Composition at the frontend : the user centric approach

Nassim Laga, Emmanuel Bertin, Noel Crespi

Composition at the Frontend: the User Centric Approach

Nassim Laga, Emmanuel Bertin
Orange Labs
France Telecom R&D, 42, rue des Coutures,
14000 Caen France
{nassim.laga, emmanuel.bertin}@orange-ftgroup.com

Noel Crespi
Institut Telecom, Telecom SudParis,
9 rue Charles Fourier, 91011, Evry Cedex, France
noel.crespi@it-sudparis.eu

***Abstract*—User generated content (UGC) is the main characteristic of current Web 2.0. This paper summarizes our experience in applying such philosophy (user generated) in the service creation field. We summarize why current SOA did not succeed in enabling end-users to create services, and propose our approach based on frontend service composition.**

Keywords-Web 2.0; SOA; service composition; user-centric design

I. INTRODUCTION

Web 2.0 paradigm has really revolutionized the Web. Software features are no longer packaged as a single application; instead, they are split into and published as services in order to promote cross-network and cross-organizations sharing, collaboration, reusability, and integration. This is known as Service-Oriented Computing (SOC) [1]; developers create services, and make them available for optional reuse by other developers. In order to facilitate the publication and the discovery of such services over the Web, Service-Oriented Architecture (SOA) has emerged as a solution. Providers publish their services into a common registry, and third party developers discover and reuse these services. However, while current SOA technologies have definitely succeeded in enabling developers to discover and reuse services, the end-users can not combine themselves these services according to their needs. This is essentially due to the fact that SOA is initially conceived for a machine-to-machine communication and not for human-to machine-communication.

On the other hand, user devices and networks are respectively more sophisticated and reliable. The devices embed more and more hardware and software capabilities, and the operators have succeeded to significantly enhance the quality of service of their networks (bandwidth, response-time…etc). These advances in networking and devices fostered the adoption of the XaaS paradigm (Everything as a Service). As end-users do no longer care about the connectivity, applications can be hosted on the Web and accessed only on demand. Thus, the user interface of the service is displayed on the device, but its logic is running on a remote server.

In this paper we claim that technology advances provide also the opportunity of investigating a new service composition approach based on the end-user device (frontend). This is characterized by composing the services by composing their UIs. This enables us to easily interact with the end-user and enhance the intuitiveness of the composition process. This paper summarizes our experience regarding this new service composition field. We first explain it and position it regarding traditional service composition in SOA. We also provide an insight of the existing technologies. Second, we summarize our vision and the new opportunities provided by frontend composition. Third, we illustrate the frameworks we have defined and implemented. Finally, we conclude the paper with a summary of our experience.

II. SERVICE COMPOSITION BACKGROUND

As we illustrate in Figure 1, we classify existing service composition tools into a backend and a frontend service composition category. In the following subsections, we define and review existing technologies in each category.

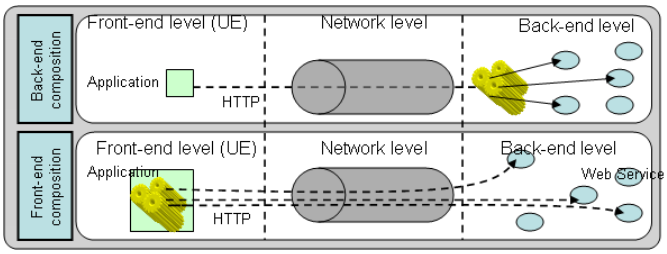

Figure 1. Backend and frontend composition approaches.

A. Backend Level

Independently of the adopted technologies (WSDL/SOAP or REST), SOA is the architecture model that enables SOC [2]. It provides the publication and discovery facilities through the usage of a common registry. Thus, developers create services, describe them (e.g. using Web Service Description Language (WSDL) [3]), and publish them into a common registry (e.g. UDDI [3]). Third party developers can then use the discovery facilities provided by the registry to discover the services they need. Finally, they invoke these services (e.g. by creating SOAP messages and sending them over HTTP requests). SOA and the enabling technologies have been successfully adopted by developers, as currently major development platforms, provide service reuse capability through WSDL and SOAP libraries (J2EE, Microsoft Visual Studio). However, ordinary end-users, without development skills, are currently left apart. Indeed, as ordinary end-users do not understand XML based files such WSDL and SOAP, it is hardly conceivable for them to compose services. Figure 2 shows this technology gap.

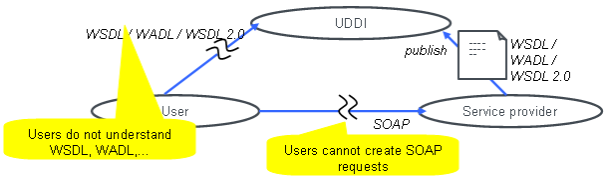

Figure 2. Limitations of backend service composition.

To tackle this limitation, and enable end-users to compose services, several composition tools have emerged on the top of SOA; we refer to them as backend composition. The goal is to facilitate and speed up as much as possible the service composition process; the process of discovering and reusing services. The adopted approach is characterized by adding semantic and semantic reasoning component the architecture. Thus, according to an end-user need, expressed for example in his natural language, an end-to-end composite service is created automatically. This composition approach provides definitely the easiest way for end-users to compose services. But, it is still limited. It requires a high expressive semantic description of services, usually performed using ontologies. Consequently, in practice, the modeling of a wide domain of knowledge is not only hard, but it also requires a continuous update. In addition, composite services that are created by this composition approach are very simple and usually based on the definition of a flowchart between request-response based services, without considering events and sessions.

Another approach for enabling end-users to compose services is characterized by providing a UI in which they can specify themselves the flow between services through a flowchart diagram. An example of such composition approach is Yahoo Pipes [4]. Yahoo Pipes introduces two interesting features. First, it uses UIs as building blocs in the flowchart definition process. Second, it does not require a high level of expressiveness in the semantic description as the composition is performed manually, by humans. However, it still has three limitations. First, it is still based on flowchart definition, which is not obvious for ordinary end-users to master, though it facilitates considerably the composition process. Second, the considered services are data oriented and request-response based. Consequently, session and event based services such as Instant Messaging and Telephoning are not considered. This limitation is due to the need of simplifying the flowchart to be accessible by ordinary end-users. Other flowchart based frameworks [5 and 6] enable session and event based composition, but this feature complicates the composition tool, and makes it addressed for developers such as Business Process Execution Language (BPEL) [7]. Third, the building blocs (UIs) are actually UIs that enable the end-users to enter the inputs required by the corresponding services. In other words, the service in itself is not natively integrated in the UI; the outputs for instance are XML-based. Nevertheless, Yahoo Pipes provide some basic UI patterns for the display (List, Map, and Images). But, services those outputs do not match the basic UI patterns defined by Yahoo Pipes can not be considered (e.g. Telephony, IM, and Video Player).

B. *Frontend Composition*

We define the frontend service composition as the process of combining services at the end-user device level; the composition logic is defined and executed at the end-user device. This approach is fostered by the computing capabilities made available at the frontend level. Here, we categorize frontend service composition into two main categories: programming language based composition, and Widgets based composition.

1) *Programming Language Based Composition*

The most basic approach, addressed for developers, to compose services at the frontend level is to use a programming language such as Java and C++. The only requirement is to be able to construct a SOAP message and send it through HTTP requests. Major programming languages even provide ready-to-use libraries to facilitate the reading of WSDL files and the construction of SOAP messages. There are even JavaScript and AJAX libraries that provide such facilities (e.g. jQuery SOAP client). As a consequence, developers can easily orchestrate services at the browser level (frontend). This composition type still remains developer-centric.

Another approach for composing services at the frontend is to use facilities provided by the device Operating System (OS). In Microsoft Windows for instance, developers of applications can dynamically (at the runtime) discover and use other application capabilities installed in the end-user machine. They use the OLE (Object Linking and Embedding) automation. Google Android OS also embeds such type of mechanism (Intent). It provides developers with the capability of invoking the functionalities that are loaded on the end-user device. For example, a contact list application can use the functionalities provided by a mailing application loaded by the end-user.

The OS-based composition mechanisms provide an interesting feature. They are more user-centric then those based on programming languages. They provide basic personalization for end-users as the composition happens only between services that are installed by the end-user on his device. For example, in the Intent based composition of Google Android, if a contact list service is composed with a mailing service (e.g. Gmail), the end-user can change the mailing service (e.g. to Yahoo mail) and the composition remains valid. Nevertheless, the end-user can not change the composition logic; the fact that the contact list service is composed with a mailing service and not with a Map service for example. This is the main limitation of this composition approach; only developers can change the composition logic.

2) *Widgets Composition*

Widgets aggregators [8] such as iGoogle[1] provide to end-users the capability of constructuting their own Web page from existing services. They use Widgets [9] as a basic and reusable element in the construction of the Web page. However, the integration between these Widgets is not widely considered as there is no standardized approach for doing so. Nevertheless, some emerging Widget aggregators (EzWeb [10] and IBM Mashup Center [11]) do provide such capabilities. These tools provide to end-users the capability of defining “wires” between two Widgets. A wire is a definition of a mapping between an output of a Widget with the input of

[1] iGoogle, http://www.google.com/ig, accessed on June 21, 2010

another. As a consequence, each time the source Widget generates a new output, the destination Widget is launched automatically, with that output as an input parameter. This Widget composition, however, is still based on a flowchart definition. It requires from the end-user to understand the concept of input, output, and the mapping between them; concepts which are not accessible for ordinary end-users.

III. FRONTEND COMPOSITION BASED ON WEB WIDGETS

In this paper we deeply investigate the potential of the Widget based composition. Our main target is to enhance the intuitiveness and the richness of this composition category. Our approach is summarized in Figure 3. Basically, we use Widgets as basic elements in the composition process. A Widget is a UI that provides access to a business logic implementation exposed as a Web service or using any other technology. These Widgets are then composed with each other to form a more innovative service and enhance the end-user experience. The UI invokes the server side part (the business logic) using AJAX requests. In order to enhance the intuitiveness the Widget based composition, we first detect semantic matching between inputs and outputs of the Widgets that are loaded on the same environment. Second, we accordingly create wires between these Widgets. Finally, we enable the end-users to modify or delete a created wire. This process enables the end-users to compose services, without having to understand the concept of input/output mapping. In addition, when compared to SOA-based composition, this approach is not only user-centric, but it also does not require a high level of semantic expressiveness as we harness the intelligence of the end-user in the composition process (since the end-user can delete wires that he considers not pertinent). Thus, the only requirement is to detect if two services could be composed, and the end-user is in charge of checking the semantic validity of the composition.

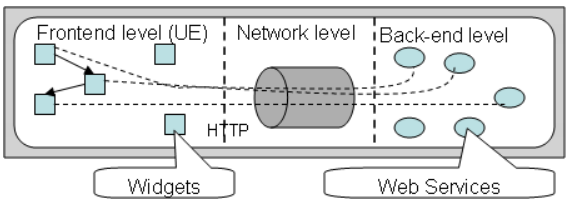

Figure 3. Widget-based Composition approach.

In order to enhance the richness of the Widget based composition, we first enable the composition based on events, and second, we introduce two innovative mechanisms: the unstructured data based composition and the cross device based composition. The former aims to enable the definition of composite services based on unstructured data. Unstructured data are data that are not formatted nor declared by the developer of the corresponding Widget. This is especially useful when considering communication services such as IM, email, and telephony, where end-users could exchange data (e.g. addresses, phone numbers, dates) that could be useful to compose with other services (e.g. Map, Contact List, Agenda). The second innovative mechanism aims to take into account the proliferation of end-user devices. It enables end-users to create composite services distributed over their devices. Basically, in addition of defining themselves composite services, end-users assign each basic element within the composite service to a preferred device. Consequently, the end-user can easily create a mashup of a mailing service and video player service, where the mailing service runs on a mobile device and the video player runs on a TV; this enables him to read emails on a mobile and attached movies on a TV.

From the technical perspectives, our solution is based on Web technologies, and is not limited to a specific application store. This is fostered by the existing Web standards (Widget, HTML, JavaScript, and CSS) from one hand, and the evolution made within mobile Web browsers fields from another hand. Services are thus created ones and executed on heterogeneous devices. The only requirement is to access these services through the Widget aggregator, which is a Web application.

IV. FRONTEND COMPOSITION MECHANISMS

This section details the Widget based composition mechanisms that we introduce to show the pertinence of frontend composition. In the first mechanism, we firstly use semantic matching to detect compatible Widgets and compose them by creating wires between outputs of some Widgets with inputs of others, and secondly, we enable the end-user to delete undesired wires or modify their types. This mechanism facilitates significantly the composition process for end-users as we detail in the next subsection. The second mechanisms we introduce enables end-users to define composite services based on unstructured data. Finally, the fourth mechanism aims to define composite services distributed over multiple devices.

A. Semantic-based composition of Widgets

1) Mechanism Goal

The goal of the semantic-based composition of Widgets is to provide an intuitive approach for ordinary end-users, without computing skills, to compose services. Our approach is characterized by composing automatically Widgets that are loaded to the same environment, according to semantic matching between them. Secondly, we provide to the end-user the capability of personalizing the composition. By deleting for example undesired wires, or modify the type of others.

2) Mechanism Design and illustration

To enable an automatic composition of Widgets loaded on the same environment, it is important to describe their capabilities. Each Widget may have one or several functionalities. Each functionality is described by its name, URL, the type of inputs it expects, and the type of outputs it may generate. The input type and the output type are described using Microformats [12] semantic dictionary. Microformats initiative is characterized by the definition of a set of formats to represent information used in Web applications. Examples of such information are: addresses, phone numbers, contact cards, and calendar events. We believe that Microformats based semantic model is a practical approach for adding semantic to Widgets. It represents a good tradeoff between expressiveness and scalability. The model, associated to the intelligence of the end-user, is expressive enough to detect if two Widgets could be composed, and it is scalable as it is not very expressive (The expressiveness is compensated by the end-user involvement in the composition).

When two Widgets are loaded on the same environment, the Widget aggregator detects the semantic matchings between outputs of each Widget and the inputs of others. As illustrated in Table I, there are three types of semantic matching [13]: exact matching, inclusion, and reverse inclusion.

TABLE I. SEMANTIC MATCHING VARIANTS.

Semantic Matching	Description
Exact	The output type (of the source Widget) is exactly the same as the input type (of the destination Widget)
Inclusion	The output type (of the source Widget) is a sub-element of the input type (of the destination Widget)
Reverse inclusion	The input type (of the destination Widget) is a sub-element of the output type (of the source Widget)

If such semantic matching is detected, a wire is created between the two Widgets. Each wire is represented to the end-user through a UI element (e.g. icon, text button). This UI element is actually a representation of the destination functionality, included in the source Widget; when the user clicks on it, the corresponding functionality is invoked using data generated by the source Widget as input parameters. When the semantic matching is not Exact (Inclusion, or Reverse inclusion), some modifications are performed on the generated data before the invocation of the functionality.

Figure 4 shows a composite service created using this mechanism. It illustrates a directory Widget composed with a telephony Widget. Thus, when the end-user searches a contact on the directory Widget, the Widget aggregator propose automatically to call that contact; and when an incoming call occurs on the telephony Widget, the framework propose automatically to search the caller on the directory Widget to have more information (e.g. name, address, email…etc).

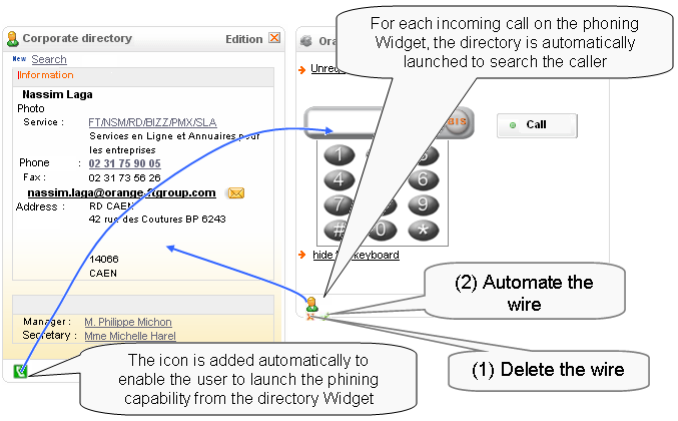

Figure 4. Semantic-based composition of Widgets.

From the technical perspectives, the Widget aggregator first creates a composite service which connects all connectable Widgets. This process is scalable as it is limited to Widgets that are loaded into the user environment. However, this may lead to the creation of wires which are not semantically valid or undesired by the end-user. Therefore, it is important to provide to end-user the capability of modifying or deleting the wires, while maintaining the mechanism as simple as possible. Hence, we propose to add two visual components to each created wire; one for deleting the wire, and one for automating the wire. An automated wire is executed each time the source Widget generates the needed output. In Figure 4 for instance, when the user choose to automate the wire from the telephony Widget to the directory, each time there will be a call in the telephony Widget, the directory Widget searches the caller.

B. *Unstructured Data based composition*

1) *Mechanism Goal*

SOA-based composition tools are all based on mapping outputs of services with inputs of others; inputs and outputs which are declared in the service description by developers when publishing their services. However, additional, and unstructured data, which are hardly expectable by developers, might be generated at runtime. This is especially true when considering end-user generated content and communication services. For example, two IM communicating end-users are likely to exchange data such as phone numbers, email addresses, and postal addresses during an IM session. These data can not be expected by the developer of the IM service. As a consequence, it is currently almost impossible to consider them in a composite service definition. Therefore, we propose in this section to enable end-users to define a composite service based on unstructured data. By relying on the frontend, we also make use of the end-user intelligence in the unstructured data extraction process. Firstly, only data related to the Widgets that are loaded on the environment are extracted, and secondly, the end-user can check whether the correctness of the extraction.

2) *Mechanism Design and illustration*

The mechanism we propose is an enhancement to the semantic-based composition of Widgets. This enhancement is characterized by three items. First, we introduce a repository of data extraction modules. These modules, when invoked, are in charge of extracting unstructured data (of a specific type) from a specified Widget. Second, we create one-to-one associations between data types present in the semantic dictionary with the data extraction modules. Finally, we define a new service composition pattern that enables the end-user to capture and reuse unstructured data. Figure 5 illustrates a composite service created using this new pattern. It illustrates an IM service composed with the telephony, Map, and agenda service, although the inputs (phone number, postal address, and date) of these services are not legacy outputs of the IM service. The creation of this composite service is performed through the Widget aggregator. First, the end-user loads the different Widgets (IM, telephony, Map, and agenda) into his environment, and second, he/she adds a data extraction module to a Widget (IM) as illustrated in Figure 6.

Figure 6 illustrates also the execution of the composite service. For each wire, in which the input data type of the destination Widget is not a legacy output of a source Widget, the Widget aggregator creates a listener, which is in charge of extracting the data needed by the destination Widget of the wire. It uses for this purpose the data extraction module that corresponds to the input data type of the destination Widget. Thus, each time the listener detects the corresponding data type within the source Widget of the wire, it extracts that data and adds an HTML element to the source Widget through which the end-user can execute the wire (send the extracted data to the destination Widget and launch the corresponding

functionality). For example, in Figure 6, each time a date is detected on the IM service, the Widget aggregator proposes to the end-user to check his availability on the agenda service.

Figure 5. Unstructured data based composition.

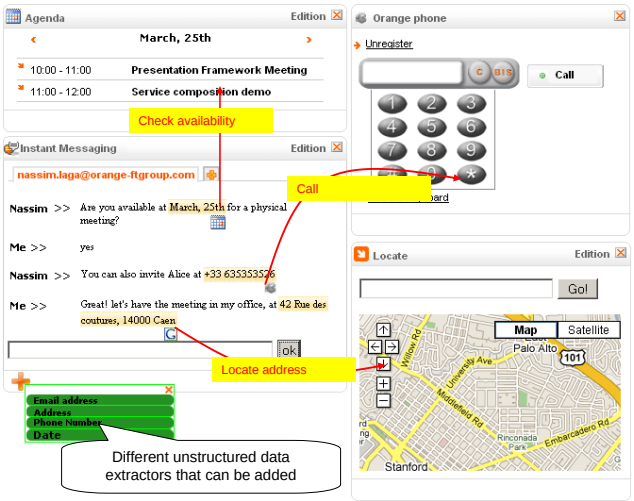

Figure 6. Illustration of an unstructured data based composition.

Current SOA based service composition tools do not enable end-users (ordinary or advanced) to make such composition of services. This is due to the fact that the composition, when addressed for ordinary end-users, creates a request-response based composite service. The composite service is executed end-to-end without interacting with the end-user. This is due to the lack of presentation layer (faces) within SOA.

C. Cross-device composition

1) Mechanism Goal

With the proliferation of end-user devices (laptops, cell phones, TV, tablets, PCs, book readers,…etc.), end-users may need to compose two Widgets loaded on two different devices. For instance, it is likely to want to play a movie attached on a web mail (running on a mobile phone), on a TV video player. Therefore, we propose in this section a distributed mechanism, running on the end-user devices, to combine the Widgets loaded on the same or different devices. We use the Web as a transport media.

2) Mechanism Design and illustration

To enable the end-user to combine Widgets loaded on different devices, while maintaining the composition process as simple as possible, we propose in this section a protocol implemented on the Widget aggregator level to enable different instances of different aggregators to exchange the capabilities of the different Widgets (see Figure 7). Figure 7 shows also that each application, complying with this protocol, may use the capabilities of the Widgets loaded on different aggregators and vice-versa. The design of this protocol is detailed in [14], and summarized here through six items:

- First, each time the user instantiates a Widget aggregator, the user is authenticated. Then, using the user identifier, the Widget aggregator creates a communication channel named "*/userId*" (if the instance is the first one), or joins the channel "*/userId*" if it is already created by another instance. The communication channel enables different aggregator instances of the same user to exchange data.
- Second, each Widget capability is defined through a quintuplet *C (Widget Name, Widget Instance Id, Functionality URL, Input Type, Device Id)*. The *Widget Instance Id* and *Device Id* fields are initialized only when the Widget is instantiated (loaded on a Widget aggregator).
- Third, each time a Widget is loaded on an aggregator instance, the corresponding capabilities are published to other instances through the communication channel "*/userId*". These capabilities are tagged "*available*".
- Fourth, each time an aggregator instance receives a capability of a Widget loaded on a different device, it checks if there are any semantic matching (based on microformats) between the input type of the capability and the outputs of the Widgets loaded on this instance. If such semantic matching is detected, the Widget aggregator adds an HTML element to the corresponding Widget; an HTML element which enables the user to launch the destination Widget loaded on another device.
- Fifth, when the user clicks on an HTML element of a wire, the corresponding data are sent to the corresponding destination Widget through the communication channel "*userId*". The sent data are tagged "*remote_call*". When the destination aggregator instance receives such data, it calls the specified functionality.
- Sixth, each time a Widget is unloaded from an aggregator instance, the corresponding capabilities are published to other instances through the communication channel "*/userId*". These capabilities are tagged "*unavailable*". When aggregator instances receive such information, they delete the wires that refer to the received capabilities (which are no longer available).

From the end-user perspective, when using Widget aggregators compliant with the protocol we define, it is very easy to create a composite service distributed over different devices. The only action needed from the end-user is to instantiate the Widgets (load the Widgets into his aggregator instances). Figure 8 shows an example of a composite service.

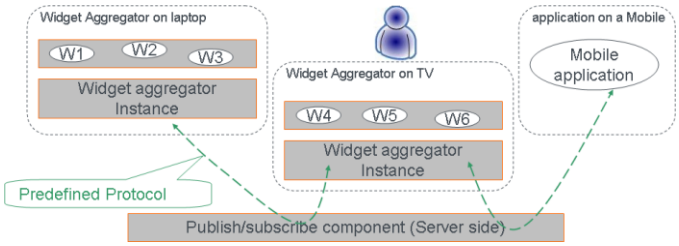

Figure 7. Cross device Widget composition basis.

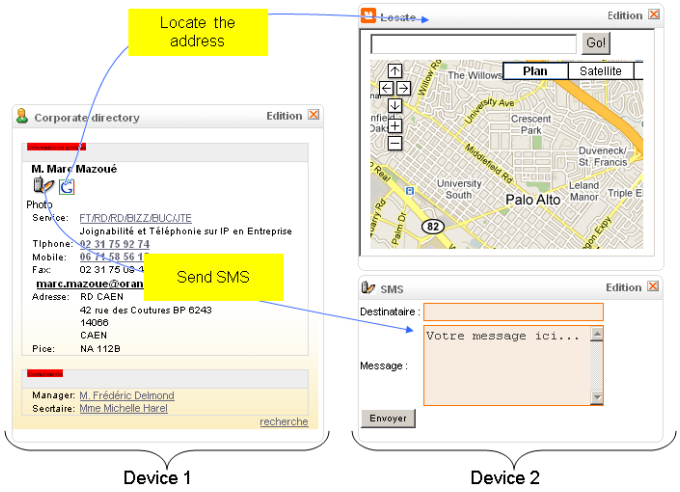

Figure 8. Illustration of the cross device composition.

This composition mechanism enhances the richness and the user centricity of the Widget composition. It enhances the richness of the Widget composition as now end-users can create composite services distributed over multiple devices, and it enhances the user centricity as the composition takes into account the proliferation of the devices. Two characteristics that do not exist today in SOA-based composition technologies addressed for end-users.

V. FRONTEND VS BACKEND COMPOSITION

The frontend composition is a new service composition approach fostered by the technology advances in devices, networks, and frontend computing capabilities. After a deep investigation of this approach, supported by the definition and implementation of several prototypes illustrated in this paper, we conclude that frontend service composition and backend service composition are complementary in some aspects and concurrent in others. Table I summarizes our comparison.

Table II: Backend vs frontend composition.

	Backend			Frontend	
	Programming language	**Flowchart**	**Automatic**	**Programming language**	**Widget based**
Richness of composite services	Yes	No	No	Yes	Medium
Richness of composite service UI	Yes	No	No	Yes	Yes
Developer centricity	Yes	No	No	Yes	No
User centricity	No	Yes	Yes	Yes (basic)	Yes
Semantic heavyness	No semantic	Not heavy	Yes (heavy Semantic)	No semantic	Not heavy
Events Consideration	Yes	No	No	Yes	Yes
Composition during sessions	Yes	No	No	Yes	Yes

Service creation by end-users is part of Web 2.0 paradigm as concluded by Tim Oreilly in [15] ("Trusting users as co-developers", "Leveraging the long tail through customer self-service"). In addition, allowing end-users to compose services has a positive impact on business process management as we detailed in [16]. Obviously, the end-user may create composite services that are not guaranteed to work. Nevertheless, the approach we have proposed in this paper relies on the reusability of composite services based on UI. This means that the errors that could happen when composing services are the same as those that could happen when the end-user manually compose them (using manual keyboarding, or copy and paste). This type of errors is usually controlled by the used basic services through an additional step to check the validity of the provided inputs.

REFERENCES

[1] M.P. Papazoglou, P. Traverso, P. Dustdar, and F. Leymann, Service-Oriented Computing Research Roadmap, technical report/vision paper on Service oriented computing European Union Information Society Technologies (IST), 2006.

[2] H. Luthria, F. Rabhi, Service oriented computing in practice: an agenda for research into the factors influencing the organizational adoption of service oriented architectures, Journal of Theoretical and Applied Electronic Commerce Research, v.4 n.1, p.39-56, April 2009.

[3] E. Newcomer, "*Understanding Web Services: XML, Wsdl, Soap, and UDDI*" Addison, Wesley, Boston, Mass., May 2002.

[4] Yahoo Pipes, http://pipes.yahoo.com/pipes/, accessed on June 25, 2010.

[5] J.C. Yelmo, J.M. del Alamo, R. Trapero, P. Falcarm, Y. Jian, B. Cairo, C. Baladron, "*A user-centric service creation approach for Next Generation Networks*," In Innovations in NGN: Future Network and Services, 2008. K-INGN 2008.

[6] B. Bhushan, et al. "Spice unified architecture," SPICE FP6 project, Architecture Delivrable, http://www.ist-spice.org/documents/SPICE_WP1_unified_architecture_Phase%202.pdf , accessed on June 25, 2010.

[7] A. Tony, et al. "*Business Process Execution Language for Web Services*" http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf

[8] N. Laga, E. Bertin, N. Crespi, "*A unique interface for web and telecom services: From feeds aggregator to services aggregator*," In ICIN 2008, Bordeaux, France, 20-23 October 2008.

[9] M. Caceres, Widgets 1.0 Requirements, W3C Working Draft, 2007.

[10] J. Soriano, "*Fostering Innovation in a Mashup-oriented Enterprise 2.0 Collaboration Environment*," UK, sai: sisn.2007.07.024, Vol 1, No 1, Jul 2007, pp 62-68.

[11] IBM Mashup Center, http://www-01.ibm.com/software/info/mashup-center/, accessed on June 25, 2010.

[12] Microformats initiative, http://microformats.org/, accessed on June 25, 2010.

[13] F. Lécué, A. Léger, "*Semantic Web Service Composition Based on a Closed World Assumption*," Web Services, 2006. ECOWS '06. 4th European Conference, pp.233-242, Dec. 2006.

[14] N. Laga, E. Bertin, N. Crespi, "*Widgets to facilitate service integration in a pervasive environment*," In Proceedings of Internation Conference on Communications (ICC) 2010, Cape Town, South Africa, May 23-27.

[15] Tim O'Reilly, "*What Is Web 2.0, Design Patterns and Business Models for the Next Generation of Software*" September 30, 2005. Available from: http://www.oreillynet.com/lpt/a/6228 (accessed on June 25, 2010).

[16] N. Laga, E. Bertin, N. Crespi, "Business process personalization through Web widgets", to appear in International Conference on Web Services, ICWS 2010, July 2010.