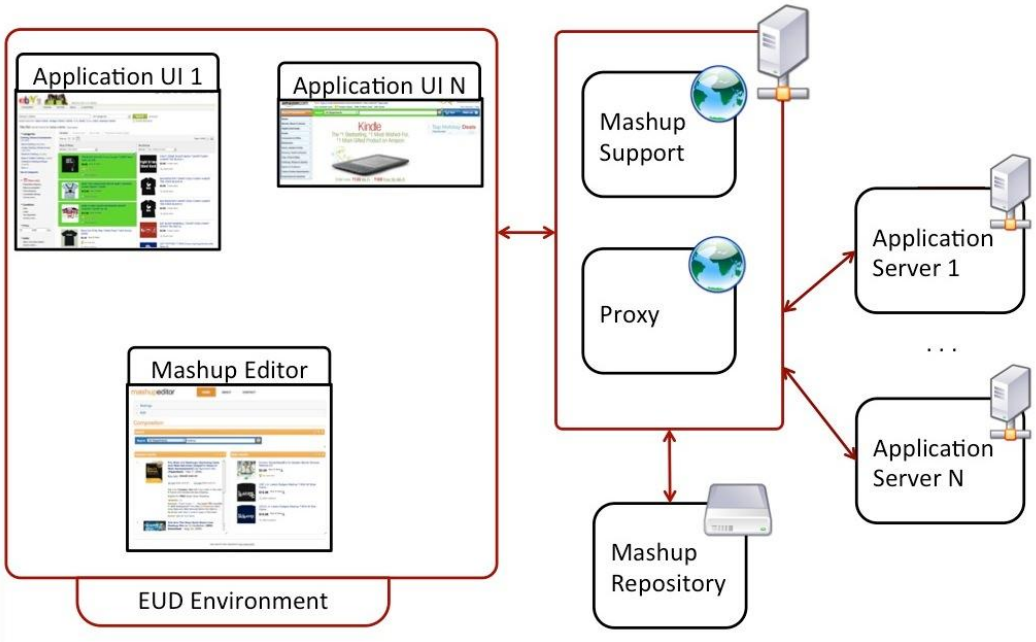

Fig. 4. Architecture of the Mashup platform.

The *EUD Environment* allows users to interactively create connections among widgets (components originally belonging to the source Web applications) and to manage the created mashups. This client-side part is implemented in HTML and JavaScript, with the support of jQuery and jQueryUI libraries. In general, the technologies we have used are standard and compatible with existing browsers, which ensure platform-independence.

Widgets are the basic building blocks of the mashup. Elements within a widget maintain the behaviour and functionality of the original application. Widgets are defined by the following properties:

- *Title*, defined by the user at creation time.
- *Author*, username of the creator.
- *Creation and last modification time*.
- *Original URL* that has generated the HTTP request from which the output components have been obtained.
- *Set of application input parameters.* If the widget accesses server-side functionalities, then the following information is maintained for each parameter of the request query string (GET method) or the request content (POST method): ID, value and a flag indicating whether the value has been inserted by the user or is automatically generated by the application.
- *Input/Output Component*, i.e. the HTML code of the user-selected components (including CSS).
- *Set of label tags*, to facilitate the widget search within the repository.
- *Set of connections parameters*, representing functional links among the connected widgets. For each connection created, an entry is added to this list containing the target widget ID and the IDs of the parameters involved in the connection.

4.2 How the Platform Works

4.2.1 Proxy-based navigation and component selection

In order to be used as resources for mashup creation, Web pages need to be accessed through the annotation proxy that is part of the environment. The proxy parses the original HTML page and performs the following main steps:

- Enhances the page with a set of JavaScript functions for subsequently managing mashup functionalities.

- Finds the selectable components of the page, i.e. all those HTML elements that might be used as "pieces" for a mashup. Containers/functional blocks such as DIV, TABLE, FORM, etc. are examples of such elements. Then, the following operations are applied to them: a unique id is assigned, if they do not have one. The id is needed to subsequently identify the elements, e.g. when they are selected for inclusion in a mashup; event handlers to manage mashup-related functionalities are added to each relevant element: onclick and onmouseover/onmouseout attributes are defined in order to allow direct selection with the mouse. If the element already has definitions for such attributes, the mashup functionalities do not replace the original ones, but the enhancements are "appended" to them. Modifies all the external http/https references within the page by converting each of them into the proxy URL plus the original reference specified as HTTP parameter. Reference modification aims to assure that the browser is able to solve all the resources and to avoid cross-domain exceptions in HTTP requests (i.e. made by AJAX scripts). References are modified according to their type: Links in HTML tags (e.g., values of "src"/"href" attributes) are easily obtained by querying the document object via available library functionalities and then converted (the queries are made through objects/methods from "javax.xml.xpath"); References within "style" or "script" elements are first identified through a rule that relies on a set of regular expressions, and then converted.

- Responds to the client with the annotated page that embeds the mashup environment capabilities and still preserves its original functionalities.

In general, the proxy has shown good reliability in handling even Web pages using more recent dynamic techniques (such as AJAX scripts, websockets, and new client-side frameworks). There are still a few very particular cases in which there may be some issues in properly annotating links within the JavaScript code, when the links are created through dynamic concatenations of arbitrary pieces of string fragments.

The first user step to creating a new mashup is to directly select components from the user interfaces of the source applications. To this end, the annotations and scripts previously inserted by the proxy are exploited. In detail, mouse hovering highlights the component in grey, while a mouse click selects the component and highlights it in green. When a component is un-hovered or unselected, the injected scripts are able to restore its previous aspect, since the original style (e.g. background colour, border, etc.) is saved.

As soon as users are satisfied with the selected components, they send them to the MashupEditor by clicking on the icon appearing on the right-hand part of the selection. From a technical point of view, such sending is performed by invoking a procedure (previously injected onto the page by the proxy server) that serializes the whole Document Object Model (DOM) of the page, including the content of previously filled in form fields, and finally sends it to the Mashup Support together with the list of IDs of the selected components. Within the server, all the elements that were not selected are eliminated from the page DOM, so as to obtain a *partial DOM,* which is saved locally. The partial DOM is sent to the MashupEditor, which asks the user to specify a label/title, and then displays it in a *Mashup Widget*.

The platform keeps the following widgets' information in a widget descriptor: position and size within the Editor, path of the HTML content file, list of input parameters, and set of IDs of the content elements.

Figure 5 shows the steps required by the Mashup Environment to create widgets from existing applications:

1. The user enters a URL to be accessed through the MashupEditor. A new URL is automatically created by concatenating the address of the proxy servlet and the URL specified by the user. A new browser window is activated to access the new URL.
2. The new window sends an HTTP request to the proxy.
3. The proxy connects to the application server, whose address is specified in the request parameter.
4. The proxy receives the application server response and annotates its content (HTML and related resources) by including the scripts for direct components selection.
5. The proxy passes the selected content to the new window, which is displayed.
6. The user selects a set of elements from the new window and triggers the DOM forwarding to the Mashup Support, which extracts the selected elements.
7. The MashupEditor is notified by the Mashup Support about the newly created widget, and allows the user to specify a name for it through a dialog box.

Users are not required to select a component from the first page displayed by a given application. Indeed, they can freely browse different pages. Every time the user changes the current page, the steps from 2 to 5 are performed, until the user sends a selection to the Mashup environment.

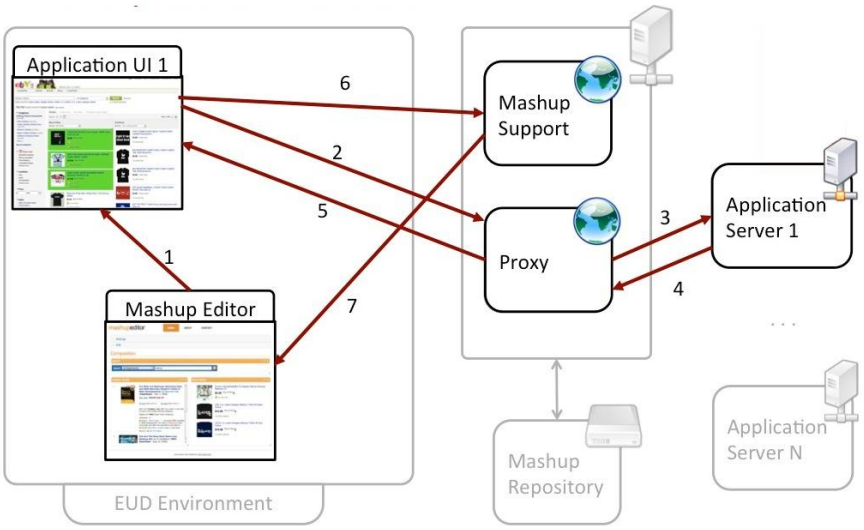

Fig. 5. Action sequence for creating a widget.

4.2.2 Extracting parameters for remote functional access

One important requirement is to still allow users to access the server-side functionalities of the original applications from the new mashup. For this purpose, at run time the Mashup Support provides the correct application parameters in the HTTP requests generated by the widgets. The identification of such application parameters is performed by monitoring the HTTP requests during proxy navigation before widget creation. In the case of the GET method, the query string is parsed; in the case of the POST method, the request content is considered.

Before creating the widget, whenever the user fills in a form element, a proxy-injected script marks it as *edited*. Every time the user interaction generates a request to the Web application server, the same script sends all the *edited* input elements' data to the Mashup Support in background. For each Web page currently open, the Mashup Support maintains only the last set of edited inputs and deletes the previous set whenever it receives a new one. The current page content is generated by sending the set of input values to the corresponding application server.

When users send their selection to the Mashup Environment in order to create a new widget, the Mashup Support checks whether there is a previously saved set of edited inputs associated with the corresponding application. If so, the widget content displayed on the Mashup Environment is made up of two parts, see an example in Figure 6: the *input* part (1), the *output* part (2).

Fig. 6. A Mashup Widget.

The Mashup Environment maintains the values that the user entered in the form elements before including the components in the mashup. At this point, the widget is ready to receive new inputs and update the content in the output part accordingly. In the example widget, the user only needs to change the search keyword (*middle east*) and press the button in order to get the new results.

4.2.3 Connecting widgets

The interactions described in the previous section allow users to create a set of widgets from existing Web components that work independently. The subsequent step is to create a connection between widgets obtained from different applications to support the information flow across them. Our aim is to make such a mechanism intuitive, relying on concepts and actions that users without programming knowledge are able to understand and handle. The solution we propose consists of identifying components' connections through a copy-paste metaphor, where users demonstrate the correspondence between values referring to elements of different applications. By interpreting such user actions, the Mashup Support is able to identify the corresponding connections between the different application components.

Our environment can exploit the copy-paste metaphor in two ways. In the first one, the copy-paste is performed between two input elements belonging to two (or even more) different applications. In this case, the user indicates that the input provided to one application should be considered as

an input also for the other one(s), then generating a parallel content update in the widgets that have been connected. In the other case, some output of one application is copied and then pasted on the input element of another one. The goal in this case is to indicate that the resulting outputs of the former application should be considered as inputs for the latter one.

Among the reasons we opted for relying on the copy-paste metaphor (instead that on e.g. the drag-and-drop) is indeed that it resembles to what the user is likely to manually do when searching for the same keyword on two search engines, or when using parts (e.g. keywords, sentences) of a Web page as input for a Web search. We believe to have achieved an interesting trade-off between usability and expressivity: the simple copy-paste metaphor could limit the expressivity of the solution, but it maximises intuitiveness and expands the pool of potential users.

4.2.3.1 Connecting Widget Inputs

As explained in the previous section, the connections in the Mashup Environment rely on sharing some request parameters among applications, or on linking an output value of a widget with a request parameter of another one. We rely on the assumption that end users do not necessarily understand the notion of HTTP request. Nevertheless, they are likely to understand that the content shown in a widget depends on the values inserted in the input elements. Therefore, we can exploit the HTML input forms (text fields, text areas, drop down lists, radio buttons, etc.) as a user perceivable representation of the HTTP input parameters. We thus identified the copy-paste metaphor as an intuitive way to allow users to specify that two different elements should share the same values. In summary, the connection mechanism supported by the Mashup Environment consists of the following user actions (see Figure 7):

1. Press the action recording button to change the state of the Mashup Environment
2. Select the content of some element from the source widget *A*
3. Copy the content of the selected element
4. Select the correspondent input element in the target widget *B*
5. Paste the value into the selected input element in the widget *B*
6. Press the stop button to end the recording phase and reset the Mashup Environment state

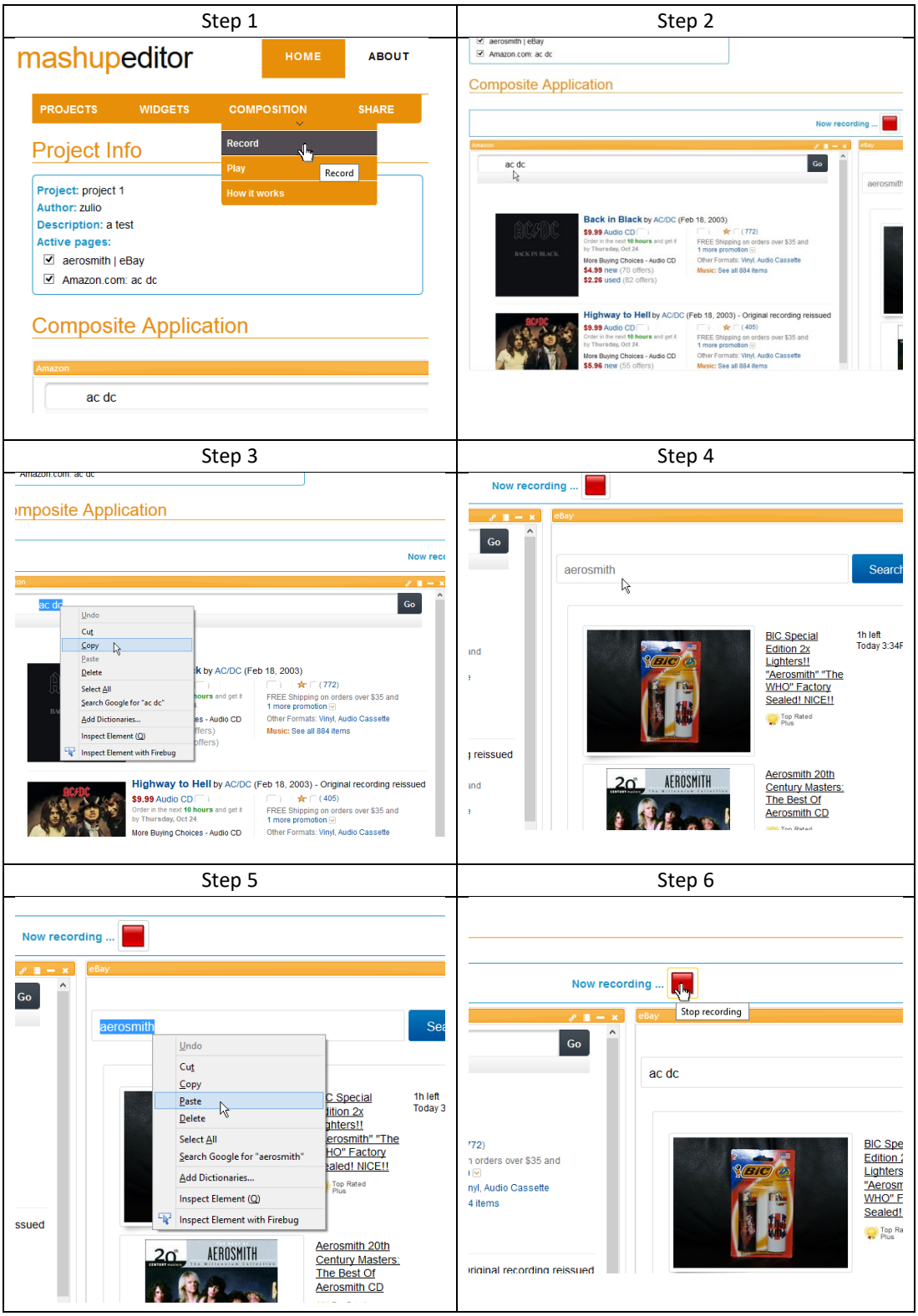

Fig. 7. Widgets connection steps.

In this way, the environment records the actions performed by the users in order to be performed automatically afterwards for updating the widgets. After these steps, the widgets A and B are connected in such a way that the widget A input values are automatically reused for updating the widget B (parallel composition) as well.

After having set up the connections, the content shown by both widgets is updated by interacting with a single widget (i.e. the form on the left-hand part of Figure 8). Figure 8 shows the resulting mashup application with the search results for the keyword “potter” shown in both widgets. In addition, it is worth noting that the field selected by the user through the paste operation (in this case the eBay search field) is automatically removed by the Mashup Support. This is because that field is no longer needed by the application, since the search on eBay will be performed according to the input provided through the Amazon form.

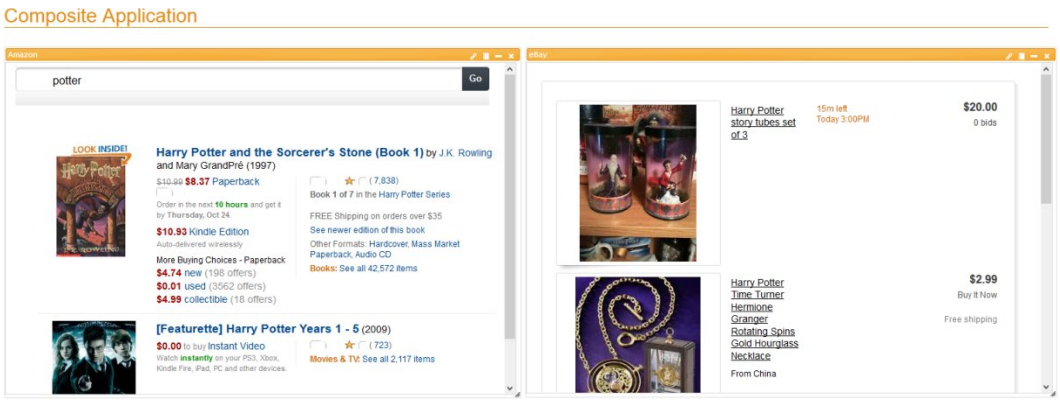

Fig. 8. Parallel search on two widgets

An additional feature of the Mashup environment allows users to review the connections they have previously created. This functionality is activated by the play action, available under the connection menu. When triggered, a special visualization modality shows existing connections among the available mashup widgets (see Figure 9): the input components of all the available widgets are shown and highlighted in green, while red arrows represent connections between components.

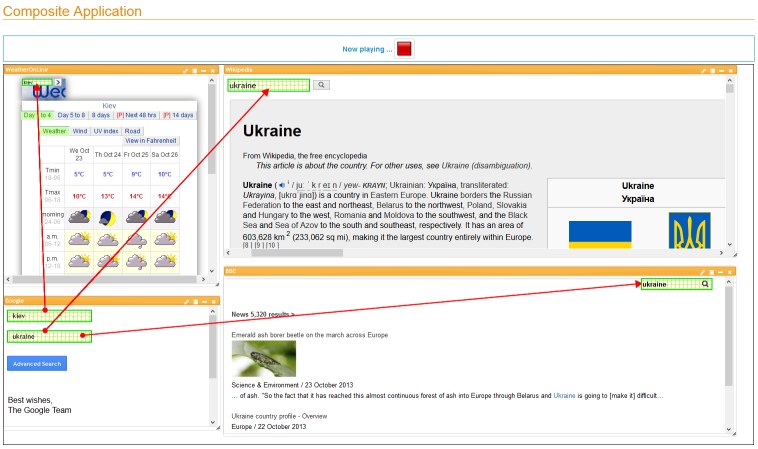

Fig. 9. Connections visualization for the available widgets.

4.2.3.2 Connecting Widget Output and Input

A further way to combine contents/functionalities across widgets through the copy-paste metaphor is supported. It allows users to select and copy relevant portions of text from the output of a first widget, and then paste them into an input of a second widget (sequential composition). For example, this functionality aims to easily reuse (parts of) the query result of an application for filling in the input form of another one.

The need for such a support comes from a common interaction practice, where several applications are accessed in order to get more information about items, places, people, etc. An example of such situation is the user browsing a conference Web site, looking at the organizing committee and wanting to search for publications of each member. The publication search may be manually done on a digital library by iteratively copying name/surname in the search field of the library and submitting the form. This implies performing a huge number of clicks and switching many times from the two browser tabs/windows. Similarly, someone interested in buying music of the ‘80s, but without knowing the exact artists/albums/song names could not be able to easily find the wanted items on a conventional online mp3 store. To this purpose, s/he would most probably access some online library listing compilations of that period. The author names would be then copy-pasted, one-by-one, into the search field of the mp3 store in order to get a list of related songs/albums for sale.

From the user point of view, the metaphor we rely on for describing such sequencing of actions is again the copy-paste because, as explained in the above examples, reflects what users actually do when searching on the Web for a predefined list of keywords. However, there are minor differences in how an intelligent system providing automatic support might furnish the output (when the mashup is run), distinguishing the case of pasting into a generic service (e.g. a search engine) from the case of pasting an address into a georeferencing service. We describe in detail how such automatic support works, its benefits in terms of user experience and limitations.

In general, the output of a first widget can be used as input for a second one in order to automatically access the latter and obtain output from it based on the output of the former. As indicated by the previously mentioned example scenarios, it often happens that users rely on more than one application to refine an online search. In these cases the user creates two widgets in the Mashup Environment: one containing the output whence to get the textual information to be used as the parameter for refinement, while the other one contains the input and output part of the application for refining the search.

Considering the example of an author search, the first widget may contain an unordered list of the committee members extracted from the conference website. The second widget is obtained by selecting the search results from a digital library. In order to connect the two widgets, the user starts the recording mode, then she selects and copies the name/surname of a committee member in the first widget, and pastes it on an input field in the second widget. The system interprets these actions as the definition of a sequence: the user provides the input to the second widget (and in turn manipulates its output) by selecting one of the results in the first widget. After turning off the recording mode, the system reacts initially by highlighting the user selection and related elements on the first widget. The user selection is basically the deepest HTML element containing the selected text. Related elements are HTML containers “similar” in structure and style to the ones involved by user selection. Such elements, identified by a procedure explained in the appendix, contain the names/surnames of the committee members, and are also integrated with additional events: when an highlighted element is clicked, its content is automatically sent to the form of the

second widget and the submit is invoked. This makes the second widget quickly show the publications of the selected person. The benefit of such a support is that, thanks to a single copy-paste action, multiple searches can be performed on the second widget by simply clicking on relevant elements on the first widget (instead of copy-pasting each element text content and submitting). Figure 10 shows an example of output-to-input connection between a table extracted from European Capitals page of Wikipedia and WeatherOnLine.co.uk website. A sample city is copied (see Figure 10 A) and pasted from Wikipedia to WeatherOnline. As soon as the recording mode is stopped, the system highlights the HTML containers of all the cities in Wikipedia (see Figure 10 B). The city containers are also enhanced with onclick events. Note that, at this point, only the right widget has been updated. Afterwards, clicking on a city results in the left widget showing the weather conditions/forecast for that city (in the example, the clicked city is *Nicosia*, see Figure 10 C).

Fig. 10. Output-to-input connection (output of right widget is connected to input of left widget): copying sample text from right widget (A), getting highlighting of automatically recognized related elements (B), clicking one of the elements for using it as input for left widget (C).

Georeferencing widget output is a particular case of using the output of a first widget as input for a service contained on a second widget (and then to obtain output from it). Map-based mashups are commonly used to locate places on maps.

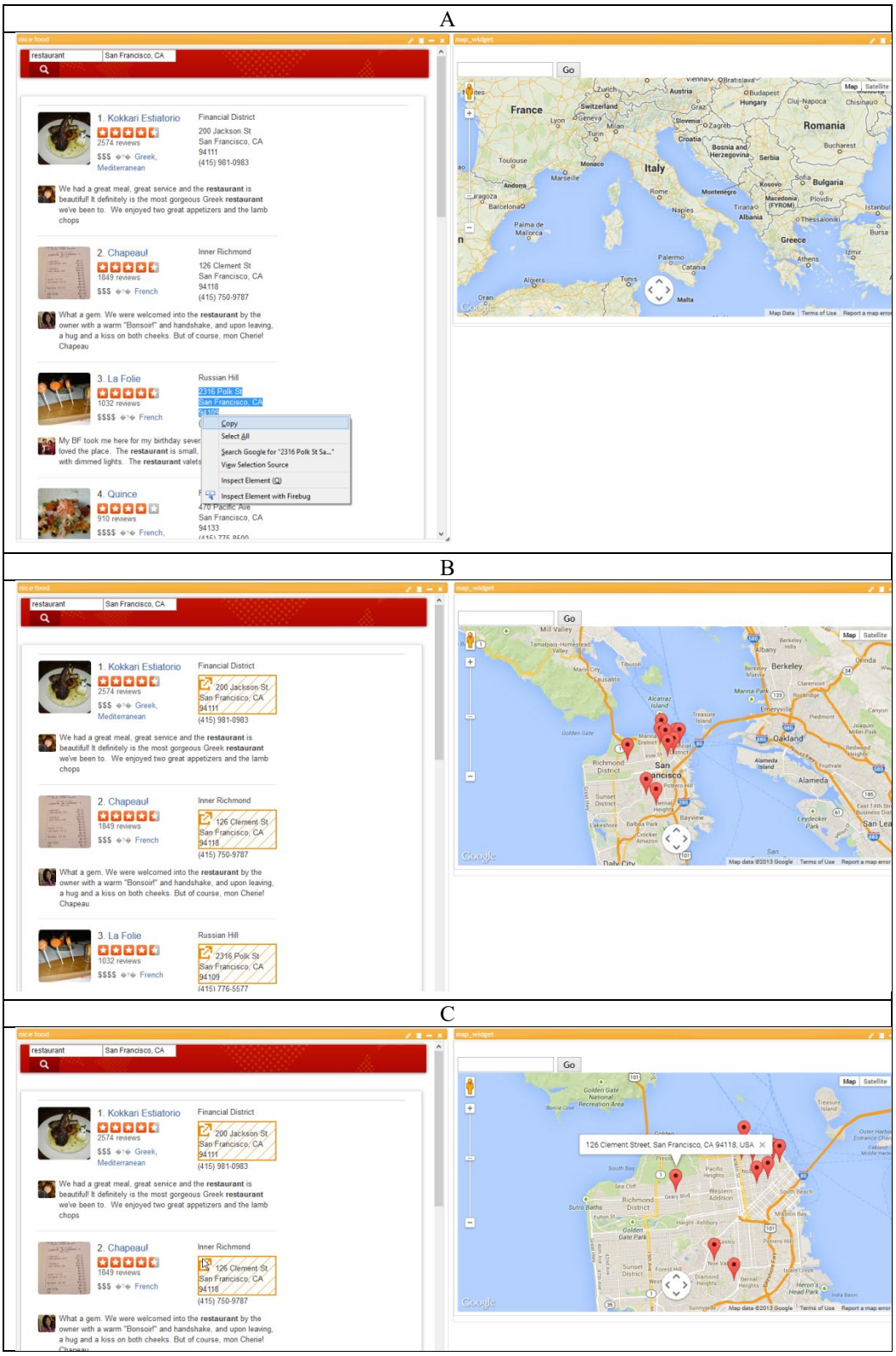

Fig. 11. Connecting output to map input: copying address from output (A), highlighting recognized addresses and georeferences (B), locating address in the map by element selection (C).

For this reason, we have opted for defining an ad hoc widget embedding a Google Maps frame, without affecting too much the generality of the support. The map widget can be shown or hidden anytime and may be used standalone or in conjunction with another widget, in order to locate the results of that widget.

A connection between a generic widget and the map is created in the same way as it occurs between two generic widgets: some text is copied from the first widget and pasted into the map input field. The system assumes that the user wants to automatically georeference the content of the first widget, and that the selection represents an example of input for the map (see an example in Figure 11 A). The system then finds all the elements similar to the selection and highlights them (see Figure 11 B). So far, the behaviour is consistent with connection between two generic widgets. The peculiarity of the map widget is in the way information can be displayed: the map indeed allows showing more than one element at the same time (i.e. an arbitrary number of pins can be deployed). Clicking on one of the highlighted elements (i.e. addresses) on the first widget triggers the widget panning in such a way as to show the pin corresponding to the selected element in the map centre and displaying the pin label (see Figure 11 C).

In the case of mashup involving a map, the system is also able to update the widgets whenever the output of the first widget changes. Referring to the previously presented example, after having found some restaurant in San Francisco, the user might search for tacos in San Diego. In this case (see Figure 12), the system furnishes the filtered result and promptly finds and highlights the addresses of tacos shops in San Diego in the search widget. The map widget is also pinned accordingly.

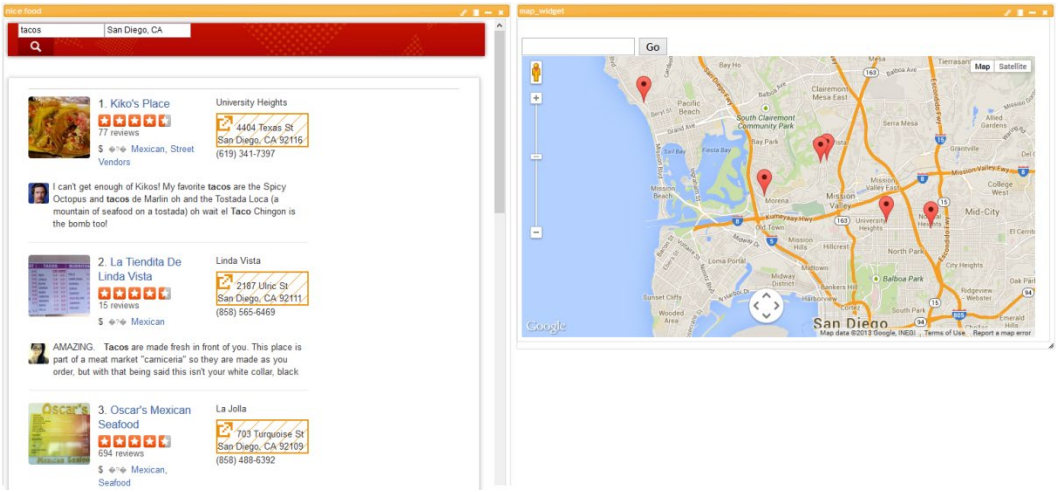

Fig. 12. Connections consistency preserved after performing a search: addresses for tacos shops are automatically highlighted on the new search results page, and are pinned in the map.

4.2.4 Complete mashup functional behaviour

The Mashup widgets are real applications that provide updated content every time the user submits new values to some input fields. Figure 13 shows the steps performed to carry out the update activated by new inputs. First of all, the input values are extracted from the selected fields by the platform and sent to the Mashup Support (1). After that, for each input element, the connected widgets are considered for update. Then, for each widget to be updated, the value of the input field is replaced in the GET query string or in the POST content, and an HTTP request is sent through the Proxy (2) to the Application Server (3). The response is received first by the Proxy (4) and then forwarded to the Mashup Support (5). The response content is then filtered by the Mashup Support according to the component(s) that were selected in order to refresh the associated widget in the MashupEditor (6). In particular, the Mashup Support is able to extract from the application response exactly those elements that correspond to the component selected by the user. For example, the user may select and include in the mashup only the first three elements of the search results. Only the first three results will then be displayed when new searches are performed through the mashup.

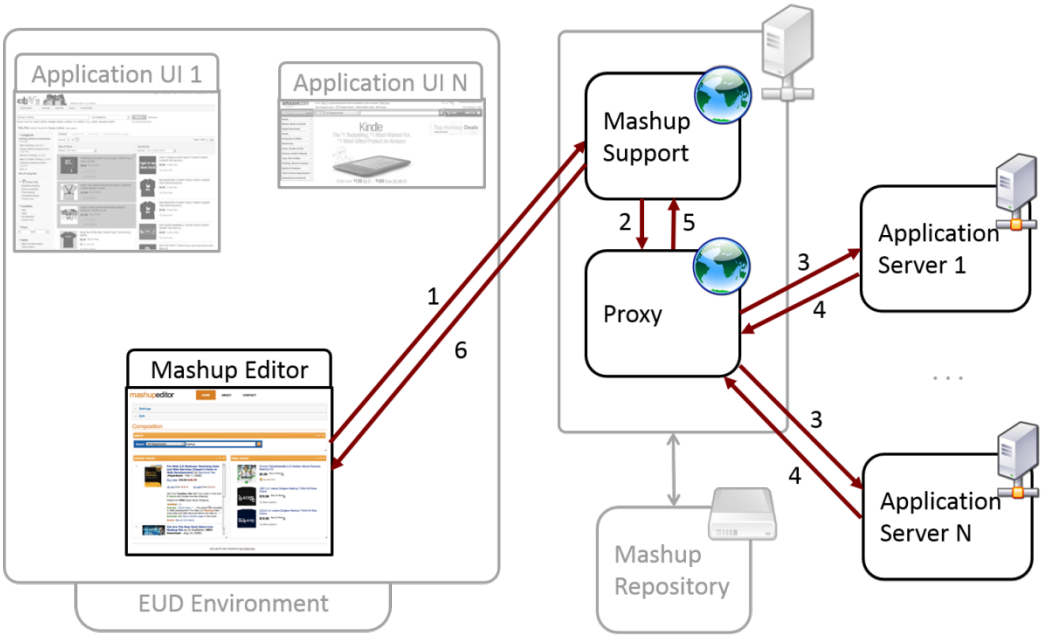

Fig. 13. Components update

4.2.5 Management of Dynamic Web Applications through the proxy support

Web technologies are in continuous evolution, thus we have extended our solution in order to work with more recent features, which are more and more used (https protocol, websockets, etc.). For this purpose, the proxy does not alter the original page functionalities, nor the protocols used to access the page itself and the related resources. In detail, if a resource is originally referred through "http", it is then accessed via "http" through the proxy as well. The same applies to https. For example, an online store can be initially accessed through the home page via http, but it then may require access via https as soon as the user requests to log in. Switching to https is usually done in order to protect confidential data (i.e. user credentials) exchanged between client and server. In our platform whenever the client and the application server are using HTTPS, the Mashup Environment routes all traffic through the proxy module using HTTPS as well.

The proxy is also able to manage pages that utilize websockets. At annotation time (when the page is accessed by the proxy), websocket addresses starting with "ws://" are not modified by the

proxy. At execution time, the injected scripts are able to detect when the DOM changes, e.g. when new elements are added by the application logic. In modern Web pages, such modifications are often due to data coming from AJAX requests and/or websocket connections. When such changes are detected, the injected scripts check whether the newly added/modified DOM elements need to be annotated with the scripts supporting the mashup environment. If so, then annotation is performed client side by the proxy-injected JavaScript. This strategy allows the user to interact with (i.e. select and forward to the Mashup Environment) the elements that are dynamically modified by the application logic.

The proxy provides session persistence to the accessed Web applications during mashup editing. This is achieved by a cookie storage mechanism that binds the internal session (client-proxy) with the cookies of the external session (proxy-application server). For example, in order to exploit a webmail widget in a mashup the user has to login only once, i.e. in the webmail home page. Afterwards, it is possible to create the webmail widget and interact with it without having to re-login at each request, since user session cookies are persistent in the Mashup Support.

The following example summarizes how the above mentioned techniques are put in practice through the steps illustrated in Figure 14 (a video is available at http://youtu.be/Szs7mQJU_iU).

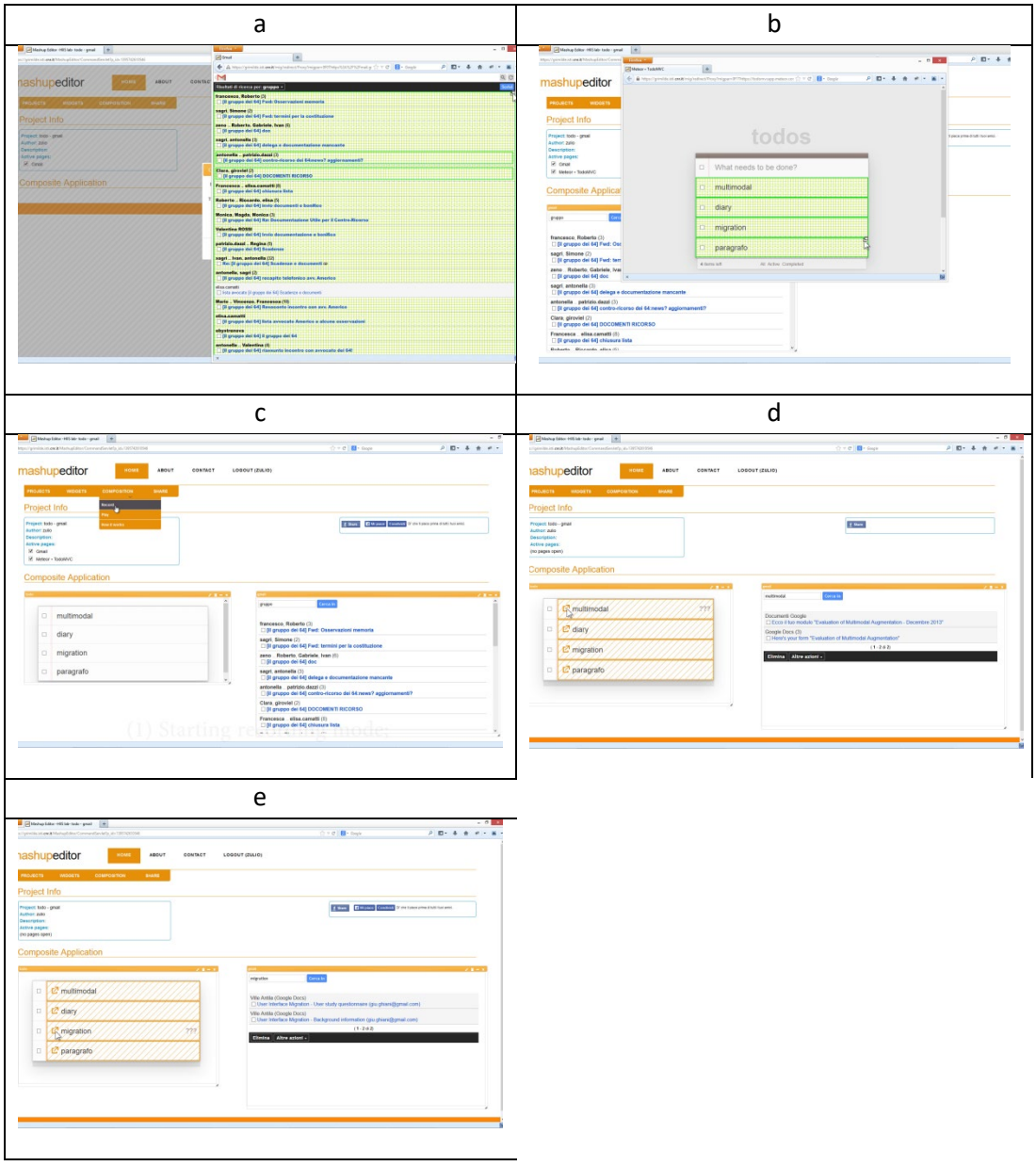

Fig. 14. An example mashup that involves Web applications relying on https protocol, session persistence and websockets.

In a) the user selects the block containing the inbox email search results, and subsequently forwards the selection to the Mashup Environment (the mobile version of Gmail.com has been used for this purpose). In b), parts of an interactive “to do list” (todo in the following) Web

application[7] are selected and, in turn, sent to the Mashup Environment. The todo application is highly dynamic, and its underlying logic adds HTML elements to the document in an unforeseeable manner. From a technical point of view, such elements dynamically added were not initially annotated by the proxy (because they were not available in the original page). However, as soon as the application logic adds them to the document, a JavaScript that was inserted by the proxy is able to annotate them as well in order to allow user selection. The resulting mashup is visualized in c). The user performs the connection phase by copy-pasting a sample element from the todo list to the Gmail search input, and all the elements similar to the copied one are orange-highlighted by the Environment. Step d) shows the result of clicking on one highlighted element of the todo list, i.e. "multimodal": the element text content is used as keyword to parameterize a search within the Gmail inbox folder. The search results are two emails whose subject contains "multimodal". Step e) is analogous to d), but keyword "migration" is used instead.
As explained above, d) and e) imply querying Gmail for the inbox messages which, in turn, requires the user to be authenticated, i.e. to be logged into Google. In the example, the user has logged in immediately after opening the Gmail page through the proxy. Our Mashup Environment is able to provide login persistence thanks to the cookie-store mechanism, where the internal session (between the client and the proxy) is bound to the external one (between the proxy and the application server).

4.3 Personal and Multiuser capabilities

The Mashup Platform can be used by several users. A personal repository is available to subscribers in order to save their resulting mashups, and it is possible to share them through social networks.

4.3.1 Personal Widget Library

The environment supports personal widget libraries, where it is possible to save widgets directly at Web navigation time, thus reducing the effort for creating new composed applications. Users can

[7] http://todomvcapp.meteor.com

load any previously saved widget from their library in order to create their mashup. A personal library is created when the user registers to the platform, and is not accessible to other users.
For adding a widget to the library, a button on the widget upper bar opens the form for adding the widget, and providing related information. Specifying a short widget title is mandatory, while providing a longer description is recommended in order to give hints about how the widget can be used. In addition, it is possible to specify tags for providing categories that facilitate a widget search.

4.3.2 Mashup Management

Mashups are managed in a similar way to widgets, and are characterized by the following parameters: name, description, author, unique ID, creation and last modification date, set of relevant tags (to facilitate search within the repository), set of widgets belonging to the project.
Each mashup descriptor is saved in an XML file that complies with a specific XML Schema Definition (XSD). A folder tree is also created that stores HTML and CSS files constituting the widgets.
The Mashup Support allocates an individual working space to each user. An arbitrary number of subscribers can thus exploit the Mashup Environment (by creating, saving, loading widgets/mashups) at the same time.

4.3.3 Widget shared repository and social network support

The EUD environment also supports the possibility to share the created mashups in a centralized repository, where it is possible to search for and reuse mashups created by other people.
The first step to share a mashup is to save it in the shared repository. In order to do this, users have to fill in a form similar to the one that is used for saving the mashup in their personal library, specifying a name, description and a set of tags for providing categories to the application.
The saved mashup is accessible through two main channels: the first one is a search engine internal to the EUD environment. The second one is provided by the connection to a social network, and we considered Facebook for this purpose. Indeed, it is possible to access the EUD environment using the Facebook credentials. When a user shares a mashup, the EUD environment

posts the news on the user's Facebook wall, together with the application link. In addition, the shared application is enhanced with a set of social features that are shown in Figure15. The user can: i) rate the application from 1 to 5, ii) comment on the application, iii) share the link on his/her wall, iv) like the application and v) send a private message to friends with the application link.

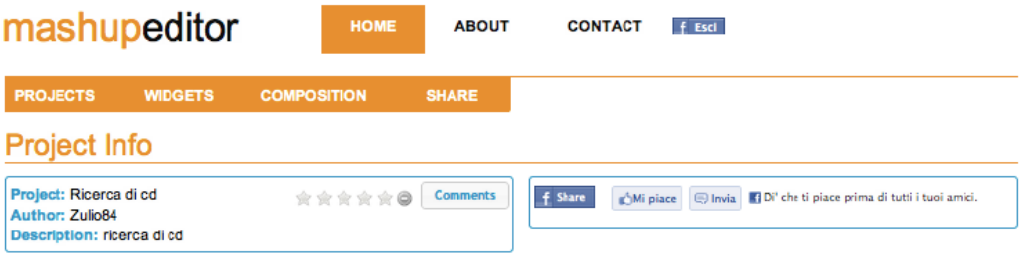

Fig. 15. Social features

The social features are available not only to the user who created the mashup, but they are especially useful for people who will reuse it. A study on how Web users share CoScripter scripts is presented in (Bogart et al., 2008), showing that even non-programmers can fruitfully build on top of macros defined by others. As discussed in (Cao et al., 2011), sharing is a way for users to get mutual help in starting to find a solution for a particular problem, as well as in refining their initial ideas. In general, social features stimulate communication among the different users and a collaborative enhancement of the different mashups.

5 EVALUATION

We have carried out two user tests for the evaluation of the mashup platform. The first one was more a formative evaluation, which provided also useful suggestions for some small improvements, while the second one considered a more engineered version of the platform. In the first user test, the aim was to assess the MashupEditor's features and to elicit users' informal feedback, which could be useful for improving the tool. The focus was particularly on understanding to what extent the graphical environment for exploiting the intelligent support facilitates the creation, composition, sharing, and execution of mashups. In the second evaluation,

we collected quantitative data for both post-task and post-test analysis using standard questionnaires (respectively NASA-TLX (Hart and Staveland, 1988) and SUS (Brooke, 1996)), together with a qualitative evaluation of the different environment features.

5.1 First User Test

Before starting the trial, each participant was first given a quick introduction about the MashupEditor, then s/he was provided with the scenario description.

In the proposed scenario, the user is a music collector and often buys CDs online. Before purchasing, s/he usually searches for information on interesting artists on Wikipedia. Then, s/he searches for their releases on LastFM (an online music catalogue) and finally buys the CDs from Amazon or eBay.

According to this scenario, we proposed a set of 4 tasks to accomplish, which allowed the users to test the different features. The tasks were selected in order to obtain feedback on real usage scenarios for the proposed environment.

At the end of each task, the test participants answered a set of questions, which allowed them to rate the different aspects we considered relevant for the features just used.

In the tasks the users had to: (i) create a new application/mashup consisting of two widgets taken from the Web, run it and save one of the widgets in their personal library; (ii) add a widget previously created by another user into their mashup, update the connections among the widgets and share the project; (iii) load a mashup previously shared by another user and modify it by adding a widget from their personal library; (iv) create a project and share it on Facebook, and finally load and run a shared project by another user on the social network.

Seventeen users, 10 females and 7 males, aged between 19 and 30 (avg.: 25.9, std.: 2.7) were involved in the study. None of them were programmers. Three participants held a Master, 6 a Bachelor and 8 a High School Degree. All of them declared to surf the Web mostly by means of a

PC, from 1 to 12 hours per day (avg.: 6.6, std.: 3.4). Sixteen users were members of one or more social networks, most of them were Facebook users.

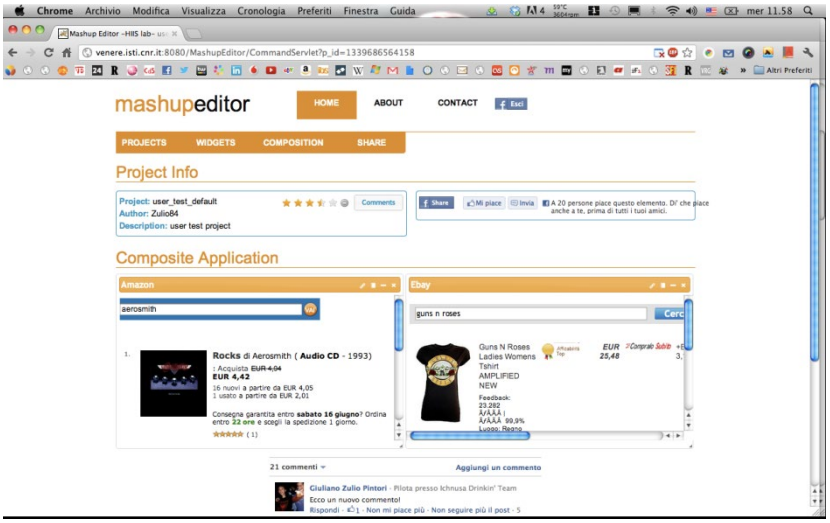

Fig. 16. Example mashup developed during the test.

After the trial, users answered an online questionnaire providing personal information and rating several aspects of the following MashupEditor features on a 5 points Likert scale (with 5 as most positive score). The questionnaire contained items for assessing different tools aspects.

The *Intuitiveness* aspect included the items related to widget creation/connection/loading, component selection, project loading/searching/sharing, and the corresponding averages were between 3.6 and 4.5, indicating a good comprehension of the mashup representation.

Satisfaction was related to widget connection visualization, personal widget library, hiding of connected components, project loading/running/rating, Facebook login/wall-post/project-loading/comments, and the corresponding averages were between 4.2 and 4.8. The users found the mashup features useful and they were overall satisfied by the resulting end-user developed application, considering the tasks at hand.

For assessing *Appropriateness*, we asked users to declare whether various features were considered to be appropriate. Facebook project sharing was perceived as appropriate by all the users, the mechanism to modify a shared project by 86% of them and the re-share project feature by 95% of them, while 68% of the users declared that it could be appropriate to integrate the MashupEditor with other social networks.

Regarding the perceived *Lack of Information*, 18% considered the amount of descriptive data shown in the window supporting search of mashup projects in the repository to be insufficient, and 12% declared that the mechanism to share and load a project should allow the project owner to prevent editing by other users.

The results from this user test were encouraging: the end-users without programming experience were able to identify the relevant parts of existing Web applications and connect them for solving the proposed scenarios. In general, the users involved in the study learned the platform functionalities in a short time, and their interaction improved as they became familiar with them.

From the post-test comments, it can also be concluded that users needed additional feedback in the MashupEditor UI for distinguishing and controlling the different Web applications visited through the proxy (those where it was possible to select components) and those visited directly. In addition, users found it annoying that page parts were highlighted even when they do not want to select them. Therefore, we modified the environment to include a list of the Web applications currently visited through the proxy, and we added the possibility to toggle and un-toggle the component selection for each Web application.

Having established that the copy-paste metaphor was suitable for allowing end users to manipulate programming concepts, we extended the MashupEditor with more advanced functionalities described in the previous sections, i.e. the possibility of connecting the output of one widget with the input of another one, and the Map Widget. Such features were introduced also considering the test users' expressed curiosity regarding the MashupEditor's ability to deal with modern dynamic Web sites.

5.2 Second User Test

The mashups created in the second study were more complex than the first study, due to the number of widgets involved and the kind of connections requested (copy-paste from input to input or from output to input). The objective of this evaluation was to assess the MashupEditor using standard questionnaires, together with a summative evaluation of its specific features.

5.2.1 Test design

The participants were given two tasks reflecting two possible scenarios where a Web mashup would allow speeding-up the tasks. For each task, the users were required to create and run a mashup through the MashupEditor.

In task A, users tried to find information for a trip. The following instructions were given:

You are planning a trip across several cities/countries and you need to collect information about weather, news (politics, events, etc.), local attractions/history and geography about your transit/destination places. Your goal is to retrieve relevant information about a place (news, weather, history, geography) by submitting a single query, i.e. by searching for that place on one single search engine and getting results about news, weather, history, geography at the same time. To this aim you are relying on the following sources (BBC, Wikipedia and WeatherOnLine), and will create a Mashup Widget for each source .

Task B was about a search through the background of the committee members of an international conference:

You are considering to submit an article to the MUM 2015 conference and would like to investigate the background of the committee members by looking at the titles of their main publications on Google Scholar. To speed up the search process, you are building a mashup. For each source you need to create a Mashup Widget.

Before starting the interaction, the users had to read a short introduction (half page) on the MashupEditor explaining its aims, and the instructions (one page) on how to interact with it. The users kept the instructions during the interaction and were allowed to examine them if needed. We also created eleven short (10 to 40 seconds) video tutorials, and put them in a folder. The instructions provided generic indications, such as “activate recording mode to start widgets connection”. The users could try to find how to access the various functionalities by themselves (i.e. by exploring the menus) or to watch the video tutorial associated to that functionality to get an example of use in a different mashup. For each underlined part in the instructions (i.e. for the main

functionalities), a video file was available and its name was consistent with the text underlined in the instructions.

Half of the users started from task A and then performed task B, while for the others the order was inverted in order to reduce the learning effect on the second task.

After each task, the users had to fill in a NASA TLX (Task Load Index) questionnaire declaring the perceived mental/physical/temporal demand, performance, effort and frustration on a 100 points scale with 5 point steps. At the end of the trial, each user completed a SUS (System Usability Scale) questionnaire and a questionnaire to rate various aspects of the system. The participants could also answer some open-ended questions to provide criticisms and suggestions for possible improvements.

We logged the timestamps of the main interaction events and recorded the screencast with the aim of quantifying and classifying the interaction errors. By analysing the screencasts we noted how many times each video tutorial had been watched, which gave an indication of the intuitiveness of the various functionalities.

5.2.2 Test participants

Seventeen users participated in the study. They were mainly recruited among the personnel of a research institute and among university students. Ten were male and 7 female, and their age ranged between 26 and 40 (avg.: 31.9, std.: 4.2). None of them had any relevant programming experience. Six participants held a Master, 6 a Bachelor and 4 a High School Degree. One held a PhD. Users declared to browse the Web between 2 and 15 hours daily (avg.: 5.6, std.: 3.4).

5.2.3 Test results

5.2.3.1 NASA TLX

NASA TLX results are reported in Table 1 and summarised in the plot in Figure 17.

Table 1: NASA TLX results

Factor	**Task A**					**Task B**				
	Avg	Med	SD	Max	Min	Avg	Med	SD	Max	Min
Mental Load	55.9	55	17.8	15	80	56.2	60	17.3	15	75
Physical Demand	17.1	15	13.9	0	50	16.5	10	16.7	0	60
Temporal Demand	35.9	35	19.5	0	75	33.2	30	19.1	5	75
Performance	49.4	50	15.3	25	70	73.2	75	18.4	35	100
Effort	70.9	75	10.8	50	85	53.8	65	20.5	10	75
Frustration	30.3	25	15.7	5	65	30.0	30	18.9	5	65

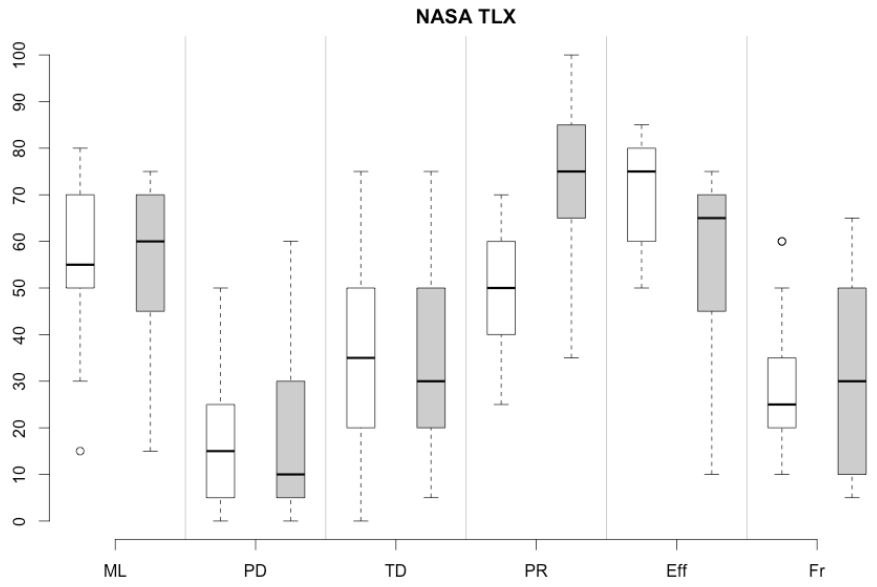

Fig. 17: NASA TLX scores.

We analysed the differences in the means for each one of the factors with a paired t-test ($\alpha = .05$). We found a significant difference between the two tasks only for Performance and Effort. For task B, the users were more satisfied with the performance ($p = .0002, c.i. = [12.09,35.55]$) and spent less effort ($p = .0016, c.i. = [3.64,30.48]$).

All users were able to complete all tasks. Analysing the different factors ratings, we notice that both tasks required a relevant mental load and effort to be accomplished. However, the levels of frustration and the performance ratings show that the tool support during the task was adequate and, especially for task B, the performance result compensated the task difficulty. In particular, the mashup resulting from task B was inspiring for our users, who appreciated the possibility to connect the output of a widget with the input of another one. As we better detail in the open-ended question section, when we request them to describe a mashup they would like to build with the tool, they included different situations where they get a list of items from one site and they look for the detail of each item in another website (e.g. compare an item price in a different website, get the technical specification for a specific instrument from the list in a music store etc.).

This confirms that with the MashupEditor interface we reached a good balance between the perceived complexity and the expressiveness of the inter-widget communication mechanism for users without programming experience: on the one hand they were both able to solve the proposed

problem and to imagine scenarios where they may exploit the tool for their needs; on the other hand, the mental load level indicates that including more advanced connection techniques may overwhelm them and increase the level of frustration.

5.2.3.2 SUS

In this case the individual scores ranged between 50 and 87.5 (avg.: 64.7, med: 65, std.: 8.5). Figure 18 shows more in detail the disaggregated scores for each questionnaire item.

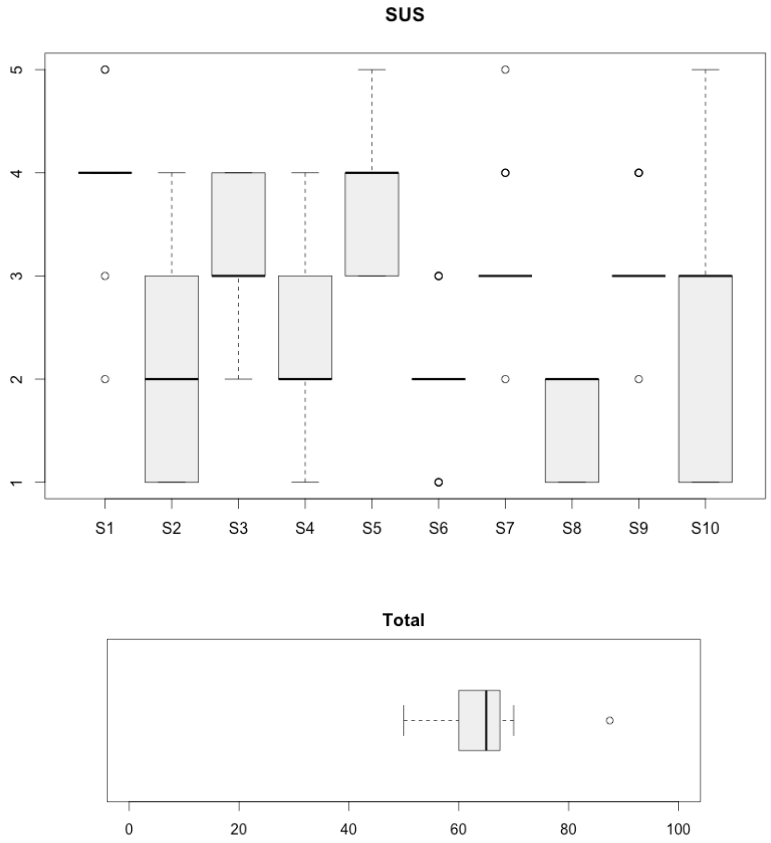

Fig. 18: Disaggregated SUS question scores (top) and aggregated score (bottom).

The users considered acceptable the overall usability of the tool, even if not completely satisfactory. In order to identify the problems encountered by users, we analysed the scores for each questionnaire item, which is shown in the box plot in Figure 18. We registered a high variability for question 2 (I found the system unnecessarily complex) and question 10 (I needed to learn a lot of things before I can get going with this system). This confirms again that users still spent some initial effort for getting started, while the answers to question 1 (I think I would like to use this system frequently) confirms that the users think that the tool is useful for them. In order to find an explanation for the variability of the ratings in questions 2 and 10, we analysed both the

usage of the video tutorials and the interaction recording, finding the critical points corresponding to the attitudes of the different users. We discuss the result of the analysis in the following sections.

5.2.3.3 General questions

The participants rated six relevant features of the MashupEditor on a 7 point Likert scale (with 7 as most positive score). The results are reported in Table 2 and are summarised in the plot in Figure 19:

Table 2: Ratings for General Questions

Feature	Avg.	Med.	SD	Max	Min
Widget creation mechanism	5.1	5	1.3	7	2
Elements selection mechanism	5.5	6	1.0	7	3
Copy-paste connection mechanism	5.5	6	0.9	7	3
Output-input connection	5.6	6	0.8	7	4
Mashup running mechanism	5.5	6	1.0	7	3
Selection forwarding mechanism	4.9	5	1.1	7	3

We can note that the core functionalities of the environment related to the composition mechanisms received satisfactory ratings, and the source of the perceived complexity is not related with any of them. We registered a higher variability for the widget creation mechanism and for the widget selection forwarding mechanism.

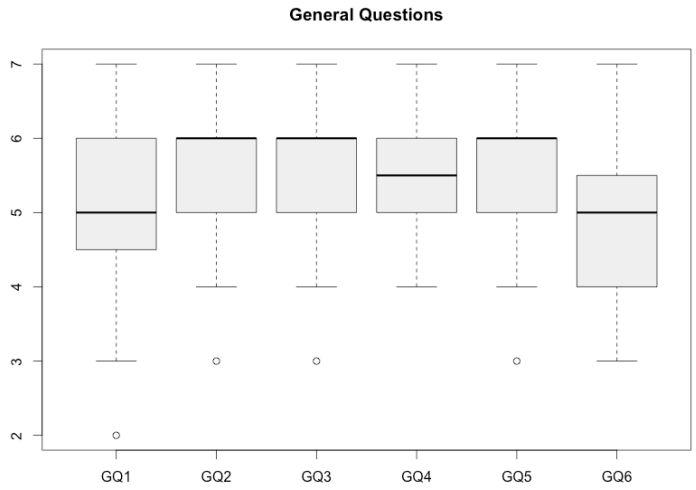

Fig. 19. General question scores

5.2.3.4 Temporal Performance

For completing task A, participants spent between 471 and 2030 seconds (avg.: 1094, std.: 392). Task B took between 391 and 1846 seconds (avg.: 955, std.: 474).

The task completion timer started as the user began reading the task description and ended when the user stopped interacting with the created mashup. The completion time thus includes any interval during which the user looked at the instructions and/or watched the video tutorials.

On average, it took 18 minutes to complete the task related to the travel scenario, requiring to create three widgets from the Web and to activate the map widget. The task about the conference committee members background took on average 16 minutes. When organizing the user study, our initial assumption was that the task B would be more complex due to the connection between an output and an input, which implies that users have first to identify a meaningful portion of text to copy from the output widget (in this case the names of committee members). However, the results show that the mean completion time for task B is slightly lower than for task A. If task A had two widgets (as task B) instead of three, the completion times would have been approximately the same. This may indicate that the copy-paste metaphor, when used by novices, performs equally when connecting two inputs or an output with an input.

5.2.3.5 Use of video tutorials

Fifteen users watched at least one video tutorial. Some users felt the need to watch the same video tutorial more than once. Details on the access to the video tutorials are reported in Figure 20.

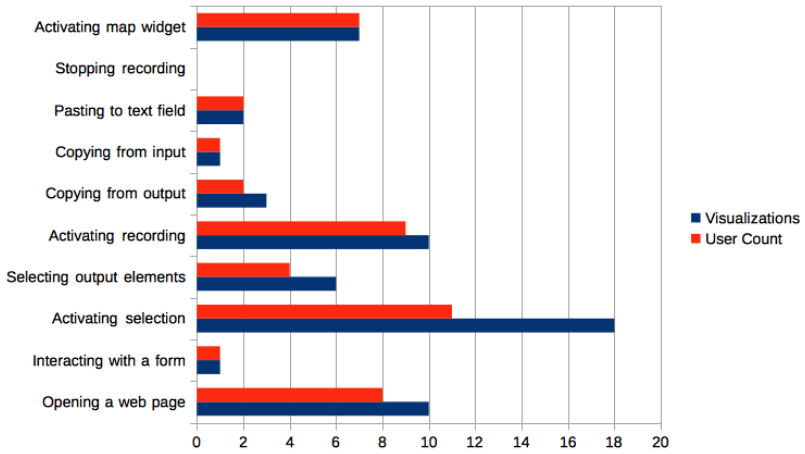

Fig. 20. Video tutorial usage summary

Users watched a video tutorial when they could not figure out how to access to a certain functionality of the MashupEditor. We thus assume that the more times a video was watched, the

less intuitive the access to that functionality is. The most accessed video tutorial was the one showing how to activate the elements selection on a Web page. The action consisted on switching from the currently browsed page to the page of the MashupEditor, finding the check box related to the current page and flagging it to enable elements selection on the page. Only six users were able to find out how to enable elements selection autonomously. This indicates that, to enable a functionality on the current page, switching between the current page and the MashupEditor is not intuitive.

In summary, we can identify the main sources of complexity for the users in two environment aspects. The first one is the page browsing mechanism that, for technical reasons, must start from the MashupEditor interface rather than from the usual Web browser address bar. There is not an immediate solution for this issue: it would be possible to provide extensions for the main browsers, but the end user should at least install and activate them in advance. The second source is the handling of the different Web pages when they are open for selecting the Web components, which may be difficult to manage. With respect to the first test version such feature was less annoying because we included in the editor the list of currently open Web applications with corresponding checkboxes to enable selection of their Web components, but it still requires further iterations for reaching an optimalusability. A possible solution is to show at the same time graphical previews of the Web pages in the MashupEditor (as happens in the default home page in some browsers), and to instrument each Web page browsed through the proxy with an additional control for enabling component selection.

5.2.3.6 Errors

By analysing the screencasts and the notes taken by the moderator, we were able to identify the errors made during the test by each user. We grouped them into five classes in order to simplify the analysis.

Type 1 – Semantics of information. Errors due to misunderstanding the semantics of the information to deal with: selection of wrong/useless elements (e.g. too little such as just a menu, or excessive amount such as the whole page).

Type 2 – Action order. Wrong order in the interaction: copy-pasting before activating recording mode for widget connection, activation/deactivation of recording mode for each single connection.

Type 3 – Missing steps. Lack of one or more interaction steps necessary for the mashup to be set-up and to run properly. This is the case of the user failing to provide a sample query to the form before creating a widget from the hosting Web page, or connecting only a subset of the mashup widgets (e.g. 2 instead of 3 or 4).

Type 4 – Wrong functionality. Errors of this type are due to the user accessing a wrong menu entry or button.

Type 5 – Wrong metaphor. Mistakes in the way a metaphor is used, such as trying to copy-paste parts of a Web page directly in the MashupEditor to create a widget, or trying to drag-and-drop a selection from a Web page to the MashupEditor.

Only three users completed the tasks without making any error. The remaining fourteen users made between 1 and 4 errors each. Details on the various types of errors are reported in Figure 21.

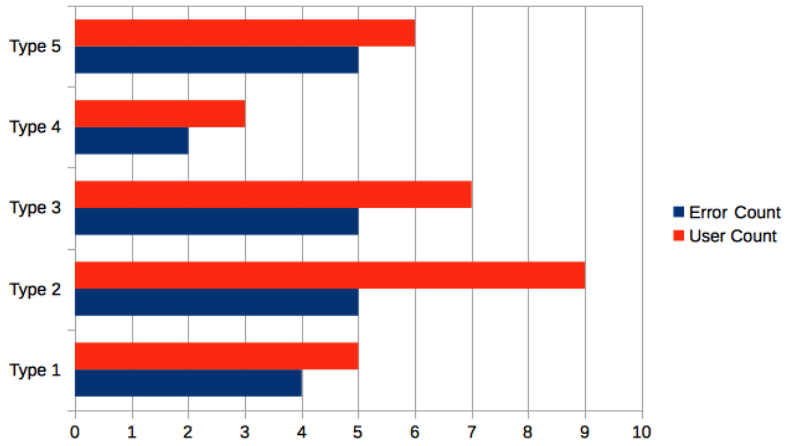

Fig. 21. Errors summary

The most frequent errors, i.e. Type 2 and 3, reveal the need for some interactive feedback in the MashupEditor listing the steps for setting up the mashup and indicating which ones still need to be performed (e.g., instructing a form, creating a widget, connecting widgets, etc.), as indicated in the answers to the open-ended questions (see the following subsection).

5.2.3.7 Open-Ended Questions

Four open-ended questions were posed.

Which example mashup would you like to build in the MashupEditor?

All the several possible example mashups indicated by the users can be actually built in the MashupEditor, meaning that users have properly understood the MashupEditor potentialities, such as e.g.: finding the same products in many e-commerce Web sites, searching for the same keyword in various blogs, searching for musical instruments information, availability, technical

specifications and tutorial at the same time), finding job positions through various engines, translating Web pages online. In addition, this indicates that our users did not identify situations where they would need more complex connections between the different data sources: none of them asked for including string manipulation for the inter-widget communication.

What are the positive aspects you found while using the MashupEditor?

Possibility to create/move/resize widgets, simplicity in learning the main features, information aggregation capabilities and time saving in multiple searches, were among the positive aspects highlighted. The possibility of selecting which parts of Web pages to add to a widget was also mentioned.

What are the negative aspects you found while using the MashupEditor?

According to some users, switching repeatedly between the browsed Web page and the MashupEditor is confusing and this confirms our findings from the questionnaires and video tutorial usage data. Additional criticisms were about the low intuitiveness of the way to access some functionalities because of the lack of tooltips in menu entries, and about the impossibility to copy-paste directly from an existing Web page to the MashupEditor.

Do you have any suggestion for improving our mashup platform?

Some users would like to work on a single window (i.e. the one of the MashupEditor), seeing Web pages and selecting components on that window.

Users also suggested to improve menu entries consistency, add a contextual menu to be activated with right click and to include a "guided" mode to create mashups for novices.

In conclusion, the summative evaluation results show that the copy-paste metaphor for creating applications with the MashupEditor has a good balance between its simplicity and its expressive power if we consider end-users without programming skills. Further work is needed for helping the users in manipulating effectively the different Web pages used as data sources.

6 CONCLUSIONS AND FUTURE WORK

We have presented an environment for creating Web mashups, described its architecture and how its intelligent support works, and shown various examples. By exploiting a copy-paste metaphor and underlying intelligent support, the environment allows direct manipulation of existing Web applications in order to create new ones without requiring any particular programming skill.

Moreover, the environment provides support for sharing and collaborating in Mashup development. Shared mashups can be run by others and/or modified and, in turn, shared again.

The enabling platform is Web-based and accessible from any browser. As already mentioned, we have opted for an architecture based on an annotation proxy, which supports access through any browser. We indeed aim to make our environment compatible with an as wide as possible set of devices and browsers, without requiring client side add-ons. In general, the proxy has shown good reliability in handling even Web pages using more recent dynamic techniques (such as AJAX scripts, websockets, and new client-side frameworks). There are still a few very particular cases in which there may be some issues in properly annotating links within the JavaScript code when the links are created through concatenations of arbitrary pieces of string fragments.

Two user studies have been carried out aiming to assess the usability of the environment functionalities, including its social networking features and to collect suggestions for possible improvements. The first test provided encouraging feedback and we collected a set of suggestions for some improvements, also in terms of the coverage of the types of Web applications that should be supported. The improved version of the environment was then used for a second test in which we focused more on the usability of the resulting environment, also using widely adopted usability metrics. Overall, the results seem positive, also considering the underlying complexity of the tasks supported (development of new Web applications by people without programming experience). Despite the fact that users did not have experience in the field, they were all able to achieve the proposed goals, with limited number of errors. The environment was perceived positively and we also collected some suggestions for small improvements that can further improve the corresponding user experience.

Future developments will include support for facilitating debugging of the resulting applications by non-experts. We also consider to investigate how the copy-paste metaphor can be applied in multi-device mashup. In this area (Husmann et al., 2014) have proposed MultiMasher, a tool providing architectural and visual support for multi-device mashups, which is suitable for users

with programming knowledge. We want to investigate whether the introduction of our copy-paste metaphor can enable even people without such expertise to create their mashups by exploiting multiple devices.

Although artefacts (widgets or their compositions) can be shared among users, the MashupEditor was not initially conceived as a collaborative tool. Further developments may be dedicated to creating a collaborative version. To this end, we could take previous work as reference. For instance, non-invasive extension of existing applications with awareness widgets was tackled in (Heinrich, Grüneberger et al., 2012), and management of concurrency while converting single-user into multiuser applications was considered in (Heinrich, Lehmann et al., 2012). Although such works did not address mashups, the proposed techniques can be considered and extended for combined use with our metaphor for creating a multiuser version of the MashupEditor, where multiple users can cooperate at the same time in the development of the same Web mashup.

REFERENCES

Aghaee, S., Pautasso, C., 2014. End-User Development of Mashups with NaturalMash. J. Vis. Lang. Comput. 25(4), pp. 414-432.

Bogart, C., Burnett, M., Cypher, A., and Scaffidi, C., 2008. End-user programming in the wild: A field study of CoScripter scripts. IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 39-46.

Broke, J., 1996. SUS. A "quick and dirty" usability Scale, In P. Jordan, B. Thomas, and B. Weerdmeester, editors, Usability Evaluation in Industry, Taylor & Francis, pp. 189-194.

Cai, D., Yu, S., Wen, J., Ma, W., 2003. Extracting content structure for Web pages based on visual representation. In Proceedings of the 5th Asia-Pacific Web conference on Web technologies and applications (APWeb'03), Xiaofang Zhou, Maria E. Orlowska, and Yanchun Zhang (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 406-417.

Cao, J., Fleming, S. D., Burnett, M., 2011. An Exploration of Design Opportunities for Gardening End-User Programmers Ideas, IEEE Symposium on Visual Languages and Human-Centric Computing, Pittsburgh, U.S.A., 2011, pp. 35-42.

Cypher, A., 2010. End user programming on the Web. In *No code required: Giving Users Tools to Transform the Web*. Morgan Kaufmann, pp. 3-21.

Cypher, A., Dontcheva, M., Lau T., Nichols J. (Eds.), 2010. No Code required, Giving Users Tools to Transform the Web, Morgan Kaufmann, 2010.

Díaz, O., Arellano, C., Iturrioz, J., 2010. Interfaces for scripting: making Greasemonkey scripts resilient to Website upgrades. In *Proceedings of ICWE 2010*, Springer-Verlag Berlin, Heidelberg, pp. 233-247.

Dontcheva, M., Drucker, S.M, Salesin, D., Cohen, M.F., 2006. Summarizing personal Web browsing sessions. In Proceedings of the 19th annual ACM symposium on User interface software and technology (UIST '06). ACM, New York, NY, USA, pp. 115-124.

Dontcheva, M., Drucker, S.M, Salesin, D., Cohen, M.F., 2007. Relations, cards, and search templates: user-guided Web data integration and layout. In Proceedings of the 20th annual ACM symposium on User interface software and technology (UIST '07). ACM, New York, NY, USA, pp. 61-70.

Faaborg, A., Lieberman, H., 2006. A Goal-Oriented Web Browser. In Proceedings of CHI '06, ACM Press, 2006, pp. 51 - 760.

Ghiani, G., Paternò, F., Spano, L.D., 2011. Creating Mashups by Direct Manipulation of Existing Web Applications. In *Proceedings of IS-EUD 2011*, LNCS 6654, pp. 42-52, Springer-Verlag.

Hart, S. G., Staveland, L. E., 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In Advances in Psychology, Vol. 52, pp. 139-183.

Hartmann, B., Wu, L., Collins K., Klemmer, S.R., 2007. Programming by a sample: rapidly creating Web applications with d.mix. In Proceedings of UIST 2007, pp. 241-250.

Heinrich, M., Grüneberger, F.J., Springer, T., Gaedke, M., 2012. Reusable Awareness Widgets for Collaborative Web Applications – A Non-invasive Approach. In Proceedings of ICWE 2012, LNCS 7387, Springer-Verlag Berlin Heidelberg, pp. 1-15.

Heinrich, M., Lehmann, F., Springer, T., Gaedke, M., 2012. Exploiting single-user web applications for shared editing: a generic transformation approach. In Proceedings of WWW 2012, ACM New York, pp. 1057-1066.

Hoyer, V., Gilles, F., Janner, T., Stanoevska-Slabeva, K., 2009. SAP Research RoofTop Marketplace: Putting a Face on Service-Oriented Architectures. In Proceedings of ICWS 2009, IEEE, pp. 107-114.

Husmann, M., Nebeling, M., Norrie, M.C. 2014. MultiMasher: Providing Architectural Support and Visual Tools for Multi-device Mashups. In Proceedings of WISE 2014 WISE 2014, LNCS 8787, Springer International Publishing Switzerland, pp. 199–214.

Huynh, D., Miller, R.C., Karger, D., 2006 Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. In Proceedings of UIST 2006, pp. 125-134.

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A.F., Burnett, M.M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B.A., Rosson, M.B., Rothermel, G., Shaw, M., Wiedenbeck, S., 2011. The state of the art in end-user software engineering. ACM Comput. Surv. 43(3): 21 (2011).

Leshed, G., Haber, E.M., Mattews, T., Lau, T.A., 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In Proceedings of CHI 2008, pp. 1719-1728.
Lieberman, H., Paternò, F., Wulf, W. (Eds), 2006. End-User Development, Springer Verlag, ISBN-10 1-4020-4220-5, 2006.

Lin, J., Wong, J., Nichols, J., Cypher, A., Lau, T.A., 2009. End-user programming of Mashups with Vegemite. In Proceedings of IUI '09, ACM New York, 2009, pp. 97-106.

Matera M., Picozzi M., Pini M., Tonazzo M.: PEUDOM: A Mashup Platform for the End User Development of Common Information Spaces. ICWE 2013: 494-497

Microsoft Popfly. http://www.popfly.ms/

Miller, R.C., Bolin, M., Chilton, L.B., Little, G., Webber, M., Yu, C.-H., 2010. Rewriting the Web with chickenfoot. In *No code required: Giving Users Tools to Transform the Web*. Morgan Kaufmann, 39-62.

Nebeling, M., Leone, S., Norrie, M.C., 2012. Crowdsourced Web engineering and design. In Proceedings of ICWE 2012, Springer-Verlag Berlin, Heidelberg, pp. 31-45.

Nichols J., Hua, Z., Barton, J., 2008. Highlight: a system for creating and deploying mobile Web applications. In Proceedings of the 21st annual ACM symposium on User interface software and technology (UIST '08). ACM, New York, NY, USA, pp. 249-258. DOI=10.1145/1449715.1449757 http://doi.acm.org/10.1145/1449715.1449757

Nichols, J., Lau, T., 2008. Mobilization by demonstration: using traces to re-author existing Web sites. In Proceedings of the 13th international conference on Intelligent user interfaces (IUI '08). ACM, New York, NY, USA, pp. 149-158.

Soi, S., Daniel, F., Casati, F., 2014. Conceptual Development of Custom, Domain-Specific Mashup Platforms. TWEB 8(3): 14:1-14:35..

Soriano, J., Lizcano, D., Cañas, M.A., Reyes, M., Hierro, J.J., 2007. Fostering innovation in a mashup-oriented enterprise 2.0 collaboration environment. In *System and Information Science Notes*, 2007.07.024, Vol 1, No 1, Jul 2007, pp. 62-69.

Tschudnowsky, A., Pietschmann, S., Niederhausen, M., Hertel, M., Gaedke, M, 2014. From Choreographed to Hybrid User Interface Mashups: A Generic Transformation Approach. In Proceedings of ICWE 2014, LNCS 8541, Springer International Publishing Switzerland 2014, pp.145-162.

WSO2 Mashup Server. http://wso2.com/products/mashup-server/

APPENDIX - UNDERLYING INTELLIGENT SUPPORT

In this section we further detail the underlying intelligence in the mashup environment. Each one of the following subsections describes the main steps leading to mashup creation/execution, detailing how intelligence is implemented.

Widget Creation from Page Content

After interacting (e.g., browsing, searching, etc.) with the Web application the user can enable elements selection from the mashup environment and choose which ones to transfer. The selected elements are extracted from the Web page and forwarded to the mashup environment in order to be inserted as a widget. The extraction starts on the client side, but most of the process is carried out on the server side, which is able to support better performance.

The extraction starts when the user clicks the button for sending the selection to the environment. A JavaScript procedure serializes the page, including its state (e.g. form fields content) and forwards the result to the mashup support, together with the list of selected elements. Only JavaScript functions are used to serialize/send the page, thus making the procedure compatible with standard browsers.

The mashup support first creates a server-side object from the incoming serialization, corresponding to the document received by the client. Then, it analyses this representation in order to find all the selected elements on the page and mark them with a flag. For each selected element, all the ancestors and the successors in the page hierarchical structure are marked as well.

After labelling, a pruning step removes all the elements but the marked ones from the document. Keeping all the successors for each selected element allows maintaining, in the partial page (i.e. in the mashup widget), the element content as it appears in the original one. For instance, in the original page, the user may select a search result item (e.g., the first one) by clicking on the main DIV element containing it. A TABLE inside the selected DIV, e.g. to display price/availability/shipping time, is also considered to be part of the user selection.

In our initial experiments, we included a reduction phase to avoid considering the ancestor elements in the path between the selection and the document root. This simplifies the partial

document, but the layout of the resulting widget is often affected because of the loss of information (i.e. style attributes can be inherited of the ancestor elements). Thus, we have opted to forego the reduction phase in order to keep as much style information as possible in the partial document. The complexity of page manipulation is the main reason why we opted for performing this procedure server side.

Figure 22 shows an example of the (sub-)tree resulting from the selected elements: only the two DIVs identified as “item_a” and “sugg_2” (circled with full lines) have been explicitly selected by the user from the original page. However, the extraction procedure includes their ancestors and successors (circled with dotted lines) in the partial tree that will set up the widget content. Thus, in general, the set of elements belonging to the widget includes the ones explicitly selected by the user from the original page, but is not limited to them.

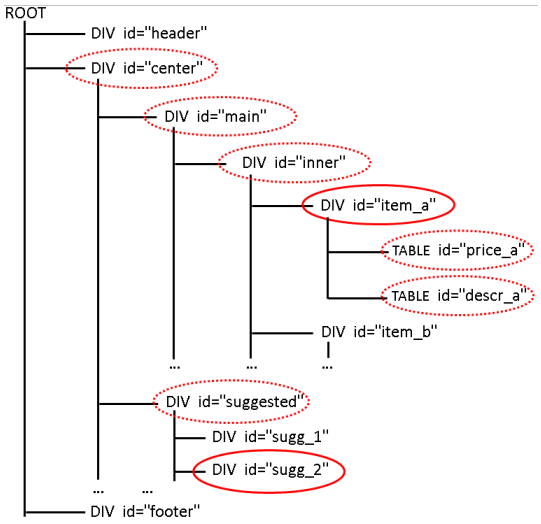

Fig. 22. An example page tree, the selected elements and the associated elements that will be included in the resulting sub-tree.

Recording user actions to connect widgets

Previous work addressing the issue of how to connect components (Ghiani et al., 2011) required users to explicitly associate the form input fields with the HTTP parameters in order to connect input/output components of the newly created widgets. The concerns that arose during previous user studies suggested that average users are not likely to be aware of such underlying HTTP parameters, nor of how they may relate to each other.

We have thus identified a solution aiming to simplify the widget connection process as much as possible. The aim is to require the user to deal only with the connections across the widgets, without having to cope with how the connections are made at the implementation level. To this end, we have relied on the widely known copy and paste metaphor. Having, for instance, widgets A and B both with input and output components, user actions of copying something from the user input element of widget A into the user input element of widget B is interpreted by the system as the user wishing that the input of A be used also as input to B, thus requesting output from both. Action recognition relies on special events provided by the jQuery JavaScript library, i.e. oncopy and onpaste that allow the environment to monitor user interaction. The actions are sent to the Mashup Support through a command servlet. In the Mashup Support, such actions are finally analysed and "translated" into a connection object that is added to the list of connections of the mashup descriptor structure. Whenever a request coming from a widget is detected (e.g., a new search), the Mashup Support:

- updates the content of the originating widget (i.e. the one making the request) by querying the application server and filtering the response;
- accesses the mashup descriptor in order to find out the widgets connected to the originating one;
- maps the actual parameters (i.e. those coming from the originating widget) into the formal parameters of the connected widgets, thus creating one query for each connected widget;
- updates the content of the connected widgets by forwarding the queries to the application servers and filtering the responses.

Intelligent Filtering when Executing the Mashup

The elements selected by the user on a Web page during widget creation are described by a list of corresponding HTML identifiers. At widget creation time, such information is used by the mashup support to produce the "partial" page for the widget. However, that information is not sufficient

for finding the corresponding elements in similar pages that may be generated during the mashup execution. The reason is that, across similar pages, elements with the same semantics can have different ids. Indeed, while the main parts of the page (e.g. upper bar, footer, central container, search form, etc.) are likely to have consistent ids within the same Web application, others such as the search results of an online store might have an id that depends on the item on sale (and not on the position of the item as displayed in the list). Thus, the Mashup support must be able to understand what elements generated by the server side functionalities are relevant for the widgets. More reliable mechanisms that go beyond a simple id-based filtering are thus needed to ensure relevant filtering mechanisms on the widget.

To this end, we have set up a filtering algorithm that provides the user with consistent output by exploiting the following information: full path within the document tree (including the position with respect to sibling nodes for each element between the root and the selected one), type, class, in addition to the element id. The algorithm is executed every time the user runs the mashup by performing a new query, and is executed once for each widget belonging to the mashup.

For each involved widget the algorithm takes as input the incoming page, generated by the application server according to the query, the original page and the selection list, which contains the ids of the elements explicitly selected by the user from the original page when creating the widget.

The main cycle of the algorithm consists of determining whether there is an element matching the id in the incoming page for each element in the selection list. If the incoming page contains an element matching the id, then that element is considered to be equivalent to the selected one. Otherwise, the algorithm aims to find the element in the page that best matches it. The best match is calculated by comparing the path of the selection list element in the original page with the structure of the current page. The path is defined as the sequence of elements from the root to the element considered. The comparison is based on a similarity concept that iteratively takes into account several features of the ancestor elements (attributes, type, sibling position, as explained in

the following). The first check of the algorithm aims to identify whether there are containers with the same ids of the element considered in the two pages. To this end, two variables are used within the inner cycle described in the following: page element refers to the currently examined element of the incoming page, and path node is the currently considered element in the path of the original page.

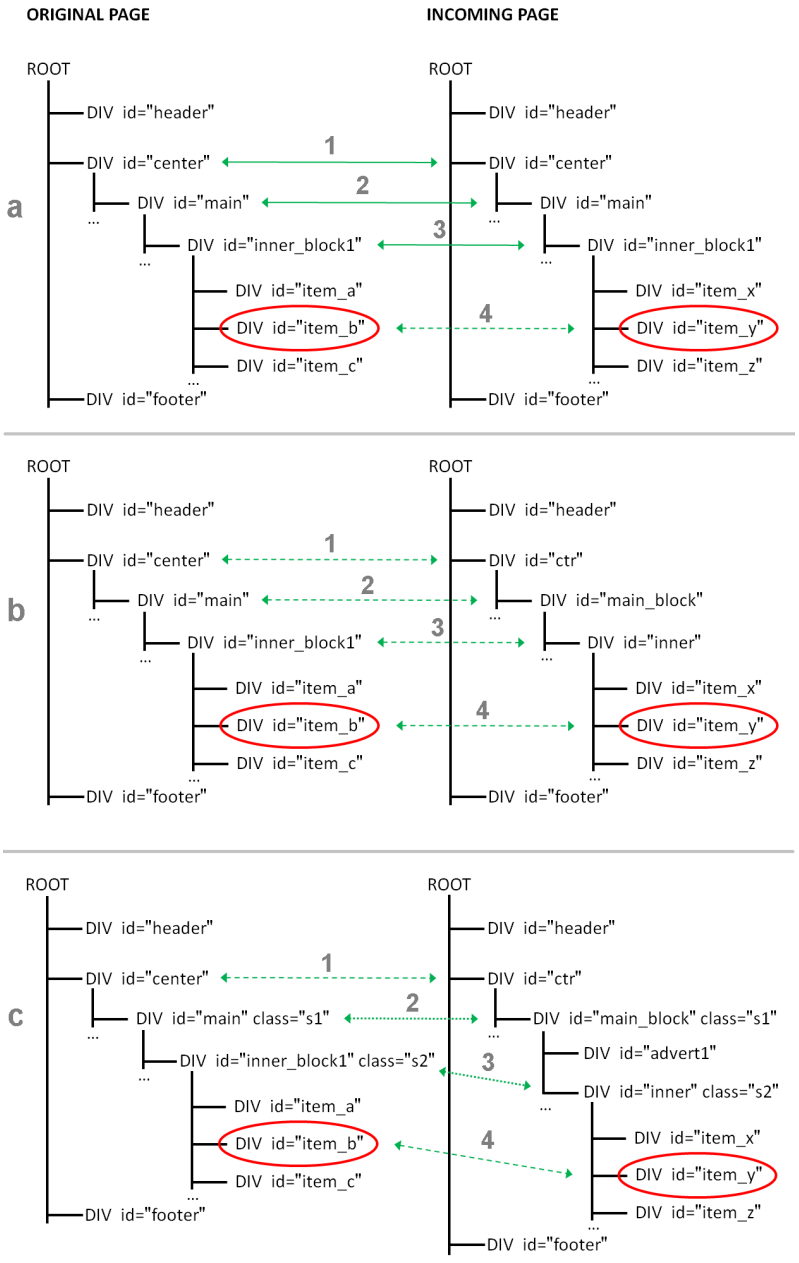

Fig. 23. Three different example cases for the filtering algorithm.

Initially, the current path node is the first node in the path between the original page root and the selection list element in the current iteration, and the current incoming page element is the incoming page root. The main steps are the following:

1) The current path node is compared with the children of the current incoming page element. Two different situations can occur:
    - A child of the current incoming page element has the same id as the current path node (see Figure 23 a – steps 1, 2, 3), in this case the child element is assigned to the page element variable.
    - None of the current page element children have the same id as the current path node. In this case, the most similar child of the current page element is selected according to a "matching score": the highest score is given to a type match (i.e., the element tag name), and a lower one to a class match. One of the following cases can occur:
        - One or more children of the current incoming page element have a positive score: the one with the greatest score is assigned to the page element variable (see Figure 23 c – steps 2, 3).
        - If the matching score is zero for all the current page element children (i.e. the current page element has no child with type and class matching with the current path node), then the algorithm considers the sibling position: the child of the current incoming page element having the same sibling index of the current path node is assigned to the page element variable (see Figure 23 a – step 4; b – steps 1, 2, 3, 4; c – steps 1, 4). The sibling index is also considered when all the children have the same matching score (e.g. identical tag name and/or class attribute).
2) If there are still nodes in the path, then the subsequent node in the path is assigned to the path node variable, and the algorithm continues from 1. Otherwise, the current page element is considered to correspond to the one selected by the user.

This algorithm provides the list of elements of the incoming page that correspond to the elements of the selection list. In order to create the partial incoming page that will be displayed on the widget, the procedure described before is executed: for each corresponding element, the ancestors and the successors are included in the partial page. The resulting partial page is then provided back to the mashup environment, which displays it in the associated widget.

Three typical situations for the filtering algorithm are shown in Figure 23. The left and right part of the figure show the simplified structures for the original page (from where the element was selected) and the right part shows the incoming page (from where the corresponding element has to be extracted), respectively. All the cases considered are characterized by the lack of id correspondence between the element selected on the original page (id="item_b") and the corresponding one in the incoming page (id="item_y").

The path of the selection list element considered consists of all the elements between the node and the document root in the original page: in the a case, the full path consists of the three DIVs "center", "main" and "inner_block1" as id. Full lines express (ancestor) matching by id, dashed lines indicate matching by sibling position, and class attribute matching is indicated by dotted lines.

In the *a* situation, the filtering algorithm assumes in the first three iterations that respectively the DIVs "center", "main" and "inner_block1" of the original page match with the DIVs of the incoming page having the same id. In the last iteration, no DIV with id="item_b" is found in the incoming page. In addition, it is useless to compare the tag name and/or the class attribute because all the children have the same tag name and the class is not defined. The filtering thus exploits the sibling position to select the element.

In the *b* case, the id matching always fails. The filtering thus relies on the sibling position for identifying the three nodes in the path, as well as for the selected element.

In the *c* case the id matching also fails at any step. The first matching is given by the sibling position of the first node, as no class attribute is defined. In order to find the second corresponding element in the path, it is possible to rely on the class matching (class="s1"). The same applies to the third node in the path (class="s2"). Sibling matching is done in the last step.

An approach similar to ours was adopted by Sifter (Huynh et al., 2006). Selection in Sifter is also done by highlighting/clicking HTML containers, links and texts directly on the visited Web page. After successful recognition, the system is able to manage queries to the application server and to merge the results in order to provide the user with desired information formatted in the preferred manner. Sifter's authors report that the delay for filtering few tens of result is around 30 seconds. Since most computations and communications, in our case, are managed server side (i.e. in the proxy), we achieved a more reasonable response time even when the mashup is made up by several applications. The Sifter filtering algorithm keeps track of the XPath for the selected nodes in order to identify corresponding nodes in the incoming pages. Our approach is more general: our intelligent algorithm, rather than explicitly saving the original XPath, considers the DOM of the original page. This allows our environment to implicitly maintain additional information about the path to the desired node, such as the sibling position of all the ancestors. When iteratively exploring the incoming page tree in search of the best intermediate candidate node (i.e. the current node in the path), our algorithm does not necessarily expect to find a node with the same tag name as the one in the original path. Indeed, if there is no tag name matching, the algorithm considers the best intermediate candidate according to a similarity score based on class name and/or sibling position matching. The reason for this choice is that modern Web pages, though often generated through predefined templates, tend to have some differences. For instance, HTML structures slightly differ according to the type of item they refer to (e.g., books vs. DVDs).

Differently from Sifter, our solution is thus more general since it can be more flexibly applied to a wide range of Web applications. For instance, the user may connect the search form of a news Web site with the main container of a weather Web site. While searching for a city, the mashup

would show news about that city and the weather conditions (e.g. temperatures, humidity) for that place. If the name of a country is instead prompted in the news form, the weather widget would show the country map annotated with temperatures of cities/regions along the country. Thus, the connected weather widget would still be able to work though the type of content displayed is rather different (a table containing detailed weather conditions vs. a map summarizing the country conditions).