Distributing the User Interface Logics along Actionable Components: the EFESTO Approach

Giuseppe Desolda[1], Carmelo Ardito[1], Maristella Matera[2], Maria Francesca Costabile[1]
[1]Dipartimento di Informatica, Università degli Studi di Bari Aldo Moro
Via Orabona, 4 – 70125 – Bari, Italy
{giuseppe.desolda, carmelo.ardito, maria.costabile}@uniba.it

[2]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano
P.zza L. da Vinci, 32 – 201233 – Milano
maristella.matera@polimi.it

Abstract — **Developing interactive systems is a very tough task. In particular, the development of user interfaces (UIs) is one of the most time consuming aspects in the software lifecycle. Software development is more and more moving toward composite applications. In this paper, we present a mashup model that enables the integration at the *presentation layer* of specific *UI components*. As application of this model, a mashup platform has been developed that allows non-technical end users to create component-based interactive workspaces via the aggregation and manipulation of data fetched from distributed online resources, also enabling the collaborative creation and use of distributed interactive workspaces. It is shown in this paper how the developed platform permits the rapid prototyping of interactive applications enabling the access to Web services and APIs.**

Keywords - Human-Centric Service Composition; Mashup Model.

I. INTRODUCTION

The development of user interfaces (UIs) is one of the most time consuming aspects in the creation of interactive systems. Proper reuse mechanisms for building UIs have become evident, especially as software development is more and more moving toward composite applications [1]. To respond to this need, in this paper we propose a mashup model that enables the integration at the *presentation layer* of "Actionable UI components", which are components equipped with both data visualization templates and a proper logic consisting of functions to manipulate the visualized data. The goal of the model is to reduce the effort required for the development of *interactive workspaces* [2], by maximizing the reuse of UI components.

In our approach, UI components not only constitute "pieces" of UIs that can be assembled together into a unified workspace; each single component can be set to provide views over the huge quantity of data exposed by Web services and APIs available online or by any data source, even personal or locally provided. With respect to the original definition of UI components [3, 4], we promote the notion of Actionable UI component, which also include UI Components that provide varying functions to allow end users to manipulate the contained data.

DOI reference number: 10.18293/DMS2016-034

Our approach can be positioned into the research context related to facilitating the access to data sources through visual user interfaces, a problem that has been attracting the attention of several researchers in the last years [5, 6]. An ever-increasing number of resources that provide content and functions in different formats through programmatic interfaces is available. The efforts of many research projects have thus focused on letting laypeople, i.e., users without expertise in programming, to access and exploit the available content [7, 8, 9]. With this respect, the reuse of easily programmable UI components is a step forwards the provision of environments facilitating the End-User Development (EUD) of service-based interactive workspaces [10]. In general, EUD refers to the involvement of end users in the software development process, in order to modify and even create software artifacts [10, 11, 12]. EUD activities range from simple parameters setting to integration of pre-packaged components, up to extending the system by developing new components. Such a reuse is typical of Web mashups [1], a new class of applications that can be created by integrating components at any of the application stack layers (presentation, business logics, data).

The very novelty introduced by mashups is the possibility to synchronize components equipped with a UI at the presentation layer, for example by means of event-driven composition techniques. Thanks to the possibility of reusing and synchronizing ready-to-use UI components, the mashup has resulted into an effective paradigm to let end users, even non-experts in technology, compose their interactive Web applications. In the last two decades, several mashup tools characterized by different composition paradigms have been proposed. One of the pioneers (even if it is not available anymore) was Yahoo!Pipes [8] that, in the attempt to better support end users, provided a visual editor to access services and operators visualized as visual modules that could be combined into a canvas pane by means of 'pipes' through drag and drop actions. A different paradigm was implemented in NaturalMash, which allows users to express in natural language what service(s) to use and how to synchronize them [7]. However, problems related to the use of a natural language still remain. A different mashup technique avoids the use of APIs and enables data integration allowing users to act directly on the Web page UI elements, which are considered interactive artefacts that can be combined through a set of mashup

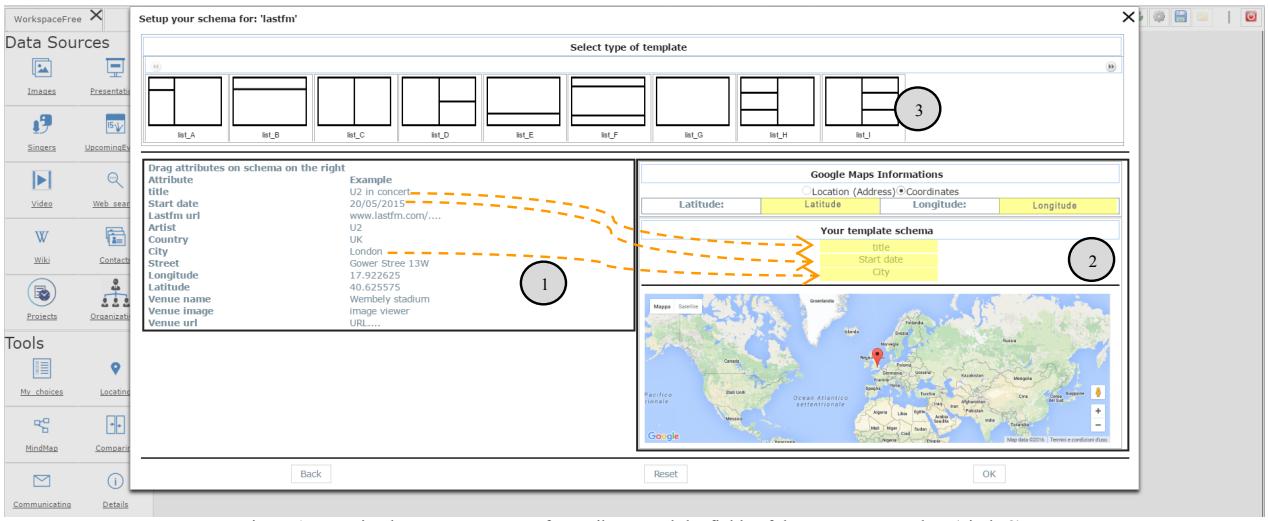

Figure 1. Mapping between some Last.fm attributes and the fields of the *map* user template (circle 2).

operations [13]. The inconvenience of this approach is that it is difficult to generalize due to the heterogeneity of web pages. Other mashup tools and mashup techniques are described in [1]. Moreover, a recent article surveys web service composition techniques and tools [14].

In the last years we have been working extensively on mashup platforms that, by exploiting end-user development principles, address the creation of component-based interactive workspaces by non-technical end users via the aggregation and manipulation of data fetched from distributed online resources [2, 15], also enabling the collaborative creation and use of distributed interactive workspaces [16]. The platform prototype keeps improving on various aspects, based on field studies performed with real users that are revealing new requirements and features useful to foster the adoption of mashup platforms in people daily activities. Based on these experiences, in which we observed people creating easily their interactive applications, in this paper we aim to stress the importance of such platforms as tools for the rapid prototyping of interactive applications enabling the access to Web services and APIs. In particular, the main contribution of this work is a model for UI component mashup that other designers and developers can adopt to develop mashup platforms as tools to easily create interactive workspaces whose logic is distributed across different synchronized components.

The paper is organized as follows: Section II illustrates the main functionality offered by the platform for the creation of interactive workspaces. Section III highlights how the supported modus operandi is made possible thanks to some abstractions, and in particular to the notion of *actionable UI components*, around which the whole platform design has been conceived. In particular, we stress how the adoption of such conceptual elements leads to the notion of *distributed User Interface*, as an interactive artefact that can be assembled according to lightweight technologies and that leverages on the logics of self-contained actionable UI components. Section IV is about Domain-Specific Languages (DSLs) we introduced to describe the main elements of a mashup platform that can guide the dynamic instantiation and execution of the distributed UIs. Section V complements Section III by providing some technical details on how the model elements are implemented in the EFESTO platform architecture. Section VI concludes the paper and outlines future work.

II. The EFESTO Platform

This section describes the most important features of our mashup platform, called EFESTO by showing how it is used to create a mashup. The platform name was inspired by Efesto, a god of the Greek mythology, who realized magnificent magic arms for other Greek gods and heroes. Analogously, the EFESTO platform aims to provide end users with powerful tools to accomplish their tasks. In order to get the reader's attention, such features are written in **bold** in this section and formalized in the model reported in Section III.

A. Mashup of Data Sources

In order to describe how EFESTO works, a scenario is reported in which Alan uses the platform to create a mashup that satisfies his information needs. Alan is a non-technical user, i.e., he does not know programming language and he is not familiar with technical terms of computer science.

Alan is going to organize his summer holidays, but he did not yet decide whether to go to Paris or Rome. Regardless the destination, Alan would like to attend at least a concert during his holidays. Thus, he uses EFESTO to create a new application (mashup) that retrieves and integrates information about music events, possibly coming from different sources, and presents the results through a visual representation he selects. Specifically, Alan starts looking for pertinent services among those registered in the platform. A wizard procedure guides him to make a selection from a popup window where services are presented by category (e.g., videos, photos, music, social). Alan clicks on the music category and, among the music services shown, he selects Last.fm, a service that provides information on music events of a specific singer. EFESTO provides different visual templates, called User Interface Templates (**UI Template**), that the user can select in

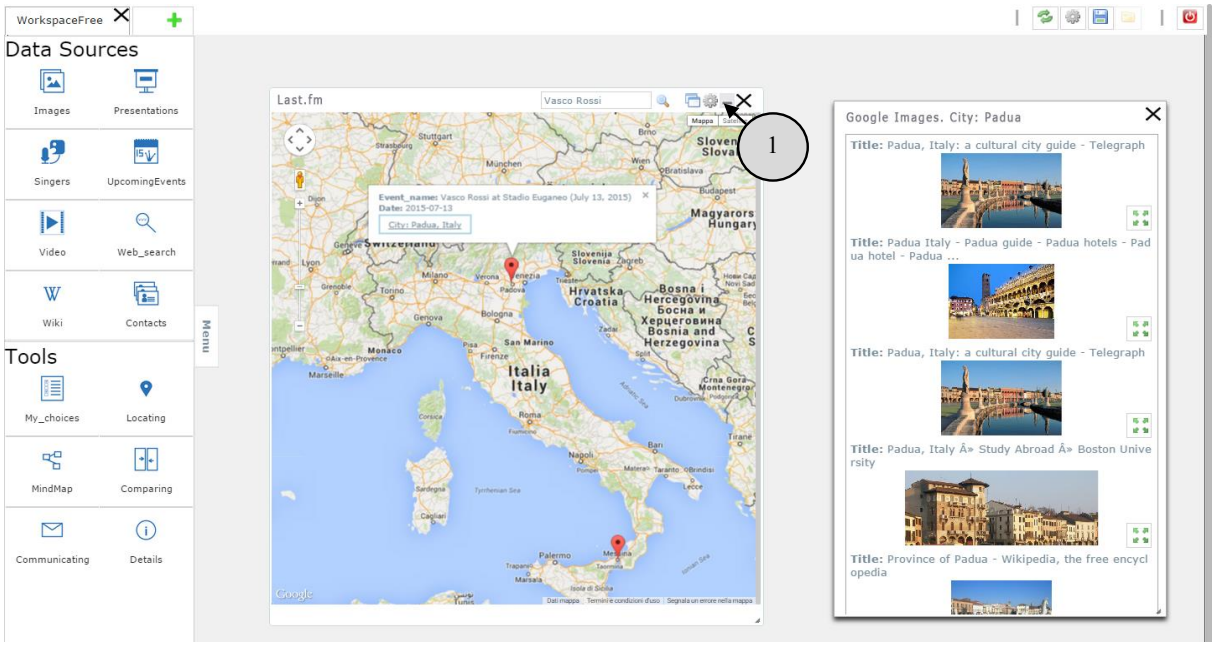

Figure 2. UI Component originated from Last.fm data source visualized as a map and joined with Google Images to show pictures of each city event.

order to display the results of the application he is creating. Alan actually selects a *map* as UI Template, since he wants to visualize the retrieved music events geo-localized in a map.

Among the different data attributes of the Last.fm dataset, Alan has to select those he is interested in, i.e., those that will be considered by the application he is creating. EFESTO enables Alan to make this selection by direct manipulation of elements shown in the user interface of his workspace. In fact, all Last.fm data attributes are visualized in a panel on the left (see Figure 1, circle 1). To make the attributes more understandable, the system also shows some example values. Alan wants his application to consider latitude and longitude of the location where a music event will be performed, so that this location will be visualized in the resulting map. Thus, Alan drags & drops the latitude and longitude Last.fm attributes into the respective fields (called **Visual Renderers** [4]) of the map UI template (Figure 1, circle 2). Alan wants also to visualize, when required, additional details about a musical event. For this, among the available UI templates for text layout (Figure 1, circle 3), he chooses a *table* with three rows and one column, since he wants to visualize three more attributes, namely *title*, *start date* and *city*. To make this possible, he selects each of these three attributes from the left panel (Figure 1, circle 1) and drops it in the visual renderers of the UI template (highlighted in yellow in Figure 1, circle 2).

After performing this mapping phase, Alan saves the mashup in the platform. From now on, this mashup is a **UI Component** in the user workspace, which is immediately executed in the Web browser. This UI Component is called "Last.fm" (by default, it takes the name of the source service but the user can rename it) and is represented as a map, as shown in the central panel in Figure 2. By typing "Vasco Rossi" in the search box (thus making a **query**), the **results set** of forthcoming events of this singer are visualized as pins on the map. The map is shown with a proper zoom level so that all the retrieved events are visualized. This zoom level can obviously be varied by the user. By clicking on a pin representing a music event, details of that event (i.e., the attributes *title*, *start date* and *city*) are shown.

Alan can later update the created mashup by integrating data coming from other **data sources** through **union** and **join** data mashup **operations** [15]. Since a non-technical user is not familiar with union and join operation, EFESTO let the user perform such operations, again, through wizard procedures and drag&drop actions. For example, Alan wants to retrieve more music events than those provided by Last.fm. He then integrates Last.fm with Eventful (another service retrieving music events). Technically, this is a union operation. Alan acts directly on the Last.fm UI component previously created by clicking on the gearwheel icon in the toolbar (pointed by circle 1 in Figure 2) and choosing the "Add results from new source" menu item. A wizard procedure now guides Alan in choosing the new service, Eventful in this example, and in performing a new mapping between the Eventful attributes and the UI template already used in the previous mashup. The newly created mashup (UI Component) is shown in the same fashion reported in Figure 2 but now, when queried with an artist name, this UI Component visualizes results gathered both from the Last.fm and Eventful services.

Another data integration operation available in EFESTO is the join of different sources; it is useful to satisfy user's desire of further integrating the mashup with new data available in other services. For example, Alan would like to show images of the location where music events are held. Last.fm does not provide such images but Alan can retrieve them from Google Images. Technically, this operation is a join between the

Last.fm city attribute and Google Images. EFESTO supports Alan in a very simple way. Alan clicks on the component gearwheel icon and choses the "Extend results with new data" menu item. A new wizard procedure guides him while choosing (a) the service attribute to be extended (City in this example), (b) the new data source (Google Images) and (c) how to visualize the Google Images results. From now on, when clicking on the city name in the map info window, another window visualizes the Google Images pictures related to the selected city, as shown in the right panel of Figure 2.

Another operation available in EFESTO is the *change of visualization* for a given UI component. Alan, in fact, during the interaction with Last.fm, decides to switch from the *map* UI template to the *list* UI template (see the result in Figure 3, circle 1). To perform this action, he clicks on the gearwheel icon in the Last.fm toolbar and chooses the "Change visualization" menu item. A wizard procedure guides Alan to (a) choose a UI template (*list* in this case), and (b) drag&drop the Last.fm attributes onto the UI template, as already described with reference to Figure 1.

B. A polymorphic data source

Despite the wide availability of data sources and composition operations, sometimes users can still encounter difficulties while trying to accommodate different needs and desires. Let us suppose, for example, that during the interaction with EFESTO Alan wants to get details about the artists of the music events, such as genre, starting year of activity and artist photo. Among those services registered in the platform, Alan does not find any that satisfy this new information need. Thus, he should go to the Web for a usual (manual) search for the specific information. However, it might happen that, even on the Web, there are no APIs providing such information.

In order to overcome this drawback, EFESTO provides a new **polymorphic data source** that exploits the wide availability of information structured in the Linked Open Data (LOD) cloud. It is called polymorphic because, when it is composed with another source S, it is capable of modifying its set of attributes depending on the source S, in order to better fulfil the users' needs. In contrast, the standard data sources (YouTube, Wikipedia, etc.) provide the same set of attributes independently of the composing source S. Lack of space prevents us to provide more details about the creation of the polymorphic data source. The interested reader may refer to [17]. DBpedia has been chosen as initial LOD cloud thanks to the vast amount of information it provides.

Thus, Alan can join the Last.fm artist attribute with the DBpedia-based polymorphic data source. The platform now shows a list of attributes related to the musical artist class (available in the DBpedia ontology), and Alan enriches the current UI Component with the attributes *genre*, *starting year of activity* and *artist photo*. Henceforward, Alan can find a list of upcoming events and also visualize artist's information when clicking on the artist's name. What has been described also shows why the DBpedia-based data source is called "polymorphic". In fact, differently from pre-registered data sources (e.g., Google Images) that provide a pre-defined, invariable set of attributes, the system provides users different attributes of the data of the DBpedia-based data source; such attributes are automatically selected depending on the attribute in the origin data source it is bound to. For example, if the Alan's join starting point is the attribute *city*, attributes like *borough*, *census*, *year*, *demographics* would be provided by the DBpedia-based data source.

C. Task-Enabling Containers

Our field studies [2, 16] revealed that mashups generally lack data manipulation functions that end users would like to exploit in order to "act" on the extracted contents, e.g. functions that allow to perform tasks such as collecting&saving favourites, comparing items, plotting data items on a map, inspecting full content details, organizing items in a mind map in order to highlight relationships. In this section, we remark another very innovative feature of EFESTO: it offers tools that enable specific tasks, allowing users to manipulate the information in a novel fashion, i.e., without being constrained to pre-defined operation flows typical of pre-packaged applications.

In order to perform more specific and complex sense-making tasks, a set of Tools are available in the left-panel of the workspace (see Figure 3, circle 4). These Tools are added to the workspace by clicking the corresponding icon. Let us describe an example of their usage with reference to our scenario. Alan is looking for hotels in Rome located nearby the places where upcoming musical events will be held. According to his strategy, he is more interested in finding a good hotel and then look for possible musical events to attend. First, he adds the *Hotel* data source into his workspace (see Figure 3, circle 5) and then performs a search by typing "Rome" in the *Hotel* search bar. After including the *Comparing* tool in the workspace, Alan drags&drops inside it the first five hotels from the *Hotel* UI component. The *Comparing* tool supports Alan in the identification of the most convenient hotels, which are now represented as cards providing further details, such as average price, services and category (see Figure 3, circle 2). Afterwards, he drags&drops three hotels from the *Comparing* tool inside the *Locating* tool (Figure 3, circle 3) in order to visualize them as pins on the map. Finally, Alan performs a search on the *Last.fm* data source by using "Rome" as keyword and then moves all the results, i.e. the upcoming musical events, inside the *Locating* container. The map now shows pins indicating both the hotels and the upcoming musical events in Rome. Alan can now easily identify which musical events are close to the hotels he has previously chosen. However, it could happen that Alan adopts a different strategy. He wants to first identify upcoming musical events and then the hotels nearby. He starts by retrieving musical events with *Last.fm* (see Figure 3, circle 1) and then moves some events inside the *Comparing* tool in order to choose the best ones based on musical genre and artists. Afterwards, he drags&drops some of the compared events inside the *Locating* tool and finally adds into this tool the hotels returned by the *Hotel* data source.

As shown in the previous example, the tools provided in EFESTO allow users to interact with information within dedicated *containers*, which enable specific tasks. Thus, we call them Task-Enabling Containers (TECs). To create such flexible environments, a model has been presented in [18] that

permits easy transition of information between different contexts; this model implements some of the Transformative User eXperience (TUX) principles described in [19, 20].

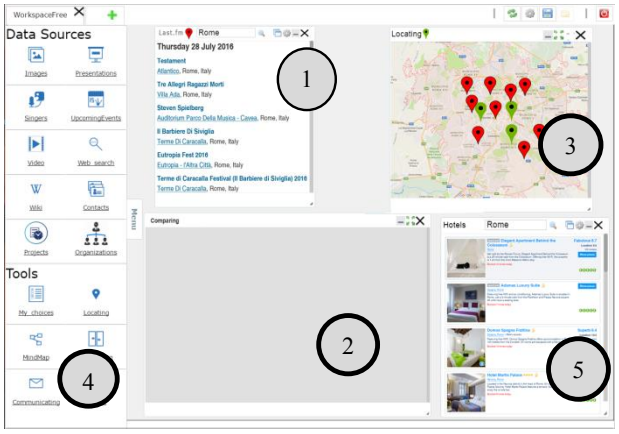

Figure 3. Use of some tools available in EFESTO to manipulate mashup data.

III. Model for UI Component Mashup

The main contribution of this paper is a model highlighting the most important components that make a mashup tool an environment where UIs can be built by reusing and synchronizing the logic of different pieces of UI. The goal is to provide designers and software engineers with a model that guides them during the development of mashup platforms for non-programmers. The proposed model refines and extends the one presented in [4], where the authors defined the modelling abstractions on which their composition paradigm is based. Our model has been iteratively refined by adding further components starting from requirements we gathered

during our research, e.g., a different way to integrate service data by means of more powerful data mashup operations join and union, the integration of some Transformative User eXperience principles [19, 20], and the polymorphic data sources based on Linked Open Data. The new model is depicted in Figure 5. In the following, we report the definitions of the most salient concepts that contribute to the notion of distributed UIs.

Definition 1. ***UI Component.*** It is the core of the model since it represents the main modularization object the user can exploit to retrieve and compose data extracted from services. It supplies a view according to specific UI Templates (see Definition 2) over one or more services whose data can be composed by means of data mashup operations and also allows the interaction with services data and functions thanks to its own UI (see Figure 4). In addition, two or more UI components can also be synchronized according to an event-driven paradigm: each of them can implement a set E of events that the user can trigger during the interaction with its user interface, and a set A of actions activated when events are performed on others UI components.

Definition 2. ***UI Template.*** It plays two fundamental roles inside the UI component: first, it guides the users in materializing abstract data sources by means of a mapping between the data source output attributes and the UI template visual renderers; second, at runtime, it displays the data source according to the user mapping. A *UI Template* can be represented as the triple

$$uit =< type, VR>$$

where *type* is the template (e.g., list, map, chart) selected by the user while *VR* is a set of *visual renderers*, i.e., UI elements that act as receptors of data attributes.

Definition 3. ***Actionable UI Component (auic).*** In addition to visualizing Web service data, *auic* also supply task-related functions for manipulation and transformation of data items retrieved from a source along user-defined task flows [18].

An *auic* can be defined as a pair:

$$auic = <TF, uit>$$

where TF is the set of *functions* for manipulation and transformation of data, while uit is a UI templates used to visualize data according to user's task.

Definition 4. ***Event-driven Coupling***. It is a synchronization mechanism among two UI components that the users define according to an event-driven, publish-subscribe integration logic [3]. In particular, the users define that, when an event is triggered on a UI component, an action will be performed by another UI component. This enables reusing the logic of single UI components, still being able to introduce some new behaviour for the composite UIs.

Figure 4: Example of UI Component that shows musical events on Google Maps.

More in general, given two UI Components uic_i and uic_j, a coupling is a pair:

$$c=< uic_i (<output>), uic_j (<input>)>$$

Definition 5. ***Presentation Template.*** It is an abstract representation of the workspace defining the visual organization of the UICs included in the interactive workspace under construction. For example, the UI components can be freely located or can be constrained to a grid schema, where in each cell only one UI Component can be placed.

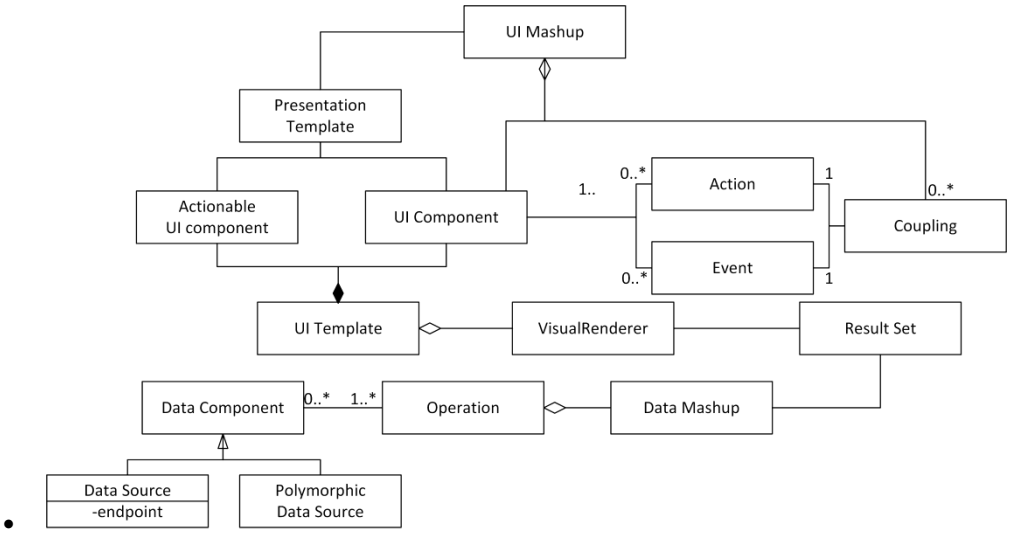

Figure 5: The mashup model.

Definition 6. ***UI Mashup***. A UI Mashup is the final interactive application built by the end users by means of the integration of different UI components within a workspace. It can be formalized as the tuple:

$$UI\_Mashup =< UIC, C, PT, AUIC>,$$

where UIC is the set of UI Components integrated into the workspace, C is the set of couplings the users established among UIC, PT is the workspace template chosen to arrange the UIC and AUIC is the set of Actionable UI Components to manipulate data extracted from UIC.

The following definitions are reported to clarify how actionable UI components are instantiated by means of data extracted from data sources.

Definition 7. ***Data Component.*** It is an abstract representation of the resource that can be used to retrieve data. In particular, *dc* is a triplet:

$$dc=<t, I, Out>$$

where *t* indicates the type of resource, for example *REST Data Source* or *Polymorphic Data Source* in our model, *I* indicates the set of input parameters to query the resources, *Out* indicates the set of output attributes. Data can be retrieved from data sources and aggregated through the following operations:

Definition 8.a ***Selection***. Given a data component *dc*, a selection is a unary operator defined as:

$$\sigma_C\ (dc) = \{r \in dc \mid result\ r\ satisfies\ condition\ C\}.$$

where r is a result obtained by querying the data component dc and C is a condition used to query dc.

Definition 8.b ***Join.*** Given a couple of data components $dc_i =< ep_i, q_i, A>$ and $dc_j =< ep_j, q_j, B>$, a *Join* is a binary operator defined as:

$$dc_i \mid><\mid_{ai} dc_j = \{(a_1, \ldots, a_n, \sigma_C\ (dc_j)) \mid C: q_j = a_i\}$$

Definition 8.c ***Union.*** Given a couple of data components $dc_i =< ep_i, q_i, A>$ and $dc_j =< ep_j, q_j, B>$, a *Union* is a binary operator defined as:

$$dc_i\ U\ dc_j = \{ x \mid x \in dc_i\ or\ x \in dc_j \}$$

The result of the applying one or more operations is a data mashup, i.e., the composite result set whose rendering and manipulation is possible by means of UI components and Task-Enabling Containers.

Definition 9. ***Data Mashup***. It is the results of the integration of data extracted by different data components. It is a pair:

$$dm=<DC,O>$$

where DC represents the set of data components involved in the composition; O is the set of operations (e.g., join and union) performed between data components in DC.

Data mashup represents an important advance w.r.t. the original model presented in [4] where data mashup was conceived just as a *visual aggregation* of different data sources by means of union and merge sub-templates. In that case, the result of the data mashup could not be reused with other UI templates. In our model, the data mashup is a new integrated result set published as a new data source. This new data source can be used in the platform as a new source that can be visualized by using UI templates.

IV. Platform Descriptors

In order to make the previous abstractions concrete in the implemented platforms, we defined some Domain-Specific Languages (DSLs) inspired to EMML [21]. New languages were adopted instead of EMML because the composition logic implemented in the EFESTO refers only to a small sub-set of the composition operators available in EMML. Each of these

new languages allow us to define internal specifications of the main elements (e.g., UI components, service, UI template) that can guide the dynamic instantiation and execution of the distributed UIs.

In Figure 6**Errore. L'origine riferimento non è stata trovata.** it is reported an example of our XML language specifying a UI component that renders a data mashup consisting in a union between two services (YouTube and Vimeo) and a join of the unified services with a third service (Wikipedia). In the XML file, the tag *unions* has two children, *services* and *shared*. The *services* tag summarizes the unified services. Each service is reported in a *service* tag. In particular, the *service* tag has the attribute *name* that indicates the name of the data source. This value is used by the mashup tool to retrieve the source details to perform the query. The *shared* tag describes the alignment of the attributes of the unified data sources. For example, it has two children called *shared_attribute*, each of them with two children *attribute* that represent the service attributes that are mapped in a *UI template.*

```
<composition join="true" union="true">
 <unions>
  <services>
   <service name="lastfm">
    <attribute name="title" path="title">title</attribute>
    <attribute name="Start date" path="startDate">startDate</attribute>
    <attribute name="City" path="venue.location.city">venue.location.city</attribute>
   <service>
   <service name="songkick">
    <attribute name="EventTitle" path="title">event_title</attribute>
    <attribute name="EventDate" path="startDate">event_date</attribute>
    <attribute name="EventCity" path="venue.location.city">event_city</attribute>
   <service>
  </services>
  <shared>
   <shared_attribute name="title">
    <attribute from_service="lastfm">title</attribute>
    <attribute from_service="songkick">event_title</attribute>
   </shared_attribute>
   <shared_attribute name="Start date">
    <attribute from_service="lastfm">startDate</attribute>
    <attribute from_service="songkick">event_date</attribute>
   </shared_attribute>
   <shared_attribute name="City">
    <attribute from_service="lastfm">venue.location.city</attribute>
    <attribute from_service="songkick">event_city</attribute>
   </shared_attribute>
  </shared>
 </unions>
 <joins>
  <join>
   <service name="youtube" />
   <input>title</input>
   <extendedAttributes>
    <attribute name="Title" path="title">title</attribute>
    <attribute name="Owner" path="path">path</attribute>
   </extendedAttributes>
  </join>
 </joins>
</composition>
```

Figure 6: An example of UI component descriptor codified with our XML language.

Each service reported in the *service* tag is detailed in a separate service descriptor XML file. In Figure 7**Errore. L'origine riferimento non è stata trovata.**, the YouTube service descriptor is reported: inside the root tag called *service*, there are the tags *source*, *inputs*, *params*, *attributes* and *flags.* The first three nodes represent all the information useful to query a data source. The fourth node, *attributes*, describes the instance attributes. The last node, *flag*, is introduced to solve the heterogeneity problem of the data sources. In fact, the remote web services typically send the results by using JSON file but the list of results if formatted in different ways (e.g. inside a JSON array).

```
<service name="youtube" type="JSON" Auth="none">
  <source name="youtube"
          url="https://www.googleapis.com/youtube/v3/search?"
          type="GET">https://www.googleapis.com/youtube/v3/search?
  </source>
  <inputs>
    <input name="q"></input>
  </inputs>
  <params>
    <param name="key">AIzaSyCy5Be6QWdqQ</param>
  </params>
  <separator>&amp;</separator>

  <attributes>
    <attribute name="Title" path="snippet.title">Fear of the Dark</attribute>
    <attribute name="Video" path="id.videoId">Video player</attribute>
    <attribute name="PublicationDate" path="snippet.publishedAt">2009-05-12</attribute>
  </attributes>
  <flags>
    <flag name="is_array">false</flag>
    <flag name="array_name">$.items[*]</flag>
  </flags>
</service>
```

Figure 7: An example of service descriptor codified with our XML language.

Another XML descriptor introduced in our model regards the UI Template. In Figure 8, the *list* UI Template has been reported. It is characterized by a set of sub-UI templates (different types of lists). In particular, the root node, *template*, has an attribute name that indicates the template name. The root has a set of children that describe different alternatives to visualize the UI template.

```
<template type="list">
  <sub_template name="list_A">
    <column width="30%">
      <component name="0" height="20%"/>
      <component name="1" height="80%"/>
    </column>
    <column width="70%">
      <component name="2" height="100%"/>
    </column>
  </sub_template>
  <sub_template name="list_B">
    <column width="100%">
      <component name="0" height="20%"/>
      <component name="1" height="80%"/>
    </column>
  </sub_template>
</template>
```

Figure 8: An example of list UI template descriptor codified with our XML language.

The UI template descriptor is linked with the VI schema through the XML mapping descriptor. An example of mapping is reported in Figure 9. In this descriptor, the root node, *mappings*, has two attributes: *templatetype* and *templatename*. The first one recalls the name of a UI Template (e.g. *list*), the second one the name of its sub-template (*list_A*).

```
<mappings template_type="list" sub_template="List_A">
  <mapping vr="0" attribute="Title" path=".Title">
  <mapping vr="0" attribute="Author" path=".Author">
  <mapping vr="0" attribute="Video" path=".Video">
</mappings>
```

Figure 9: An example of mapping descriptor

V. From the Model to the Platform Architecture

The model presented in Section III guides designers and software engineers in developing mashup platforms targeting to non-programmers. That model highlights the main concepts of a mashup platform without emphasizing technical aspects. In this section, we report a high-level overview of the architecture of the EFESTO mashup platform, in order to illustrate how it implements the mashup model.

The architecture is characterized by tree-layers (Figure 10). On top there is the *UI layer* whose main feature is a visual language that allows end users to perform mashups without requiring technical skills. Such language is based on *UI Components* that use *UI Templates* and *Actionable UI Components* to allow users to visualize and manipulate data extracted from remote sources. *UI layer* runs in the user's Web browser and communicates with the *Logic* and *Data layer* that run on a remote Web server.

The *Logic Layer* implements components that translate the actions performed by end users at the *Interaction Layer* into the mashup executing logic. In particular, the *Mashup Engine* is invoked each time an event, requiring the retrieval of new data or the invocation of service operations, is generated. The *Event Manager*, instead, manages the UI Components coupling. In particular, when users define a synchronization between two UI Components A and B, it instantiates a listener that waits for an event on A that, when triggered, causes the execution of an action on B, according to the coupling defined by the user.

The *Data Layer* stores the XML-based descriptors described in Section IV into proper repositories. In addition, at this layer there are the remote data sources that reside on different Web servers.

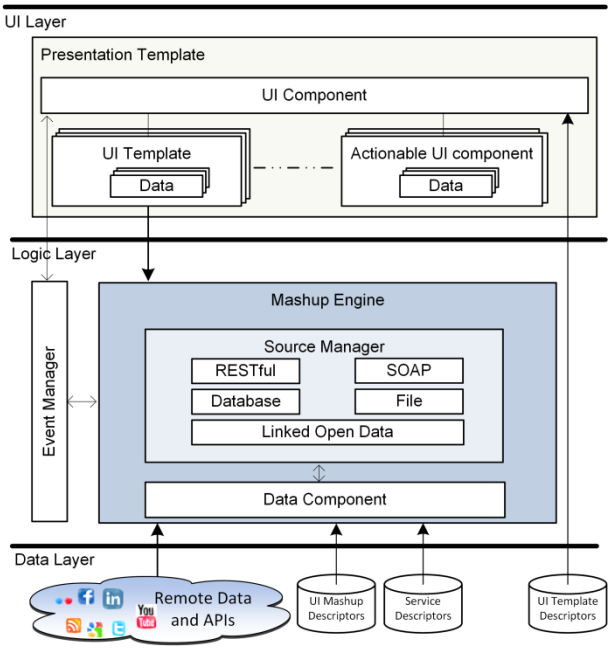

Figure 10. An high-level overview of the EFESTO three-layer architecture

VI. Conclusion

This paper discusses some abstractions that can promote mashup platforms as tools to easily create (i.e., even by end users) interactive workspaces whose logic is distributed across different components that are however synchronized with each other. One of the main contributions of mashup development is the introduction of novel practices enabling integration at the presentation layer in a component-based fashion - an aspect that was never investigated before. Some papers, indeed, discuss and motivate the so-called UI-based integration [4, 22, 23] as a new component-based integration paradigm that privileges the creation of full-fledged artifacts, also equipped with UIs, in addition to the traditional service and data integration practices that instead mainly act at the logic and data layers of the application stack. Along this direction, this paper highlights how interactive artifacts can be composed by reusing the presentation logics (i.e., the UIs) and the execution logics of self-contained modules, the so-called UI Components and Actionable UI Components, providing for the visualization of data extracted by data sources and for data manipulation operations through task-related functions.

The paper capitalizes on the experience gained in the last years by the authors in the development of prototypes of mashup platforms, but it also aims to propose a systematic view on concepts and techniques underlying mashup design and on the way such concepts materialize into composition paradigms and architectures of corresponding development tools, independent of specific approaches and technologies and, thus, of more general validity. Our current work is devoted to enrich the EFESTO platform by means of tools for actionable components and to customize the platform to other application domains, so that further validation studies will be performed.

I. References

[1] Daniel, F., Matera, M.: Mashups: Concepts, Models and Architectures. Springer (2014)

[2] Ardito, C., Costabile, M.F., Desolda, G., Lanzilotti, R., Matera, M., Piccinno, A., Picozzi, M. (2014). User-Driven Visual Composition of Service-Based Interactive Spaces. In *Journal of Visual Languages & Computing* 25(4), 278-296

[3] Yu, J., Benatallah, B., Saint-Paul, R., Casati, F., Daniel, F., Matera, M. (2007). A framework for rapid integration of presentation components. In: *Proc. of WWW '07*. Banff, Alberta (Canada). May 8-12. pp. 923-932

[4] Cappiello, C., Matera, M., Picozzi, M. (2015). A UI-Centric Approach for the End-User Development of Multidevice Mashups. In *ACM Transaction Web* 9(3), 1-40

[5] Spillner, J., Feldmann, M., Braun, I., Springer, T., Schill, A. (2008). Ad-Hoc Usage of Web Services with Dynvoker. In Mähönen, P., Pohl, K., Priol, T. *Towards a Service-Based Internet - ServiceWave 2008*. (Vol. 5377, pp. 208-219)

[6] Krummenacher, R., Norton, B., Simperl, E., Pedrinaci, C. (2009). SOA4All: Enabling Web-scale Service Economies. In: *Proc. of ICSC '09*. Berkeley, CA (USA). 14-16 September. pp. 535-542

[7] Aghaee, S., Pautasso, C. (2014). End-User Development of Mashups with NaturalMash. In *Journal of Visual Languages & Computing* 25(4), 414-432

[8] Pruett, M.: Yahoo! pipes. O'Reilly (2007)

[9] Hirmer, P., Mitschang, B. (2016). FlexMash–Flexible Data Mashups Based on Pattern-Based Model Transformation. In Daniel, F., Pautasso, C. *Rapid Mashup Development Tools - Rapid Mashup Challenge in ICWE 2015*. (Vol. 591, pp. 12-30)

[10] Fischer, G. (2009). End-User Development and Meta-design: Foundations for Cultures of Participation. In Pipek, V., Rosson, M.B., de

Ruyter, B., Wulf, V. *End-User Development - Is-EUD 2009*. (Vol. pp. 3-14)

[11] Costabile, M.F., Fogli, D., Mussio, P., Piccinno, A. (2006). End-User Development: The Software Shaping Workshop Approach. In Lieberman, H., Paternò, F., Wulf, V. *End User Development - Is-EUD 2006*. (Vol. 9, pp. 183-205)

[12] Costabile, M.F., Dittrich, Y., Fischer, G., Piccinno, A.: End-User Development. Vol. LNCS 6654, Springer-Verlag, Berlin / Heidelberg, 2011 (2011)

[13] Daniel, F. (2015). Live, Personal Data Integration Through UI-Oriented Computing. In: *Proc. of ICWE '15*. Rotterdam, the Netherlands. 23 – 26 June 2015. pp. 479-497

[14] Lemos, A.L., Daniel, F., Benatallah, B. (2015). Web Service Composition: A Survey of Techniques and Tools. In *ACM Computing Survey* 48(3)**,** 1-41

[15] Desolda, G., Ardito, C., Matera, M. (2015). EFESTO: A platform for the End-User Development of Interactive Workspaces for Data Exploration. In Daniel, F., Pautasso, C. *Rapid Mashup Development Tools - Rapid Mashup Challenge in ICWE 2015*. (Vol. 591, pp. 63 - 81)

[16] Ardito, C., Bottoni, P., Costabile, M.F., Desolda, G., Matera, M., Picozzi, M. (2014). Creation and Use of Service-based Distributed Interactive Workspaces. In *Journal of Visual Languages & Computing* 25(6)**,** 717-726

[17] Desolda, G. (2015). Enhancing Workspace Composition by Exploiting Linked Open Data as a Polymorphic Data Source. In Damiani, E., Howlett, J.R., Jain, C.L., Gallo, L., De Pietro, G. *Intelligent Interactive Multimedia Systems and Services (KES-IIMSS '15)*. (Vol. pp. 97-108)

[18] Ardito, C., Costabile, M.F., Desolda, G., Latzina, M., Matera, M. (2015). Making Mashups Actionable Through Elastic Design Principles. In Díaz, P., Pipek, V., Ardito, C., Jensen, C., Aedo, I., Boden, A. *End-User Development - Is-EUD 2015*. (Vol. LCNS 9083, pp. 236-241)

[19] Latzina, M., Beringer, J. (2012). Transformative user experience: beyond packaged design. In *interactions* 19(2)**,** 30-33

[20] Beringer, J., Latzina, M. (2015). Elastic workplace design. In *Designing Socially Embedded Technologies in the Real-World*. (Vol. pp. 19-33)

[21] Viswanathan, A. (2010). Mashups and the enterprise mashup markup language (EMML). In *Dr. Dobbs Journal*

[22] Daniel, F., Matera, M., Yu, J., Benatallah, B., Saint-Paul, R., Casati, F. (2007). Understanding ui integration: A survey of problems, technologies, and opportunities. In *Internet Computing, IEEE* 11(3)**,** 59-66

[23] Yu, J., Benatallah, B., Casati, F., Daniel, F. (2008). Understanding Mashup Development. In *IEEE Internet Computing* 12(5)**,** 44-52