End-User Composition of Interactive Applications through Actionable UI Components

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

Maristella Matera
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 to access and manipulate data is a very tough task. In particular, the development of user interfaces (UIs) is one of the most time-consuming activities in the software lifecycle. This is even more demanding when data have to be retrieved by accessing flexibly different online resources. Indeed, software development is moving more and more toward composite applications that aggregate on the fly specific Web services and APIs. In this article, we present a mashup model that enables the integration, at the *presentation layer*, of specific *UI components* providing visualizations and manipulation functions on top of data retrieved by online data sources. This model has been exploited to develop a mashup platform that allows non-technical end users to create component-based interactive workspaces via the aggregation and manipulation of data fetched from distributed online resources. Due to the abundance of online data sources, facilitating the creation of such interactive workspaces is a very relevant need that emerges in different contexts. This article shows how the developed mashup 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 activities in the creation of interactive systems. The need for proper reuse mechanisms for building UIs has become evident in the last years, especially as software development is moving more and more toward component-based applications [1]. A considerable number of resources are also available online. Thus, easy and effective mechanisms to create UIs on top of the offered data are required. In this article, we propose a mashup model that enables the integration at the *presentation layer* of "Actionable UI components". These 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 into a unified workspace. Each single component can also provide views over the huge quantity of data exposed by Web services and APIs available online or from 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 components*, which introduce varying functions to allow end users to manipulate the contained data.

Our approach is located in 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 recent years [5, 6]. An ever-increasing number of resources is available that provide content and functions in different formats through programmatic interfaces. The efforts of many research projects have thus focused on letting laypeople, i.e., users without expertise in programming, access and exploit the available content [7, 8]. In this respect, the reuse of easily programmable UI components is a step towards the provision of environments facilitating the End-User Development (EUD) of service-based interactive workspaces [2]. In general, EUD refers to the involvement of end users in the software life cycle, in order to modify and even create software artifacts [9, 10]. EUD activities range from simple parameter setting to the integration of pre-packaged components, up to extending the system by developing new components. Reusing is typical of Web mashups [1], a class of applications that emerged in the last decade, which can be created by integrating components at any of the application stack layers (presentation, business logics and data). The term mashup was originally coined in music, where mashup indicates a song created by blending two or more songs, usually by overlaying the vocal track of one song seamlessly over the instrumental track of another.

The real novelty introduced by Web mashups is the possibility to synchronize components at the presentation layer by considering elements of their UI, 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 in an effective paradigm to let end users, even non-experts in technology, compose their interactive Web applications.

Over the last years, we have been working extensively on a mashup platform called EFESTO that, by exploiting end-user

development principles, addresses 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, 11]. This platform also enables the collaborative creation and use of distributed interactive workspaces [12]. The platform prototype keeps improving from various aspects, based on field studies performed with real users who reveal new requirements and features that are useful to foster the adoption of mashup platforms in people's daily activities. Based on these experiences, in which we observed people creating their interactive applications easily, in this article we aim to stress the importance of this type of platforms as tools for the rapid creation of interactive applications enabling the access to Web services and APIs. In particular, the main contribution of this article is a model for a UI component mashup that other designers and developers can adopt to develop mashup platforms as tools to easily compose 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 a 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 discusses the 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. On the basis of the related literature, Section VI presents the classification dimensions for mashup tools and discusses how EFESTO is characterized w.r.t. such dimensions. Section VII concludes the article 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 made 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 capture 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 Tyrion uses the platform to create a mashup that satisfies his information needs. Tyrion is a non-technical user, i.e., he does not know programming language and he is not familiar with technical terms of computer science.

Tyrion is going to organize his summer holidays, but he hasn't decided yet whether to go to in the US. Regardless of the destination, Tyrion 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, Tyrion 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). Tyrion clicks on the music category and, among the music services shown, he selects *SongKick*, 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 order to display the results of the application he is creating. Tyrion 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 SongKick dataset, Tyrion has to select those he is interested in, i.e., those that will be considered by the application he is creating. EFESTO enables Tyrion to make this selection by direct manipulation of elements shown in the user interface of his workspace. In fact, all SongKick 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. Tyrion 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, Tyrion drags & drops the *latitude* and *longitude* SongKick attributes into the respective fields (called **Visual Renderers** [4]) of the map UI template (Figure 1, circle 2). Tyrion 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 *Event_name*, *Artist* 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, Tyrion saves the mashup in the platform calling it "Upcoming events by artist name". From now on, this mashup is a **UI Component** in the user workspace, which is immediately executed in the Web browser and represented as a map, as shown in the central panel in Figure 2. By typing "Maroon 5" 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 *Event_name*, *Artist* and *City*) are shown.

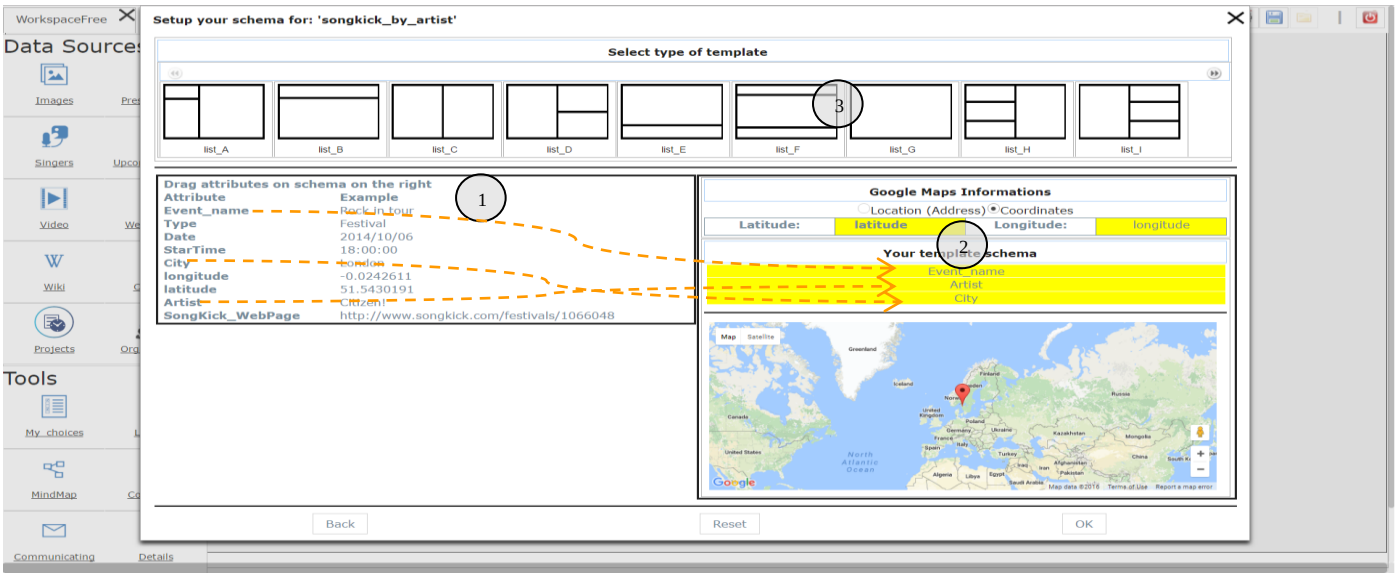

Figure 1. Mapping between some SongKick attributes and the fields of the map user template (circle 2).

Tyrion can later update the created mashup by integrating data coming from other **data sources** through **union** and **join** data mashup **operations** [11]. 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, Tyrion wants to retrieve more music events than those provided by SongKick. He then integrates SongKick with *Eventful* (another service retrieving music events). Technically, this is a union operation. Tyrion acts directly on the SongKick 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 Tyrion 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 SongKick 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, Tyrion would like to show artist video. SongKick does not provide such video but Tyrion can retrieve them from *YouTube*. Technically, this operation is a join between the SongKick *artist* attribute and YouTube. EFESTO supports Tyrion in a very simple way. Tyrion 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 (*artist* in this example), (b) the new data source (YouTube) and (c) how to visualize the YouTube results. From now on, when clicking on the artist name in the map info window, another window visualizes the YouTube videos related to the artist, as shown in the right panel of Figure 2.

Another operation available in EFESTO is the *change of visualization* for a given UI component. Tyrion, in fact, during the interaction with SongKick, 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 SongKick toolbar and chooses the "Change visualization" menu item. A wizard procedure guides Tyrion to (a) choose a UI template (*list* in this case), and (b) drag&drop the SongKick 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 Tyrion 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, Tyrion 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 [13]. DBpedia has been chosen as initial LOD cloud thanks to the vast amount of information it provides.

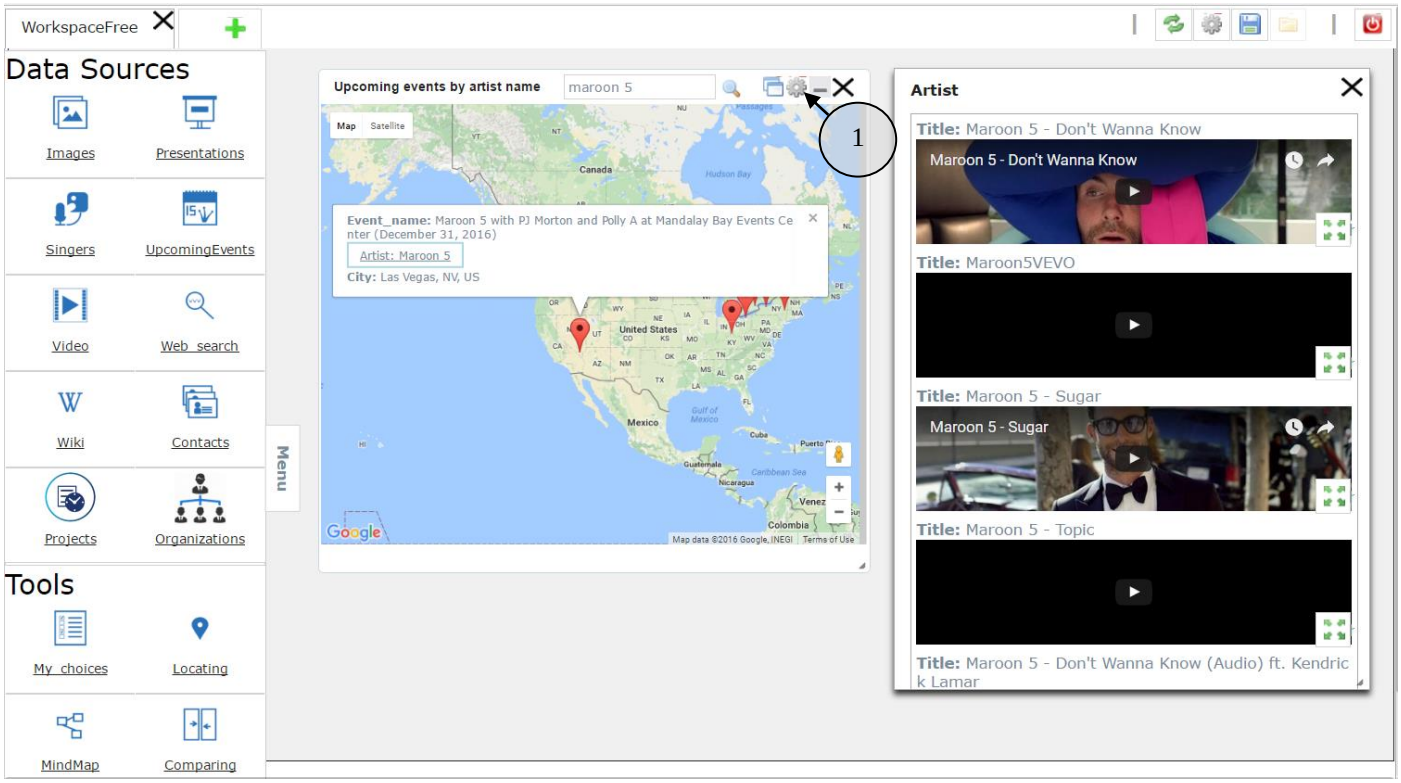

Figure 2. UI Component originated from SongKick data source visualized as a map and joined with YouTube to show artist video.

Thus, Tyrion can join the SongKick 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 Tyrion enriches the current UI Component with the attributes *genre*, *starting year of activity* and *artist photo*. Henceforward, Tyrion 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., YouTube) 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 Tyrion's join starting point is the attribute *city*, attributes like *borough*, *census*, *year*, *demographics* would be provided by the DBpedia-based data source.

C. *Actionable UI Component*

Our field studies [2, 12] 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 is 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. Tyrion is looking for hotels in New York 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 "New York" in the *Hotel* search bar. After including the *Comparing* tool in the workspace, Tyrion drags&drops inside it the first five hotels from the *Hotel* UI component. The *Comparing* tool supports Tyrion 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, Tyrion performs a search on the *SongKick* data source by using "New York" 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 (red ones) and the upcoming musical events in New York (green ones). Tyrion can now easily identify which musical events are close to the hotels he has previously chosen. However, it could happen that Tyrion adopts a different strategy. He wants to first identify upcoming musical events and then the hotels nearby.

He starts by retrieving musical events with *SongKick* (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 *actionable component*, which enable specific tasks. Thus, we call them Actionable UI Components. To create such flexible environments, a model has been presented in [14] that permits easy transition of information between different contexts; this model implements some of the Transformative User eXperience (TUX) principles described in [15, 16].

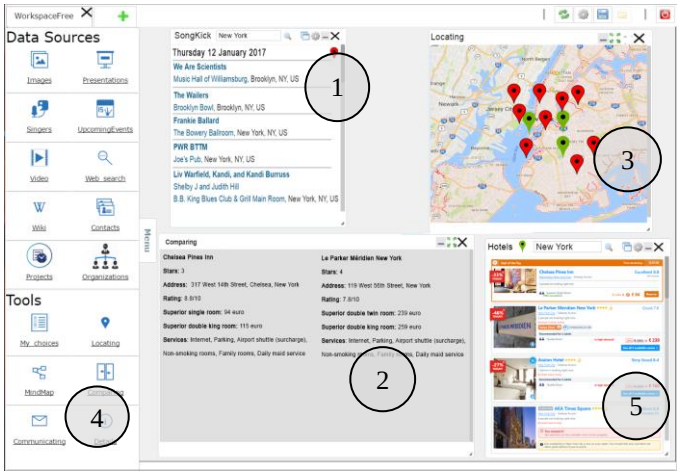

Figure 3. Use of some tools available in EFESTO to manipulate mashup data.

III. Model for UI Component Mashup

The main contribution of this article 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, namely: i) a different way to integrate service data by means of more powerful *join* and *union* operations for data mashup; ii) the *Actionable UI Components* that implement some Transformative User eXperience principles [15, 16]; iii) 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. According to [3], a UI component is a JavaScript/HTML stand-alone applications that can be instantiated and run inside any 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. In our approach, a UI component also allows the interaction with services data and functions thanks to its own UI (see Figure 4). More specifically, 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. 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.

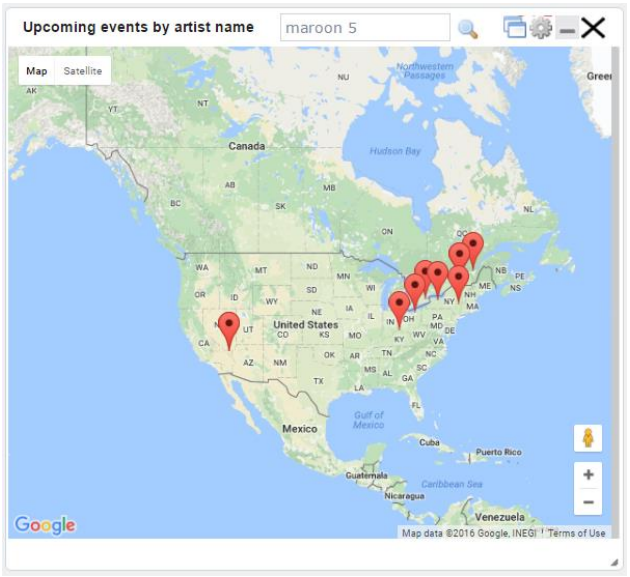

Figure 4: Example of UI Component that shows musical events on Google Maps.

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 [14].

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.

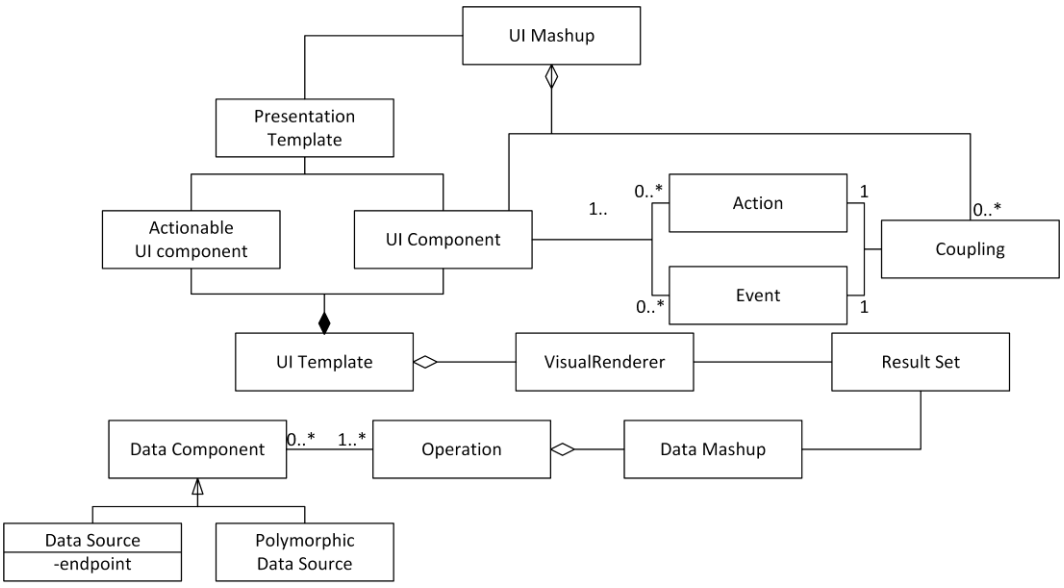

Figure 5: The mashup model.

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. 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.

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:

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

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 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 Actionable UI Components.

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 [17]. 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.

```
<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.

In Figure 6 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.*

Each service reported in the *service* tag is detailed in a separate service descriptor XML file. In Figure 7, 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 non-programmers. The 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, the *UI layer* provides and manages the 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. The *UI layer* runs in the user's Web browser and communicates with the *Logic* and *Data layer* that run on a remote Web server.

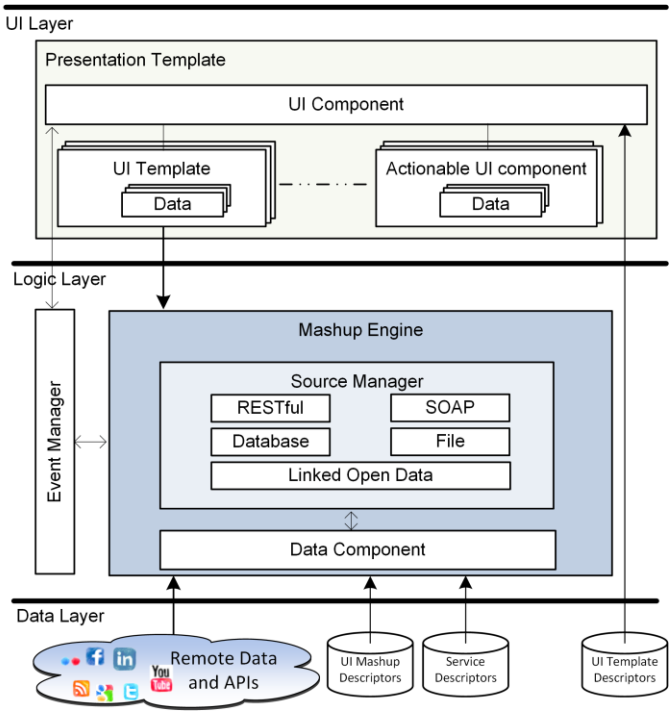

Figure 10. An high-level overview of the EFESTO three-layer architecture

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.

VI. Classifying Dimensions and EFESTO Characterization

A book that is a comprehensive reference for mashups, authored by Daniel and Matera, was published in 2014 [1]. The authors systematically cover the main concepts and techniques underlying mashup design and development, the synergies among the models involved at different levels of abstraction, and the way models materialize into composition paradigms and architectures of corresponding development tools. Some other publications also report several mashups and mashup tool features. For example, a recent review of tools, languages and methodologies for mashup development is presented in [18]. In this paper, the authors identify a set mashup and mashup tool features, taking into account other surveys and literature reviews. In [19] the design space of mashup tools has been proposed. The authors survey more than 60 articles on mashup tools, pointing out that only 22 tools are online. Based on these 22 tools, they propose a model focused on the main perspectives occurring in the design of mashup tools. On the basis of this model and taking into account what is reported in [1, 18], in the rest of this section we provide a further characterization of EFESTO in relation to the dimensions that most characterize mashup tools. The considered dimensions are reported in Table 1, indicating with * the ones derived from the design issues in [19].

Table 1. Mashup tool dimensions. The * indicates the dimensions derived from [19].

	Dimensions	Categories
Tool	Targeted end users *	*Non-Programmers - Local Developers - Expert Programmers*
	Automation degree *	*Full Automatic - Semi-Automatic – Manual*
	Liveness Level *	*1 – 2 – 3 – 4*
	Interaction Paradigm*	*Editable Example – Form based – Programming by example – Spreadsheets – Visual DSL – Visual Language (Iconic) – Visual language (Wiring, Implicit control flow) - Visual language (Wiring, Explicit control flow) – WYSIWYG – Natural Language*
	License	*Open Source – Commercial*
	Runtime environment	*Desktop – Mobile – Cloud*
	Supported Resources	*RESTful – SOAP – smart things – file – database – CSV – excel – smart things*

Targeted end users. In terms of programming skills, the end users range from non-programmers to experienced programmers, with in the middle professional end-users without programming skills, but who are interested in computers and technology, also called local developers [20]. Typically, the tools for experienced programmers are very powerful but less easily usable. On the contrary, the tools for non-programmers have simplified mechanisms that sacrifice the expressive power of the tools.

Non-programmers are users without any skill in programming and represent the majority of web users. The tools they are interested in are the ones that don't require learning/use of programming languages and technical mechanisms common for ICT experts and engineers (e.g., the use of logical operators and complex process flows). Thus, non-programmers should be provided with tools that limit their involvement in the development process to small customizations of predefined mashup templates, or the execution of parameterized mashups. A mashup tool for non-programmers has been described in [21]: it supports the development of adaptive user interfaces that react to contextual events related to users, devices, environments and social relationships. In particular, non-programmers can define the context-dependent behavior by means of trigger / action rules.

Local developers are users with knowledge in ICT technology and software usage without having skills in computer programming. Typically, those target users are willing to explore software and thus tools can provide composition functionality where mashups can be assembled from scratch by composing predefined components or by customizing and changing existing examples and templates. To do this, mashup tools for local developers have to provide a high level of abstraction that ideally hides all the underlying technical complexity of the mashup development. An example is the platform presented in [22], where the author proposes a new perspective on the problem of data integration on the web, the so-called surface web. The idea is to consider web page UI elements as interactive artefacts that enable the access to a set of operations that can be performed on the artefacts. For example, a user can integrate into his personal web page a list of videos gathered from YouTube and he can also append a list of Vimeo videos. This data integration can be improved by means of filtering and ordering mechanisms. These operations can be achieved, for example, by pointing and clicking elements (YouTube and Vimeo video lists), dragging and dropping them into a target page (e.g. personal Web page) and choosing options (filtering and ordering).

Programmers are users with an adequate knowledge of programming languages. They are the only users who can compose complex, rich in features, and powerful mashups, by means of tools that also provide Web scripting languages for developing more complex, customized mashups. EFESTO and the model it implements is strongly oriented to non-programmers and local developers. In fact, the mashup platforms we are interested in designing are devoted to non-technical users, providing them with a composition paradigm that fits their mental model.

Automation degree. This dimension refers to how much the mashup creation can be supported by the tool on behalf of its users. For this reason, the author of [19] identified two categories: *semi-automation* and *full-automation*. A new category, *manual*, has been introduced to indicate tools without support in mashup creation.

Tools that support a *semi-automated* creation of mashup partially support users, providing low levels of guidance and assistance. A semi-automated tool requires users to have more skills, but guarantees a high degree of freedom in creating a mashup that satisfies their needs.

A *full automation* in mashup development reduces the direct involvement of users in the development process, since users are strongly guided and assisted in the process, and play a supervisory role of just providing input or validating mashup results. These tools require a short learning curve and decrease the effort in mashup development. However, these facilities limit the possibility of creating a mashup that fits all the user needs. An example of a full-automated tool is *NaturalMash*, a tool that allows users to express in natural language what services they want in their mashup and how to orchestrate them [7]. To ensure the accuracy of the expressed user requests, NaturalMash limits the user with a controlled natural language (a subset of a natural language with limited vocabulary and grammar).

The *manual* category refers to those tools that do not provide any support to the users during the mashup creation. For example, Yahoo! created a console to formulate queries in a YQL language to perform data-mashup. In this tool, no help or assistance is given to the users because they have to formulate their queries following the YQL syntax. If the query is expressed correctly the the JSON or XML result is produced, otherwise a syntax error is shown.

With respect to this dimension, EFESTO supports a full automation degree. In fact, the tool composition paradigm is grounded on wizard procedures that guide the end users in creating a widget on top of a web service or in composing different web services by means of operations like join or union.

Liveness Level. The concept of liveness for visual languages presented in [23] was also adopted in the mashup domain [19]. In particular, the authors of [19] used the four liveness levels to express the tool complexity.

Level 1 expresses the Flowchart as ancillary description: in this case tools are used to compose a mashup as a non-runnable prototype that is not directly connected to any kind of runtime system. This prototype has just a user interface, but does not implement any functionality. If on one hand these tools don't require technical or programming skills, one the other hand an execution environment is necessary to execute the prototype. Microsoft Visio enables the creation of prototype mashups. The resulting prototypes can be completed with data and executed by Microsoft Excel [24].

Level 2 expresses the executable flowchart: tools in this category produce a mashup design blueprint with sufficient details to give it an executable semantics. The consistency (logical, semantical or syntactical) of the produced mashups

can be verified. However, the development of mashups through these tools requires skills in programming, since users need to define low-level technical details and thus their use is limited only to programmers. For example, *Activiti* is a lightweight workflow and Business Process Management (BPM) Platform characterized by features such as modeler, validation and remote user collaboration.

Level 3 expresses the edit triggered updates: in this case mashup tools are characterized by the development of mashups that can be easily deployed into operation. Users produce their mashups without devoting too much effort in the manual deployment typically by using two environments: one for the mashup editing and another for mashup execution. The deployment of the mashup under development in the editing environment could be obtained, for example, by clicking a run button that produces a deployment in the execution environment. An example of a mashup tool with these features is *JackBe Presto*, characterized by a design environment to model the mashup and a runtime environment used for debugging and monitoring purposes.

Level 4 expresses the Stream-driven updates: it is assigned to those tools that support live modification of the mashup code, while it is being executed, without differences between editing and execution. In this way, the mashup development is very fast and does not require particular programming skills. This approach is implemented in *DashMash*, a mashup tool that allows Web service creation and synchronization by means of an event-driven paradigm [25], without distinction between editing and execution time. With respect to this dimension, EFESTO supports live modification of mashup, since it blends into a single environment both the editing and the execution phases (level 4 - Stream-driven updates). In fact, the end users edit and run their mashups in the same environment, this tool exactly, without switching between two or more different environments. This mechanism is in line with our goal of proposing a mashup tool for non-technical end users.

Interaction Paradigm. One of the most important aspects that affects mashup tool adoption is the interaction paradigm to compose Web services. Actually, this dimension is called *Interaction Technique* in [19]. This is one of the most critical aspects that has limited the wide adoption of mashup tools in recent years, since the interaction paradigm proposed by several tools was not suitable for non-technical people. In the following, the most adopted interaction paradigms are reported.

The *Domain Specific Language* class includes technical interaction techniques since it refers to script languages targeted to solve specific problems for specific domains. In fact, these languages are characterized by textual syntax, sometimes similar to existing programming languages. Since these languages are very similar to programming languages, they require users to have strong knowledge and skills. An example is *Swashup*, a Web-based development environment for a textual Domain-Specific Language (DSL) based on the Ruby on Rails framework (RoR) [26].

A simpler but less powerful alternative is the class of *visual programming language*s, i.e. programming languages that use visual symbols, syntax, and semantics. In [19] the authors identify two sub-dimensions of visual programming languages: *visual wiring languages* and *iconic visual languages*. In the case of wiring languages, mashup tools visualize each mashup component or each mashup operation (e.g. filtering, sorting, merging) as a box that can be wired to other boxes. The mashup tools adopt, in most cases, the visual wiring mechanism, since this is the most explicit, thanks to the one-to-one relationship between the control flow and the data from one activity to another and the visual boxes wired to each other. Tools that implement iconic visual languages translate objects of mashup language into visual icons. In this way, if the icons are properly designed, users are facilitated in understanding how to compose a mashup.

The class of *WYSIWYG* (What You See Is What You Get) interaction mechanisms permits the creation and modification of a mashup on a graphical interface, without any need to switch from an editor environment to an execution environment (similar to the Liveness Level 4). These tools are very useful and suitable for non-programmers, since users have the mashup creation under control. However, these mechanisms sometimes represent a limitation, since users cannot access advanced features like filtering and conversion, that are typically hidden in the tool backend and thus one not available to the users.

An alternative is the class of *Programming by Demonstration* interaction techniques that allow the programming of a computer by giving an example of a particular task. Typically, these interactions are very useful to reduce or remove the need to learn programming languages and therefore they are also adopted in the context of mashup tools. With these techniques, users can 'show' to the mashup tools how a mashup should be. The tools are then in charge of converting the given example into a runnable program, i.e. a mashup.

Another class of techniques, similar to the previous one, is *Programming by Example Modification*, that consists in allowing users to modify a mashup instead of starting from scratch. If the tool provides an adequate set of examples, in most cases the customization of one of the available mashups requires a little effort by users.

An alternative class of interaction technique is *Spreadsheets*, one of the most popular end-user programming approaches to store, manipulate and display data. Tools that implement spreadsheets are oriented towards data mashups, but typically produce data visualization, thus they cannot build a mashup with their own user interface.

The last example is *form-based* interaction. Tools that adopt this interaction ask users to fill out forms to create an object or to edit an already existing one. Since the form filling is a common practice today on the Web for all kinds of users, mashup tools that implement this technique are easy to use by a wide range of users. However, these tools cannot produce complex mashups.

With respect to this dimension, EFESTO implements a WYSIWYG interaction mechanism to make the mashup modification more simple. In fact, during the wizard procedures that assist the users in editing their mashups, all the

Web service details are always visible and under the control of the end users in a WYSIWYG fashion.

License. Several mashup tools are conceived as research projects published in a public repository and/or available as runnable tools on a site. However, also commercial products appeared over time, thus, from the license perspective, two types of tools can be identified: open source and commercial.

In the case of *open source tools*, the community of users is composed of project contributors, i.e. programmers that participate in the tool development, and by end users, i.e. people that just use the tool. As is common in many open source projects, support and quality of mashup tools are sometimes quite low, since they are born as research projects and there are not adequate funds and interest in maintaining and updating the projects over time.

In the case of *commercial tools*, the development and update of the tools are performed by ICT companies that provide these tools for free or for payment. For example, in the case of *Netvibes* [27], the tool can be used for free to aggregate general Web services or for payment by agencies and enterprises providing them with advanced features like the possibilities to sell social dashboards to clients (agencies) or use personal data inside the dashboard (enterprises). With respect to this dimension, EFESTO is the result of academic research and thus released as open source software.

Runtime environment. Similar to the device location dimension presented in the previous section, different devices can be used to create a mashup with a tool. The desktop PCs are the most common environments on which mashup tools run, since they are equipped with wide screens that offer enough space to visualize mashup components.

However, in some cases, also mobile devices are used to create mashups. For example, the *Atooma* app transforms a smartphone into a 'personal assistant', since the users can automate all the manual operations they usually perform with their phone, e.g. combine Wi-Fi, Mobile Data, Facebook, Twitter, Instagram, Gmail and other services. In particular, with the Atooma app the users can simply create automations exploiting an event-action paradigm, that allows the definition of rules following the syntax "IF something happens DO something else".

With respect to this dimension, EFESTO runs on different environments that include tablets, desktop PCs and large interactive displays. The tool 'fits' the device on which it runs, optimizing the UI and functions, depending on the hardware peculiarities and constraints (e.g. display size, interaction methods, etc.).

Supported resources. This dimension is related to the type of resource dimension identified in the mashup dimensions. In fact, in order to create a mashup with different services, mashup tools have to support different types of services, such as the ones previously identified (e.g. RESTful and SOAP). The more types of resources the tool is able to support more flexible and powerful the tool is.

With respect to this dimension, the mashup tool described in this article implements a mashup engine that permits the mashup of different data sources such as RESTful web services, CSV files and databases. The modularity of this engine allows the easy integration if the tool with other types of data sources.

VII. Conclusion

This article discusses some abstractions that can promote mashup platforms as tools that permit the easy creation (i.e., even by non-technical end users) of 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 of available service and data at the presentation layer, in a component-based fashion - an aspect that has been investigated before. Some papers, indeed, discuss and motivate the so-called UI-based integration [4, 28, 29] as a new component-based integration paradigm, that privileges the creation of fully-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. In this direction, this article 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 Actionable UI Components, providing for the visualization of data extracted from data sources and for data manipulation operations through task-related functions. A model is also provided to describe the most salient elements that enables the integration at the presentation layer of Actionable UI components.

By capitalizing on the experience gained in recent years by the authors in the development of prototypes of mashup platforms, this article aims to propose a systematic view on concepts and techniques underlying mashup design and on the way such concepts materialize into the composition paradigms and architectures of corresponding development tools, independently of specific approaches and technologies and, thus, of more general validity. Our current work is devoted to enriching the EFESTO platform by means of tools for actionable components and to customizing the platform to other application domains, so that further validation studies will be performed.

References

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

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

[3] Yu, J., Benatallah, B., Saint-Paul, R., Casati, F., Daniel, F. and Matera, M. (2007). A framework for rapid integration of presentation components. In *Proc. of International Conference on World Wide Web (WWW '07)*. ACM, 923-932.

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

[5] Spillner, J., Feldmann, M., Braun, I., Springer, T. and Schill, A. (2008). Ad-Hoc Usage of Web Services with Dynvoker. In *Towards a Service-Based Internet - ServiceWave 2008*. Lecture Notes in Computer Science 5377. Springer Berlin Heidelberg, 208-219.

[6] Krummenacher, R., Norton, B., Simperl, E. and Pedrinaci, C. (2009). SOA4All: Enabling Web-scale Service Economies. In *Proc. of International Conference on Semantic Computing (ICSC '09)*. IEEE Computer Society, 535-542.

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

[8] Hirmer, P. and Mitschang, B. (2016). FlexMash–Flexible Data Mashups Based on Pattern-Based Model Transformation. In *Rapid Mashup Development Tools - Rapid Mashup Challenge in ICWE 2015*. 591. Springer Verlag, 12-30.

[9] Lieberman, H., Paternò, F. and Wulf, V. (2006). *End User Development*, Springer.

[10] Fischer, G. (2009). End-User Development and Meta-design: Foundations for Cultures of Participation. In *International Symposium on End-User Development - Is-EUD 2009*. Lecture Notes in Computer Science 5435. Springer Berlin Heidelberg, 3-14.

[11] Desolda, G., Ardito, C. and Matera, M. (2015). EFESTO: A platform for the End-User Development of Interactive Workspaces for Data Exploration. In *Rapid Mashup Development Tools - Rapid Mashup Challenge in ICWE 2015*. Communications in Computer and Information Science 591. Springer Verlag, 63 - 81.

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

[13] Desolda, G. (2015). Enhancing Workspace Composition by Exploiting Linked Open Data as a Polymorphic Data Source. In *Intelligent Interactive Multimedia Systems and Services (KES-IIMSS '15)*. Smart Innovation, Systems and Technologies 40. Springer International Publishing, 97-108.

[14] Ardito, C., Costabile, M. F., Desolda, G., Latzina, M. and Matera, M. (2015). Making Mashups Actionable Through Elastic Design Principles. In *End-User Development - Is-EUD 2015*. Lecture Notes in Computer Science 9083. Springer Verlag, 236-241.

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

[16] Beringer, J. and Latzina, M. (2015). Elastic workplace design. In *Designing Socially Embedded Technologies in the Real-World*. Computer Supported Cooperative Work Part I. Springer, 19-33.

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

[18] Paredes-Valverde, M. A., Alor-Hernández, G., Rodríguez-González, A., Valencia-García, R. and Jiménez-Domingo, E. (2015). A systematic review of tools, languages, and methodologies for mashup development. *Software: Practice and Experience*, 45, 3, 365-397.

[19] Aghaee, S., Nowak, M. and Pautasso, C. (2012). Reusable decision space for mashup tool design. In *Proc. of ACM SIGCHI symposium on Engineering Interactive Computing Systems (EICS '12)*. ACM, 211-220.

[20] Nardi, B. A. (1993). *A small matter of programming: perspectives on end user computing*. MIT Press.

[21] Ghiani, G., Paternò, F., Spano, L. D. and Pintori, G. (2016). An environment for End-User Development of Web mashups. *International Journal of Human-Computer Studies*, 87, 38-64.

[22] Daniel, F. (2015). Live, Personal Data Integration Through UI-Oriented Computing. In *Engineering the Web in the Big Data Era*. Lecture Notes in Computer Science 9114. Springer International Publishing, 479-497.

[23] Tanimoto, S. L. (1990). VIVA: A visual language for image processing. *Journal of Visual Languages & Computing*, 1, 2, 127-139.

[24] Wright, S., Bakmand-Mikalski, D., bin Rais, R., Bishop, D., Eddinger, M., Farnhill, B., Hild, E., Krause, J., Loriot, C. and Malik, S. (2011). Designing Mashups with Excel and Visio. In *Expert SharePoint 2010 Practices*. Springer, 513-539.

[25] Cappiello, C., Matera, M., Picozzi, M., Sprega, G., Barbagallo, D. and Francalanci, C. (2011). DashMash: A Mashup Environment for End User Development. In *Web Engineering - ICWE 2011*. Lecture Notes in Computer Science 6757. Springer Berlin Heidelberg, 152-166.

[26] Maximilien, E. M., Wilkinson, H., Desai, N. and Tai, S. (2007). *A domain-specific language for web apis and services mashups*. Springer.

[27] Netvibes. Retrieved from https://www.netvibes.com/. Nov 26th, 2015

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

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