A UI-Centric Approach for the End-User Development of Multidevice Mashups

CINZIA CAPPIELLO, MARISTELLA MATERA, and MATTEO PICOZZI, Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano

In recent years, models, composition paradigms, and tools for mashup development have been proposed to support the integration of information sources, services and APIs available on the Web. The challenge is to provide a gate to a "programmable Web," where end users are allowed to construct easily composite applications that merge content and functions so as to satisfy the long tail of their specific needs. The approaches proposed so far do not fully accommodate this vision. This article, therefore, proposes a mashup development framework that is oriented toward the End-User Development. Given the fundamental role of user interfaces (UIs) as a medium easily understandable by the end users, the proposed approach is characterized by UI-centric models able to support a WYSIWYG (What You See Is What You Get) specification of data integration and service orchestration. It, therefore, contributes to the definition of adequate abstractions that, by hiding the technology and implementation complexity, can be adopted by the end users in a kind of "democratic" paradigm for mashup development. This article also shows how model-to-code generative techniques translate models into application schemas, which in turn guide the dynamic instantiation of the composite applications at runtime. This is achieved through lightweight execution environments that can be deployed on the Web and on mobile devices to support the pervasive use of the created applications.

Categories and Subject Descriptors: D.1.7 [**Visual Programming**]: User Interfaces; D.2.6 [**Programming Environments**]: Interactive Environments; D.2.2 [**Design Tools and Techniques**]: Computer-Aided Software Engineering; H.3.5 [**Online Information Services**]: Web-Based Services; H.5.3 [**Group and Organization Interfaces**]: Web-Based Interaction

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Mashups, model-driven mashup development, end-user development, multidevice mashups, data fusion, personal information management, Web interfaces

1. INTRODUCTION

Past years have seen an evolution in the way information seeking applications are constructed and deliver responses to users' needs. Also enabled by the huge availability of online resources, one rising trend is to allow the users, not necessarily skilled programmers, to get rapid access to diverse resources offering functionality and data on the Web, and create new value by integrating them into simple but situated applications, the so-called *mashups* [Daniel and Matera 2014]. The proliferation of mobile devices,

Authors' addresses: C. Cappiello, M. Matera, M. Picozzi, Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Piazza Leonardo da Vinci, 32 - 20133 - Milano, Italy; emails: {cinzia.cappiello, maristella.matera, matteo.picozzi}@polimi.it.

capable of running software applications pervasively, is also fueling the end users' desire of participating in the development of their own artifacts to satisfy the "long tail" of specific and unexpected information needs not always addressed by common applications. The scenario emerging around mashup development is, therefore, characterized by a strong potential for user-centric innovation that promotes users as producers of significant value. Despite this potential, the research on Web mashups, and especially on composition paradigms and tools, has not produced substantial improvements for the end users. The consequence is that still only few experts are able to create their own applications by programming manually the service integration [Namoun et al. 2010a].

We directly experienced some limits of mashup development practices in the context of the research we have been conducting in recent years. We have tried to define composition paradigms and tools through which specific end-user communities, exploiting the potential of mashup technologies, could construct applications responding to the flexibility of tasks characterizing their working environments [Cappiello et al. 2011b; Ardito et al. 2012]. Our research has addressed more than just unexperienced users; rather, we have, in general, investigated how paradigms for fast prototyping, making intensive use of high-level abstractions, could ease the access to the plethora of contents available online today. Through a series of user studies, we came to the conclusion that this goal can be reached only by methods and tools that hide technicalities to make technologies accessible to the end users. This need is even more accentuated in the mobile context, where even expert programmers need to get acquainted with very specific technologies for application development that also vary substantially depending on the target mobile device. Therefore, there is a need for composition paradigms that on one hand should abstract from implementation details and on the other hand should support different kinds of integration logics for the creation of full-fledged service-based applications running on different devices.

1.1. Contributions

In light of the previous considerations, we believe there are numerous opportunities for research and development in the area of End-User Development (EUD) of mashups. In our previous work, we investigated the integration of *user interface (UI) components* at the presentation layer [Yu et al. 2007; Cappiello et al. 2011a, 2011b], showing how Web mashups can be composed by synchronizing the UI of prepackaged applications. Some field studies that we conducted to observe real users using our tools [Ardito et al. 2012] highlighted the need for an "open platform," where users would be enabled to create new UI components by selecting pertinent data from different sources and by choosing how to visualize the integrated result set. This article, therefore, introduces a UI-centric composition paradigm that enables the creation of multidevice UI components and UI mashups. The focus is on the following issues:

—*User-driven development process*. We argue that the user-driven creation of mashups is more challenging than the provider-driven development of services. The main issue is to help users understand easily the features of the available services, the way they can be integrated, and the effect that each service may have on the overall composition. We, therefore, propose representations of services and of service composition that abstract from programmatic interfaces or communication protocols while they enhance role expressiveness. We investigated these aspects since the beginning of our research on Web mashups. This article further clarifies our perspective on EUD and provides a systematic and integrated view on the overall mashup life cycle and the set of modeling abstractions we have been defining around the core concepts of UI components and UI mashups.

—*UI-centric abstractions for resource composition*. We propose a new UI-centric model for the composition of data mashups, based on associating *data renderers*, exposed by Web services, to *visual renderers* available in some predefined *UI templates*. Creating a UI for an integrated dataset is one of the most challenging tasks in the construction of UI components and UI mashups; rarely, mashup tools provide facilities for UI definition. The UI templates and the composition model proposed in this article try to alleviate this task. They give life to an "integration-by-example" paradigm, through which the composers program the integration of heterogeneous resources by expressing at the UI level examples of what they would like to experience during the execution of the final application. With respect to our previous work, the composition model discussed in this article introduces several novelties—from the possibility for the end user to express visually data integration operations, to the new UI-centric flavor that we assign to these operations, to the possibility for the end users to define from scratch new UI components.

—*Multidevice deployment of composite resources*. We propose a *Domain-Specific Language (DSL)* that encapsulates the fundamental constructs of the composition model, yet abstracting from specific UI templates and execution environments. A model-driven engineering process is thus enabled to transform "examples" of data integration, visually defined by users, into *application schemas*, specified according to the DSL syntax, that can be interpreted and instantiated on multiple devices. The need for deploying mashups on different devices (tablet and smart phones, desktop PCs, multitouch screens) was identified during field studies that we conducted to validate our initial Web platform [Ardito et al. 2012]. This article illustrates the new modeling abstractions and a platform architecture that cover the new emerged requirements.

—*Lightweight integration of resources.* The main challenge of the visual composition paradigm illustrated in this article is not the definition of a new data integration or service orchestration model; rather, our approach especially aims to promote abstractions that (1) capture and simplify the most salient aspects of composition making them suitable for the average end users, (2) can be handled by lightweight architectures not requiring dedicated integration platforms and protocols, and (3) can be ported on different client devices (even the mobile ones with limited computing capabilities).

—*Proof of concepts.* We illustrate a reference architecture for the execution of UI components and UI mashups on Web and mobile platforms. Through performance tests, we show that the integration logic can be adopted even under the limitations posed by mobile devices. By reporting on the results of two user studies, we discuss (1) the suitability of the visual integration paradigm with respect to the capabilities of average end users and (2) the usability of the generated mashups. Through a systematic usability inspection, we quantify the usability of the composition paradigm, also taking into account the complexity of the user tasks.

1.2. Article Organization

This article is organized as follows. Section 2 introduces some background concepts and clarifies the rationale of our research. Section 3 illustrates our mashup development process and the way its different activities can be performed through interactive environments offered by our tools. Section 4 defines the modeling constructs, the integration models, and the algorithms on which our approach is based. Section 5 illustrates the architecture of the composition platform. Section 6 reports on the evaluation that we conducted to assess the technological feasibility of the integration techniques, and the usability of both the composition paradigm and the final generated applications. Section 7 summarizes the EUD principles our work is inspired by and the lessons we

learned from our research on EUD of mashups. Section 8 finally draws our conclusions and outlines our future work.

2. RATIONALE AND BACKGROUND

Web mashups are "composite" applications constructed by integrating ready-to-use functions and content exposed by public or private services and Web APIs. The mashup composition paradigm was initially exploited in the context of the consumer Web for creating applications rapidly by reusing programmable APIs and content scraped out from Web pages. Soon, the potential of such lightweight integration practice also emerged in other domains. For example, platforms for *enterprise mashups* [Jhingran 2006] were proposed as tools for the enterprise users—not necessarily technology-skilled—to compose their dashboards for process and data analysis.

Service composition has been traditionally covered by powerful standards and technologies that, however, can only be mastered by IT experts [Ro et al. 2008]. What makes mashup development different from plain Web service integration is the possibility, deriving from recent Web technologies, to merge ready-to-use resources at the client-side, thus with reduced efforts and without the need of complex integration platforms. Mashup development also emphasizes novel issues, such as the composition at different layers of the application stack of heterogeneous resources that make use of different technologies. In particular, the *integration at the presentation layer* is the very innovative aspect that mashups bring into the Web technology scenario. It enables the creation of full-fledged Web applications whose UI can be easily achieved by synchronizing the UIs of different ready-to-use components. Mashup development can thus be considered an alternative solution to service composition that can help realize the dream of a "programmable Web" [Maximilien et al. 2007] even by non-programmer users.

2.1. Mashups and User-Driven Innovation

Because of its intrinsic value as development practice to let end users produce new value, mashup composition is in line with the so-called "culture of participation" [Fischer 2009]; users are enabled to evolve from passive consumers of applications to active co-creators of new ideas, knowledge, and products. There is indeed a specific driver at the heart of the user participation to the mashup phenomenon: user driven innovation, that is, the desire and capability of users to develop their own things, to realize their own ideas, and to express their own creativity [von Hippel 2005].

According to recent works published in literature [Latzina and Beringer 2012], there is also an increasing need to replace fixed applications with elastic environments that can be shaped up flexibly, to accommodate different situational needs. New design principles are emerging [Latzina and Beringer 2012] to promote paradigms where end users can access contents and functions and flexibly use and compose such resources in several situations and across several applications and access devices. If the composition activity turns out to add significant new value, the advantage for the users is that they achieve effective applications matching exactly their needs. Additionally, the providers of the original resources can integrate the user innovation back into their core products [Iyer and Davenport 2008] and improve their services to fulfill users' requirements without the need of carrying out the iterative experimentation generally required to identify requirements and develop and test a new product. The end users are entirely in charge of these aspects because they are enabled to create solutions that closely meet their needs.

Such innovation potential is, however, not adequately supported by the approaches for mashup composition proposed so far [Daniel et al. 2011]. The research on mashups has been focusing on enabling technologies and standards, with little attention on

easing the mashup development process. In many cases, mashup creation still involves the manual programming of the service integration. Research teams and industrial players tried to define simplified composition paradigms, mostly based on visual notations and lightweight design and execution platforms running on the Web. However, many of such projects failed because of their inadequacy with respect to some key principles defined and experimentally validated in the EUD domain [Namoun et al. 2010a; Casati et al. 2012]. According to the EUD vision, enabling a larger class of users to create their own applications requires the availability of intuitive abstractions, interactive development tools, and a high level of assistance [Burnett et al. 2004; Liu et al. 2007]. In particular, to support the user-driven innovation potential, the challenge is to let users concentrate on the conception of new ideas, rather than on the technicalities beyond service composition. In other words, users should be enabled to easily access resources responding to personal needs, integrate them to compose new applications, and simply run such applications without worrying about what happens behind the scenes.

2.2. UI-Centric Composition of Mashups

Our claim is that EUD of mashups can be promoted by UI-centric approaches where composers are allowed to focus on user-oriented artifacts (i.e., user interfaces they are able to use rather than on programmatic interfaces). Nevertheless, operating at the UI level must offer the possibility to express how to integrate components so that programming constructs can be derived to guide the automatic generation and execution of the resulting application. Some projects (e.g., Dynvoker [Spillner et al. 2008] and SOA4All [Krummenacher et al. 2009]) already had a similar intuition and focused on easing the creation of UIs on top of services to provide a direct channel between end users and services. However, such approaches addressed the interaction with single services, but they did not investigate the integration of multiple services.

The integration dimension has been investigated by a considerable body of research on mashup tools, the so-called *mashup makers*. Such tools support the integration of components at different layers of the application stack, especially at the data and presentation layer. Many tools offer composition paradigms based on graphical notations that abstract relevant mashup development aspects and operations. The mashup composers define diagrams to express the internal logic of a mashup, without the need for writing code. The diagrams are then automatically translated into application code.

Among the most prominent proposals, *Yahoo!Pipes* (http://pipes.yahoo.com) is an online platform supporting the integration of RSS feeds through a dataflow composition language. Users specify the URI of some data sources into dedicated input boxes. Then, they connect such boxes and define diagrams, the so-called *pipes*, to specify how data parameters propagate from one source to another to achieve filtering and merge of the involved result sets. Different modeling constructs can be connected to determine the flow of parameters and also the order of activation of components. A preview of the mashup output is possible within the Web editor, where a panel shows the resulting RSS feeds in a textual format. Simple visualizations, in form of HTML tables or maps (in case of geolocalized feeds) can also be associated with the results. Ready pipes can be stored on the Yahoo!Pipes online repository and accessed via a unique URL, which allows one to execute the mashup and fetch the resulting RSS feeds.

JackBe Presto (http://jackbe.com/products) also adopts a pipes-like approach for data mashups but with a greater set of operators with respect to Yahoo!Pipes and a larger coverage of data source types. Its composition environment, *Presto Wires*, offers a diagrammatic notation to express several data transformation operators (e.g., filter, sort, join, group, annotate, merge, and split), on heterogeneous types of data sources. This notation is an abstraction of Enterprise Mashup Markup Language

(EMML), an XML-based DSL promoted by the Open Mashup Alliance (OMA, http://www.openmashup.org/) for the definition of portable and interoperable data mashups, with a special focus on enterprise mashups [Open Mashup Alliance (OMA) 2013]. Presto Wires allows the developer to choose from different data integration constructs and to drag and drop them onto a visual canvas. Similar to Yahoo!Pipes, it offers a preview panel showing the computed dataset. Complex processing logic can be also embedded as scripts, whereas conditional statements and expressions are coded in XPath. Finally, ready mashups expressed in EMML can be stored on the Presto server where they can be executed by an EMML engine. The output is an integrated dataset, whose visualization can be configured only outside Wires, in a dedicated visualization dashboard that supports a portal-like aggregation of UI widgets (the so-called *mashlets*) offering a chart-based rendering of data.

mashArt [Daniel et al. 2009] also enables the creation of graph-based mashup models. Different from other tools, it proposes an integration approach, called *universal integration*, which allows the integration of data, application logic, and UI components inside one and the same modeling environment. The integration logic is based on an event-driven, publish-subscribe paradigm [Yu et al. 2007], where operations of some components subscribe to the occurrence of events in other components. Components need to be prepackaged and wrapped to expose events and operations. Expressing the actual mashup logic then requires defining dataflow connectors that associate source events and target operations. The result is an XML-based specification of component orchestration. The layout of the mashup can be finally defined by coding manually, outside the tool, an HTML template with a set of placeholders for the display of UI components.

With respect to manual programming, the previous platforms certainly facilitate mashup development. However, to some extent, they still require the composer's understanding of the integration logic (e.g., dataflow, parameter coupling, and composition operator programming). In other words, such tools tried to supply abstractions for non-programmers, but they just provided diagrammatic representations of the technology-driven model for APIs invocation and composition. For example, in Yahoo!Pipes, each service is represented by its URI and the XPath expressions for content extraction, whereas in mashArt, each component is represented by means of the events and operations it exposes. Another critical aspect is that building the mashup UI requires its manual coding or the adoption of external tools.

Asking the users to define a dataflow or control-flow logic, even assuming they would be able to understand these technical concepts, is not realistic [Angeli et al. 2011]. Some studies about users' expectations and the usability of mashup composition environments [Namoun et al. 2010a] highlighted fundamental issues concerning the conceptual understanding of service composition. First of all, end users do not think about connecting services. They do not even know what a Web service is or how it can be invoked trough its programmatic interface, although they are able to understand what a service can offer if they interact with any kind of service UI. Additionally, they do not understand the difference between application design and execution. Therefore, they feel extremely disoriented when they are required to define diagrams that do not resemble at all the application they aim to build. This is true even when diagrams are meant to provide abstractions for technical concepts.

Taking into consideration all these issues, our research has been focusing on natural, visual composition paradigms for the integration of components at the presentation layer. By "natural," we mean that the composers deal with interactive artifacts that are closer to their mental model of the application they want to construct. The composition paradigm requires acting directly on the user interface of the mashup in a kind of live programming paradigm where each composition action has a visual feedback on the

artifact under construction. In our previous work [Cappiello et al. 2011b], we showed how the end users can be enabled to compose Web mashups by synchronizing the behavior of UI components. In Yu et al. [2007], we defined UI components as JavaScript/HTML stand-alone applications that can be instantiated and run inside any common Web browser and that, unlike Web services or data sources, are equipped with a UI that enables the interaction with the underlying service via standard HTML. The characteristic of such class of components is that their UI allows users to interactively navigate and manipulate the component's content, also invoking business logic operations. UI components are in line with the Widget standard [Càceres 2012] recently proposed by the W3C. However, whereas the W3C standard just focuses on formats and metadata for packaging Web apps, UI components also expose an event-driven, publish-subscribe logic that gives them the capability to react (i.e., to enact operations that change their state) to events raised by other components. Events generate output parameters, which become input parameters for the coupled operations. Parameter passing thus enables an update of the status of the target component.

Other approaches successively proposed similar composition paradigms. For example, the *FAST* mashup framework [Lizcano et al. 2013] allows users to define "narrative" descriptions of the mashup they want to compose, and the composition environment, based on natural language processing techniques, identifies keywords and generates suggestions on the components to be used. The addition of components into the composition workspace is possible in a drag&drop fashion, and the added components are immediately executed, interactively showing their UI and dataset. A natural language specification of mashup requirements is also adopted by the *NaturalMash* tool [Aghaee et al. 2013]. Based on natural language sentences, the platform deduces which components should be added in the mashup to fulfill the expressed user goals; such components are then immediately added and run into the user workspace. However, one problem observed with these tools is that users have to learn how to express their goal in natural language because a specific syntax is required. Additionally, whereas the selection and inclusion of single components is performed "naturally," the orchestration of components is based on a dataflow diagrammatic definition, very similar to the one adopted in mashArt for expressing event-driven component couplings, which requires users to couple input/output parameters.

The work presented in this article capitalizes on previous results on natural composition paradigms, but it goes one step forward by proposing a new paradigm to integrate easily data coming from different data sources into "pervasive" (not only Web-based) UI components. This contribution is an opportunity for "opening" mashup platforms and increasing their flexibility with respect to the user needs. One observed pitfall for mashup tools, including the platforms discussed earlier, is indeed the difficulty for the composer, especially if not a programming expert, to add new ready components fulfilling their composition needs. The platform presented in this article tries to solve this issue by offering a method for the rapid definition of data mashups and their UI. Our work in particular aims to promote the notion of UI-centric data integration, as a paradigm in which the UI of the final application guides querying and integrating different result sets.

Our main challenge is not the definition of new methods for data integration or service orchestration. We indeed recognize that other platforms already offer powerful solutions. However, although rich in the offered operations for data integration, in some cases such platforms require complex and heavy architectures for the creation and execution of mashups, but especially their composition paradigms do not focus at all on the UI of services and mashups. Our UI-centric abstractions aim to offer an alternative solution to make the composition process abstract with respect to technical details: *interactive* (i.e., based on live programming and immediate feedback mechanisms)

and *light* (i.e., not requiring dedicated integration platforms and protocols and easily portable on different client devices, even the mobile ones that are characterized by limited computing capabilities). The UI centric nature of the composition process, moreover, intrinsically supports the construction of the UI layer of the final mashup—a feature rarely supported by other approaches.

2.3. Multidevice Mashups

The EUD of mashups that can be executed on different devices is a novel aspect covered by our research, which enlarges the possibility for the users to run their applications pervasively, thus creating a continuum across different usage contexts. In the field of EUD of mobile applications, some approaches investigated the possibility of authoring mobile apps directly on the mobile devices [Häkkilä et al. 2005; Davies et al. 2010]; however, especially due to the well-known screen limitations of such devices, the proposed editors enable the configuration of very simple applications, the so-called *microservices* [Danado et al. 2010], with poor contents and functions and not addressing at all the composition of remote services and APIs. Recently proposed services, like IFTTT (If This Than That, https://ifttt.com/wtf) and Atooma (http://www.automa.com), enable users to synchronize the behavior of different apps through simple conditional statements. However, they do not support at all the integration of different datasets and of corresponding UIs.

Chaisatien et al. [2011] illustrate a mobile generator system that, in line with our perspective on the user-driven development of mashups, offers a desktop environment that automatically generates the code of the mobile application. This approach is based on a publish-subscribe paradigm for service synchronization that recalls our model for UI mashup composition [Yu et al. 2007; Cappiello et al. 2011b]. The platform aims to support fast prototyping because it is able to automatically generate a large part of the application code; however, it requires the users to write scripts to structure their applications, thus featuring those design barriers that programming languages typically bring with them. Another aspect is that although this approach enables content extraction from Web documents, it does not support content integration. We believe this is instead a fundamental feature, especially for the mobile usage context where the quick access to heterogeneous content by means of integrated views can greatly improve the user experience.

With respect to the previous works, one distinguishing feature of our approach is the capability of abstracting from specific technologies of the target applications. In line with the Model-Driven Engineering (MDE) philosophy, the work presented in this article focuses on the generation of application schemas complying with a DSL and on their interpretation in different execution platforms by means of native engines. This is a very relevant feature, as recent studies on device share and traffic share report on a generally observed attitude of users to access applications through different devices (desktop and mobile) [Lipsman and Aquino 2013].

3. A UI-CENTRIC COMPOSITION PARADIGM

Our mashup development method allows the users to compose different resources at different levels of granularity, operating iteratively on the UI of the interactive artifacts to be created. The resulting process is represented in Figure 1(b), where it is also compared with the mashup life cycle typically supported by the majority of the mashmaker tools (Figure 1(a)) [Daniel et al. 2011]. Having a mashup idea, the composer can start by selecting ready-to-use *UI components*, equipped with a predefined dataset and a UI, to compose a *UI mashup* in an interactive dashboard supporting the inclusion of components and their synchronization at the UI level. As reported in Figure 1(a), the selection and composition of prepackaged components are also covered by other

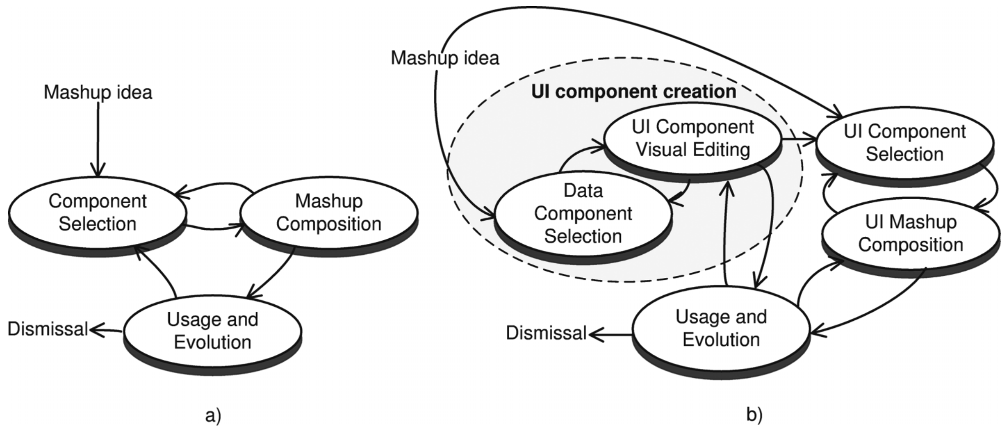


Fig. 1. (a) Mashup life cycle as supported by the majority of mashmaker tools [Daniel et al. 2011] and (b) the life cycle supported by our approach, also covering the selection of contents from data sources and their integration into UI components.

approaches (e.g., Lizcano et al. [2013] and Daniel et al. [2009]). However, as illustrated next in this section, our framework allows the composer to create new UI components from scratch by integrating content retrieved through local or remote data sources.

The creation of UI components is a distinguishing feature of our composition approach. Another relevant aspect is that user-created UI components can be executed as self-contained applications on multiple devices and can expose an event-driven logic that makes them amenable to the synchronization with other components as well. The rest of this section shows, by means of a reference example, how all these different activities can be performed through an interactive Web environment offered by our enabling platform.[1]

3.1. Selection of Data Components and UI Templates

The user who wants to create a new application starts by selecting one or more *data components* and their predefined queries registered into the composition platform. Data components are basic elements of our composition approach that provide access to REST/SOAP Web services.

As an example, let us suppose the user is interested in retrieving information about music events in a given city (e.g., Chicago) and in composing a mashup to be run on an Android smart phone. Let us also assume that a number of relevant data components are already registered in the platform. Through a Web-based design environment, the user visually browses the available data components and selects the one publishing pertinent information (e.g., *Upcoming*) and one of its parametric queries (e.g., the one retrieving events in a specified city). He can also set parameter values to query the service; for example, he inserts "Chicago" as the value of the city he is interested in.

As represented in Figure 2, a *data panel* on the left displays the results retrieved through the selected query. The panel on the right, instead, is devoted to show a *UI template* that the user can select among different alternative templates for data presentation. In the example of Figure 2, the user selects a list-based template. Each list item is represented by three visual elements, namely $< title, subtitle, picture >$. These are the template visual renderers (vrs) that the user can fill in with some of the

[1]The video available at http://youtu.be/Y15ACpxNoZ8 shows our system at work in the different phases of mashup composition.

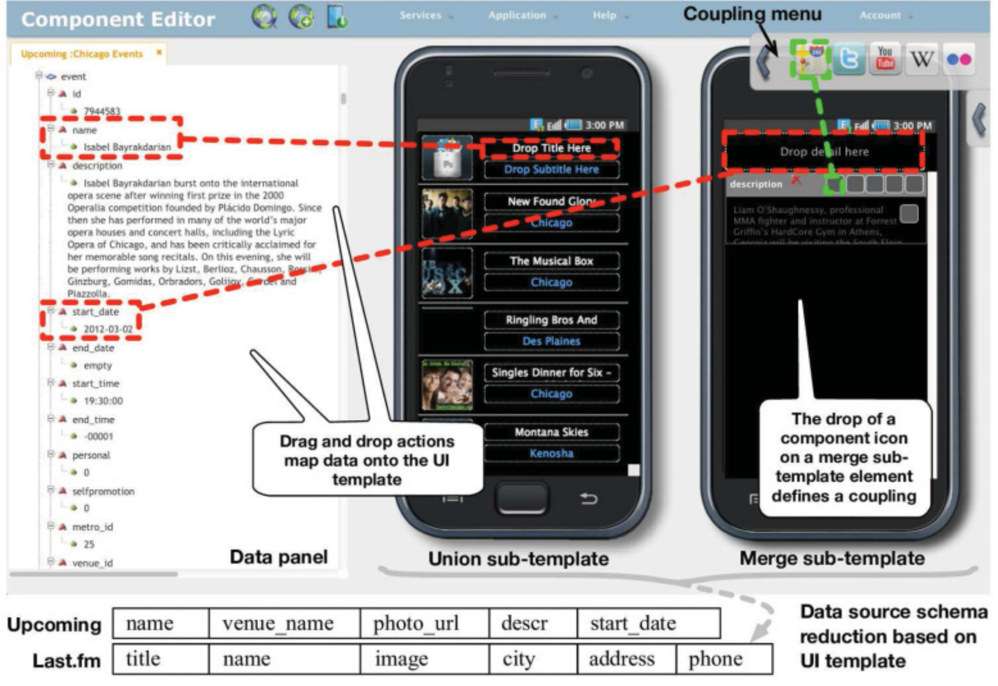


Fig. 2. Schematic representation of the visual mapping activity. Drag&Drop actions allow the user to reduce the data source schemas into local source schemas and to build a global schema guiding the integration of multiple result sets.

data attributes displayed in the data panel. The list showing these items is called the *union subtemplate*. In the final application, it will display the union of the datasets retrieved through the queries selected by the user. The UI template also includes a second view, the *merge subtemplate*, that allows the user to map, on an arbitrary number of additional visual renderers, further attributes that he might want to add as details of the items shown in the union subtemplate. In the final application, the merge subtemplate is devoted to display joins of different datasets.

3.2. Visual Mapping

After the selection of data components and UI templates, the user proceeds with the *visual mapping* activity. Through drag&drop actions he associates data items, visualized in the left-hand data panel, with visual elements of the right-hand UI template. Data item selections represent "examples" of the data the user would like to retrieve and visualize through the final application; such examples are translated by the visual editor into projection queries on the data components to be executed at runtime when the integrated application has to be instantiated on the target device.

Thus, step by step, visual mapping actions define *reductions* of the original data schemas exposed by the components. The two tables at the bottom of Figure 2 exemplify the schema reduction operated by the user on the service *Upcoming* and on an additional data source, *Last.fm*, that the user has selected to retrieve further info on Chicago's music events. Such reductions are automatically coded in a mashup schema expressed in an XML-based language. When the created application is executed, the mashup schema guides the fusion of the retrieved result sets extracted by the two involved services and their visualization according to the selected UI template. The

composer is not required, however, to be familiar with these service-related concepts, as schemas (both the reduced data schemas and the mashup schema) are implicitly deduced from his visual mapping actions.

3.3. Couplings with UI Components and Device-Local Services

Once the integrated dataset and its visualization are in place, the user can also define couplings between visual renderers and other UI components available in the platform. A coupling synchronizes the event of selecting a visual renderer with the invocation of an operation exposed by another UI component or a device-local service. The aim is to further enrich the composition by enabling the exploratory access to complementary data, functionality, and visualizations that can enhance the fruition of the core integrated dataset. For example, the event of selecting a geo-localized music event might activate the visualization of the event venue on a map by means of a map service. Also, on a smart phone the selection of a phone number can activate the dialing of the selected number.

As shown in Figure 2, a tool bar is displayed closed to each visual renderer in the merge subtemplate to highlight in form of icons the components that can be coupled with the parameters carried by the selection of the visual renderer. To create a coupling, the user moves one such icon on the desired visual renderer.

3.4. App Execution on Multiple Devices

Each visual action performed by the user contributes to the automatic generation of a platform-independent schema specifying the organization of the application. Once the composition is completed, the user saves the schema in the platform repository. In the example of Figure 2, the user saves the created schema and names it *My events*. He will then be able to download the schema and properly execute the corresponding app on any device where a proper runtime environment is installed, according to a kind of "polymorphism" enabled by model-to-code generative techniques. The created schema indeed guides client-side execution engines to instantiate dynamically the designed mashup as a standalone app (e.g., on mobile devices) or, as described next, as a UI component within larger UI mashups.

3.5. UI Component Synchronization within Larger UI Mashups

This article especially focuses on the models and UI-centric mechanisms for creating multidevice UI components through data integration. However, to give a complete overview of the capabilities of our framework, we here summarize the main features of the paradigm for UI components synchronization [Cappiello et al. 2011b].

Figure 3 illustrates a UI mashup for searching information about music events, created within an interactive Web workspace devoted to the synchronization of UI components. To create a UI mashup, the user selects components from the visual palette on the left and includes them in the central canvas. Some components, so-called *wrapped components*, are prepackaged (i.e., provided with the initial installation of the platform or added, if needed) by platform administrators. Some other components are self-created by the user through the visual mapping paradigm. In the example of Figure 3, the components based on GoogleMaps, Wikipedia, and Flickr are wrapped. The *My events* component is instead the one created by the user, as illustrated. Although *My events* was initially created as an app for an Android smart phone, thanks to the polymorphism of the created integration schema, in the example the component is instantiated as an HTML table within a $< div >$ container of the workspace Web page.

As regards the composition paradigm, a live programming paradigm is also adopted at this level. Thanks to the intermixing of application design and execution, the users are provided with immediate feedback about the effect of their composition actions.

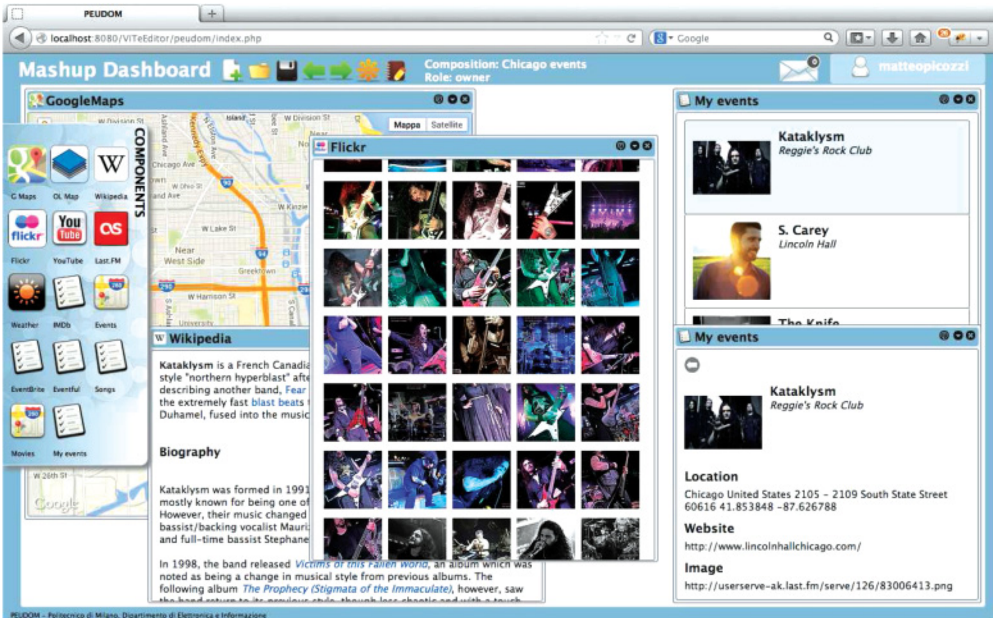


Fig. 3. Interactive Web workspace for the synchronization of UI components [Cappiello et al. 2011a, 2011b, 2012].

The selection of a component in the visual palette is immediately followed by the visualization of the component UI; the user can thus start interacting with the component, accessing and browsing its dataset and figuring out how the component works. Possible couplings with components already included in the workspace are automatically identified based on compatibility rules [Cappiello et al. 2011b, 2012] and visually highlighted through different colors of the component border (i.e., green if a coupling is possible, red if not). A coupling can be defined by dragging one component onto another. In the example of Figure 3, two couplings synchronize the selection of an artist in the *My events* component with the display of a corresponding page in Wikipedia, and with the display of related images in Flickr.

4. MODELING ABSTRACTIONS

In this section, we define the modeling abstractions, schematically represented in the metamodel illustrated in Figure 4, on which our composition paradigm is based. Such abstractions are transparent to the users performing the mashup composition. Their systematic definition, however, facilitates the understanding of how the visual actions operated by the users are transformed into models and how models are in turn parsed by interpreters that instantiate the respective applications on the fly. Our approach indeed complies with Model-Driven Engineering methods where modeling abstractions guide the design of applications while generative layers mediate between high-level visual models and low-level technical engines that execute the mashup.

4.1. UI Components

The fundamental blocks of our composition paradigm are UI components.

Definition 4.1 (*UI Component*). A UI component is a self-contained software module that is bound to one or more services providing data and/or functionality and is equipped with its own UI (its *concrete UI*). A UI component also exposes an event-driven

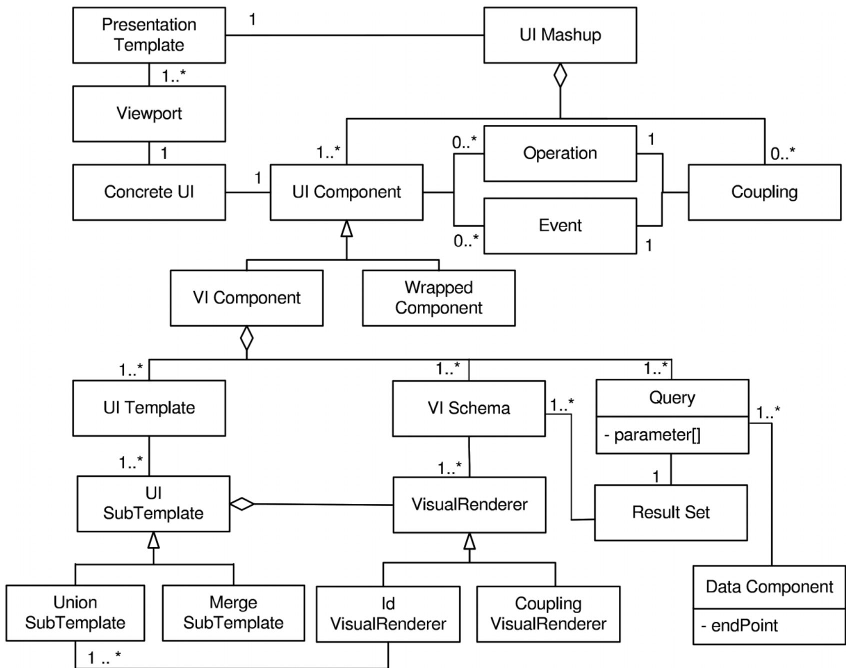


Fig. 4. Main modeling elements enabling the UI-centric composition paradigm.

logic characterized by a set of events, E, that can be generated by the user interaction with its concrete UI and can transport parameters, and a set of operations, O, that some other components' events can activate to change its status when a synchronized behavior within a composite application is needed.

As an example, all the components included in the Chicago events mashup illustrated in Figure 3 are UI components. The specificity of UI components with respect to other components (e.g., Web services) is indeed the presence of a UI as a means for the users to interactively navigate and manipulate the component's content and invoke business logics operations.

4.2. UI Mashups

A UI mashup is a composite application that integrates UI components at the presentation layer. It reuses and synchronizes the UIs of the involved components. It also mediates possible data mismatches occurring when synchronizing the components, but it leaves the responsibility of data and business-logic management to each individual component. More precisely, a UI mashup can be characterized as follows:

Definition 4.2 (*UI Mashup*). A UI mashup can be defined as $UI_Mashup = <UIC, C, PT>$, where UIC is the set of UI components included in the mashup, C is the set of *couplings* that determine the synchronized behavior of components within the mashup, and PT is the presentation template adopted to organize the UI of the mashup.

Components couplings are channels for intercomponent communication, defined according to an event-driven, publish-subscribe integration logic [Yu et al. 2007; Cappiello et al. 2011b]. They establish how the occurrence of published events cause the execution

of subscribed operations, which in turn result into a state change in the target component. More precisely:

Definition 4.3 (*Components Coupling*). Given two UI components, uic_s and uic_t, a *coupling* synchronizing their behavior is a pair $c =< e_{uic_s}(< output_parameters >), o_{uic_t}(< input_parameter >) >$, representing the subscription of an operation of the target UI component, o_{uic_t}, to an event raised by the source UI component, e_{uic_s}, and more specifically to the output parameters the event might transport.

For example, in the mashup of Figure 3, the coupling $< selectArtist_{MyEvents}(ArtistName), showImages_{Flickr}(SearchKey) >$ is defined to synchronize the selection of an artist's name in the *My events* component with the display in the Flickr component of images related to that artist. The selection event generates as output parameter the *ArtistName*, and this data item is coupled to the *SearchKey* input parameter of the Flickr operation. Such a parameter passing enables the update of the status of Flickr every time a new artist is selected in *My events*.

It is worth noting that UI mashups intrinsically have a presentation layer derived by the aggregation of each single component's UI. As represented in Figure 4, the resulting UI can be managed through a template (e.g., a grid-based HTML layout for the Web) where each component's UI is visualized within a viewport,[2] that is, a window or any other viewing area on the screen (e.g., an HTML `div` or `iframe` on the Web), hosting the visualization and execution of the component.

Later in this section, we further characterize UI components as wrapped UI components, which are prepackaged and included into the platform in form of scripts wrapping the original services, and VI components, which are visually defined from scratch by end users through the visual mapping paradigm.

4.3. Wrapped UI Components

Wrapped UI components are widgets that are prepackaged by expert developers who manually program wrappers for the remote access to the underlying Web services and APIs. When the accessed resources are not natively equipped with a UI, wrapping is also aimed to create a UI and to program the event-driven logic needed for achieving synchronization at the presentation level. In the example of Figure 3, the components related to GoogleMaps, Flickr, and Wikipedia are all wrapped UI components. For GoogleMaps, the wrapper supports the instantiation of its native UI within a $< div >$ tag, the invocation of the API methods triggered by the user interaction, and the logic for event handling needed for the execution of the overall mashup. For the other two components, wrappers also implement ad-hoc UIs that are not provided by the original service; wrappers are thus in charge of populating HTML markup with content fetched from the services and also manage any successive interactive access to the component data.

4.4. VI Components

Visual Integration (VI) components are created by the users by mapping visually result sets extracted by one or more data components to UI templates.

4.4.1. Data Components

Definition 4.4 (*Data Components*). A data component provides read-only access to a remote or local data source. It can be defined as a pair $d_comp = <ep, Q>$, where *ep* represents the *endpoint* of the underlying service (e.g., the URI of a RESTful service) and Q represents the set of predefined parametric queries.

[2]http://www.w3.org/TR/CSS21/visuren.html#visual-model-intro.

To enable the invocation of a data component at runtime, its endpoint and the queries are specified in a descriptor that is created when the component is registered into the platform. The registration is supported by a form-based user interface where the user specifies the needed configuration data; configuration files are then generated. For example, the *Last.fm* data component used in our reference scenario is defined on top of the Last.fm RESTful API. Its registered endpoint is the URI *http://ws.audioscrobbler.com/2.0/* where the API is published. Q includes some parametric queries, for example, the one based on an operation to retrieve music events located in a specified city. Queries are expressed as HTTP GET requests, for example, *"?method=geo.getevents&location=location_str&api_key=lastfm_api_key."* At runtime, requests are instantiated by considering actual parameter values specified during the composition or also during the component execution, for example, the value *"chicago"* for the *location_str* parameter and the API key *0697XXX*.

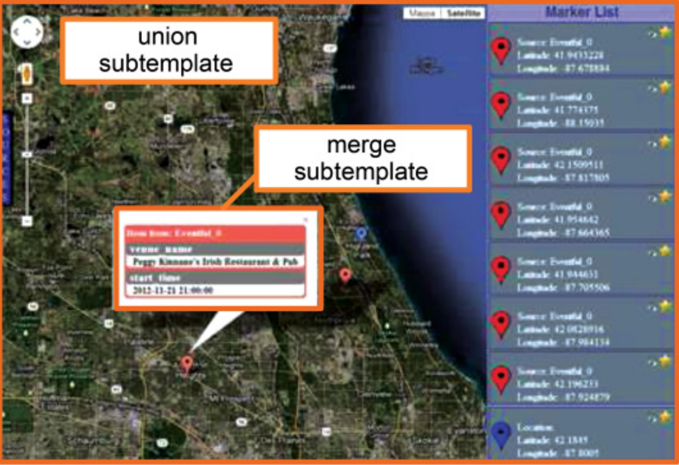
4.4.2. UI Templates

Definition 4.5 (*UI Template*). A UI template is an "abstract" representation of the UI of a VI component. It is adopted during the visual mapping activity to guide the selection and integration of data retrieved through data components. At runtime, it is then used as a basis to instantiate the "concrete" UI displaying the integrated dataset. The concrete UI organization reflects the one of the UI template, but it can still assume different layout styles. More precisely, a UI template can be characterized as $UItemp = <type, VR, templE>$, where:

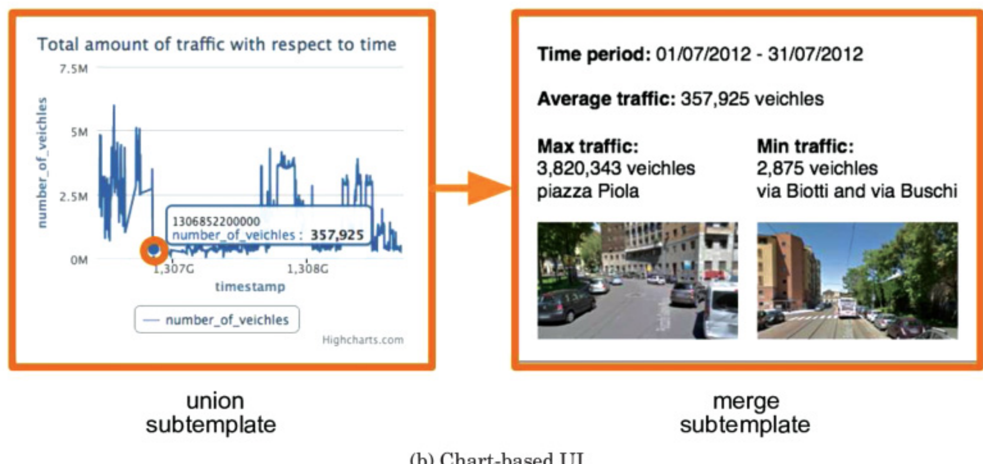
—*type* is the template class (e.g., list, map, chart) selected by the user.
—VR is the set of visual renderers, vr_k, that is, the elements that provide visual placeholders for single data attributes or for the aggregation or fusion of data attributes extracted from data components. The way visual renderers are displayed in the final application is specific for each UI template type; they can be Points of Interest (POIs) in a map, text fields in a list-based UI, or point in a scattered plot. At a higher level of abstraction, each vr can be considered merely as a "receptor" of data attributes, independent of its specific rendering in the component's concrete UI.
—$templE$ is the set of events that at runtime can be raised by the selection of template visual renderers. For example, one may assume that some visual renderers be associated with an *onClick* event transporting as parameter the displayed data item. VI components thus inherit the set of events of the UI template they are based on.

Visual renderers can be grouped in two different subtemplates:

—*union visual renderers* (uvr_k), included in the union subtemplate, are in charge of displaying few characterizing attributes, representing data items concisely. Some uvr_k, called *ID visual renderers*, display key attributes used as identifiers of data instances. The number of union visual renderers is predefined and depends on the visual organization of the UI template. At least one ID visual renderer has to be included, as at runtime the associated data attributes are exploited to join different result sets and detect and manage duplicates.
—*merge visual renderers* (mvr_k), included in the merge subtemplate, display additional details for the data instances selected in the global subtemplate. Different from the union subtemplate, the number of merge visual renderers is not defined a priori, as the users can arbitrarily add visual renderers to display data items or even choose not to have a merge subtemplate in their applications. Some mvr_k can play the role of *coupling visual renderers*, meaning that at runtime their selection can generate events carrying as parameters the displayed data.



(a) Map-based UI



(b) Chart-based UI

Fig. 5. Examples of map and chart UIs organized into union and merge subtemplates.

The choice of structuring UI templates into the two subtemplates just described enables a specific policy for data integration that will be described later in this section. It is also effective with respect to the usability of the resulting applications. Indeed, it is coherent with the well-known “global-detail” pattern [Card et al. 1999], which suggests providing a data overview first and then details on demand for the only items selected by the users in the overview. This pattern is very common in Web application design [Ceri et al. 2007]; more recently, it has being effectively adopted in mobile app design as well [Burigat and Chittaro 2013]. It is also applicable in almost any type of visualization. For example, Figure 5 illustrates the adoption of the two subtemplates in two other types of UIs:

—*Map-based UI template*: The map-based UI represented in Figure 5(a) displays the same result set displayed as a list in Figure 2. The map globally displays all the retrieved data items, each one as a POI highlighted on the map; a pop-up balloon then displays further details when a POI is selected.

—*Chart-based UI template*: The chart in Figure 5(b) represents traffic data for some intersections in the city of Milan [Picozzi et al. 2013]. The result set is globally represented by one or more series in the chart, each one corresponding to a measure (e.g., the average number of vehicles) computed for different values of a dimension (e.g., time). The selection of a series then leads to more detailed information, for example, to the visualization of descriptive statistics computed by aggregating the values in the series.

4.4.3. VI component Constituents.

Definition 4.6 (*VI Component*). Given the availability of data components and UI templates, a *VI* component can be defined as the tuple $VIcomp =< selQ, UI_template, VI_schema, E, O >$, where:

—$selQ$ is the set of queries that the user selects from each involved data component to gather the VI component data. For example, the VI component *My events* in the example of Figure 3 is associated with two parametric queries to select data instances from *Last.fm* and *Upcoming* based on the value of the *city* attribute (e.g., *Chicago*);
—$UI_template$ is the user-selected UI template associated with the component for the visualization of its integrated dataset. In the *My events* component, the selected $UI_template$ is the *list*.
—VI_schema (Visual Integration schema) represents the set of mappings between data items extracted through the queries in $selQ$ and visual renderers characterizing the $UItemplate$. Independently of the adopted UI template, a VI schema is a tuple, $VI =< vr_1, vr_2, \ldots, vr_n >$, where each vr_k represents queries to retrieve and/or fuse the associated data attributes.
—E and O are the sets of events and operations exposed by the component to make it comply with the event-driven logic needed for UI synchronization. $E \subseteq templE_{UItemplate}$, that is, E is derived from the events associated with the UI template (see Definition 4.5). For example, for the *My events* component, E includes the selection of the list items in the merge subtemplate. $O \subseteq selQ$, that is, O is derived from the set of queries exposed by the involved components. In the *My events* component, an operation that updates its status is, for example, the one that queries the underlying data sources searching for events based on a specified city name.

It is worth noting that, while for wrapped UI components the execution logic is blurred in the programmed wrapper and merely depends on the opportunistic strategy adopted by the programmer, in VI components a unique execution logic, replicated according to the technology of the target execution environments, is used to interpret the user-created schema and generate, through model transformations, the code for the component instantiation and execution.

4.5. Construction of the VI Schema

This section explains how the modeling elements illustrated thus far enable the construction of VI schemas controlling the execution of VI Components. The creation of VI components is based on the integration of result sets coming from different data components; thus, the creation of a VI schema is comparable to a data integration problem. Data integration is usually modeled as a triple $< G, S, M >$, where G is the global schema, defining the structure of the final integrated dataset, S is the heterogeneous set of source schemas, and M is the mapping between G and S that associates each element of G with a query over one or more sources in S [Lenzerini 2002]. We now illustrate how we approach this problem, given the specificities posed by our UI-centric composition approach.

	$\underline{uvr_1}$	$\underline{uvr_2}$	uvr_3	mvr_1	mvr_2	mvr_3	mvr_4	mvr_5
Upcoming	name	venue_name	photo_url	descr	start_date	$\perp$	$\perp$	$\perp$
Last.fm	title	name	image	$\perp$	$\perp$	city	address	phone
	Union mapping			Merge mapping				

Fig. 6. Global integration schema constructed through the union and merge mapping. Attributes from the service local schemas are renamed according to the visual renderer names.

In order to increase the openness of our tools and to give to the end users the freedom to select the data they deem pertinent, we assume that the source schemas for the registered data components are not known a priori (i.e., specified/described by an expert designer in the data component descriptors). Given a data component selected by the user, the queries in $selQ$ are run and a source schema is derived by interpreting the returned result set. The result set representation in the data panel of the design environment (see Figure 2) thus provides users with a situational source schema, $sits_i$, that corresponds to the set of attributes retrieved through the executed query and not necessarily to the source schema as exposed by the service provider.

The user selection of interesting data attributes in the data panel then defines the source local schema, $ls_i \subseteq sits_i$, while the association of such attributes with the visual renderers in the UI template represents the mapping M between the source schema $sits_i$ and the global schema G. Indeed, the global schema G in our approach is structured according to the set of visual renderers in the UI template, with each visual renderer representing one or more queries on the source local schemas. G, however, is not completely defined a priori. This because while the set of visual renderers in the union subtemplate is known from the definition of the UI template, the structure of the merge subtemplate is totally undefined as it is incrementally constructed as soon as the user adds attributes in the subtemplate.

More formally, let us consider the set of data sources $\{s_1, \ldots, s_j\}$ selected by the user. Associating some attributes of a data source s_i with the visual renderers of a UI template corresponds to specifying assertions of the form $VR \rightarrow Q_{s_i}$ expressing that the data visualized by each element vr_k is retrieved by a projection query Q_{s_i} over s_i. The set of all the attributes $\{s_i.a_h\}$ extracted through Q_{s_i} determines the source local schema ls_i, that is, the reduction of the situational source schema that actually contributes to the construction of the integrated dataset.

Coherently with the structure of UI templates, ls_i also consists of two parts:

—uls_i: $\forall s_i.a_h \in uls_i, uvr_k \rightarrow s_i.a_h$, that is, the subset of attributes mapped to the union visual renderers; and
—mls_i: $\forall s_i.a_h \in mls_i, mvr_k \rightarrow s_i.a_h$, that is, the set of attributes mapped to the merge visual renderers.

Let us consider our reference example illustrated in Figure 2, where a list-based UI template is adopted for the integration of the two services $s_1 = Upcoming$ and $s_2 = Last.fm$. By selecting attributes from the two services and mapping them to the visual renderers of the list UI template, the user defines the local schemas $ls_1 =< name, venue_name, photo_url, descr, start_date >$ and $ls_2 =< title, name, image, city, address, phone >$. In particular, as reported in Figure 6, the first three attributes of each schema (i.e., $< name, venue_name, photo_url >$ for s_1 and $< title, name, image >$ for s_2) are associated with the same union visual renderers in the global schema ($< uvr_1, uvr_2, uvr_3 >$), while the other remaining fields are mapped to distinct merge visual renderers.

The table in Figure 6 represents how the attributes are renamed according to the visual renderers they are associated with. The same attribute names are assigned to

```
<sources>...</sources>

<filters>...</filters>

<union type="list">
 <vr name="uvr1" key label="Title" src="Upcoming" query="@name"/>
 <vr name="uvr2" key label="Subtitle" src="Upcoming" query="/@venue_name"/>
 <vr name="uvr3" label="Image" src="Upcoming" query="@photo_url"/>
 <vr name="uvr1" key label="Title" src="Last.fm" query="/title"/>
 <vr name="uvr2" key label="Subtitle" src="Last.fm" query="venue/name"/>
 <vr name="uvr3" label="Image" src="Last.fm" query="venue/image[position()=4]"/>
</union>

<merge>
 <vr name="mvr1" label="Description" src="Upcoming" query="@description"/>
 <vr name="mvr2" label="Start date" src="Upcoming" query="@start_date"/>
 <vr name="mvr3" label="City" src="Last.fm" query="venue/location/city"
                                                  coupling="twitter|flickr|wiki"/>
 <vr name="mvr4" label="Address" src="Last.fm" query="venue/location/street"
                                                  coupling="maps|flickr"/>
 <vr name="mvr5" label="Phone" src="Last.fm" query="venue/phonenumber"
                                                  coupling="dialer|addressbook"/>
</merge>
```

Fig. 7. An excerpt of the XML-based VI schema for our reference example.

all the attributes associated with the union subtemplate (e.g., uvr_1, uvr_2, and uvr_3), while distinct names (e.g., from mvr_3 to mvr_5) are assigned to the attributes associated with the merge subtemplate. The result consists of the two local schemas $ls_1 =< uvr_1, uvr_2, uvr_3, mvr_1, mvr_2 >$ and $ls_2 =< uvr_1, uvr_2, uvr_3, mvr_3, mvr_4, mvr_5 >$. The two data sources are then overlapped on the basis of the attributes associated with the ID visual renderers,[3] in the example uvr_1 and uvr_2.

The global schema G is obtained as a *Universal Relation* [Naumann 2002], i.e., as the union of all the attributes in the local schemas. The assumption at the basis of the Universal Relation is that attributes with the same name in the schema refer to the same property [Naumann 2002], and there is no reason to replicate them. Therefore, in the example described above $G = ls_1 \bigcup ls_2 =< uvr_1, uvr_2, uvr_3, mvr_1, mvr_2, mvr_3, mvr_4, mvr_5 >$.

4.5.1. Domain Specific Language. VI schemas are expressed into an XML-based specification, that is automatically generated by the design environment without requiring any intervention by the end user. The specification is based on a Domain Specific Language that defines how each composition action has to be translated into rules for data integration and UI synchronization to be executed at runtime for the automatic instantiation of the final mashup. Figure 7 shows a simplified fragment of the XML-based specification generated for our reference example. The global integration schema is given by the specified visual renderers. In particular, for each vr_k:

—The `label` attribute specifies the label to be displayed on the concrete UI. Labels can be defined by the users when they perform the visual mapping.
—`src` specifies the data components providing the mapped data attributes, while `query` specifies the corresponding queries. If multiple data components and queries are specified for a given vr_k, as represented in Figure 7 for uvr_1, uvr_2, and uvr_3, corresponding data union and fusion policies are applied at runtime.
—For merge visual renderers, the `coupling` attribute specifies the list of additional UI components subscribed to the selection of the visual renderer (in the example, Twitter, Flickr, and Wikipedia for the `City` visual renderer). Each coupling is then interpreted as a pair $< mvr_k.event.parameter, UIcomp_i.operation >$, specifying that

[3]The indication of the key attributes can derive from predefined settings in the UI template definition. The user can modify these settings through a function offered by the design environment.

an operation of the UI component $UIcomp_i$ subscribes to the parameter generated by the selection of the merge visual renderer.

The settings to invoke data components (URI and parameters for the initial selection queries) are automatically added at the beginning of the schema (not reported in figure for brevity). Data component properties may also include the definition of data filters defined by the user and proposed at runtime to progressively refine the result set. Finally, to support the synchronization of the VI component with other UI components, the VI schema also specifies (1) *events* (the selection of visual renderers that carry as parameter the displayed data items) and (2) *operations* (the queries selected by the users for the creation of the integrated dataset).

4.6. Policy for Data Union and Fusion

The construction of VI schemas at design time is complemented with a policy for querying at runtime the involved data components and for performing the union and fusion of the obtained result sets. Figure 8 illustrates the different steps.

When a VI component is executed, the involved data components are queried according to the local schemas specified in the VI schema. The union of all the data items extracted by considering the union local schema, uls_i, of each involved data component is computed (Step 1 in Figure 8) and rendered in the union subtemplate (Step 2). Duplicates, that is, data items returned by different data sources referring to a same real-word entity, may still exist. Their detection is performed only when the user selects a specific instance from the union subtemplate (Step 3); in this case, indeed the merge subtemplate has to visualize the fusion of all the attributes extracted from the different sources that refer to the selected instance. The process, therefore, goes on with a data fusion procedure that we call *Data Fusion on Demand* (Steps 3–7 in Figure 8): with the aim of reducing the computational effort, we indeed contextualize duplicate detection and fusion to the only instances actually selected by the user at runtime. This choice prevents us from executing data fusion operations a priori, that is, for any instance displayed in the union subtemplate.

Algorithm 1 specifies the adopted data fusion strategy, which consists of an initial setup, a duplicate detection phase, and the final fusion and deduplication. In the setup, for each result set RS_i represented in an XML/JSON format, we build an array. For example, each tuple of the result sets illustrated at the top of Figure 8 will correspond to an array element. Hence, retrieving an instance or the value of its key attributes consists in a direct access to an array element. A Comparison Set (CS), which is the input for the successive duplicate detection phase, is then constructed as a new array. In particular, given the user-selected item *si* belonging to a given result set RS_{si}, CS is built by including all the data items in the result sets involved in the composition but RS_{si}. The choice of not including RS_{si} originates from the assumption that a service would not return duplicated items. For example, in Figure 8, *si* belongs to the *Upcoming* result set. Thus, CS is initialized with the only items from the *Last.fm* service. If another service is included in the composition (e.g., *Eventful*), CS would consist of the union of the two result sets retrieved from *Last.fm* and *Eventful*.

In the duplicate detection phase (Step 4 in Figure 8), the key attributes of si, K_{si}, are then compared with the key attributes of all the instances in CS, K_{cs_i}. An in-depth search is performed among the CS elements. Regarding the comparison, the function $IsSimilar(K_{si}, K_{cs_i})$ computes a similarity score sc. If sc is greater than a specific threshold, the items are considered similar.[4] In our current implementation, the

[4]Possible values for *sc* range from 0 to 1. In our experiments, after executing the data fusion algorithm with different threshold values, we determined that 0.965 is a value that effectively reduces the number of false positives and negatives.

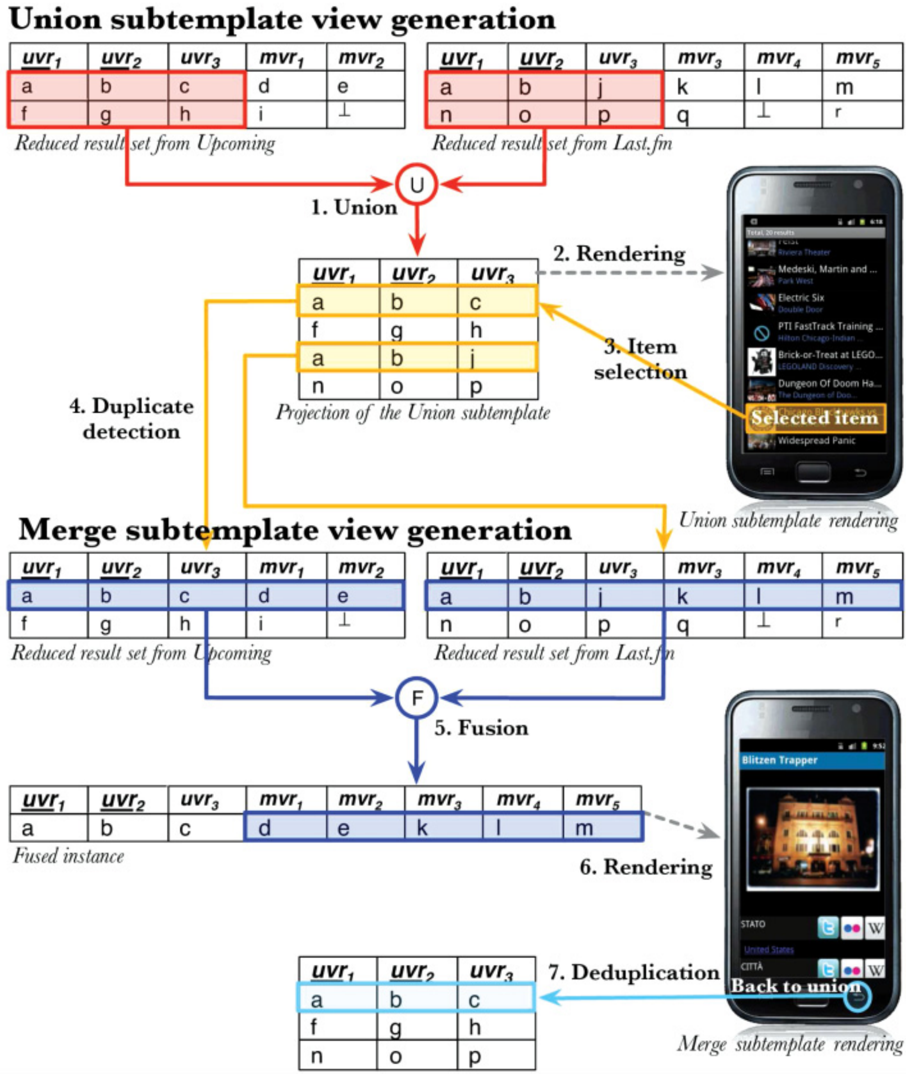


Fig. 8. Main steps for the union and fusion of data.

function is based on the Chapman's Soundex metrics [Zobel and Dart 1996; Bleiholder and Naumann 2008], whose algorithm encodes strings according to their pronunciation. Similar-sounding names are assigned with the same code. String comparison then takes into account such phonetic encoding.[5] The complexity for the computation of this measure is linear with the length of the strings to be compared [Zobel and Dart 1996]. Since in our case the compared strings have a limited length (in our reference scenario we for example compared event titles), we thus assume a constant cost for the similarity function.

[5]Soundex is the most common phonetic algorithm, is widely used as an alternative to exact-matching, and is also provided as a built-in function in many DBMS products like Oracle and MS SQL. Some studies also proved it works better than others measures (e.g., the Levenshtein index) for long sentences.

ALGORITHM 1: Data Fusion On Demand

RS_i: result sets for the i-th data component
ls_i: local schema for the service s_i
si: the instance selected by the user at runtime
CS: comparison set, including all the instances to be compared with si to identify duplicates
cs_i: i-th item of CS

$GetLocalSchema(cs_i)$: returns the local schema of the origin data source of the instance cs_i
$GetPrimaryKey(cs_i)$: returns the primary key values of the instance cs_i
$GetMergeAttributes(ls_i)$: extracts from the local schema ls_i the attributes in mls_i
$AddMergeAttributes(mls_i, mls_j)$: adds the attributes in mls_j to mls_i
$IsSimilar(K_i, K_j)$: returns true if two passed primary keys are similar, based on the adopted similarity measure

```
begin
    // Initialization of the comparison set CS
    forall the RS_i, si ∉ RS_i do
        add RS_i to CS
    end
    // Initialization of ls for the origin data source of si
    ls_si ← GetLocalSchema(si)
    // Search for similar items with the comparison set CS
    K_si ← GetPrimaryKey(si)
    forall the cs_i ∈ CS do
        K_cs_i ← GetPrimaryKey(cs_i)
        // Similarity Evaluation
        if IsSimilar(K_si, K_cs_i) then
            ls_cs_i ← GetLocalSchema(cs_i)
            Fuse(ls_si, ls_cs_i)
            remove cs_i
        end
    end
end

Fuse(ls_si, ls_i) {
    mls_i ← GetMergeAttributes(ls_i)
    mls_si ← GetMergeAttributes(ls_si)
    AddMergeAttributes(mls_si, mls_i)
}
```

In the fusion phase (Step 5 in Figure 8), if a similarity match is detected between si and an item cs_i, the two items are fused; si is now composed of the attributes in its union local schema, the attributes in its merge local schema, plus the attributes in the merge local schema of the similar instance. The merge subtemplate is rendered by displaying the new extended merge local schema (Step 6). Once the user returns to the visualization of the union subtemplate, the policy for handling duplicates updates the visualized union set. In particular, the identified duplicates are dropped out (Step 7) and the only instance selected by the user is kept in the union list.

It is evident that the complexity of the previous algorithm is especially related to the comparisons needed for the identification of duplicates. The best case corresponds to the use of one single data component, which does not require the construction and the analysis of CS. Of course, the number of comparisons grows with the number of data components included in the mashup. However, our choice to compare the only

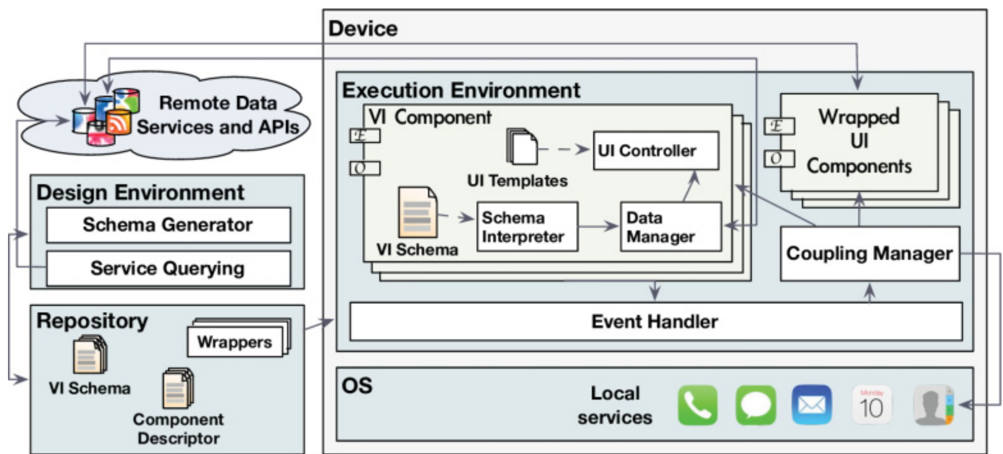


Fig. 9. Platform architecture.

instance selected by the user, si, with all the other instances in CS allows us to keep the complexity linear to the cardinality of CS. Our algorithm is indeed organized as a linear search of the user selected item in the CS array. In Section 6, we will in particular show how the series of comparisons required by the duplicate detection phase is feasible in a reasonable amount of time even on mobile devices with limited computing capabilities. This is also due to the result pagination policies adopted by services – service responses generally do not exceed few hundreds items. A further optimization of the algorithm would consist in stopping the duplicate search within a given result set as soon as a match in that result set is detected, under the assumption that no duplicates exist in a same result set.

5. REFERENCE ARCHITECTURE

This section illustrates the reference architecture of our current platform prototype. We especially concentrate on the execution engine in charge of running the created mashups on multiple devices. Therefore, we specifically highlight the modules for instantiating and synchronizing UI components.

To keep the overall approach lightweight, our architecture is strongly based on a client-side execution logic. Of course, the same logic could be moved at the server side, thus having a thin client with the only responsibility of handling the presentation of a mashup and its event-driven logic. This is needed, for example, when persistence and evolution of data and application schemas have to be managed.[6] However, we believe that a lightweight architecture is fundamental to encourage end-user development. Moreover, because of the "interaction-intensive" nature of the resulting applications, a server-side logic would imply a high volume of of requests to be managed by the server that would downgrade performances. Finally, we chose to perform data fusion on the client, and to not manage integrated datasets materialized on the server (see, e.g., Bozzon et al. [2012]), to enable the user to access at any moment fresh data, as they are made available by service providers. Figure 9 illustrates the main architectural components.

The **Platform Repository** stores data component descriptors, wrappers and descriptors for pre-packaged UI components, as well as the schemas of the user-created artifacts, namely VI components and UI mashups.

[6]A recent extension of our platform to support resource sharing and collaboration among multiple users adopts a server-side logic for storing materialized views of datasets [Matera et al. 2013; Ardito et al. 2013].

The **Design Environment** (DE) is an Ajax Web application. It includes a *Service Querying* module, in charge of querying the data components and transforming the retrieved data into the visual representation offered by the data panel, and a *Schema Generator* module, which translates the visual mapping actions into the XML-based VI schema and stores it on the platform repository. Based on the mashup target device, the DE can be detached from or can be run on the execution environment of the final mashup. For example, in case of mashups to be executed on mobile phones, it would be convenient to compose the mashups on other devices with a larger screen (e.g., a desktop PC), not directly on the target devices.

The **Execution Environment** (EE) runs on the device where the mashup is executed and interprets the generated mashup schema. The schemas are downloaded on the client from the platform repository together with the wrappers for the execution of prepackaged UI components, if needed. The EE itself can be downloaded from the platform repository where it is available as a native application for the different mobile operating systems or as a client-side script for the Web browser.

Once a mashup schema is downloaded, a *Schema Interpreter* parses it and invokes other modules in charge of managing the different application aspects. Based on the VI schema, the *UI Controller* (UIC) translates the specified visual renderers into code for generating the corresponding UI layout. For example, if Android is the target operating system, the set of visual renderers is translated into the Android layout markup language handling the generation of screens—so-called *activities*. Based on the visual mapping rules defined for each visual renderer, the *Data Manager* (DM) queries the involved data sources and stores the retrieved data in a local data repository. If needed, it handles the execution of data union and fusion. The UIC then handles the population of each visual renderer with the resulting data.

At runtime, the previous modules manage the construction of the component's concrete UI and the initial dataset. Other modules are needed to execute the resulting composition. An *Event Handler* (EH) acts as an event bus. It listens to the events generated by the interaction at the UI level with visual renderers and communicates them to the subscribed components. If the event is the selection of a union visual renderer, the EH triggers possible data fusion procedures executed by the DM module; alternatively, if the event occurs on a merge visual renderer on which a synchronization coupling is defined, the EH delegates to the *Coupling Manager* module the invocation of the operations in the subscribed components. On mobile devices, the Coupling Manager also handles the synchronization with the device local services.

6. EVALUATION

In order to assess the feasibility and the validity of our approach, we evaluated the performance of the generated mashups in the worst possible cases, that is, for their execution on devices with limited computation capabilities when processing a high number of data items. We also evaluated the usability of the composition paradigm through (1) a user study focusing on the performance of a sample of users and their satisfaction when using our composition platform and (2) an expert-based heuristic evaluation also taking into account the complexity of the composition tasks. We also preliminarily evaluated the usability of the generated mashups.

6.1. Performance of the Data Fusion Technique

Performance evaluation concentrated especially on the data fusion technique, being it the most time-consuming operation to be executed at runtime. We focused on the most strict conditions, namely the execution of data fusion at the client side, with the client being a smart phone. We built a mobile mashup for the reference example described in Section 3 by integrating data from the services *Last.fm*, *Upcoming,* and *Do Staff*

Table I. Comparison of Average Data Fusion Times for the Two Test Cases on a Smart Phone and a Desktop Web Browser

Number of items	Mobile: 2 services – mean time [s]	Mobile: 3 services – mean time [s]	Desktop: 2 services – mean time [s]	Desktop: 3 services – mean time [s]
100	4.90	4.02	0.04	0.03
200	9.31	6.71	0.05	0.04

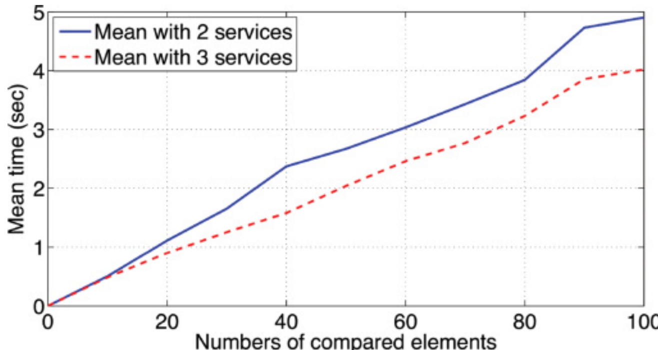


Fig. 10. Mean execution time for data fusion on the mobile device: time increases linearly with the number of performed comparisons.

Media. We thus executed the resulting app on a Samsung Galaxy SII with a dual-core processor (1.2GHz ARM Cortex-A9) and 1GB RAM. The component was also executed in the Firefox 24.0 Web browser on a Mac-Book Pro with OS X 10.8.5, 2.5GHz Intel Core i5 CPU and 8GB of DDR3 RAM.

We considered two test cases: (1) a mashup integrating two data components, *Last.fm* and *Upcoming*, and (2) the inclusion of a third data component, *Do Staff Media*. In both cases, during the execution of the mashup, we measured the time elapsed from the selection of an item in the union subtemplate until the visualization of its details in the merge subtemplate. The measured time thus refers to the execution of duplicate detection and data fusion.

For each test case, we considered different numbers of compared items (from 10 to 200). Given the result set of each service, we selected the items for the comparison set on the basis of their text attributes, to include the best, medium, and worst cases with respect to the Chapman's Soundex similarity metrics. The comparison attribute was the one displayed by the *Title* visual renderer (i.e., the title of a concert).

As reported in the first two columns of Table I, processing 100 items retrieved from two different services on the mobile device took 4.90s, while it took 4.02s with three services. Similarly, processing 200 items retrieved from two and three services took 9.31s and 6.71s, respectively. This shows that the execution time does not depend much on the number of services, rather it is influenced by the number and the length of texts to be compared. In fact, the third added service (*Do Staff Media*) returned data items with a text length much shorter than the other two services, and this reduced the average time needed for the comparison. The bottleneck, therefore, is given by the metrics used to estimate the similarity of the comparison attributes.

Figure 10 compares the average execution time on a mobile device for the two test cases. The time is linear in the number of the compared items—the higher the number of items, the higher the number of comparisons for duplicate detection. This shows that the execution time can be, for example, reduced through an adequate pagination

of the result set. The current implementation of the execution engine for Android smart phones, for example, allows the users to set some limits on the number of items to be downloaded by each service so that the execution time is optimized in relation to both the data download and the execution of comparisons needed for fusing data.

We also ran the first test scenario (two services, 200 items in the comparison set) in a Firefox Web browser executed on a desktop computer. Table I compares the performance of data fusion when executed on the mobile device and in the Web browser of a desktop PC. The execution in the Web browser took, of course, a much lower time (5.421*ms* in average), but the behavior of the Web application was coherent with the one observed for the Android app.

In relation to memory consumption of the generated app, it is important to note the following:

—The size of the XML-based VI schema is 10kB, on average, and the schema is parsed once, when the application is launched. Hence, we can assume that downloading and parsing the schema do not have significant impact on the memory consumption of the overall execution.
—The application data consists of the result sets coming from the involved data components and the UI components (if any synchronization coupling is defined). Our evaluation focused only on the memory usage by the data components. Indeed, actuating a synchronization coupling, starting from the integrated result set, allows the user to navigate to additional (multimedia) content offered by UI components. However, the time needed for querying the component and downloading its result set does not depend on the application logic of our execution environment and would not be different if the user would access the same content by using dedicated apps or the Web browser.

To estimate an order of magnitude for the size of the downloaded dataset, we ran the app to search music events from *Last.fm* and *Upcoming*. We repeated the search for the 10 biggest cities in the United States. The observed average dimension of the integrated result set was 356.11kB, with a maximum of 536.8kB. We can, therefore, say that the memory consumption is reasonable even on mobile devices with limited capabilities.

6.2. Usability of the Composition Paradigm

We assessed the usability of the composition paradigm through a user study that involved 36 participants with varying work activities (e.g., students, employees, housewives), age (18–70), and skills. All had experience with the Internet and the mobile world. Seventeen users were "experts," that is, with some programming skills but without any previous experience with service composition. Nineteen users were "novice," that is, never exposed to technical activities. The study specifically focused on the effectiveness and intuitiveness of the composition paradigm, trying to measure such factors in terms of efficiency, ease of use, and user satisfaction. We expected all users to be able to complete the experimental tasks. However, we also expected a greater efficiency (e.g., a reduced time for task completion) and a more positive attitude (in terms of perceived usefulness, acceptability and confidence with the tool) by expert users. Their technical background could indeed facilitate the comprehension of the experimental tasks and improve the perception of the control over the composition method and thus their general satisfaction.

Users were asked to fill in a pretest questionnaire to gather data on their knowledge about computer science, services, and mashups. Then a training session followed in which one researcher of our group gave a 10-minute demonstration to introduce the participants to the design environment and the basic composition mechanisms. All the

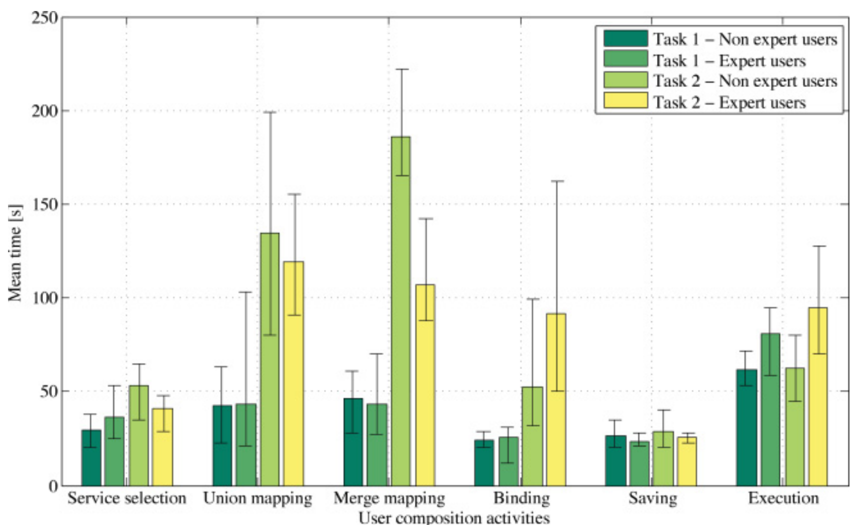


Fig. 11. Completion time of the different composition steps within the two evaluation tasks for novices and expert users.

users were then asked to perform two composition tasks with incremental difficulty. Task 1 consisted in creating a component by using only one service and the list UI template. The aim was to let the users become familiar with the basic visual mapping mechanism. Task 2 was more demanding, asking the users to visually integrate the data retrieved by two services by using a map-based template. Two researchers observed the interaction and took notes of the difficulties encountered by the participants and of their comments. After the completion of each experimental task, users were asked to use the application they had just composed, to check the results of their composition. At the end, they filled in a satisfaction questionnaire.

Efficiency. All participants were able to complete both tasks without showing severe difficulties. Figure 11 illustrates the average time for the completion of the two tasks, distinguishing among the different phases in the composition process. The times denote a satisfying ease of use of the system; however, from the collected qualitative data, it resulted that some users complained about the initial difficulty in interpreting the service result set shown in the data panel. Most of the time spent for the visual mapping was indeed due to interpret the service results and especially to identify relevant data attributes. However, all the users, and especially the experts who knew what service querying generally requires, remarked that the visual representation makes service querying easier in comparison to manual programming, especially because it hides the XML or JSON syntax generally returned by services.

No statistically significant differences in task completion time were found between experts and novices. A Wilcoxon-Mann-Whitney test showed that technology expertise was not discriminant for both Task 1 ($U = 15.5$, $p = .55$) and Task 2 ($U = 17$, $p = .42$). The average time to complete Task 1 was about 3.83 minutes for novices and 4.23 minutes for experts, whereas for Task 2 it was about 8.61 minutes for novices and 7.99 minutes for experts. The difference in completion times for the two tasks reflects the greater number of steps in Task 2 with respect to Task 1.

The lack of significant differences between the two groups does not mean that the experts performance was bad. Indeed, the completion times for Task 2, which is the most significant in terms of complexity, revealed a trend toward a better performance of

the expert users. Completion time for Task 1 was unexpectedly higher for experts; the reason, however, was especially the curiosity that experts showed and their engagement in the composition tasks—they wanted to understand exactly how the method works and, as also shown by the higher times for the execution of the created app during Task 2 (see Figure 11), they were very diligent in checking the matching of the created application with the previously performed composition. The novices instead trusted more the correctness of the tool; some of their comments indeed expressed that having a direct feedback of their composition actions helped them checking that the tool was performing correctly.

The user efficiency was also analyzed along the number of actions performed to complete the two tasks; the aim was to assess whether deviations occurred due to lack of orientation. Also for this variable, no significant differences were found between experts and novices both for Task 1 ($U = 17.5$, $p = .31$) and for Task 2 ($U = 18$, $p = .31$), although, even in this case, the analyzed data revealed a trend toward a better performance of expert users. In general, based on what we observed during the test, the absence of significant differences for all the analyzed variables can be interpreted as an indication that the tool enables even inexperienced users to complete composition tasks in a limited time, independently of the users' technical experience.

Ease of use. This dimension was assessed through some questions in the postquestionnaire that asked users to rate how easy it was to identify and include services in the composition and to define visually data integration and service coupling. We also asked users to score globally the ease of use of the method. Users could modulate their evaluation on a 5-point scale. The reliability of the questions was good ($\alpha = .81$). The correlation between the detailed questions and the ease-of-use global score was also satisfying ($\rho = .58$, $p < .001$). This highlights the good external reliability of the measures. On average, the users judged very good the ease of use of the tool ($mean = 3.7$, $meanS.E. = .58$—the scale was from 1 = very negative to 5 = very positive). Also, a t-test did not find significant differences between novice and expert users ($t_{(46)} = .875$, $p = .39$).

User satisfaction. It was assessed by using a semantic-differential scale that required users to judge the method on 12 aspects. Users could modulate their evaluation on a 5-point scale (1 = very negative to 5 = very positive), whose reliability was assessed as satisfying ($\alpha = 0.73$). A user-satisfaction index, SI, was computed as the mean value of the score across all the 12 items; its average value was very good ($mean = 4.18$, $meanS.E. = 0.46$). Moreover, a question asked users to globally score the method on a 10-point scale (1 = very negative to 10 = very positive). The mean value of the global satisfaction, GS, is very good ($mean = 8.3$, $meanS.E. = .86$). The correlation between the SI index and the GSscore is satisfying ($\rho = .41$, $p < .015$).

The last two questions asked users to judge their performance as mashup developers and to indicate the percentage of requirements they believed they had satisfied with their composition, based on the observation of the application they were able to generate. This metric can be considered as a proxy of confidence. On average, users stated to be very confident about their performance as app developers, being able to cover the 92% of the requirements specified by the two experimental tasks ($min = 50\%$, $max = 100\%$, $meanS.E. = 13\%$). They also felt very satisfied about their performance as composers ($mean = 3.4$, $meanS.E. = .52$; 1 = very negative, 4 = very positive). We consider this as an indication of the perceived effectiveness of our composition method.

In summary, contrarily to what we expected, the experimental data revealed that the lack of technology skills did not influence the user performance or the perception on the ease of use of the tool. Qualitative data also revealed a very good satisfaction by the expert users. We initially expected expert users to be bored by the composition tasks

because they were familiar with programming. Instead, several observations revealed they were enthusiast about the method, recognizing it could help them speed up the creation of applications. They, in particular, recognized several advantages related to the possibility to create applications portable on different devices. They indeed observed how even expert programmers could have difficulties mastering the technologies on different execution platforms.

Threats to Validity. We now analyze some issues that may have threatened the validity of the user study, also to highlight under which conditions the study design offers benefits that can be exploited in other contexts, and under which circumstances it might fail.

Internal validity can be threatened by some hidden factors compromising the achieved conclusions:

—*Learning effect.* A learning effect could, in general, offer an advantage to the participants during the experiment, enabling them to improve their performance along the series of assigned tasks. In our study, this factor was alleviated by asking each subject to apply each of the two tasks starting from different services; different order combinations of services were considered for the different users.
—*Subject experience.* It was alleviated by the fact that none of the subjects had any experience with our tool. We further tried to alleviate this threat by not disclosing much information during the short training session performed prior to the experiment.
—*Method authorship.* We eliminated the biases that different facilitators running the experiment could introduce, as we had the same instructor for every session of the study. In this way, we avoided any variability in the initial training as well as in the way users had been observed.
—*Information exchange.* Since the study took place over 10 days, it is difficult to be certain whether the involved subjects did not exchange any information. However, participants were recruited from different working and living contexts; thus, for many of them, it was difficult to know each other and communicate. The participants were asked to return all the material (e.g., the description of tasks) at the end of each session. We asked participants coming from the same contexts (e.g., students of a same course or employee of a same company) to perform the test in the same day, in close or even parallel sessions.
—*Understandability of the material.* Possible misunderstandings expressed by the users in the different sessions were cleared up without, however, providing details that could help them accomplishing the tasks.

External validity refers to the possible approximation of truth of conclusions in the attempt to generalize the results of the study in different contexts. With this respect, the main threats of our study are:

—*Representativeness of the results.* Given the context in which the study took place, and especially the nature of the assigned tasks (the tasks referred to the search of music events in the Milan area), the results could be representative with regard to casual users and to the leisure domain (mono-method bias). We cannot predict the effect of the use of the tool in other domains, by knowledgeable users and for the creation of more critical applications (e.g., in an enterprise context). Given the generality of the composition paradigm with respect to the involved services, we can, however, suppose that the experience of users in a given domain would not contrast the results of our study, but rather it would accentuate the trend toward a positive user experience in terms of efficiency, effectiveness, and satisfaction. It is also true that our aim was to assess the usability of the interaction paradigm, not the capability of the tool to improve the user productivity in given domains. Therefore,

we believe the representativeness of the results is not much affected. To alleviate this threat, we have already started carrying out a series of new user studies to validate other customizations of our platform in specific working domains [Ardito et al. 2012, 2014].

—*Complexity of composition tasks.* To limit the duration of the experiment, we decided to use relatively small composition tasks applied to few representative data sources. We also limited the number of UI components the users could choose among. Therefore, we were not able to assess how easy it is for the users to identify the right services. It is worth noting that in this study we were not much interested in aspects related to selection of services, which we instead investigated in other works [Cappiello et al. 2012]. However, it could be interesting to assess through further studies the usability of the assistance mechanisms that our tool offers for service selection.

Construct validity might have been influenced by the measures that we applied in the quantitative analysis and by the reliability of the questionnaire. We alleviated the first threat by adopting measures, such as efficiency, that are commonly employed in user studies [Dix et al. 2003]. In addition, the subjective measures, such as perceived ease of use and perceived satisfaction, are based on the Technology Acceptance Model (TAM) [Davis 1989], a well-known and thoroughly validated model for evaluating information technologies. The reliability of the questionnaire was tested by applying the Cronbach test to each set of closed questions intended to measure subjective variables. As reported in the previous section, all the values obtained were higher than the acceptable minimum threshold (>0.70) [Maxwell 2002].

Conclusion validity refers to the validity of the statistical tests applied. In our study, this was alleviated by applying the most common tests that are employed in the empirical software engineering field [Juzgado and Moreno 2001].

6.3. Task-Weighted Usability of the Composition Paradigm

In addition to evaluating the usability of the composition paradigm, we also conducted an expert-based usability inspection to analyze in detail the flow of the composition tasks and the way the complexity of the tasks influences the usability of the composition environment. For the analysis, we adopted the method illustrated in Dyk and Renaud [2004], which proposes some measures to weigh the outcome of heuristic evaluations of Web sites by some factors that take into account the nature of the tasks (i.e., their repetition and complexity).

By surveying works on the usability evaluation of mashup tools [Sarraj and Troyer 2010; Namoun et al. 2010b; Aghaee and Pautasso 2013], we identified a set of usability criteria specific for this class of applications. We then selected the most relevant ones and revised their formulation to take into account the peculiarities of our composition process. The resulting set of criteria is reported in Table II. We finally associated the criteria with composition phases and detailed steps occurring in each phase. In particular, we considered three main phases and in total six different steps that correspond to the composition steps we also focused on during the user study described in the previous section:

—**Phase 1—Service Selection**: It refers to the initial steps for the users to get oriented into the composition editor (**S1**) and to make a decision on their composition goal through the selection of data components and UI templates (**S2**). The choice of the UI template is trivial, while the service selection might be more demanding due to the need of understanding what content is provided by the different services registered into the platform. Steps S1 and S2 can be iterated till the user makes a decision.

Table II. Heuristics for the Different Phases and Steps of the Composition Process

Phase 1: Service Selection	
Steps	*Criteria*
S1	**Ph1_C1**. Is it clear, based on the adopted notation and visual mechanisms, how users can access the available data sources and a UI templates?
S1	**Ph1_C2**. Does the mashup tool force a user to start by defining new queries/components before s/he can do anything else?
S2	**Ph1_C3**. How easy is it to identify what kind of data the available data components are able to provide?
S2	**Ph1_C4**. How easy is to identify what kind of visualizations the available UI templates are able to provide?
S2	**Ph1_C5**. If two different elements have to be compared (data components and/or UI templates), can you see them at the same time?
S2	**Ph1_C6**. Does the system inform users of the success or failure of their selection actions?
Phase 2: Visual Mapping	
Steps	*Criteria*
S3	**Ph2_C1**. Are possible composition actions easy to understand?
S3	**Ph2_C2**. Does the mashup tool allow the user to fool around or make sketchy things when one is not sure which way to proceed?
S3	**Ph2_C3**. Is it clear what a user should do to complete the visual mapping task?
S4	**Ph2_C4**. Do associating data items to UI template seem especially complex or difficult to work out (e.g., when combining data from several services)?
S3	**Ph2_C5**. Can the user easily go back to undo some operations (e.g., addition of a data item, addition of a component)?
S3	**Ph2_C6**. Does the mashup tool use an easy composition paradigm which makes it difficult to make mistakes?
S3	**Ph2_C7**. Does the system inform users of the success or failure of their visual mapping actions?
S5	**Ph2_C8**. Are user actions linked to changes in the interface? Can users easily check the effect of their composition actions?
S5	**Ph2_C9**. Is it easy to find out the progress made, or check what stage of the work has been reached?
S5	**Ph2_C10**. Is it possible to try out partially completed versions of the mashup?
Phase 3: App execution	
Steps	*Criteria*
S6	**Ph3_C1**. Is it clear how to store the app schema on the remote repository?
S6	**Ph3_C2**. How easy is it to download and install the app on the user device?
S6	**Ph3_C3**. Does the system inform users of the success or failure of their actions?

—**Phase 2—Visual Mapping**: It includes the steps for identifying how to accomplish the composition tasks, that is, which are the relevant data items and the visual renderers they can be associated with (**S3**), for executing the visual mapping actions (**S4**), and for checking if the result of the visual mapping corresponds to the initially defined goals (**S5**). Given the live programming paradigm offered by our platform, Steps S2–S5 can be iteratively repeated by the users.
—**Phase 3—Mashup Execution**: It consists of saving the mashup schema and executing the corresponding app (**S6**), as a component in the Web dashboard or as a self-contained app in mobile execution environments. Saving the VI component schema is trivial, while opening it within any execution environment can be more complex, especially if the component has to be executed on a mobile device.

Following the method defined in Dyk and Renaud [2004], three independent evaluators (three researchers of our department already exposed to our method) analyzed the Web design environment and assigned a usability score to each criteria. They, in particular, gave a score to each single page enabling the execution of the steps the heuristic referred to, according to a scale of four values, ranging from 0 = feature never

Table III. Task-Weighted Usability Scores for Phase 2 (Visual Mapping)

Steps	Criteria	Score	Max Score	Score/Max	R+C	TWF	UC
S3	Ph2_C1	12.00	18.00	0.67	1+0.5	1.67	0.08
S3	Ph2_C2	12.00	18.00	0.67	1+0.1	1.22	0.06
S3	Ph2_C3	12.00	18.00	0.67	0.8+0.8	1.78	0.08
S4	Ph2_C4	7.00	9.00	0.78	1+1	2.2	0.12
S4	Ph2_C5	6.00	9.00	0.67	1+0.1	1.2	0.06
S4	Ph2_C6	9.00	9.00	1.00	1+0.1	1.2	0.09
S4	Ph2_C7	9.00	9.00	1.00	1+0.1	1.2	0.09
S5	Ph2_C8	9.00	9.00	1.00	1+0.1	1.2	0.09
S5	Ph2_C9	9.00	9.00	1.00	1+0.1	1.2	0.09
S5	Ph2_C10	9.00	9.00	1.00	0.8+0.1	1	0.07
Raw score	**80.34**						
Task-weighted score	**82.45**						

available to 3 = feature universally available. Each evaluator then assigned a global score to each criteria summing up the scores given to each single page. The global scores of all evaluators were in turn summed up to compute the final score.

Table III reports the measures we computed for the different heuristics for Phase 2. We report the detailed values of the measure only for this phase because it is the central phase of our composition process and also the most demanding for the users. For example, given the criteria $PH2_C1$, the sum of all the scores assigned by reviewers to the three pages for accessing the list of available services and UI templates is 12 (column *Score*). The column *MaxScore* then reports the maximum score that could be assigned to that criteria, while the next column reports the ratio between the *Score* and *MaxScore*.

Besides assigning usability scores to each criteria, the evaluators also assigned suitable values to the repetition (R) and the complexity (C) of each step. R was quantified through values ranging from 0.1 = low occurrence to 1 = regular occurrence through the interaction. Similarly, C was quantified with values ranging from 1 = high complexity to 0.1 = low complexity. For example, the criteria $Ph2_C1$ was assigned with $R = 1$ and $C = 0.5$ because it is highly repetitive and has a medium complexity.

The ratio $\frac{R+C}{max(R+C)}$ was then computed to indicate how important a particular criteria is, also taking into account the repetition and the complexity of the task it refers to. In Table III, these values for all the criteria are reported in the column *TWF* (*Task Weighted Factor*).

For each criteria, the final usability coefficient was then computed as

$$UC = \frac{Score}{MaxScore} \cdot \frac{R+C}{\sum(R+C)}$$

The overall usability scores for each phase were then computed as follows:

—The percentage raw (i.e., non–task-weighted) usability score is given by

$$RawScore = \frac{\sum(Score)}{\sum(MaxScore)} \cdot 100;$$

—The percentage task-weighted usability score is given by

$$Task\ Weighted\ Score = \sum UC \cdot 100.$$

As reported in Table III, the two measures for Phase 2 are 80.34 and 82.45, respectively. In other words, considering the nature of tasks led to an increase of the overall usability score. This is an indication of the attention we paid on the usability of the tasks with a higher repetition and complexity.

The scores totaled for the other two phases are:

—Phase 1: $RawScore = 76.7$, $TaskWeightedScore = 73.94$;
—Phase 3: $RawScore = 77.78$, $TaskWeightedScore = 64.71$.

The values for Phase 1 indicate in general a good usability level, even though the task-based weighting revealed that perhaps more attention could have been given to some specific tasks. In particular, this was true for the Step S2 ("Decision on goals based on service and UI template selection"), which totaled an $R+C$ score of 0.33 for the criteria *Ph1_C5* related to the comparison of different result sets or UI templates. We are indeed aware that this activity in our composition environment is still complex for the end user because result sets from different services (and also different UI templates) are shown in separate tabs. The user is, therefore, forced to go back and forth through different tabs if he wants to compare them.

Similar observations also apply to the score computed for Phase 3. In particular, the task-weighted score is downgraded by the evaluation of criteria *Ph3_C2*, which relates to the download and installation of the created app. This activity might result indeed difficult for novice users and we are aware the evaluated version of the platform does not provide adequate assistance.

6.4. Usability of the Generated Mashups

In the early phases of the platform design, we conducted a preliminary assessment of the usability of the generated mashups. Through a survey, we asked the users to judge one such mashup and to provide feedback about its functionality and organization. Besides defining a usable composition paradigm, we indeed wanted to make sure the target applications would be usable too.

We especially addressed mashups to be run on smart phones, as we believe the user experience with mobile applications is more critical given the limited capabilities of the target devices. In order to reach a reasonable number of users, we created an animated mockup emulating the interaction with a generated application, and we published it online together with a questionnaire aimed to gather the opinions of the users. The mockup was organized as a guided tour: the users could emulate the interaction with the mashup by clicking on the available buttons and links and proceed step by step through the different application screens. The emulated application allowed the users to choose among two different mashups (one about music events, the other about movies) to simulate the situation in which the user creates and stores on the platform repository multiple application schemas. Each mashup was configured to retrieve and integrate data from multiple services, according to the data fusion policy described in the previous sections. The users could also refine the selection of components (both data and UI components) and choose which services had to be actually used at runtime. They could also configure dynamically some filters on the retrieved data.

Forty-eight users responded to the survey. They were students, employees, managers, and workers, and their age ranged from 20 to 35 years. The results were very positive. Eighty-two percent found that interacting with the application was easy and intuitive. On a scale of 5 values (from 1 = not easy at all to 4 = easy and 5 = very easy), 59% of users judged it "very easy," and 23% "easy." The majority of the users then especially appreciated some characterizing features:

—Eighty-two percent of the participants appreciated the possibility to further personalize the application by choosing at runtime the data sources to be considered for data extraction. In particular, 50% of users found it excellent and 32% good.
—Ninety percent of users appreciated the possibility to dynamically select the UI components to be actually coupled at runtime with the integrated dataset.

—Ninety percent of the participants judged excellent some mechanisms provided by the applications to filter the integrated result set (e.g., the one based on user-specified keywords).

Such results reveal a very positive users' attitude about some features that intrinsically characterize our approach (e.g., the flexibility in the choice of the data and UI components) and also some other mechanisms specifically added to ease the fruition of data (e.g., the possibility to personalize filters).

It is worth noting that the usability of the generated applications was also indirectly assessed through the last question in the survey filled in by composers during the study on the usability of the composition paradigm (see previous section). This question asked the users to judge their experience as composer of the created application. The reaction to this question was very positive, and this can be considered an expression of satisfaction not only about the composition paradigm but also about the created application.

7. LESSON LEARNED

We conclude this article by outlining some guidelines for the EUD of mashups that we derived first by analyzing the most notable EUD requirements studied and validated in different user-centric studies [Green and Petre 1996; Burnett et al. 2004; Lieberman et al. 2004; Ko et al. 2004; Wu et al. 2007] and also by "learning" from real users while observing their behavior. We indeed conducted several user studies that allowed us to analyze working communities using our platform to accomplish their specific tasks [Cappiello et al. 2011b; Ardito et al. 2012, 2013; Picozzi et al. 2013].

Closeness of mapping. In order to help users understand the features provided by the available components, the effect that each component may have on the overall composition, the way it can be integrated with other components, it is important to come up with representations of services that abstract from technical details and instead increase the expressiveness of the roles that services play in the mashup. In other words, the effect that can be achieved through the inclusion of components needs to be emphasized [Green and Petre 1996]. As proposed by our method, one solution could be to let users manipulate (e.g., add, remove, or modify) visual objects and visual properties. Also, actions on visual objects should immediately produce the visualization of their effects on the data and the UI of the corresponding component. Therefore, users can operate on service visualization properties rather than configuring technical details for service invocation and integration. Experiments with real users, conducted by us within our research project [Cappiello et al. 2011a] as well as by other independent researchers [Namoun et al. 2010a, 2010b], assessed that this kind of visual approach, focusing on some forms of immediate visualization of services, increases the user-perceived usefulness and ease of use. As highlighted by the user study described in this article, self-efficacy (i.e., the user perception of being in control of the composition process) is also enhanced.

Progressive evaluation. In order to further enhance the users' perception of and the control on the effects that services and composition actions have on the application under construction, it is important to provide feedback on "how the user is doing" [Green and Petre 1996]. The end users are not able to distinguish between design and execution phase; therefore, since the very first composition action, the user must be able to see a running application and to observe incrementally the effect of any other subsequent composition action. Immediate execution paradigms, based on WYSIWYG representations, where each single composition action is observable through the immediate execution of the modified application, are therefore good candidates to

support progressive evaluation. These mechanisms also avoid the so-called *premature commitment* because the user is not forced to make decisions without being able to observe and evaluate the effect of such decisions. Together with closeness of mapping mechanisms, they also contribute to enhance self-efficacy.

Composition assistance. In order to further smooth design barriers, it is also important to aid those users that do not have sufficient development knowledge. Composition can be assisted or guided in multiple ways, for instance, by providing default system-driven service couplings, when possible, while also giving to the users the right level of control to modify the automatically provided solutions, or offering recommendations for further services and composition patterns that can fit the current composition. This article does not focus on this specific aspect; however, we assessed the effectiveness of both these kinds of assistance mechanisms in our previous work [Cappiello et al. 2012]. In general, as also proved by other independent studies [Namoun et al. 2010b], end users find helpful any kind of hints that the system is able to provide during the mashup composition.

Abstraction gradient. Another fundamental ingredient is to accommodate different users skills and attitudes and also varying composition contexts. In other words, users must be provided with different abstraction levels [Green and Petre 1996], to ensure a "gentle slope of difficulty" [Lieberman et al. 2004]. Different "composition styles" can be offered to reflect different composition granularities. For example, our platform enables the end users to embed prepackaged, ready-to-use components into a composition workspace, possibly taking advantage of preconfigured rules for the automatic definition of component couplings [Cappiello et al. 2011b], without any need of defining additional settings for components execution and integration. However, the users can also define additional integration logic to synchronize the UIs of the different components [Yu et al. 2007] as well as build their own UI components. Other approaches also found it useful to provide different representation metaphors accommodating different levels of complexity. For example, the NaturalMash platform [Aghaee et al. 2013] offers a natural language paradigm to express desiderata on components to be added in a mashup. The tool then offers a WYSIWIG representation of components similar to the one offered by our platform as well as an additional wired notation to let the users express component couplings.

Domain-specific focus. In order to allow users to understand the possibilities offered by the mashup platform and to make sense of the services and components that are available for composition, it is important to restrict the platform to a well-defined domain the user is comfortable with. That is, it is important to develop a general platform that can be, however, easily customized as far as the offered components and the provided composition features are concerned. In this regard, our approach allows the end users to create their own components. Domain specificity, however, in some cases, requires the intervention of experts to customize the composition platform. In this respect, our approach specifically promotes metadesign scenarios [Fischer et al. 2004], since component creation can also be performed by domain experts who exploit their domain knowledge to create sensible components for the end users. Moreover, the separation between the mashup UI and the mashup application logic, which is enforced by our Visual Integration model, facilitates the adoption of different UIs, thus different interaction metaphors, to accommodate the needs and the background of specific users' communities. The customization of UI templates is indeed a requirement emerged in some studies we have been conducting to validate some aspects of our methodology related to metadesign and domain specificity [Ardito et al. 2012].

Elasticity. So far, software systems have been conceived as prepackaged sets of data, functionality, and visualizations that software developers build for the end users. Elastic systems [Latzina and Beringer 2012] diverge from such idea and try to promote paradigms where contents, functionality, and presentations on different devices are totally decoupled from each other and from specific contexts of use. They emerge at use time, depending on the actions for data exploration and composition that the users perform while fulfilling their situational information needs. Elasticity, in other words, goes beyond the adoption of easy-to-use composition paradigm, but it goes beyond, aiming to define flexible frameworks that can enlarge the validity of applications across varying needs. The idea is to capitalize on a proper separation of concerns for not constraining a platform to a specific context of use. Software design patterns, first of all Model-View-Controller (MVC), already addressed separation of concerns. However, elasticity focuses not on programming practices to facilitate the development and maintenance of the final applications. Rather it aims to empower the end users to shape up their applications dynamically. The approach discussed in this article is totally in line with the notion of elasticity. We pursue it especially thanks to the adoption of UI templates that act as elastic containers adaptable to different interaction metaphors and different access devices. This aspect allowed us to focus on the definition of general composition paradigms characterized by natural, UI-centric mechanisms that exploit separation of concerns to facilitate the specialization to different domains, users characteristics, and usage situations.

8. CONCLUSION

In this article we introduced our perspective over the EUD of mashups. We discussed how a UI-centric model, mainly based on UI templates, can enable a lightweight integration process for the definition of unified data views and their synchronization with remote APIs or device local services. For the execution of the resulting mashups, we adopt an "on-demand" data fusion technique; thus, the efforts required for comparing and merging different datasets concentrate only on the information the user is really interested in. Indeed, we also aim to achieve lightweight architectures and integration policies, mainly deployable on client devices that might also have limited computing capabilities. Some validation experiments proved that this choice requires reasonable computational costs and is effective for the end users. The article also discussed how different architectural choices (e.g., privileging data integration on the server) can easily replace the current client-side logic and further optimize the mashup execution performance.

The composition method illustrated in this article mainly addresses casual end users with limited (or even totally absent) programming experience. To simplify the composition, the platform makes use of some default settings, for example, related to the operations and the events that each VI component exposes. User studies highlighted that despite these simplifications, which limit in a sense the freedom of the composers, expert users also felt comfortable when using the platform. On the one hand, experts might prefer more flexibility as they are used to programming their applications without any constraint; on the other hand, we are also very confident that few extensions to the composition paradigm can fulfill their expectations. For example, they can be enabled to configure advanced features of components, such as more complex queries, an extended event-driven logic, and more sophisticated policies for data fusion and conflict resolution. Our current work is devoted to improving the design environment along this direction, trying to achieve a full-fledged approach accommodating different expertise levels.

The possibility to deploy applications that can be realistically used by the end users has been assessed in different usage contexts where our platform has being used:

—When supporting a community of managers at the Milan Municipality in monitoring the city tourism services through the sentiment analysis of user-generated contents [Cappiello et al. 2011b].
—When supporting a community of professional guides of an archaeological park in creating applications executable on mobile devices and multitouch screens. Such applications help the guides in their presentations during the park visit and allow them to share material with the park visitors [Ardito et al. 2012, 2013].
—When supporting city traffic planners by generating synchronized visualizations of traffic data gathered by multiple city sensors [Picozzi et al. 2013].

We are also aware of some limitations of our approach:

—First of all, the openness of the platform that we wanted to strengthen through the creation of VI components could be hindered by the need for registering data components. This is true even if the registration activity is easy to perform. The standardization of methods for API invocation and instantiation, or the definition of API ecosystems characterized by homogeneous programming interfaces would help a lot with this respect. These are topics the research community is now starting investigating. However, the current heterogeneity of formats and protocols necessarily demands for some descriptive layers through which the platform can understand how to query services.
—The addition in the platform of wrapped UI components might be a further, even stronger barrier; different of data component registration, UI component wrapping is totally out of reach for casual users and requires expert developers to perform the initial population of the platform with relevant UI components. For this reason, also in relation to the registration of data components, we envisage the adoption of our platform in metadesign scenarios, where other stakeholders (i.e., expert programmers and domain experts) are supposed to configure the platform for its initial use by the end users. However, we expect that the method for the creation of VI components be fruitful for the addition of new components by the end users.
—The definition of visual templates currently requires the creation by hand of HTML/JavaScript presentation templates. We are, however, working on the creation of an environment where visual templates can be defined visually. Also, the dynamic (i.e., at runtime) shift from one template to another, to pursue the notion of elasticity, requires assessing if the nature of data makes it possible the visualization change and identifying how a same dataset could be visualized through different visualization styles. In our current prototype, the user is in charge of assessing these aspects. We are, however, defining techniques to support a full polymorphism of the integrated result sets.
—Native runtime engines optimize the application execution on different devices, but they can also limit portability. For this reason, we are now improving our mashup execution logic in the Web browser, by exploiting HTML5 APIs for the invocation of remote and device-local services, and responsive CSSs and media queries for optimizing the presentation on devices with varying screen size.

Despite these known limitations, the conducted user studies encouraged us as they showed that users appreciated our approach and were able to use the platform with efficiency and satisfaction. Such studies also allowed us to collect very useful feedback on possible extensions of the approach. For example, the need of collaborative mechanisms for sharing and co-creating artifacts largely emerged. Some recent extensions of our platform already go toward this direction [Matera et al. 2013; Ardito et al. 2014]; we are now working to further improve and validate such new features.

We are also designing new extensions to enable the definition of mashups whose composition and execution is "distributed" along multiple devices. This entails an evolution of the mashup logic, from a single-instance, stateless application to a long-lasting, stateful application where both the application schema and the state representation need to be maintained across multiple instances and different sessions. Such a new logic will also require the revision of the composition models and of the interactive composition paradigm, to take into account the concurrent contributions of multiple users to the creation of applications that can be in turn distributed across several users and execution devices.

REFERENCES

Saeed Aghaee and Cesare Pautasso. 2013. Guidelines for efficient and effective end-user development of mashups. In *Proceedings of the 4th International Symposium on End-User Development (IS-EUD'13)*, Yvonne Dittrich, Margaret M. Burnett, Anders I. Mørch, and David F. Redmiles (Eds.). Lecture Notes in Computer Science, Vol. 7897. Springer, 260–265.

Saeed Aghaee, Cesare Pautasso, and Antonella De Angeli. 2013. Natural end-user development of web mashups. In *Proceedings of the 2013 IEEE Symposium on Visual Languages and Human Centric Computing.* IEEE, 111–118.

Carmelo Ardito, Paolo Bottoni, Maria Francesca Costabile, Giuseppe Desolda, Maristella Matera, Antonio Piccinno, and Matteo Picozzi. 2013. Enabling end users to create, annotate and share personal information spaces. In *Proceedings of the 4th International Symposium on End-User Development (IS-EUD'13).* Lecture Notes in Computer Science, Vol. 7897. Springer, 40–55.

Carmelo Ardito, Paolo Bottoni, Maria Francesca Costabile, Giuseppe Desolda, Maristella Matera, and Matteo Picozzi. 2014. Creation and use of service-based Distributed Interactive Workspaces. *J. Vis. Lang. Comput.* 25, 6 (2014), 717–726. DOI:http://dx.doi.org/10.1016/j.jvlc.2014.10.018

Carmelo Ardito, Maria Francesca Costabile, Giuseppe Desolda, Maristella Matera, Antonio Piccinno, and Matteo Picozzi. 2012. Composition of situational interactive spaces by end users: A case for cultural heritage. In *Proceedings of the 7th Nordic Conference on Human-Computer Interaction (NordiCHI'12).* ACM, 79–88.

Carmelo Ardito, Maria Francesca Costabile, Giuseppe Desolda, Rosa Lanzilotti, Maristella Matera, and Matteo Picozzi. 2014. Visual composition of data sources by end users. In *Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI'14)*. ACM, 257–/260. DOI:http://dx.doi.org/10.1145/2598153.2598201

Jens Bleiholder and Felix Naumann. 2008. Data fusion. *ACM Comput. Surv.* 41, 1 (2008).

Alessandro Bozzon, Stefano Ceri, and Srdan Zagorac. 2012. Materialization of web data sources. In *SeCO Book*, Stefano Ceri and Marco Brambilla (Eds.). Lecture Notes in Computer Science, Vol. 7538. Springer, 68–81.

Stefano Burigat and Luca Chittaro. 2013. On the effectiveness of Overview+Detail visualization on mobile devices. *Pers. Ubiquitous Comput.* 17, 2 (2013), 371–385. DOI:http://dx.doi.org/10.1007/s00779-011-0500-3

Margaret M. Burnett, Curtis R. Cook, and Gregg Rothermel. 2004. End-user software engineering. *Commun. ACM* 47, 9 (2004), 53–58.

Marcos Càceres. 2012. *Packaged Web Apps (Widgets): Packaging and XML Configuration* (2nd ed.). W3C Recommendation. Retrieved from http://www.w3.org/TR/widgets/.

Cinzia Cappiello, Florian Daniel, Maristella Matera, Matteo Picozzi, and Michael Weiss. 2011a. Enabling end user development through mashups: Requirements, abstractions and innovation toolkits. In *Proceedings of the 3rd International Symposium on End-User Development (IS-EUD'11).* Lecture Notes in Computer Science, Vol. 6654. Springer, 9–24.

Cinzia Cappiello, Maristella Matera, Matteo Picozzi, Florian Daniel, and Adrian Fernandez. 2012. Quality-aware mashup composition: Issues, techniques and tools. In *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology (QUATIC'12).* IEEE Computer Society, In print.

Cinzia Cappiello, Maristella Matera, Matteo Picozzi, Gabriele Sprega, Donato Barbagallo, and Chiara Francalanci. 2011b. DashMash: A mashup environment for end user development. In *Proceedings of the 11th International Conference on Web Engineering (ICWE'11).* Lecture Notes in Computer Science, Vol. 6757. Springer, 152–166.

Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. 1999. *Readings in Information Visualization: Using Vision to Think*. Academic Press.

Fabio Casati, Florian Daniel, Antonella De Angeli, Muhammad Imran, Stefano Soi, Chritopher R. Wilkinson, and Maurizio Marchese. 2012. Developing mashup tools for end-users: On the importance of the application domain. *IJNGC* 3, 2 (2012).

Stefano Ceri, Maristella Matera, Francesca Rizzo, and Vera Demaldé. 2007. Designing data-intensive web applications for content accessibility using web marts. *Commun. ACM* 50, 4 (2007), 55–61.

Prach Chaisatien, Korawit Prutsachainimmit, and Takehiro Tokuda. 2011. Mobile mashup generator system for cooperative applications of different mobile devices. In *Proceedings of 11th International Conference on Web Engineering (ICWE'11)*. Lecture Notes in Computer Science, Vol. 6757. Springer, 182–197.

José Danado, Marcin Davies, Paulo Ricca, and Anna Fensel. 2010. An authoring tool for user generated mobile services. In *Future Internet (FIS'10)*. Lecture Notes in Computer Science, Vol. 6369. Springer, Berlin, 118–127. DOI:http://dx.doi.org/10.1007/978-3-642-15877-3_13

Florian Daniel, Fabio Casati, Boualem Benatallah, and Ming-Chien Shan. 2009. Hosted universal composition: Models, languages and infrastructure in mashart. In *Conceptual Modeling - ER 2009, Proceedings of the 28th International Conference on Conceptual Modeling (ER'09)*. Lecture Notes in Computer Science, Vol. 5829. Springer, 428–443.

Florian Daniel and Maristella Matera. 2014. *Mashups - Concepts, Models and Architectures*. Springer. DOI:http://dx.doi.org/10.1007/978-3-642-55049-2

Florian Daniel, Maristella Matera, and Michael Weiss. 2011. Next in mashup development: User-created apps on the web. *IT Professional* 13, 5 (2011), 22–29.

Marcin Davies, Anna Fensel, François Carrez, Maribel Narganes, Diego Urdiales, and José Danado. 2010. Defining user-generated services in a semantically-enabled mobile platform. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications and Services (iiWAS'10)*. ACM, 333–340.

Fred D. Davis. 1989. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Q.* 13, 3 (Sept. 1989), 319–340. DOI:http://dx.doi.org/10.2307/249008

Antonella De Angeli, Alberto Battocchi, Soudip Roy Chowdhury, Carlos Rodríguez, Florian Daniel, and Fabio Casati. 2011. End-user requirements for wisdom-aware EUD. In *Proceedings of the International Symposium on End-User Development (IS-EUD'11)*, Maria Francesca Costabile, Yvonne Dittrich, Gerhard Fischer, and Antonio Piccinno (Eds.) Lecture Notes in Computer Science, Vol. 6654. Springer, 245–250.

Alan Dix, Janet E. Finlay, Gregory D. Abowd, and Russell Beale. 2003. *Human-Computer Interaction* (3rd ed). Prentice-Hall, Upper Saddle River, NJ.

Gerhard Fischer. 2009. End-user development and meta-design: Foundations for cultures of participation. In *Proceedings of the 2nd International Symposium on End-User Development (IS-EUD'09)*. Lecture Notes in Computer Science, Vol. 5435. Springer, 3–14.

Gerhard Fischer, Elisa Giaccardi, Yunwen Ye, Alistair G. Sutcliffe, and Nikolay Mehandjiev. 2004. Meta-design: A manifesto for end-user development. *Commun. ACM* 47, 9 (2004), 33–37.

Thomas R. G. Green and Marian Petre. 1996. Usability analysis of visual programming environments: A 'Cognitive Dimensions' framework. *J. Vis. Lang. Comput.* 7, 2 (1996), 131–174.

Jonna Häkkilä, Panu Korpipää, Sami Ronkainen, and Urpo Tuomela. 2005. Interaction and end-user programming with a context-aware mobile application. In *Proceedings of the INTERACT 2005, IFIP TC13 International Conference.* Lecture Notes in Computer Science, Vol. 3585. 927–937.

Bala Iyer and Thomas H. Davenport. 2008. Reverse engineering Google's innovation machine. *Harvard Busines Review* 86, 4 (2008), 58–69.

Anant Jhingran. 2006. Enterprise information mashups: Integrating information, simply. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. ACM, 3–4.

Natalia Juristo Juzgado and Ana María Moreno. 2001. *Basics of Software Engineering Experimentation*. Kluwer.

Andrew Jensen Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'04)*. IEEE Computer Society, 199–206.

Reto Krummenacher, Barry Norton, Elena Paslaru Bontas Simperl, and Carlos Pedrinaci. 2009. SOA4All: Enabling web-scale service economies. In *Proceedings of the 3rd IEEE International Conference on Semantic Computing (ICSC'09)*. IEEE Computer Society, 535–542.

Markus Latzina and Joerg Beringer. 2012. Transformative user experience: Beyond packaged design. *Interactions* 19, 2 (2012), 30–33.

Maurizio Lenzerini. 2002. Data integration: A theoretical perspective. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* ACM, 233–246.

Henry Lieberman, Fabio PaternÚ, and Volker Wulf. 2004. *End User Development*. Human-Computer Interaction Series, Vol. 9. Springer.

Andrew Lipsman and Carmela Aquino. 2013. *Mobile Future InFocus - 2013*. White Paper. ComScore. Retrieved from http://www.comscore.com/Insights/Press_Releases/2013/2/comScore_Releases_the_2013_Mobile_Future_in_Focus_Report.

Xuanzhe Liu, Gang Huang, and Hong Mei. 2007. Towards end user service composition. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC'07).* IEEE Computer Society, 676–678.

David Lizcano, Fernando Alonso, Javier Soriano, and Genoveva López. 2013. A web-centred approach to end-user software engineering. *ACM Trans. Softw. Eng. Methodol.* 22, 4 (2013), 36.

Maristella Matera, Matteo Picozzi, Michele Pini, and Marco Tonazzo. 2013. PEUDOM: A mashup platform for the end user development of common information spaces. In *Proceedings of the 13th International Conference on Web Engineering (ICWE'13).* Lecture Notes in Computer Science, Vol. 7977. Springer, 494–497.

E. Michael Maximilien, Hernán Wilkinson, Nirmit Desai, and Stefan Tai. 2007. A domain-specific language for web APIs and services mashups. In *Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC'07).* Lecture Notes in Computer Science, Vol. 4749. Springer, 13–26.

Katrina Maxwell. 2002. *Applied Statistics for Software Managers*. Prentice Hall.

Abdallah Namoun, Tobias Nestler, and Antonella De Angeli. 2010a. Conceptual and usability issues in the composable web of software services. In *ICWE Workshops*, Florian Daniel and Federico Michele Facca (Eds.). Lecture Notes in Computer Science, Vol. 6385. Springer, 396–407.

Abdallah Namoun, Tobias Nestler, and Antonella De Angeli. 2010b. Service composition for non-programmers: Prospects, problems, and design recommendations. In *Proceedings of the 8th IEEE European Conference on Web Services (ECOWS'10).* IEEE Computer Society, 123–130.

Felix Naumann. 2002. *Quality-Driven Query Answering for Integrated Information Systems*. Lecture Notes in Computer Science, Vol. 2261. Springer.

Open Mashup Alliance (OMA). 2013. OMA EMML Documentation. Retrieved from http://www.openmashup.org/omadocs/v1.0/.

Matteo Picozzi, Nervo Verdezoto, Matti Pouke, Jarkko Vatjus-Anttila, and Aaron J. Quigley. 2013. Traffic visualization—applying information visualization techniques to enhance traffic planning. In *Proceedings of the International Conference on Computer Graphics Theory and Applications and International Conference on Information Visualization Theory and Applications (GRAPP & IVAPP'13).* SciTePress, 554–557.

Agnes Ro, Lily Shu-Yi Xia, Hye-Young Paik, and Chea Hyon Chon. 2008. Bill organiser portal: A case study on end-user composition. In *WISE Workshops.* Lecture Notes in Computer Science, Vol. 5176. Springer, 152–161.

Wael Al Sarraj and Olga De Troyer. 2010. Web mashup makers for casual users: A user experiment. In *iiWAS*, Gabriele Kotsis, David Taniar, Eric Pardede, Imad Saleh, and Ismail Khalil (Eds.). ACM, 239–246.

Josef Spillner, Marius Feldmann, Iris Braun, Thomas Springer, and Alexander Schill. 2008. Ad-hoc usage of web services with Dynvoker. In *Proceedings of the 1st European Conference towards a Service-Based Internet (ServiceWave'08).* Lecture Notes in Computer Science, Vol. 5377. Springer, 208–219.

Tobias Van Dyk and Karen Renaud. 2004. Task analysis for e-commerce and the web. In *The Handbook of Task Analysis for Human-Computer Interaction*, Dan Diaper and Neville Stanton (Eds.). Lawrence Erlbaum, Mahwah, NJ, 68–81.

Eric von Hippel. 2005. *Democratizing Innovation*. MIT Press.

Jen-Her Wu, Yung-Cheng Chen, and Li-Min Lin. 2007. Empirical evaluation of the revised end user computing acceptance model. *Computers in Human Behavior* 23, 1 (2007), 162–174. DOI:http://dx.doi.org/10.1016/j.chb.2004.04.003

Jin Yu, Boualem Benatallah, Régis Saint-Paul, Fabio Casati, Florian Daniel, and Maristella Matera. 2007. A framework for rapid integration of presentation components. In *Proceedings of the 16th International Conference on World Wide Web (WWW'07)*. ACM, 923–932.

Justin Zobel and Philip Dart. 1996. Phonetic string matching: Lessons from information retrieval. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'96)*. ACM, New York, NY, 166–172. DOI:http://dx.doi.org/10.1145/243199.243258

Received November 2013; revised November 2014; accepted February 2015