Developing front-end Web 2.0 technologies to access services, content and things in the future Internet

Juan Alfonso Lara\textsuperscript{a}, David Lizcano\textsuperscript{a}, María Aurora Martínez\textsuperscript{a}, Juan Pazos\textsuperscript{b}

\textsuperscript{a} Universidad a Distancia de Madrid, UDIMA, 28400, Collado Villalba, Madrid, Spain
\textsuperscript{b} Facultad de Informática, Universidad Politécnica de Madrid, 28660, Boadilla del Monte, Madrid, Spain

ABSTRACT

The future Internet is expected to be composed of a mesh of interoperable web services accessible from all over the web. This approach has not yet caught on since global user-service interaction is still an open issue. This paper states one vision with regard to next-generation front-end Web 2.0 technology that will enable integrated access to services, contents and things in the future Internet. In this paper, we illustrate how front-ends that wrap traditional services and resources can be tailored to the needs of end users, converting end users into prosumers (creators and consumers of service-based applications). To do this, we propose an architecture that end users without programming skills can use to create front-ends, consult catalogues of resources tailored to their needs, easily integrate and coordinate front-ends and create composite applications to orchestrate services in their back-end. The paper includes a case study illustrating that current user-centred web development tools are at a very early stage of evolution. We provide statistical data on how the proposed architecture improves these tools. This paper is based on research conducted by the Service Front End (SFE) Open Alliance initiative.

1. Introduction

Service-Oriented Architectures (SOA) have attracted a great deal of interest in the last few years. SOAs increase asset reuse, reduce integration expenses and improve the agility with which businesses respond to new demands [1].

Until now, however, mainstream development and research into SOA has focused mainly on middleware and scalability, service engineering and automating service composition using BPM (Business Process Management) technologies. Little or no attention has been paid to service front-ends, which we believe to be a fundamental part of SOAs [2]. Consequently, SOAs remain on a technical layer hidden to end users.

End users access tens of web portals every day to search for information, invoke services, make queries, etc. All of these are very time consuming. End users make use of web services and resources as rendered by service providers. Despite advances in SOA, Business-To-Consumer (B2C) solutions are confined to web applications developed by a professional programmer and offered to end users, even though there is generally a gap between the provided application and end-user needs.

Users in many Information and Communication Technology (ICT) areas have started using languages and tools (like spreadsheets) that do not require programming skills to develop their own software solutions. This approach is known as End-User Development (EUD). Users have tried to export these ideas to web applications development based on commercial mashup portals (like JackBe Presto, iGoogle, OpenKapow, etc.), visual languages for generating complex data sources in Rich Site Summary (RSS) format (like Yahoo! Pipes), etc. But end users have a problem with this technology: users that have no service-oriented programming or computing skills are unable to use the resources available on the Internet. This results in an epistemological problem: users are unable to learn to use, access and manage these resources to solve everyday problems in support of their routine work. This is the open problem that we intend to address in this paper.

The evolution of Web 2.0-based portals and user-friendly interfaces has led to a major advance in service usability. However, existing web-based service front-ends fall short of end-user expectations [3]. Applications and information portals are still based on monolithic, inflexible, non-context-aware, non-customizable and unfriendly user interfaces. Consequently, end users do not really benefit from the advantages of modularity, flexibility and composition promoted by service orientation [4]. Also service front-ends are not built using formal engineering methods; they are constructed ad hoc without tools that could speed up time to market.
In this paper, we present an architecture designed to enable users without programming skills to build composite web applications whose back-end is based on web services. To do this, the architecture wraps current web services using a front-end adapted to user needs and provides a visual composition model for users to orchestrate, invoke and tailor services to their individual needs. We use this next-generation service front-end architecture in a case study to examine the following Research Questions (RQ):

- **RQ1:** Are end users able to build a composite web application using today’s EUD tools and initiatives?
- **RQ2:** Does the proposed architecture increase the number of end users that manage to build a composite web solution?
- **RQ3:** From the qualitative viewpoint, do potential users find the proposed architecture usable, operational and stable?

The architecture is structured around several visual components or building blocks of varying levels of abstraction, ranging from web mashups to visual elements like forms, buttons, etc. The main component is a gadget. A gadget is a component that uses a visual interface to abstract web service or resource invocation. Gadgets can be visually created, tailored and interlinked by users to manage a web service or a set of web services that performs an atomic function through an interface tailored to user needs. A gadget might be a component that outputs a map location from an address input, an application for purchasing transport tickets or an email inbox filter. The architecture offers modules for creating gadgets and for integrating and intercommunicating gadgets with each other for use as a web application, for cataloguing and sharing, etc. Section 2 discusses the service front-end problem and the background of our proposal. Section 3 proposes an architecture enabling the creation of new ecosystems where all stakeholders will be able to collaboratively develop capabilities and innovate new operating procedures by mixing and integrating available services. Section 4 presents our implementation of the proposed architecture. Section 5 sets out the results of an empirical study carried out to validate this architecture, illustrating its success compared to other approaches. Finally, Section 6 discusses the results, and Section 7 explains the main conclusions of this research.

2. State of the art

The provision and consumption of information-intensive electronic services across corporate boundaries has attracted considerable interest over the past few years. For example, the web services technology protocol stack [1] was expected to act as efficient and agile "plumbing [⋯] for information systems to interact without human involvement" [5]. Following the design principles suggested by SOAs, web services provide a uniform, system-independent way for interlinking dispersed electronic services. While technology and standards are important for achieving the vision of a globally networked, agile service economy, it has been widely recognized today that they are not sufficient on their own [3]. Analyses of today’s cross-organizational service interconnections following the SOA paradigm have resulted in the identification of the following major weaknesses [6]:

- **Rigid and process-oriented composition.** SOAs' inherent potential has not yet been unleashed. Adherence to merely process-oriented design guidelines leads to rigid applications that cause huge reprogramming efforts in the event of changes. As in the 1970s, where the prevalence of spaghetti code-like software programming led to application unmanageability and unchangeability (the "software crisis" [7,8]), the application of inflexible service orchestration techniques (e.g., based on Business Process Execution Languages (BPEL)) prevents SOAs from being truly agile today. Not until the principles of structured programming and object-oriented programming were employed could the crisis be resolved. Today's SOA context calls for a novel design methodology that incorporates the principle of modularity, emphasizes structural rather than process-oriented organization and paves the way for a more declarative (as opposed to imperative) approach to service orchestration.
- **Non-interoperability.** A second major issue concerning today’s SOAs is service interoperability. Sometimes referred to as the corporate household problem, information objects, defined as service input or output messages, are based on highly proprietary specifications. The resulting semantic and syntactic heterogeneity causes significant mapping efforts when different services are to be interlinked. This often leads to errors and increased costs. In order to enhance the productivity of service-oriented computing, novel composition support tools are required that draw on metadata attached to service interfaces and help to automate parts of mapping tasks.
- **Limited retrievability.** Another roadblock on the way to a networked service economy in today’s Internet is the dearth of comprehensive, trustworthy and widely accepted service registries. In fact, users are dependent on a number of intermediaries that provide rich navigation, as well as improve transparency and thus fulfill institutional functionality. Future intermediaries should continuously gather data about registered services in order to provide potential users with detailed insights into an otherwise baffling market of electronic resources. Besides transparency, navigability is required to optimize the services that meet a user’s needs.
- **Mute and autistic service interfaces.** Technologies such as the above web services protocol stack aim at supporting the setup of loosely coupled application interactions especially in a professional context and assume users to be technically sophisticated. Web Service Description Language-based interfaces, for example, do not cater for a rich interaction between machines and human users, but rather focus on automated machine-to-machine interoperation. Also, services are seldom context aware; parameters specifying environmental conditions are not considered during service provision, thereby degrading the service quality that users experience. Finally, currently available service retrieval platforms, composition and utilization do not allow for the explicit knowledge sharing or implicit derivation of recommendations from user behaviour. Future platforms will have to account for such social aspects of service-oriented computing in order to facilitate productivity.

Many projects have tried to address these or a subset of these problems. For example, the European Union Seventh Framework (EU FP7) FAST project¹ aimed to provide an innovative visual programming environment to facilitate the development of next-generation composite user interfaces. Another EU FP7 project, OPEN² aimed to develop an environment enabling people to continue to perform their tasks on the move or using other interaction devices. The strategic Networked European Software and Services Initiative (NESSI) project, EzWeb³ aimed to develop an enriched enterprise mashup platform and key technologies for use in the construction of the front-end layer of a new generation SOA architecture. The European Union Sixth Framework (EU FP6) InContext project⁴ aimed at developing a novel scientific approach to the problem of enabling diverse individual knowledge workers at separate organizations to work in effective team collaboration

---

¹ FAST Project, http://www.fast-project.eu.
with one another. The Eureka CELTIC MyMobileWeb project\(^5\) was the open source reference implementation of a next-generation content and application adaptation platform for the mobile web. All these projects were part of the SFE Open Alliance. The SFE Open Alliance aims to integrate the results of relevant research and development projects in order to produce open specifications and an open source reference implementation of components for an envisioned Web 2.0 platform to access services, contents and things in the future Internet.

All these projects attest to there being major interest in building an architecture for using web services tailored to end-user needs. However, none has managed to meet the needs of end users throughout the entire service-based composite web applications development cycle.

Most of these projects and architectures share a common idea; tailor web services for users in order to overcome the obstacles preventing users without advanced programming skills from accessing SOAs. Keidt and Kemper [9] pioneered work in this respect, proposing a mechanism for tailoring web services to the context of use from which they are invoked. The problem with this proposal is that they merely tailor the services, but no support is provided for users to parameterize these services or create a new service-based application without having to modify the actual service source code, which calls for high-level programming knowledge.

Many other papers focused on how to offer users a platform from which to be able to visually create a web application from its front-end. Such platforms, commonly referred to as mashup platforms, have empowered users to develop small web applications. These platforms were what motivated this research. Major manufacturers, like Yahoo!, Google, Microsoft, Kapow or Apple, sold data or web component mashup development portals like Yahoo! Pipes and Dapper, iGoogle, PopFly, OpenKapow, RoboMaker, Apple Dashboards, etc. However, the experience gathered from many European projects suggests that users are still unable to develop composite applications using these portals, a point that the first research question addressed in this paper intends to investigate. Additionally, tool proliferation points to the need for a global platform, like MashArt [10], or a standard and common web components (gadgets) definition language [11], capable of using all their specialized components. The drawback with MashArt is that it does not empower users to create new web components, and its potential is confined to using and parameterizing existing services. On the other hand, the language defined by Agahee and Pautasso is able to standardize the metadata of the components in use, but does not account for the possibility of interlinking or orchestrating the components, meaning that such metadata are insufficient for a future where users can interlink web services.

Recent work addressing the above weaknesses is starting to focus on front-ends. Bianchini et al. [12,13], for example, focus on how to provide visual elements in a registry or catalogue for use for generating Web Application Programming Interfaces (APIs) based on selection patterns. They introduce only interface elements for creating front-ends. They do not empower users to attach an operational back-end to such elements to integrate interfaces with web services. Therefore, they do not solve the problem set out here.

On this ground, the design of a web service architecture that offers a front-end tailored to the needs of users whom it empowers to create their own composite solutions based on existing services is still an open problem. This problem now poses an even greater challenge than the goals pursued by the research reported above. The purpose of this paper is to define and validate this architecture.

The state of the art paints a picture where the research conducted so far has set forth a set of guidelines for addressing this challenge. These guidelines are the result of an a posteriori definition based on the developments reported by the above researchers and projects. These guidelines are listed below:

- **Empowering users to author and share their operating workspace and applications.** Traditional user-service interaction should be enhanced by facilitating the selection, creation, composition, customization, reuse and sharing of applications in a personalized operating environment [14].
- **Constructing ubiquitous and context-aware service front-ends.** New generation service front-ends should have the capability to detect, represent, manipulate, and use contextual information to seamlessly adapt to each situation, supporting human users in a more effective, personalized and consistent way [3]. Novel engineering tools and methods should be devised in order to support context-aware service front-ends. This guideline will drive the construction of novel service front-ends capable of using contextual information to influence their behaviour, thus supporting human users in a more effective and personalized way [15].
- **Capturing and exploiting user's knowledge in service front-ends.** This principle aims to exploit user domain knowledge and collective intelligence to improve service front-ends. End-user knowledge can be used to tag resources using light semantics, provide service interaction support, enrich contextual information (for example, by means of automatic user profiling) and infer new candidate processes to be later automated (on the back-end) [5].
- **Constructing a new generation of collaborative, user-centric and context-aware ICT systems.** Enterprise systems should incorporate advanced user-centric, context-aware front-ends to enable their employees and other stakeholders to exploit, and share their extensive domain expertise and their thorough business knowledge [4]. Employees, customers, developers and providers will collaborate to create and improve enterprise applications, sharing, reusing, changing and combining existing context-aware components (services, contents, things . . .) [16].

In response to this open problem, we present an architecture that helps end users to create composite web applications. It empowers end users without programming skills to use existing web services. This architecture solves the open problem and is also used to conduct an empirical study to investigate the research questions stated in Section 1.

### 3. High-level architecture proposed to achieve the Internet of services

In this section we propose a novel architecture for next-generation service front-ends. This architecture has been devised in accordance with the presented guidelines set forth in the state of the art. For the sake of clarity, we have separated the authoring and run time phases of the service front-end lifecycle (see Fig. 1).

Gadgets will be, as has been explained above, the main building blocks of this architecture. They are relatively simple self-contained front-end components focused on a single goal that can be grouped into workspaces.

Our proposed architecture for the authoring phase includes two main components:

1. A user-centric Integrated Development Environment (IDE) (Fig. 1(a)), which is a gadget authoring tool devoted to gadget design and creation. This is a visual tool that helps users with programming skills to create their own service front-end resources [17]. Using this user-oriented IDE, gadget authors will be able to visually design, reuse and share gadget screens, flows
These two tools will be supported by at least the following formalisms:

- A declarative authoring language for describing devices and modality-independent user interfaces. Traditional user interface development approaches are insufficient for supporting the new-generation service front-ends. In actual fact, traditional UI platforms and toolkits do not have the formalisms necessary to deal with context-aware service front-ends. For example, there are no declarative mechanisms for specifying how an interface should adapt according to different delivery contexts. Instead developers have to adapt UIs manually, using an ad-hoc and costly approach which does not promote reuse or standardization. We propose a layered approach to the development of service front-end UIs: device-independent abstract UI, device-dependent concrete UI and physical realization for specific devices. All layers can be represented in eXtensible Markup Language (XML) and embody a model of behaviour at a progressively finer level of detail.

- A standard format and information set (infoset) for describing gadget metadata (gadget template). This template is a machine-readable gadget description that should contain information about author names and affiliations, date, a human-readable description, pointers to the gadget source code and, most importantly, publish-subscribe metadata depicting what data items are published and consumed by gadgets (including their type, name, semantics, etc.). Finally, the template should contain context metadata about gadgets.

At run time we propose a workspace access layer (Fig. 1(c)) that is responsible for giving end users access to one or more workspaces. The layer will be in charge of rendering each user's workspace (and the gadgets it contains) as per the characteristics of the access device. To do this, we analyse the user context with respect to access device and use the mobility channel of the MyMobileWeb project that dynamically adapts the visual appearance of the workspace for optimal display on the target device. For this purpose, different displays are prepared and the access layer uses one or another depending on the device settings at run time. Devices are recognized thanks to the use of a Web 2.0 mobile device repository, called WURFL (Wireless Universal Resource File). The workspace access layer adapts the visual appearance of the workspace and the gadgets that it contains to the constraints of the target context.

This architecture does not yet exploit the remainder of the user context, such as preferences, habits, location, etc., which is a future line of research, focusing on the implementation of all the artefacts needed to support the execution of gadgets at run time, such as publish and subscribe communication mechanisms or gadget data and state persistence, providing a run time environment for gadget execution.

Additionally, there will be a set of horizontal modules specializing in different aspects that are common to the run time and authoring phases:

- A resource catalogue (Fig. 1(d)) containing all the metadata about the different building blocks in the architecture (gadgets, screens, flows, workspace, content delivery resources, application data resources, resource compositions, etc.).

- A context framework (Fig. 1(e)) implementing mechanisms to capture and exploit user context to adapt components to their ubiquitous needs.

- An identity management (Fig. 1(f)) for dealing with user sessions and identity.

- A knowledge framework (Fig. 1(g)) implementing mechanisms to capture and exploit user preferences and previous experience, to offer components adapted to their preferences.

4. Implementation of the proposed architecture

This section gives an overview of the low-level components necessary to implement the proposed architecture.

4.1. User-centric IDE or gadget authoring tool

We have created a gadget authoring tool (Fig. 1(a)) to create new gadgets. Gadgets developed using this web development tool can be used on any mashup platform, like, for example, Yahoo! Dapper, EzWeb, JackBe or OpenKapow, which acts both as a workspace editing tool and as access layer. Gadgets, referred to in the gadget authoring tool as screenflows, are assembled by
composing one or more resource representations called screens. Each screen is composed of a form and several back-end resources and/or authoring resources, such as for example, Amazon, Google, Yahoo!, eBay, Microsoft back-end resources, etc., that are offered by means of REST (REpresentational State Transfer)-like homogeneous interfaces [17]. Fig. 2 shows that the gadget authoring tool uses building blocks available in catalogues with differing degrees of abstraction, including screenflows, screens, forms or back-end resources. Each element is an aggregation of the elements lower down in the hierarchy. Also, back-end resources can be operators or wrapped services, composed of an API and invocation data. All building blocks are composed of a description, an interface (for example, an image, a HyperText Markup Language (HTML) or XML form, an interface described in XUL (XML-based User interface Language), etc.) that users can tailor using WYSIWYG (What You See Is What You Get) mechanisms as in other HTML editors, a precondition, a postcondition and the option of including a wrapped resource. A building block precondition and postcondition are composed of facts, and a fact is just a data item with which a meaning is associated. The preconditions and postconditions are used together with problem domain knowledge bases and a Rete algorithm to recommend valid dataflows and verify the validity of connections for users. The use of the preconditions, postconditions and problem domain ontologies are discussed at length in [18–20].

The gadget authoring tool is fed by a single identity manager. This manager checks the identity of all users and what contributions they make to the catalogue. This implements a federated mechanism for controlling each new published development and uses a context framework to guide the user based on the needs elicited when tools were used in the past.

6 Integrated Development Environment, http://demo.fast.morfeo-project.org, use the user account = guess, password = demo for a trial.
4.2. Workspace editing tool and run time access layer

We have also created a workspace editing tool enabling users without programming skills to create solutions to improve their routine work and satisfy their personal requirements. This tool also acts as a run time composite application access layer. Fig. 4 illustrates the workspace editing tool architecture. This architecture uses similar but even more abstract building blocks than the gadget authoring tool, including solutions, mashups, workspaces and gadgets. Again, components are composed from components lower down in the hierarchy.

The workspace editing tool offers workspace design and editing support based on catalogued gadgets (Fig. 1(b)) and run time support as a workspace access layer (Fig. 1(c)).

When the platform is acting as a development environment, it can be used to interlink gadgets. Thus, the execution of one gadget can generate useful data for the other components that are part of the mashup, leading to a dataflow among gadgets as shown in Fig. 5. Fig. 5 illustrates the building blocks that generate a dataflow on the left and elements that can act as an incoming dataflow on the right.

At run time, users can use the created workspace to manage user-defined dataflows, support the implicit invocations of web services and access any remote resources that are gadget data input sources, solving any resulting cross-platform problems, as illustrated in the screenshot in Fig. 6. Fig. 6 shows a Run time solution made up of a workspace composed of several gadgets to search a keyword in Amazon services, Web news, Flickr, Wikipedia, YouTube, etc. By creating the flows, the design environment uses the data semantics in the knowledge framework to visually recommend which outputs can be rerouted to the inputs of other components, leading to a valid dataflow.

The workspace editing tool uses a knowledge framework built into the environment. This framework helps users to share the workspace that they have built with other users to their advantage, subject to its parameterization to another problem. It also monitors and captures user behaviour at run time and design time. To do this, it creates behavioural rules that are used through the inference engine on the other platforms and are able to recommend new gadgets of interest for the workspace created by users or new components and background resources for enriching existing gadgets or creating new gadgets with the gadget authoring tool.

The environment is available for trial use.

---

This shared hierarchized catalogue meets the needs described. Thanks to this information, the workspace editing tool considers them into outputs with modified types and semantics. It also works for building in-house workspaces, as well as low-level components for building in-house gadgets. To do this, a template includes information on the supplier, gadget author, version, authors email, description, resources catalogue component icon, a social component description wiki URI (Universal Resource Identifier), platform visualization preferences (such as location, size, background colour, etc.), meta-information on the data handled for wiring this gadget with others, leading to a dataflow, and finally a link to the repository where the author has to publish future versions of the component located in a perpetual beta. We have designed special-purpose labels within an XML tree for all this meta-information. A proposal for formalizing this template has already been published by one of the authors of this research [21].

This mechanism sets up the interoperability of gadgets created by different manufacturers and tools. To do this, a programmer must set up or adapt a template in conformity with the syntax employed by the proposed architecture, which indicates gadget functionality, inputs and outputs, and their expected data types. Thanks to this information, the workspace editing tool considers the building block as an interoperable black box, provided that the component has been previously published in the resources catalogue. Publication validates the template and enables the workspace editing tool to use the external gadget as if it were a native building block.

### 4.3. Declarative authoring language and gadget template

We have created a gadget template to assure the interoperability of the gadgets implemented and used with other catalogue gadgets. To do this, a template includes information on a gadget author, business logic and input and output interface. Fig. 7 shows a specimen template for a gadget that generates two events for setting up information flows towards other gadgets and an input or slot to receive outputs from other components. The template includes information on the supplier, gadget author, version, authors email, description, resources catalogue component icon, a social component description wiki URI (Universal Resource Identifier), platform visualization preferences (such as location, size, background colour, etc.), meta-information on the data handled for wiring this gadget with others, leading to a dataflow, and finally a link to the repository where the author has to publish future versions of the component located in a perpetual beta. We have designed special-purpose labels within an XML tree for all this meta-information. A proposal for formalizing this template has already been published by one of the authors of this research [21].

This mechanism sets up the interoperability of gadgets created by different manufacturers and tools. To do this, a programmer must set up or adapt a template in conformity with the syntax employed by the proposed architecture, which indicates gadget functionality, inputs and outputs, and their expected data types. Thanks to this information, the workspace editing tool considers the building block as an interoperable black box, provided that the component has to be published in the architecture catalogue. Publication validates the template. The external component then becomes a building block available in the tool.

The structural hierarchy is federated and component subsets are grouped by creator and problem type. Each subset has a flat hierarchy composed of labelled resources. These labels form what is known as the folksonomy.

The federated hierarchy implies that the catalogue contains different publication levels or spheres, and software suppliers, companies or institutions are responsible for maintaining the catalogue and the catalogue components. Thanks to this federated division, resource maintenance and updating, it is possible to:

- limit and identify who is responsible for a resource through federation in order to identify liabilities in the event of failures and report weaknesses or bugs for repair, etc.
- maintain published resources in a perpetual beta so that the resource supplier can update resources or manage weaknesses, producing more and more advanced versions of resources, which will be notified to users that use these elements in particular developments.

As discussed in the future trends section, security is an open issue, as users using resources published by suppliers or other users should be empowered with techniques to gain information about the element developer and to establish relations of trust.

### 4.4. Architecture resource catalogue

We have developed a resource catalogue to enable the workspace editing tool and gadget authoring tool to cooperate. This shared hierarchized catalogue meets the needs described in Fig. 1(d). It is possible to publish both gadgets and entire workspaces, as well as low-level components for building in-house gadgets (forms, operators, wrapped services and web resources). Fig. 8 shows the results of a search of the catalogue elements. Each element includes a natural language description on the right of the screen, information on the input and output data used, expected types and semantics and how the component transforms them into outputs with modified types and semantics. It also includes a rating, the reputation of the publisher (entity/user) (managed by the identity manager), and contexts where it has been successfully used. Users can add labels for each component in order to establish a tag cloud for each component (guiding future users about its use) [22] and enabling the search and recommendation of components based on socially constructed folksonomies [23].

The catalogue is also useful for exploiting external resources, services and components. For this purpose, we have defined a standard template for wrapping each resource (detailed in Section 4.3). This template acts as a standard interface for converting any external item into an architecture component which can be used to create a gadget or mashup. After defining its meta-information, the component has to be published in the architecture catalogue. Publication validates the template. The external component then becomes a building block available in the tool.

As discussed in the previous section, security is an open issue, as users using resources published by suppliers or other users should be empowered with techniques to gain information about the element developer and to establish relations of trust.
that all the tools can solve. The tools actually include video tutorials illustrating how to deal with similar problems. They all have four components that solve the problem: a component for storing HTML and CSS (Cascading Style Sheets) copies of web pages and their content (such as, for example, Amazon Persistence Service), a screen scraping component (like the REST Yahoo!fetch URI), a component for comparing markup languages (like the REST DiffXML service offered by several providers) and a component for writing emails or SMS (like, for example, Google GMail Notifier).

Using a computer with Internet access, each user had to use all the tools to solve the problem. One of the authors of this paper was present while the users completed the experiment. The results of the experiment are shown in Table 2. Column N shows how many of the 100 users managed to find a valid solution that met the set requirements. It also shows the mean time taken (in minutes), the standard deviation and standard error (in minutes), the lower and upper bounds of the 95% confidence interval of the mean, the minimum and maximum recorded time. Table 2 reports the results of the other experiments similarly. It shows that less than 20% of the sample managed to achieve a satisfactory solution with any of the tools. The best results were for Apple Dashboards, but the figures are far from promising: only 18 out of 100 users knew how to use the tool properly and managed to solve a problem that, in theory, the tool is specially designed to solve, as shown in its video tutorials. The results for the other tools were even worse. Only 7% of Yahoo! Pipes users, 9% of iGoogle users, 10% of PopFly users and, finally, 16% of OpenKapow users managed to solve the problem. Note, additionally, that seven of the Yahoo! Pipes users were also successful using the other tools. The nine successful iGoogle users also managed to solve the problem with PopFly and performed best using OpenKapow and Apple Dashboards. A statistical study of the subsamples that were successful using each tool individually and all tools globally did not return any significant data to suggest that this success was due to the descriptive variables of the sample, such as gender, age, education, background knowledge or profession.

Finally, note that, if end users had been given a choice between the above range of EUD tools, less than 20% of end users could have solved the set problem, and most would have had to use OpenKapow or Apple Dashboards. It would take them on average from 45 to 68 min to solve the problem. Whether or not a user was successful using one specific tool has no effect on the study of the other tools, as the problem statement (not complexity) was altered due to the descriptive variables of the sample, such as gender, age, education, background knowledge or profession.

We chose this run-of-the-mill problem because it is a problem that all the tools can solve. The tools actually include video tutorials illustrating how to deal with similar problems. They all have four components that solve the problem: a component for storing HTML and CSS (Cascading Style Sheets) copies of web pages and containing the original status of the two web pages. These web pages do not publish RSS or notify content changes in any other way. The application should examine these web pages at a user-configurable time interval and check whether any changes have been made compared against the baseline. Moreover, the application will SMS message or email an outline of any changes made to the users mobile phone and email address. Both data should obviously be configurable at run time. The tools actually include video tutorials illustrating how to deal with similar problems. They all have four components that solve the problem: a component for storing HTML and CSS (Cascading Style Sheets) copies of web pages and their content (such as, for example, Amazon Persistence Service), a screen scraping component (like the REST Yahoo!fetch URI), a component for comparing markup languages (like the REST DiffXML service offered by several providers) and a component for writing emails or SMS (like, for example, Google GMail Notifier).

Using a computer with Internet access, each user had to use all the tools to solve the problem. One of the authors of this paper was present while the users completed the experiment. The results of the experiment are shown in Table 2. Column N shows how many of the 100 users managed to find a valid solution that met the set requirements. It also shows the mean time taken (in minutes), the standard deviation and standard error (in minutes), the lower and upper bounds of the 95% confidence interval of the mean, the minimum and maximum recorded time. Table 2 reports the results of the other experiments similarly. It shows that less than 20% of the sample managed to achieve a satisfactory solution with any of the tools. The best results were for Apple Dashboards, but the figures are far from promising: only 18 out of 100 users knew how to use the tool properly and managed to solve a problem that, in theory, the tool is specially designed to solve, as shown in its video tutorials. The results for the other tools were even worse. Only 7% of Yahoo! Pipes users, 9% of iGoogle users, 10% of PopFly users and, finally, 16% of OpenKapow users managed to solve the problem. Note, additionally, that seven of the Yahoo! Pipes users were also successful using the other tools. The nine successful iGoogle users also managed to solve the problem with PopFly and performed best using OpenKapow and Apple Dashboards. A statistical study of the subsamples that were successful using each tool individually and all tools globally did not return any significant data to suggest that this success was due to the descriptive variables of the sample, such as gender, age, education, background knowledge or profession.

Finally, note that, if end users had been given a choice between the above range of EUD tools, less than 20% of end users could have solved the set problem, and most would have had to use OpenKapow or Apple Dashboards. It would take them on average from 45 to 68 min to solve the problem. Whether or not a user was successful using one specific tool has no effect on the study of the other tools, as the problem statement (not complexity) was altered slightly for each tool to prevent any bias caused by users already being familiar with the problem and problem-solving mechanism. In any case, the number of components, component types and
Table 2
Results using current solutions and the proposed architecture.

<table>
<thead>
<tr>
<th>Tool</th>
<th>N</th>
<th>Mean of time</th>
<th>Std. dev.</th>
<th>Std. Error</th>
<th>95% lower b.</th>
<th>95% upper b.</th>
<th>Min.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yahoo!Pipes and Dapper</td>
<td>7</td>
<td>45.50</td>
<td>6.42</td>
<td>0.1469</td>
<td>40.60</td>
<td>50.54</td>
<td>30.00</td>
<td>67.00</td>
</tr>
<tr>
<td>iGoogle</td>
<td>9</td>
<td>63.20</td>
<td>8.79</td>
<td>0.2340</td>
<td>55.35</td>
<td>72.20</td>
<td>35.00</td>
<td>72.00</td>
</tr>
<tr>
<td>PopFly</td>
<td>10</td>
<td>56.40</td>
<td>11.30</td>
<td>0.1808</td>
<td>43.98</td>
<td>65.50</td>
<td>28.00</td>
<td>69.50</td>
</tr>
<tr>
<td>OpenKapow and RoboMaker</td>
<td>16</td>
<td>49.69</td>
<td>7.32</td>
<td>0.1072</td>
<td>40.50</td>
<td>58.50</td>
<td>39.00</td>
<td>71.00</td>
</tr>
<tr>
<td>Apple Dashboards</td>
<td>18</td>
<td>67.50</td>
<td>14.18</td>
<td>0.0579</td>
<td>45.50</td>
<td>70.25</td>
<td>40.00</td>
<td>87.00</td>
</tr>
<tr>
<td>Presented architecture</td>
<td>91</td>
<td>43.75</td>
<td>5.12</td>
<td>0.1961</td>
<td>37.06</td>
<td>51.22</td>
<td>27.00</td>
<td>61.30</td>
</tr>
</tbody>
</table>

Fig. 8. Resource catalogue screenshot.

component interlinking required to solve the problems were all equivalent to the original problem.

These data show that, despite their WYSIWYG mechanisms, the visual composition tools designed to empower users to exploit web services have not managed to remove the barriers that end users without programming skills come up against when developing software. After the objective statistical evaluation of the number of users that managed to solve the problem and the time that it took, we briefly surveyed the problems or weaknesses that they identified. Subjects expressed their opinions in responses to open, unstructured survey questions. The findings from the examination of the responses were as follows:

- Regarding Yahoo! tools, 95% of the sample stated that they had no way of interconnecting Yahoo!Dapper widgets with each other, whereas 76% missed the option for composing widgets based on finer-grained components. Eighty-two per cent of the sample highlighted that, apart from feeds and screen scraping-based information sources, Yahoo! Pipes failed to provide useful wrapped services for end users.

- Regarding Google, 85% of the sample stated that they were unable to establish a correct dataflow among widgets.

- Regarding PopFLy, 95% of the sample criticized the fact that they were unable to find the right elements in the catalogues and were not able to locate the right components for a particular problem.

- Regarding OpenKapow and RoboMaker (an auxiliary tool supporting OpenKapow), over 90% of the sample found that Kapow component linking and tailoring mechanisms were not handy (required programming knowledge), whereas 85% found the component search, location, parameterization and recommendation mechanisms (very similar to the ones proposed in the high-level architecture) to be very satisfactory.

- Regarding Apple Dashboards, 90% of the sample stated that the visual composition interface should not be confined merely to desktop dashboards available through Apple Dashboards. Additionally, 75% of the sample found it impossible to establish correct dataflow among the different components and gadgets.

At the end of the first phase, we undertook a second experiment, to address RQ2. The aim in this case was to demonstrate that the proposed architecture performs better than existing solutions. To do this, we used the components described in Section 4. The same sample was given a similar video tutorial to the other tools, this time on the workspace editing tool, the gadget authoring tool and the resources catalogue. Users were then set the same problem, and the results are shown in the bottom row of Table 2.

This time round, 91% of the sample solved the problem satisfactorily. They used the same technology, services and resources as for the other tools. The only difference is that the proposed architecture is used to support each tool used, and there is a joint context, identity and catalogued resource manager that exploits user capabilities to describe and locate the required building blocks based on recommendations and semantic searches.

We have also found that the required design and development time was reduced. This could, however, be due to the fact that subjects were already familiar with the kind of problem and part of the tool, for which reason we ignore this point.

The nine users that failed to solve the problem came up against the same obstacle: they did not realize that they had to compose based information sources, Yahoo! Pipes failed to provide useful other, whereas 76% missed the option for composing widgets based on finer-grained components. Eighty-two per cent of the sample highlighted that, apart from feeds and screen scraping-based information sources, Yahoo! Pipes failed to provide useful wrapped services for end users.

- Regarding Google, 85% of the sample stated that they were unable to establish a correct dataflow among widgets.

- Regarding OpenKapow and RoboMaker (an auxiliary tool supporting OpenKapow), over 90% of the sample found that Kapoor component linking and tailoring mechanisms were not handy (required programming knowledge), whereas 85% found the component search, location, parameterization and recommendation mechanisms (very similar to the ones proposed in the high-level architecture) to be very satisfactory.

- Regarding Apple Dashboards, 90% of the sample stated that the visual composition interface should not be confined merely to desktop dashboards available through Apple Dashboards. Additionally, 75% of the sample found it impossible to establish correct dataflow among the different components and gadgets.

At the end of the first phase, we undertook a second experiment, to address RQ2. The aim in this case was to demonstrate that the proposed architecture performs better than existing solutions. To do this, we used the components described in Section 4. The same sample was given a similar video tutorial to the other tools, this time on the workspace editing tool, the gadget authoring tool and the resources catalogue. Users were then set the same problem, and the results are shown in the bottom row of Table 2.

This time round, 91% of the sample solved the problem satisfactorily. They used the same technology, services and resources as for the other tools. The only difference is that the proposed architecture is used to support each tool used, and there is a joint context, identity and catalogued resource manager that exploits user capabilities to describe and locate the required building blocks based on recommendations and semantic searches.

We have also found that the required design and development time was reduced. This could, however, be due to the fact that subjects were already familiar with the kind of problem and part of the tool, for which reason we ignore this point.

The nine users that failed to solve the problem came up against the same obstacle: they did not realize that they had to compose based information sources, Yahoo! Pipes failed to provide useful other, whereas 76% missed the option for composing widgets based on finer-grained components. Eighty-two per cent of the sample highlighted that, apart from feeds and screen scraping-based information sources, Yahoo! Pipes failed to provide useful wrapped services for end users.

- Regarding Google, 85% of the sample stated that they were unable to establish a correct dataflow among widgets.

- Regarding OpenKapow and RoboMaker (an auxiliary tool supporting OpenKapow), over 90% of the sample found that Kapoor component linking and tailoring mechanisms were not handy (required programming knowledge), whereas 85% found the component search, location, parameterization and recommendation mechanisms (very similar to the ones proposed in the high-level architecture) to be very satisfactory.
needs of the sample, using the questionnaire shown in Table 3. Questions 5, 10, 11, 12 and 13 are backup questions, which have been rephrased to check that subjects are not answering without thinking and responses are consistent.

The results reveal that users have a positive impression of the proposed architecture: each item was evaluated on a 1-to-5 Likert scale, and each question scored a mean of at least four or more points. Table 4 shows the statistical feedback based on the questionnaire data. The usability of our architecture was rated as neutral with a positive tendency. We found that participants located the components that they needed and did not have to do a lot of learning to use the architecture. From a functionality perspective, we found that participants had some difficulties in finding the right element to use. Input and output definitions were self-explanatory. On the other hand, users found the procedure for publishing designed composite applications on a target platform easy. Regarding performance, our architecture was stable without any critical exception throughout the whole evaluation time frame.

There were other interesting findings. A five-minute video\(^8\) demonstrating the features of our iOS architecture was enough to teach users how to compose their applications. Participants also felt confident with the architectural terminology, which was again rated with 4 and 5 out of 5. This suggests that our architecture is easy to use. To confirm these results, we commissioned an external study. This study reached the same conclusions [24].

Finally, looking at the overall impression caused by the architecture, it is noteworthy that more than 70% of the participants rated the architecture as good or excellent and recognized the composition system's potential. A comparison of the impressions of the sample revealed an interesting point: end users had added to solve aspects that were not specified in the experiment as successful solutions to the problem and the low-level code developed for each application internally. We inspected the composition languages and low-level code developed for each application. We ran white-box tests to check that the applications met the set requirements and had similar computational efficiency, response times and robustness irrespective of the visual language used. In some cases, the compositions had dataflows that users had added to solve aspects that were not specified in the statement or were optional. Such flows led to longer response times, albeit not to problems of robustness.

### 6. Discussion of the experiment

The tests run address the stated RQs. Regarding RQ1, we have found that current EUD tools often do not empower end users without programming skills to develop a composite application on their own. The main strength of the proposed architecture is that it raises the success rate of end users developing composite applications enormously, leading to a positive response to RQ2.

Regarding RQ3, the usability, flexibility and performance of the proposed architecture appear to be good with respect to the set problem. It remains to run a series of tests on the architecture applied to real use cases in external projects. Such data will provide a better evaluation of Human–Computer Interaction (HCI) aspects of the proposed architecture. To do this, we intend to use the NEXOF-Ra project in order to demonstrate the results presented here.

Note that we analysed all the compositions that were classed in the experiment as successful solutions to the problem and the low-level code developed for each application internally. We inspected the composition languages and low-level code developed for each application. We ran white-box tests to check that the applications met the set requirements and had similar computational efficiency, response times and robustness irrespective of the visual language employed. In some cases, the compositions had dataflows that users had added to solve aspects that were not specified in the statement or were optional. Such flows led to longer response times, albeit not to problems of robustness.

### 7. Conclusion, limitations and future trends

In this paper, we have proposed a novel user-centric architecture for next-generation service front-ends. This architecture enables users to exploit their unique expertise to build applications that support their routine work in an open innovative creation process. This architecture helps users without programming skills to develop composite web applications based on front-end components, achieving much higher success rates than using existing tools. The original contributions of this research compared with the related work are as follows:

- The architecture empowers users to develop applications based on off-the-shelf and published components through a gradual visual composition process without having to use programming languages.
- The architecture is able to wrap any resource or web service for inclusion within the building blocks available to end users.

---

8 Video available from [http://www.youtube.com/watch?v=qFt2LBlxkwU](http://www.youtube.com/watch?v=qFt2LBlxkwU) (part 1) and [http://www.youtube.com/watch?v=doRhRnF8_1A](http://www.youtube.com/watch?v=doRhRnF8_1A) (part 2).
The applications adapt, at design and run time, to the users navigation context.

The architecture enables dataflows based on visual recommendations, supported by the semantics of the data handled by each component.

Applications including screen transitions can be created. Screen front-ends are considered as composable and interlinkable elements with dataflows that govern the transitions.

Application events can be handled as if they were data sources, abstracting and making event-oriented programming understandable for end users.

Such approaches will first of all unleash unprecedented potential with respect to the consumption of electronic services by different stakeholders: large enterprises may capitalize on faster application development (thereby reducing application development backlogs in ICT departments), a more agile system landscape, and the empowerment of their employees to contribute to the design of the applications they are intended to use. Small and medium-sized enterprises will be able to select and compose resources hosted by a wealth of third parties rather than paying for pre-determined, inflexible and potentially heavyweight solutions. Finally, private individuals will benefit from intuitive, unsophisticated ways to discover, remix and use the web services that they consider interesting and useful.

Besides the discussed benefits for different user groups, the novel user-centric approach will also aid the large-scale proliferation of what is often referred to as the Internet of Things (IoT): not until an agile service front-end architecture makes the information gathered from the multitude of dispersed sensors accessible and useable will the envisioned IoT materialize.

One feature of this research is that it is incremental. In our approach, end users use a visual language to compose components and connectors and build a composite application, but these components and connectors must be previously built and published in the composition tool. Therefore, the success and range of software solutions that can be created using the tool will be directly related to the range of available components and connectors. It is fairly easy to generate a lot of general-purpose components for widespread use that can be tailored to special-purpose user needs through parameterization. However, third parties will have to be encouraged to tailor their services, resources and back-ends to user manageable application domain-specific components. Many business areas have already experienced this phenomenon as a result of Web 2.0, and the trend is to offer easy-to-use data and interfaces for applications that often have little or nothing to do with the contexts of use for which they were originally created. It can be said that the more successful this type of composite applications are, the more likely businesses are to take an interest in feeding their catalogues with components to gain a market share. The more components there are, the more likely users are to make use of this type of tools. The problem is how to trigger this escalation of mutual interests.

Future work will focus on evolving several aspects of the proposed architecture.

We intend to set up an automatic tailoring mechanism capable of making components catalogued in existing end-user development tools compatible (Yahoo!Pipes, OpenKapow, iGoogle, and so on) with this catalogue of resources in order to rapidly increase the number of components available to architecture end users. Users that have access to more components will be able to build more varied and above all more complex solutions, notably improving the final product.

We aim to deliver the architecture as an open source service framework that builds on all the key guiding principles described in Section 3 and on the IoT vision. We expect this architecture to become a major hub for publishing, brokering, customizing and lastly consuming web resources on a global, cross-organizational scale, revolutionizing user-service interaction.

We are working on monitoring the behaviours of end users in order to make use of their experiences, previous developments, design time and run time habits, etc., by recording timestamps within a time series. We intend to run studies based on data mining to extract common behaviour patterns and improve the knowledge manager and recommendations received by users.

At present, the catalogue supports collaborative developments by several users and asynchronous communication channels support the comments on components published in the catalogue and their meta-information. As ends users have no software engineering knowledge, we plan to set up a peer-programming platform to enable several users to interact on the same development or receive on-line information about the software life cycle that they should enact for complex developments. To do this, we plan to exploit collaborative software development forge ideas.

The architecture exploits only a small part of the end-user context, specifically, their stored preferences and their architecture access device at either design or run time. We are working on the definition, use and management of complex contextual information, such as user location for use, especially at run time, in the created application.

We are also addressing catalogued component security and trust issues. To do this, we are exploiting the idea of SOA information technology governance. These policies and lessons learned have to be adapted to a field where any user can be a provider of components and partial or full solutions.


