Widgets and Composition Mechanism for Service Creation by Ordinary Users

Nassim Laga, France Telecom
Emmanuel Bertin, France Telecom and Telecom Sud Paris
Roch Glitho, Concordia University and Telecom Sud Paris
Noel Crespi, Telecom Sud Paris

ABSTRACT

Significant research work has been conducted in software engineering to facilitate and speed up the process of service creation by experienced developers. Recently, however, service creation by ordinary users has attracted more and more attention as non-technical people have begun to play an active role in service life cycles, especially in a Web 2.0 context. In addition, service creation by ordinary users tackles the heterogeneity, the dynamicity, and the spontaneous nature of users needs. We show that current technologies are mainly inspired by previous approaches and architectures conceived for experienced developers, which means that they are not really adequate for service creation by ordinary users. This article proposes a novel service creation environment for ordinary users. It is made up of a new Widget abstraction layer that exposes the graphical user interface of services as reusable components, and relies on a two-step mechanism to compose these services at runtime. A proof of concept prototype has been built. The new abstraction layer offers interfaces that are much more user friendly than the current service creation tools. It also enables the different capabilities of a service to be seamlessly handled throughout its usage lifecycle.

INTRODUCTION

Users' needs are dynamic and spontaneous. This means that most of their needs cannot be anticipated long in advance but just arise in a given situation. In software engineering, two human activities are still necessary to address users' needs: specifying these needs (performed by users and/or a marketing team), and developing the corresponding services (performed by an IT team). To enable more spontaneous service creation, the current trend is to guide ordinary users in specifying their needs for themselves, with a given formalism, and then to automate the service development process. For example, service creation environments like Yahoo Pipes enable users to create their own services by assembling service building blocks.

However, this service creation activity, by its very nature, must be performed at design time and not at runtime. In other words, spontaneous needs, which mostly emerge at runtime when a user is utilizing his or her services, cannot be met immediately. Instead, the user needs to quit her/his working environment and go into a service creation environment, where she/he plays a programmer role. This approach lacks dynamicity. Moreover, it still requires basic computing skills from ordinary users.

In this article, we propose a novel service creation environment (SCE) for end users. It is based on service oriented architecture (SOA) principles [1]. SOA enables providers to wrap complex software features within well defined and reusable interfaces, which are made available to third parties. This method is usually considered to be the most appropriate means to handle a high level of heterogeneity. The SCE we propose enables ordinary users to spontaneously create services in order to respond to their spontaneous needs. For instance, a user who has loaded an enterprise directory service into his/her daily working environment (to search for contacts) and a telephony service (to make calls) can directly compose these two services inside this working environment to call various contacts found by the directory service.

The proposed environment comprises a widget-based abstraction layer and a two-step service composition mechanism. A widget is a reusable graphical user interface (GUI) that gives access to one or more functionalities of a service. Ordinary users can use the abstraction layer and rely on the first step to construct composite services automatically with little effort. They can even go further by customizing the service with additional actions during the second step. Our implementation is based on Web technologies (XHTML, Microformat,

0163-6804/12/$25.00 © 2012 IEEE

JavaScript, and CSS). The next section presents the requirements and discusses the related work. In the third section, we present the essential features of this novel service creation environment, followed by its implementation and validation. In the conclusions, we summarize the key features of the proposal, introduce briefly the experiments that were conducted with it in France Telecom laboratories, and discuss the lessons learned.

Requirements and Related Work

This section begins by defining the requirements. The two categories of related work we have identified, programming-based SCE and graphical SCE, are then reviewed.

Requirements

In order to simplify the service creation process to a level that targets ordinary users, we have defined a set of requirements. The first requirement is related to the spontaneous nature of users' needs. It is very important to enable spontaneous compositions that can respond to spontaneous needs. Unlike planned needs, which can be addressed following a design-time service creation process, and which involves two human activities (specifying the need and developing it) realized by two human entities (an end user [or marketing team] and an IT team), spontaneous needs should be addressed spontaneously at runtime, directly by users.

The second requirement is related to the composition environment. Unlike experienced developers who are familiar with integrated development environments (IDEs) and other specialized software development environments, ordinary users are most familiar with their daily working environment (e.g., web portals or email applications), and very few of them will know how to use IDEs. Therefore, it is this daily working environment that should be utilized for service creation, and any required tools must be fluidly integrated into this type of environment. The third requirement is in relation to the granularity of the services. These must be functionally meaningful to ordinary users. In other words, the granularity of the services must be sufficiently high to provide an added value and be understandable to ordinary users.

Programming-Based Service Creation Environment

Service creation environments have historically been based on programming, starting from assembly languages to the current development environments, such as the J2EE and .NET platforms. In addition to providing all the tools required for creating, testing, and deploying services, current environments are usually empowered with SOAP Based Web Services [2] and RESTFul Web Services [3], which facilitate and speed up the service creation process.

From the conceptual viewpoint, these technologies enable providers to wrap complex software features within standardized and reusable interfaces. These interfaces are made available to third parties. This could be achieved in a centralized way, through a common registry supported by the development environment; or in an ad hoc way, through a service provider website, for example.

Programming-based service creation environments, based on SOAP Based Web Services or RESTful Web services, are by definition oriented to developers needs. This makes them unresponsive to several of the requirements listed above. Their main limitation is that they are based on programming APIs, which ordinary users do not understand and therefore cannot use.

Several attempts to facilitate the service creation process based on SOAP-Based Web Services have nonetheless been made in the business area. A typical approach is to rely on the service composition concept via scripting languages such as Business Process Execution Language (BPEL) [4] and Service Logic Graphs (SLG) [5]. Service composition is then the action of combining two or more services, following a logic defined by means of a scripting language. This logic is developed through mappings between the outputs of some services and the inputs of others, combined with other operators such as loops, conditions, forks, joins, and so on.

Even though these scripting languages facilitate the creation process, they are still not appropriate for use by ordinary users. First, the operations still remain too complex. Second, the corresponding tools are usually integrated into traditional development environments (e.g., the J2EE and .NET platforms) instead of being integrated in an ordinary users' daily working environment. Finally, it has been shown [6] that session control and the event-driven nature of telecommunication services are difficult to master by experienced programmers, let alone ordinary users, using SOAP-based and RESTFul web services [6].

Twelve participants were involved in the study presented in [6]; all were acquainted with SOA and Session Initiation Protocol (SIP) and had at least two years of programming experience. This study also showed that the granularity of the basic services used in service composition is an important parameter of the intuitiveness of service composition. In other words, the higher the abstraction level, the more intuitive a service creation tool will be.

Graphical-Based Service Creation Environment

To speed up the service creation process even more, and to make it accessible to ordinary users, graphic tools (e.g., Eclipse BPEL Editor and Sedna [7]) have emerged, first in the business community and then more widely in the Web 2.0 community. Some of these are based on the XML languages mentioned in the previous section (e.g., BPEL). They represent the different operations and web service calls with black boxes that are connected to each other so that the data flow between services and their execution sequences can be defined.

While these tools definitely make the creation process easier and faster, they remain targeted to experienced developers. First of all, they are based on IDEs, which are difficult for

Programming-based service creation environments, based on SOAP Based Web Services or RESTful Web services, are by definition oriented to developers needs. This makes them unresponsive to several of the requirements listed above.

While these new tools have significantly enhanced the intuitiveness of Mashup creation, they are still too complicated for ordinary users, since the composition is performed manually based on the flowchart concept.

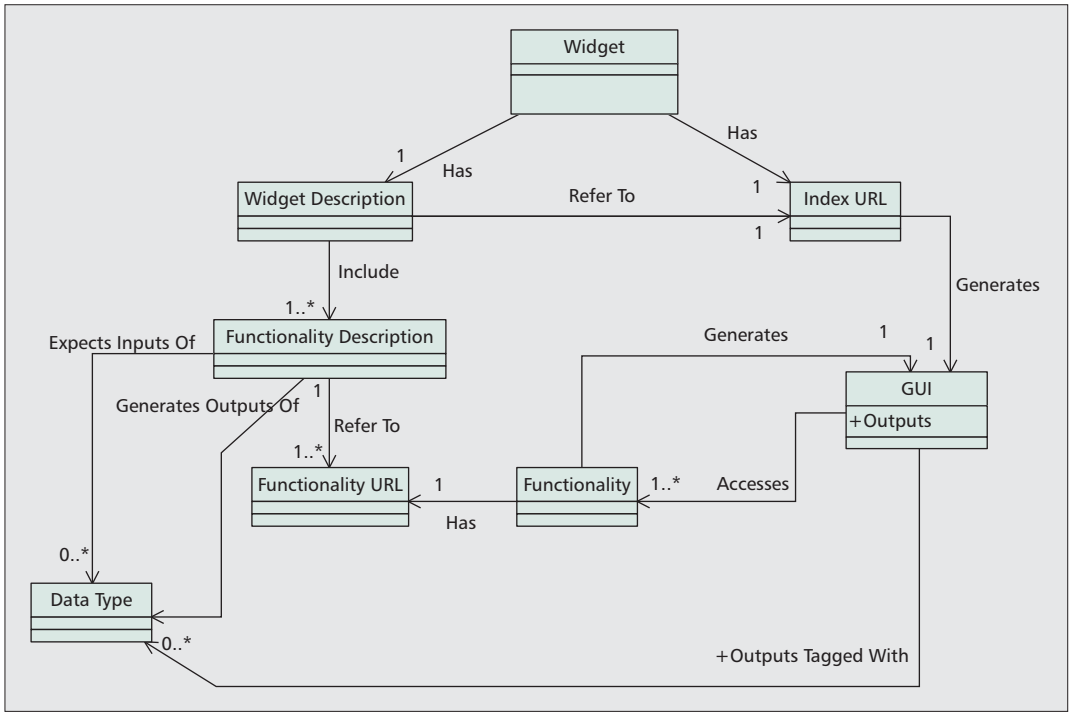

Figure 1. *Widget data model.*

ordinary users to use. Second, it is necessary to understand different computing concepts such as flowcharts, inputs, and outputs to create new services.

The successful adoption of Web 2.0 principles by ordinary users (collective intelligence, trusting the user as a co-developer) has encouraged the application of the same principles in the service creation field [8]. Concepts such as mashups, including social mashups [9] and enterprise mashups [10], have emerged. A mashup is basically a web page that composes and combines several services data sources. A typical example of a Mashup application is HousingMaps, which combines a house search service and Google Maps. Some Mashups have the peculiarity of combining the GUI of a service. For instance, all Mashups based on Google Maps reuse its GUI.

Existing mashup creation tools are mainly based on the flowchart concept. MARMITE [11] and Yahoo Pipes are examples of such an approach. Two important limitations should be highlighted in these tools. First, the flowchart basis is not intuitive enough for ordinary users. Concepts like mapping the outputs of some services and inputs of others, loops, conditions, and regular expressions (for adapting the outputs of some services with inputs of others) are not understandable by most ordinary users. The experimentation achieved in [11] confirms this assertion. Second, even though the mashup creation process is supported with an advanced GUI, accessible by ordinary users through their web browser, the UIs of the mashups that could be created are very basic, usually limited to a set of patterns like Map and RSS list.

Also based on the flowchart concepts, a new mashup creation approach has emerged, with a focus on the GUI aspects. IBM Mashup Center [12] and EZWEB [10] are examples of such an approach. These tools are based on the widget concept. The widgets are loaded into a widget environment, through which the user accesses, consumes, and optionally defines a flowchart-based composition of these widgets in order to personalize this environment according to his or her own needs and habits. While these new tools have significantly enhanced the intuitiveness of mashup creation, they are still too complicated for ordinary users, since the composition is performed manually based on the flowchart concept.

The Proposed Service Creation Environment

In this section we detail the widget abstraction layer, the two-step composition mechanism, and then we illustrate through a scenario how the different components interact.

The Widget Abstraction Layer

The widget-based abstraction layer is characterized by two components: the widget and the widget environment. The widget environment is in charge of providing a customizable user environment, where ordinary users not only access their preferred widgets, but also combine them according to their needs, processes, and habits.

The Widget — Figure 1 shows a simplified data model of a widget. Each widget has both an implementation and a description (contract). Each widget may provide one or several functionalities, which are described within the widget description file. Each functionality description contains an abstract description part and an implementation description part. The abstract description part describes the functionality, the

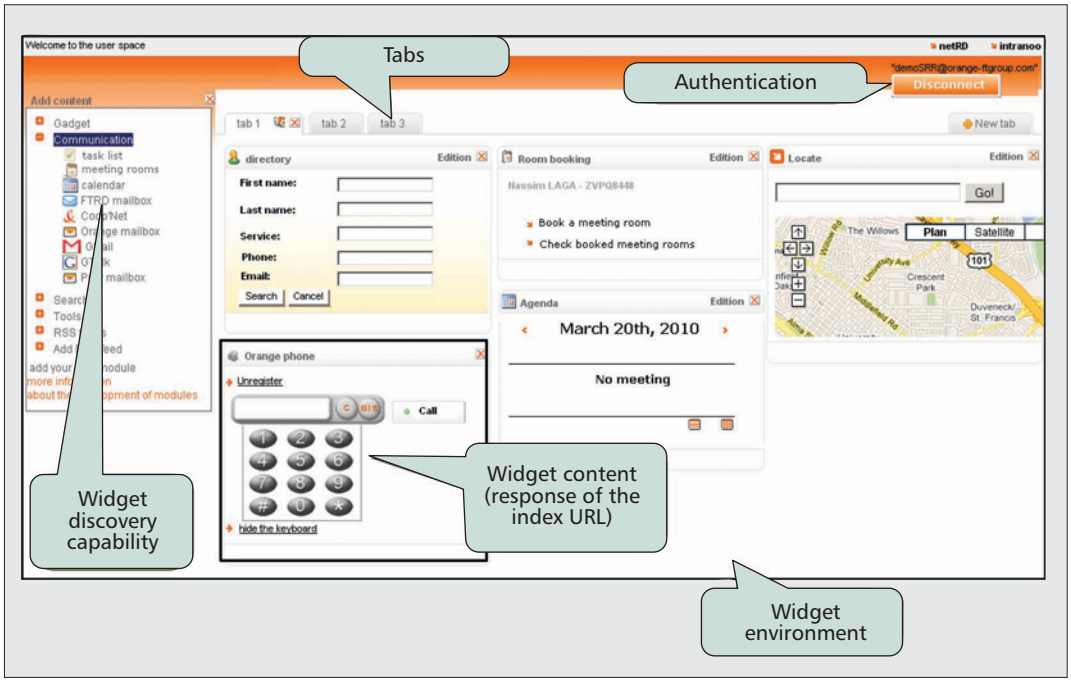

Figure 2. *Widget environment.*

inputs it requires, and the outputs it generates. The functionality, the inputs, and the outputs are described using a semantic dictionary. We decided to use Microformats [13] as the semantic dictionary. We have used for instance the hCard format to describe contact information and hCalendar to describe calendar events.

The description of the widget implementation refers to the index URL that enables the access to the widget's welcome screen; it also refers to the URL that gives access to each functionality. The GUI is an important element in widget design. Each functionality generates a GUI that includes the outputs tagged semantically using the same semantic tag included in the widget description.

Widget Environment — The widget environment is characterized by two capabilities: the widget discovery, and the widget aggregation capability. The widget discovery capability makes it possible for users to see which widgets are available. It can also include advanced widget search mechanisms, for example, based on semantic technologies, but this is out of the scope of this article. The selected widgets will be instantiated each time the user accesses the widget environment. Each user organizes their widget environment into one or several tabs, composed of one or several columns. Each widget instance is associated with the column of a given tab. The user can also move widgets from one column to another and from one tab to another. Figure 2 shows an example of the widget environment GUI.

The Two-Step Composition Mechanism

The two-step composition mechanism is performed by the widget combination component depicted in Fig. 3. It comprises four subcomponents: a communication manager, an application programming interface (API), a process manager, and a data structure that contains a list of functionalities that are present in the widget environment.

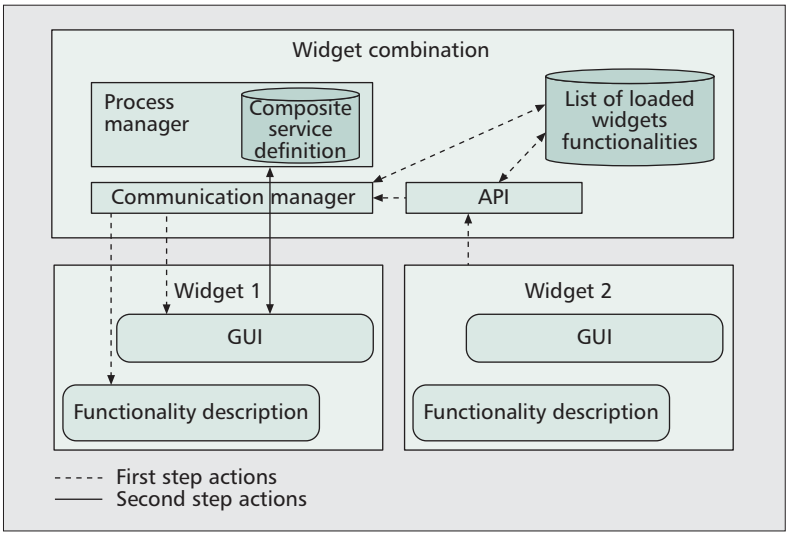

Figure 3. *The widget combination component.*

The first step of composition is realized automatically and involves the communication manager, the API, and the list of the loaded functionalities. The communication manager (detailed in [14]) is in charge of reading the functional description of each widget loaded by the user (which includes reading the URL of the functionality, the inputs expected by the functionality, and the type of outputs it will generate), and updating the list of the widgets' functionalities. In addition, at runtime, each widget can modify the functionalities it provides using the API component. This is important for

Each link is graphically presented to the user through a GUI element added to the GUI of the source Widget (the Widget that generates the data (output) needed as the inputs to launch the destination functionality).

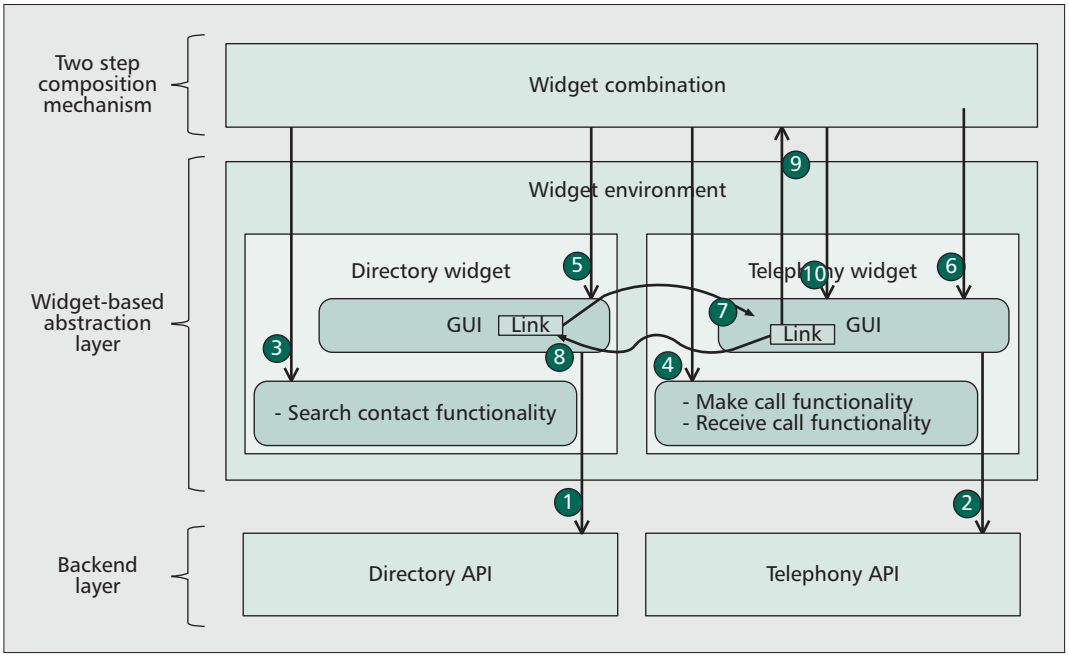

Figure 4. *High-level view of the proposed SCE.*

the widgets that manage different states and different capabilities in each state. For example, a telephony widget cannot initiate a new call when a user is already in communication with another user.

Based on the list of the loaded functionalities, the communication manager is also in charge of detecting the semantic matching between them, and creating (optional) links between the widgets. There are three types of semantic matching: exact matching, inclusion, and reverse inclusion. The matching is exact when one output type of a functionality of a source widget is exactly the same as an input type of a destination widget's functionality. It is an inclusion when the output type of a functionality of a source widget is a subelement of an input type of a functionality of a destination widget. Finally, it is a reverse inclusion when the input type of a destination widget's functionality is a subelement of an output type of a source widget's functionality.

Through this first step, the communication manager creates a composite service that connects all of the connectable widgets. This composite service is defined as a graph $G <N, L>$, where nodes N represent the list of widgets that are loaded in the widget environment, and links L represent the links between the different widgets. L is a sextuplet L (*Source-Widget, Output-Type, Destination-Widget, Destination-Functionality, Input-Type*, and *Link-Type*). Source-Widget — an element on N — is the source widget of the link. Output-Type is the type of the output that will be generated by a functionality of Source-Widget. Destination-Widget — an element of N — is the destination widget that provides the destination functionality (Destination-Functionality) of the link. Destination-Functionality is invoked when the link is executed. Input-Type is the input parameter expected by Destination-Functionality. Finally, Link-Type is the type of the link (automatic or at the user initiative).

Each link is graphically presented to the user through a GUI element added to the GUI of the source widget (the widget that generates the data [output] needed as the inputs to launch the destination functionality).

The second step of the composition mechanism mainly involves the process manager component (Fig. 3). The goal of this second step is to personalize a composite service that was created automatically in the first step. For this purpose, the process manager component maintains a composite service definition (Fig. 3), which is initialized to the graph G created in the first step. The process manager component also associates two GUI elements to each created link, which enable the user to delete the link or modify its type. By acting on the different links, the user customizes the composite service. This mechanism is detailed in [15].

End-to-End Scenario

To illustrate the end-to-end scenario, we consider the layout and indicators in Fig. 4. We assume that the user has loaded an enterprise directory widget and a telephony widget into his/her working environment. The enterprise directory widget provides a search functionality, based on either a phone number or a name. In both cases, it generates contact information that includes first name, last name, address, phone number, and so on. The telephony widget provides two functionalities: making calls and receiving calls. Both the enterprise directory and telephony widgets have access to their backend service (rows 1 and 2) via web services or any other technology. The communication between the widget and the service it renders is in charge of the widget developer.

When these two widgets are loaded into the widget environment, the widget combination

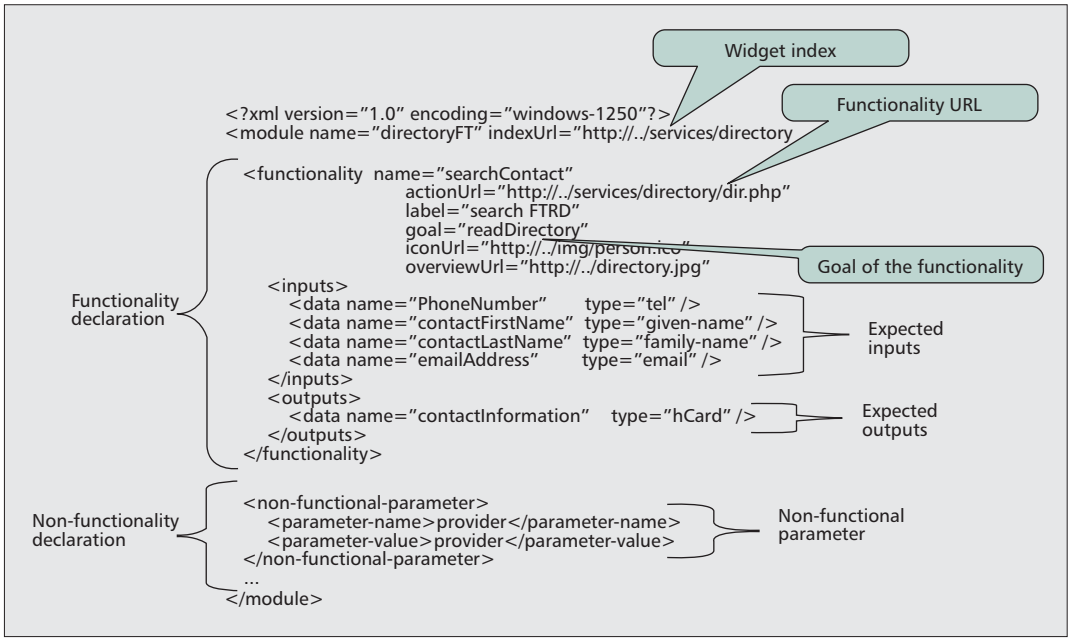

Figure 5. *Widget description file example.*

component starts the first step of the composition. First, it reads the functional description of each widget (rows 3 and 4). Based on the functional description, it detects semantic matching between the widgets. Next, it automatically creates links between them (rows 5 and 6). In our example, two semantic matchings will be detected. The first one is between the contact cards generated by the enterprise directory widget (which contains phone numbers) and the phone number expected as an input parameter by the telephony widget. The second matching is between the call session generated by the telephony widget, which contains phone numbers, and the phone number expected as an input parameter of the search functionality of the enterprise directory widget.

For each semantic matching detected, the widget combination component creates a link represented through a GUI element in the widget GUI (rows 5 and 6). In our example, a link will be created in the enterprise directory widget to the telephony widget, and another from the telephony widget to the enterprise directory widget. The former enables the user to launch the telephony widget from the enterprise directory widget (row 7), and the latter allows the user to search for caller information in the enterprise directory widget (row 8).

The second composition step involves the user. It enables a user to delete or automate the links that were created automatically in the first step. This is performed through the GUI element associated with each created link. When the user clicks on that GUI element, the widget combination component is notified, and the link will be modified/deleted (rows 9 and 10). In our example, the user may want to automate the link between the telephony widget and the enterprise directory widget so that the search functionality of the directory widget will be launched automatically for each incoming call in the telephony widget.

IMPLEMENTATION

In this section we provide the end-to-end implementation details of the widget-based SCE.

WIDGET ABSTRACTION LAYER

Widget Implementation — As detailed in the design section, each widget has a description part and an implementation part. In our case, the description part is realized through an XML file. An example is illustrated in Fig. 5. First we define the name of the widget and the index URL used to access the widget's welcome screen. Second, each functionality is defined through its URL, its goal, the input it expects, and the output it generates. Additional nonfunctional parameters can also be added.

The implementation of a widget is associated with its description file. Each widget must be accessible through the index URL specified in the description file. The implementation uses current web standards such as XHTML, JavaScript (JS), and CSS. Each functionality that is provided must be accessible through the URL as well, using the HTTP GET or HTTP POST methods, as specified in the description file. The input parameters are passed as GET or POST parameters in the HTTP request, using the parameter names specified in the description as the parameter names in the request. The outputs of each functionality are annotated in the GUI (the GUI is defined through the XHTML content rendered by the functionality URL), using the tag specified in the description file.

Widget Environment Implementation — Figure 6 shows the main components included in the widget environment implementation.

The GUI component is a web page that provides the front-end GUI. It enables the user to be authenticated (through the authentication component), and the personalization of his/her environments (by creating new tabs, loading new

The implementation of a Widget is associated with its description file. Each Widget must be accessible through the index URL specified in the description file. The implementation uses current Web standards such as XHTML, JavaScript (JS), and CSS.

The implementation of the Widget Combination component is distributed over the Widget Containers loaded on the user environment. This distribution enables the implementation to decouple the composition mechanisms from the Widget environment.

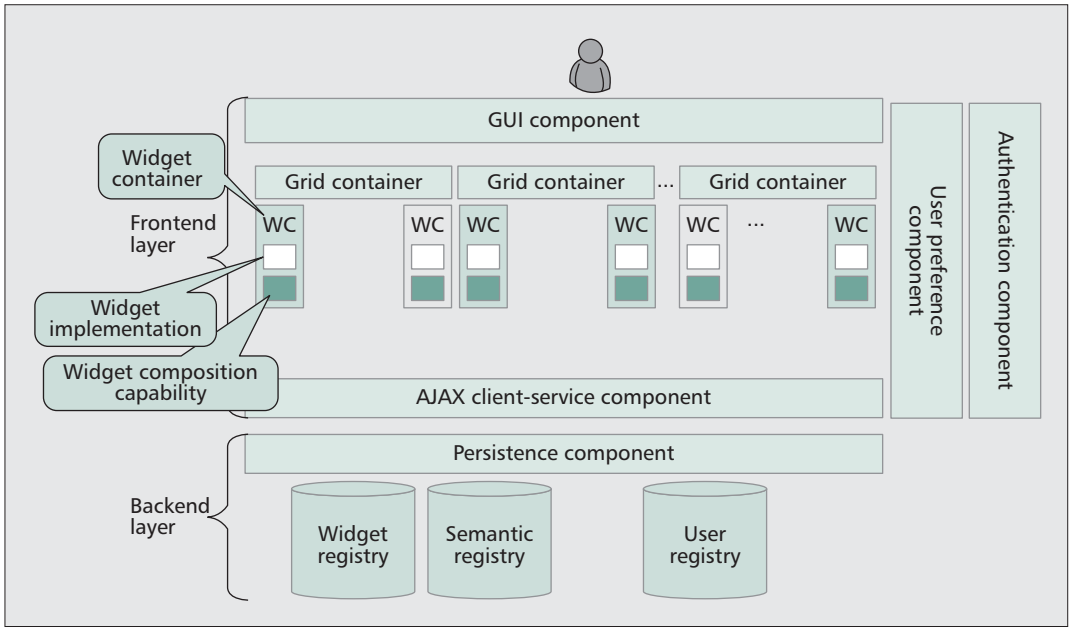

Figure 6. *Widget environment main components.*

widgets, etc.). Each tab includes a JavaScript object, named a grid container, which is a drag and drop area on the web page. It enables users to dynamically add, remove, and move widgets. Each widget is wrapped within a widget container (WC) component. The WC receives as input the widget description file URL. It extracts the index URL of the widget and invokes it. It parses the responses (XHTML-based) in order to detect special tags such as generated data and their type. The WC is in charge of managing the entire life cycle of the widgets. The WC is implemented as an extension to the widget object of the DOJO library.

The widget combination component is distributed over the WC components. It implements the two-step composition mechanism we have introduced. Its implementation is detailed in the next section.

The GUI component communicates with the back-end components (e.g., the user preference component) through the AJAX client-server component. It is a JS (JavaScript) API based on DOJO, which facilitates the interaction between the front-end components and the back-end (server side) components. It facilitates, for example, the retrieval of the list of existing widgets (to be displayed for the user on request), the description of a specific widget, and the user-related data (a list of widgets loaded on the environment, their locations in the environment, the list of tabs, etc.).

The back-end components are essentially the user preference component and the persistence component. The former is in charge of saving and loading all of the user-related parameters from the database, such as a user's preferred widgets, their place on the web page, and their configuration parameters. The persistence component provides access to the database content. This database contains essentially information about users and their credentials, widgets list, widget instances, tabs, composite services, and so on.

Two-Step Composition Mechanism Implementation

The implementation of the widget combination component is distributed over the WCs loaded on the user environment. This distribution enables the implementation to decouple the composition mechanisms from the widget environment in order to provide widget composition capability even for widget containers loaded on a third-party website.

When the widgets are loaded into the same environment, the different communication manager components, corresponding to each loaded widget, discover each other. The loaded widget then declares the capabilities it provides. This is performed either programmatically, by invoking the JavaScript function *Subscribe*, or automatically through the XML description file provided by the widget. The lists of capabilities of each communication manager are synchronized each time they are updated. This leads to the creation, by the widget container, of a GUI element in the second widget when a semantic matching is detected. This GUI element is a clickable icon representing the link between the two widgets. As illustrated in Fig. 7, the GUI element includes an icon to execute the link, an icon to delete it, and another icon to automate it. Thus, at runtime, the user can perform these actions to personalize the composite service definition.

A definition ($G < N, L >$) of the composite service created at this first step is managed by the process manager component. This definition is defined using the JSON format to facilitate and speed the processing at the web browser level. Based on this composite service, the user may invoke a widget capability from another one. This could be performed when the user clicks on a GUI element created earlier. This mechanism is illustrated in Fig. 7, in which the telephony widget invokes a search capability of

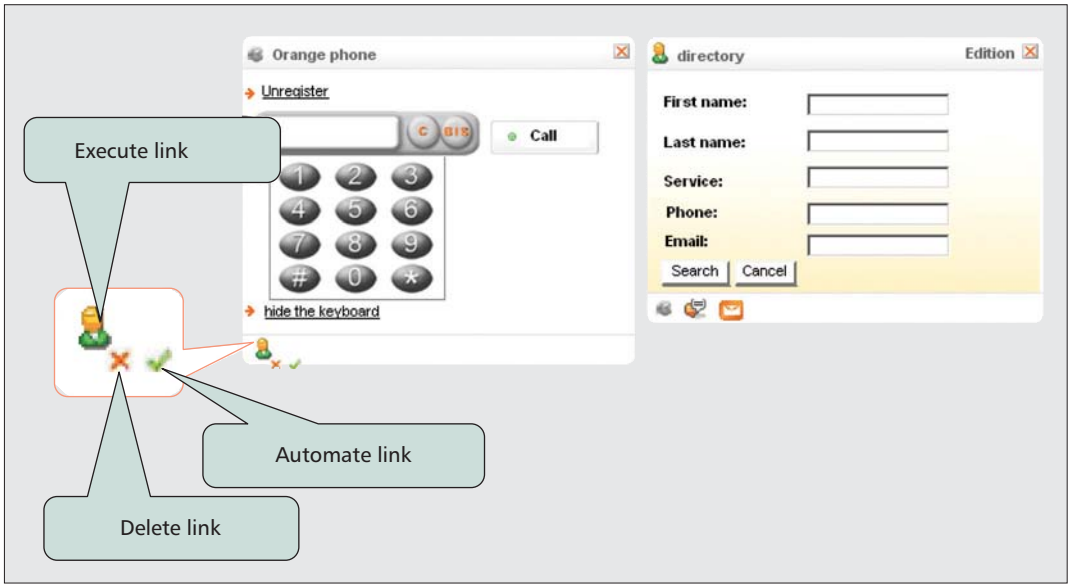

Figure 7. *Illustration of communicating Widgets.*

Our proposal is inline with the W3C standardization of Widgets. A first limitation of the current W3C standard is that it does not include the description of Widgets' functional capabilities. A second one is that it does not support Widget combination for security reasons.

the directory widget in order to display more information about a caller.

The second step of the widget composition mechanism we propose enables the user to personalize the composition created automatically based on semantic matching. This is achieved through two GUI elements added to each created link. As illustrated in Fig. 7, the first element enables the user to delete the link, and the second one lets the user automate it. In our implementation these elements appear only when the user passes through the link with the mouse.

Conclusions

In this article we have introduced a novel service creation environment for ordinary users. It relies on a new abstraction layer based on widgets. On top of this abstraction layer, we have introduced a two-step composition mechanism. The first step is automatic and relies on semantics. The second step enables manual personalization of the composite services created at the end of the first step.

Our proposal is in line with the World Wide Web Consortium (W3C) standardization of widgets. A first limitation of the current W3C standard is that it does not include the description of widgets' functional capabilities. A second one is that it does not support widget combination for security reasons. However, one must note that the second limitation should be addressed with HTML5 standardisation efforts.

An experiment of the proposed widget environment along with the composition mechanisms has been conducted in the laboratories of France Telecom. The results clearly show the usefulness of the environment in an enterprise context. Indeed, 89 percent of the 184 ordinary users involved in the experiment used the proposed platform; 80 percent of them created their own accounts and have personalized their environments by loading the appropriate widgets. Furthermore, 55 percent of them have used, or intend to use, the environment as the default starting page of their web browser. In addition, 72 percent of them considered the environment useful. The ordinary users' feedback concerning the widget composition capability was also positive and encouraging.

Important lessons have been learned from the design and implementation of this new SCE. First, by considering the service GUI as part of the reusable component, the created services are much more user friendly than existing tools. In the surveyed tools, the GUI is either created automatically, in which case it is very basic and not user friendly, or it is handled by the user him/herself, in which case ordinary users cannot create it.

The second lesson learned is related to the capability of seamlessly controlling sessions and handling asynchronous events when composing services. Indeed, since Widgets include a GUI, sessions and asynchronous events are directly managed by the GUI and consequently invisible to the composition tool. Session control and asynchronous events are transparent from the perspective of ordinary users. It should be noted that the capabilities of a widget may change during its life cycle within the widget environment. For example, a telephony widget, in its initial state, can make and receive calls; but the same widget does not have the same capabilities when a call has been established. The SCE we proposed in this article takes this issue into account.

The third lesson is the limitation of our proposal to services that have a GUI. This can be considered a limitation, since the composition scope does not include enablers such as authentication or charging. However, it is also an advantage when considering composition by ordinary users as it limits the composition scope to a level that is understandable by ordinary users.

Finally, the last lesson was in regards to the limitation of the Microformat semantic dictionary. It does not cover all types of the data that could be shared between services. For instance, Microformats does not include a specification

The results show clearly the usefulness of the environment in an enterprise context. Indeed, 89 percent of the 184 ordinary users involved in the experiment have used the proposed platform, 80 percent of them have created their own accounts and have personalised their environment by loading the appropriate Widgets.

for representing call information (caller phone number, called phone number, call state, etc.). We had to define our own format to overcome this issue.

REFERENCES

[1] T. Erl, *Service-Oriented Architecture: Concepts, Technology and Design*, Prentice Hall PTR, Aug. 2005.
[2] E. Newcomer, *Understanding Web Services — XML, WSDL, SOAP, and UDDI,* Addison-Wesley, 2002.
[3] L. Richardson and S. Ruby, *RESTful Web Services*, O' Reilly & Associates, May 2007.
[4] D. Jordan *et al.*, "Web Services Business Process Execution Language Version 2.0," http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html. OASIS Specification, 2006.
[5] B.S. Ku, "A Reuse-Driven Approach for Rapid Telephone Service Creation," *Proc. 3rd Int'l. Conf. Software Reuse: Advances in Software Reusability*, Nov. 1994, pp. 64–72.
[6] T. Moriya and J. Akahani, "Application Programming Gap Between Telecommunication and Internet," *IEEE Commun. Mag.*, vol. 48, no. 8, Aug. 2010, pp. 96–102.
[7] B. Wassermann *et al.*, "Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling," *Workflows for eScience — Scientific Workflows for Grids*, Dec. 2006.
[8] Y jung *et al.*, "Employing Collective Intelligence for User Driven Service Creation," *IEEE Commun. Mag.*, Jan. 2011.
[9] A. Maaradji *et al.*, "Social Composer: A Social-Aware Mashup Creation Environment," *ACM CSCW '10*, pp. 549–50.
[10] J. Soriano *et al.*, "Fostering Innovation in A Mashup-Oriented Enterprise 2.0 Collaboration Environment," *Sys. and Info. Sci. Notes, SIWN Int'l. Conf. Adaptive Business Sys.*, Chengdu, China, July, vol. 1, no. 1, pp. 62–69.
[11] J. Wong and J. I. Hong, "Making Mashups with Marmite: Towards End-User Programming for the Web," *Proc. SIGCHI Conf. Human Factors in Comp. Sys.*, New York, NY, pp 1435–44.
[12] IBM Mashup Center, http://www-01.ibm.com/software/info/mashup-center/, accessed on Thursday 16th, 2011.
[13] R., Khare, "Microformats: the Next (Small) Thing on the Semantic Web?," *IEEE Internet Computing*, vol. 10, no. 1, Jan.–Feb. 2006, pp. 68–75.
[14] N. Laga, E. Bertin, and N. Crespi, "Building a User Friendly Service Dashboard: Automatic and Non-intrusive Chaining between Widgets," *2009 World Conf. Services I*, 6–10 July 2009, pp. 484–91.
[15] N. Laga, E. Bertin, and N. Crespi, "Business Process Personalization Through Web Widgets," *IEEE Int'l. Conf. Web Services*, 2010, pp. 551–58.

BIOGRAPHIES

NASSIM LAGA holds a Ph.D. in computer science and telecommunications (UPMC Paris 6 University and Telecom Sud Paris, France), and M.Sc degrees in network engineering (UPMC Paris 6 University) and computer science (ESI Algiers, Algeria). He has been working for France Telecom since 2007. His research is focused on service composition, especially in the customer relationship management field. He has authored more than 10 peer-reviewed papers, and holds four patents on service composition mechanisms.

EMMANUEL BERTIN [M] holds a Ph.D. in computer science and telecommunications (UPMC Paris 6 University and Telecom Sud Paris, France), and an M.Sc. degree in telecom engineering (Telecom Bretagne graduate engineering school, France). He has been working for France Telecom since 1999 in the field of IP communication services, where he designed the first European IP Centrex offer. Since 2004, he has been involved in Orange's enterprise architecture program. He is currently senior service architect at Orange Labs and an adjunct professor at Telecom Sud Paris. His research interests include service architecture, service composition, and adaptation. He has authored more than 30 peer-reviewed papers and holds 16 patents in these areas.

ROCH GLITHO [SM] holds a Ph.D. (Tekn. Dr.) in tele-informatics (Royal Institute of Technology, Stockholm, Sweden), and M.Sc. degrees in business economics (University of Grenoble, France), pure mathematics (University Geneva, Switzerland), and computer science (University of Geneva). He is an associate professor of networking and telecommunications at the Concordia Institute of Information Systems Engineering (CIISE), Concordia University, Montreal, Canada where he holds a Canada Research Chair in End-User Service Engineering for Communication Networks. He is also an adjunct professor at several universities including Telecom Sud Paris. He worked in industry for almost a quarter of a century and has held several senior technical positions at LM Ericsson in Sweden and in Canada (e.g. expert, principal engineer, senior specialist). His industrial experience includes research, international standards' setting (e.g. contributions to ITU-T, ETSI, TMF, ANSI, TIA, and 3GPP), product management, project management, systems engineering and software/firmware design. He is a member of several editorial boards including IEEE Network and IEEE Communications Surveys and Tutorials. In the past he has served as an IEEE Communications Society distinguished lecturer, as Editor-In-Chief of IEEE Communications Magazine and as Editor-In-Chief of IEEE Communications Surveys & Tutorials. His research areas include architectures for end-users services, distributed systems, non-conventional networking, and networking technologies for emerging economies. In these areas, he has authored more than 100 peer-reviewed papers, more than 30 of which have been published in refereed journals. He also holds 24 patents in the aforementioned areas and has several pending applications.

NOËL CRESPI [SM], professor, holds a Master's from the Universities of Orsay and Canterbury, a diplome d'ingénieur from ENST-Telecom ParisTech, and a Ph.D .and a Habilitation from Paris VI University. From 1993 to 1995 he worked at CLIP, Bouygues Telecom and then joined France Telecom R&D in 1995 where he was involved in Intelligent Network paradigms for value-added services. For Orange, he led the Mobicarte prepaid service project to define, architecture, and deploy an infrastructure that now hosts more than 15 million mobile subscribers. He has played an active role in standardisation as a delegate in a number of committees and as an editor for CAMEL; he was appointed as the coordinator for France Telecom's activities for core network standardization, and later for all GSM/UMTS standards. In 1999, he joined Nortel Networks as Telephony Program manager for France and the Middle East-Africa. He was responsible for the evolution of the switching area, and led key programmes for the evolution of Nortel products. He has also worked for ETSI as an independent contractor. He joined the Institut Telecom in 2002 and is currently professor and programme director, leading the Network and Services Architecture laboratory. He coordinates the standardization activities for Institut Telecom at ETSI and 3GPP. He is also a visiting professor at the Asian Institute of Technology and is on the four-person Scientific Advisory Board of FTW, Austria. His current research interests are in service architectures, P2P service overlays, future Internet, and Web-NGN convergence. He is the author/co-author of more than 230 papers and contributions in standardization.