A component- and connector-based approach for end-user composite web applications development

D. Lizcano [a,*], F. Alonso [b], J. Soriano [b], G. López [b]

[a] Department of Computer Science, Open University of Madrid, Spain
[b] School of Computer Science, Universidad Politécnica de Madrid, Spain

ABSTRACT

Enabling real end-user development is the next logical stage in the evolution of Internet-wide service-based applications. Successful composite applications rely on heavyweight service orchestra-tion technologies that raise the bar far above end-user skills. This weakness can be attributed to the fact that the composition model does not satisfy end-user needs rather than to the actual infrastructure technologies. In our opinion, the best way to overcome this weakness is to offer end-to-end composition from the user interface to service invocation, plus an understandable abstraction of building blocks and a visual composition technique empowering end users to develop their own applications. In this paper, we present a visual framework for end users, called FAST, which fulfils this objective. FAST implements a novel composition model designed to empower non-programmer end users to create and share their own self-service composite applications in a fully visual fashion. We projected the development envi-ronment implementing this model as part of the European FP7 FAST Project, which was used to validate the rationale behind our approach.

1. Introduction

The recent evolution of IT and the software business has significant implications for software products, development and use. Over the past few years, traditional software products, sales and licence fees have declined, whereas business value and revenues have shifted to SaaS-based services (Anon., 2006). Software as a service (SaaS) is a recognized approach that emerged from the traditional application service provider (ASP) delivery method. As a result, Internet services are becoming more important than product revenues and are also including traditional product terms and ideas (marketing, support, product concept, package and distribution, and so on) (Anderson, 2006). Since the year 2000, both the business-to-business (B2B) and business-to-customer (B2C) IT economies have become more based on web services and resources. This often leads to cost savings on purchases of traditional commercial software for performing particular functionalities, like office packages and desktop applications (Lizcano et al., 2008). Service-oriented architectures (SOAs) increase asset reuse, reduce integration expenses and improve the rate at which businesses can respond to new demands. The main idea is to enable enterprises and end users to create their own software solutions by composing and orchestrating heterogeneous Internet services, enabling real end-user development (Lieberman et al., 2006) within the future Internet of Services (IoS)(Anon., 2009).

End users should be able to create their own software solution that exactly meets their requirements within a very short development time by composing a solution from heterogeneous resources and their front-ends (Davenport, 2005).

There are key proposals offering DIY (Lizcano et al., 2008) guidance on evolving SOAs to meet end-user demands and requirements, like *iGoogle*[1] (http://www.google.com/ig), *Yahoo! Pipes and Yahoo! Dapper* (http://pipes.yahoo.com/), *OpenKapow* (http://kapowsoftware.com) or *EzWeb* (http://conwet.fi.upm.es/morfeo-project/ezweb_blog/?lng=en). Their aim is to get end users to appreciate the benefits of web services by fostering composition, loose coupling and reuse on the front-end layer, and moving towards a user-centred service as opposed to the traditional B2B conception (Scaffidi et al., 2005). However, existing solutions have several weaknesses: one of them is that they fail to provide a user-based

* Corresponding author. Tel.: +34 902 02 00 03.
E-mail addresses: david.lizcano@udima.es (D. Lizcano), falonso@fi.upm.es (F. Alonso), jsoriano@fi.upm.es (J. Soriano), glopez@fi.upm.es (G. López).

[1] As of 1 November 2013, iGoogle became Chrome Productivity Tools (Chrome PT). In this paper, we use the original term, because we ran our experiment prior to this date.

front-end to enable the user-centred composition of back-end services. This has been identified as a major shortcoming of the Future Internet (Anon., 2009). Solutions tend to be limited to the combination of just content rather than applications, overlooking end users and their participation. A high dependency on the underlying computing infrastructure is another limiting factor (Schneider, 1999). Also, changes in the original wrapped applications, the back-end services or the portal infrastructure may cause a user interface (UI) failure and disable user-service interaction because there is no formal user-centred framework.

Our aim, presented in this paper, was to design and implement a *visual mashup composition framework* capable of solving these problems (as shown in the experiment reported later). This framework implements a development model, composition techniques and a visual compositional language, all of which are centred on the end user. The development model is based on connectors and components. The composition techniques should help users to solve their problem by linking components using connectors following visual guidance that abstracts end users away from programming language concepts like variables or data types. The visual compositional language should enable the composition of user-centred front-ends capable of exploiting business services. This framework, which has some similarities and overlaps with general-purpose design-by-contract approaches (Lizcano et al., 2008), offers a visual composition process that aims to enable end users to create their own real-world solutions, fostering an *open innovation process* for software development (Lizcano et al., 2009). The source of the innovation is merely the result of using a resource (for example, a service) in a new, unexpected way for a novel purpose or combined with another apparently unrelated resource. This framework was based on the FAST initiative. FAST was a STREP project partially funded under the European Commission's 7th Framework Programme, as part of NESSI (http://www.nessi-europe.com), the Networked European Software and Services Initiative. FAST is an acronym for Fast and Advanced Storyboard Tool and is a tool for merging and filtering data provided by different web services. FAST proposes a top-down approach whereby users define a high-level screen flow by way of a storyboard and then use operators, data sources and wrapped resources to define what each screen will do.

Perhaps the best way to uncover the potential of these DIY proposals is to use a running example of an everyday problem with which we are all familiar in order to illustrate what type of applications could be built using a user-centred B2C approach. This example would serve as a benchmark for exploring the strengths and weaknesses of today's tools, and the potential of the presented framework and its internal design. The example is as follows:

> *A R&D worker has to make frequent national and international trips, scheduled using a web-based personal organizer shared by all research group members. All face-to-face meetings are posted on this personal organizer, specifying the meeting date and time, venue and agenda. The researcher usually travels with two travel agencies, one specializing in high-speed trains and the other in long-distance flights, and both manage all the travel and accommodation options at a full range of hotels. The researcher adds a new trip to his personal organizer. He wants to look up and locate the meeting venue on a map. He then accesses the travel agency services and checks what travel options they offer, as well as their price. He usually compares the two options and chooses one agency or the other depending on the travel options, length of stay and price. If the trip is to last longer than a day, the researcher searches hotels near to the meeting venue and checks the prices per room and night offered by the travel agencies. It is the researcher's job to calculate how much the travel and chosen accommodation will cost, check that there is enough money available for the trip and deduct it from his personal budget. Then the researcher makes the bookings one by one. Finally, the researcher checks the Internet for information about his destination, demographics, weather forecast, etc.*

Although the researcher has access to a number of software solutions to perform this routine task (project agenda, personal organizer, travel agency services, department cash flow program, etc.), he has to use each service offering distributed and heterogeneous information separately. If there were an EUD tool that end users could use to create an architecture integrating a personal organizer, map, transport search service, hotel booking service, payment gateway and travel query service, i.e., a situational application to satisfy end-user needs, composed of an information system integrating all the above services, then the system would have a single entry-point for execution: the personal organizer and appointments (date, time and place). It would then be possible, based on a appointment recorded in the personal organizer, to synchronize all the services, searching for transport, hotels, prices, on-line payment, etc., all based on a visually rendered user-comprehensible, unified dataflow. Fig. 1 shows an example of one such application, built by an end user using the framework proposed in this article.

The remainder of the paper is structured as follows. Section 2 presents the related work and background of our research. This related work was the source of the specific requirements for generating the proposed composition model. The following sections describe the main parts of the proposed composition model: Section 3 deals with the component model, Section 4 with the composition technique, and Section 5 with the composition languages. Section 6 details the execution language used to deploy the application created by the end user for execution. Section 7 then presents the results of a study that we conducted to test the validity of our composition model. Finally, Section 8 discusses the conclusions and the limitations of the proposed approach and briefly outlines future work.

2. Related work

Companies are beginning to focus on people as the entry point to SOA, data sources, disperse resources and web services consumption (Lizcano et al., 2008). Thus they need a means to bridge the gap between people and services, which is when they come up against the traditional shortcomings of composite applications.

2.1. DIY: the web as an ecosystem of user-centred resources

A number of web user-centred composite application frameworks are beginning to proliferate. Worthy of note are IBM's solution, named SOA for people (Anon., 2008), and SAP's proposal, called SOA-People (http://www.soapeople.com). They focus on a portal framework acting as a SOA front-end to maximize people's productivity and collaboration. The increasing interest in this approach is indicative of the current importance of user-centred SOA in the business world. However, existing approaches focus on employing particular Web 2.0-based technologies to deliver a front-end to SOA rather than lending attention to the composition process and component modelling.

Other companies are focusing on a different approach. They highlight the mashup, Web 2.0 and end-user development ideas, but overlook the exploitation of real back-end business logic. The potential of these tools like *iGoogle*, *Yahoo! Pipes*, *OpenKapow*, *Apple Dashboards*, *PopFly*, *Marmite* (Wong and Hong, 2007) and *AMICO* (Obrenovic and Gasevic, 2009) is very promising. The construction of EUD web applications is one of the aims of today's DIY applications. However, as explained below and demonstrated by our experiment, non-programmer end users will find it hard to build an application like this unassisted, even if they do have access to

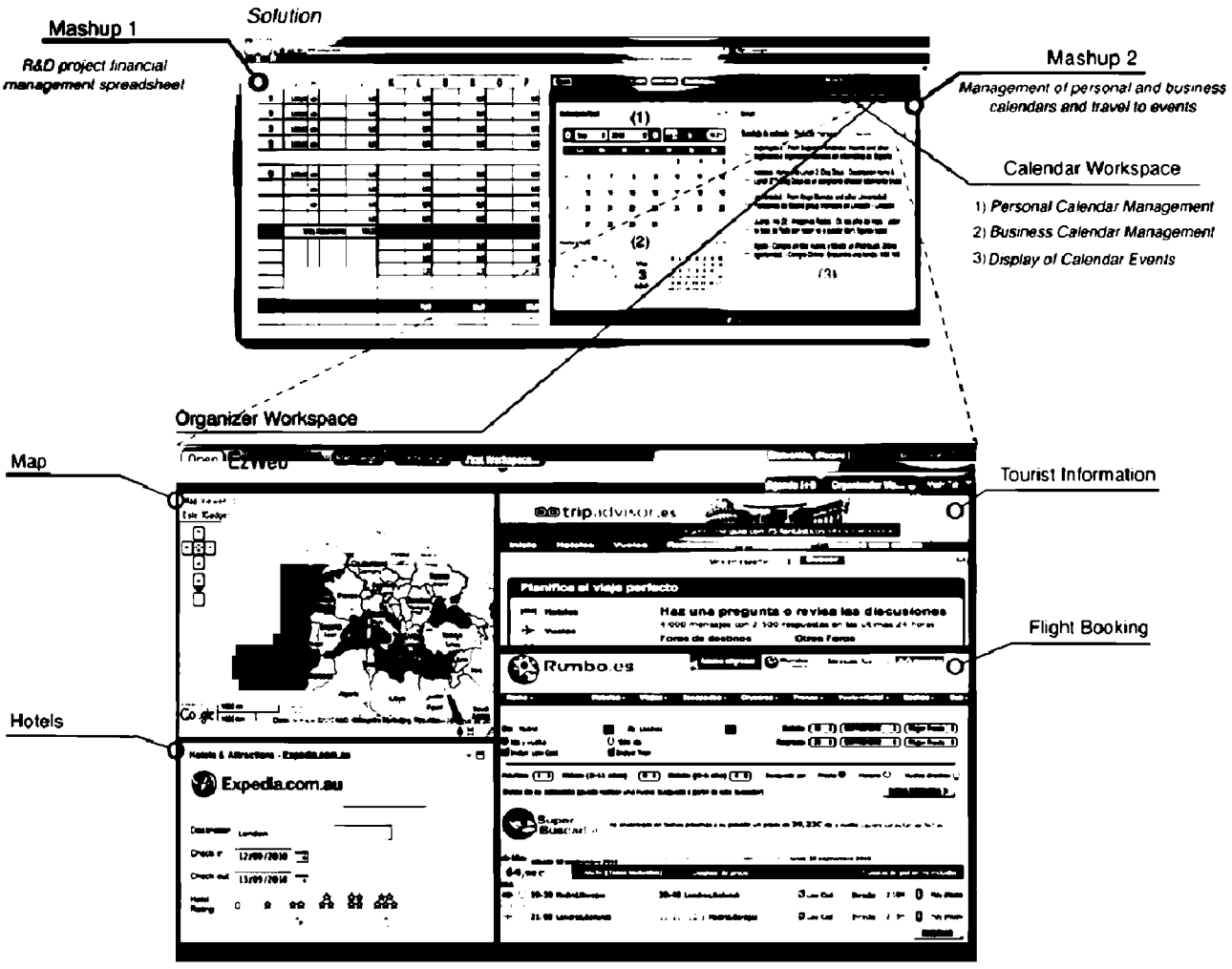

Fig. 1. Example of a real EUD web application.

Table 1
Comparison of today's most successful EUD tools.

EUD tool	Main objective	Event model	Dataflow support	Taxonomy of components	Catalogue	Actions required	Open source	Design strategy
Yahoo! Pipes and Dapper	Create complex RSS feeds	JavaScript event handler	HTTP headers, MIME contents	Operators, RSS readers and HTML visualizers	Based on taxonomies and keywords	XSD-based parameteriza-tion	No	Bottom-up
iGoogle	Create a visual mashup of separate widgets	Actions on JMI based on MOM	Pub/sub with XML contents	Visible or invisible widgets	Based on taxonomies and keywords	XSD-based parameteriza-tion	No	Top-down
PopFly	Create an office suite mashup	JavaScript event handler	Formal pre/postconditions	Screens, functions and arguments	Based on taxonomies and keywords	Knowledge of recursive calls	No	Bottom-up
OpenKapow	Create a complex portal from screen scraping	Action triggers over SOAP	Triggers and handlers of JavaScript	Screens, robots operating on a screen, and concatenators	Based on taxonomies and keywords	Activity diagrams with pseudocode control flow	No	Top-down
Apple Dashboards	Create a console command pipeline on a file	Functional programming-based RPC	Command line actions and plain text files	Functions, nodes and directed arcs	Based on taxonomies and keywords	Prototypes to be command line reprogrammed	No	Bottom-up
Proposed framework	Create visual mashup of data provided from web services	CRUD based on REST	Visual annotated data types to fulfil pre/post conditions	Screen flows, screens, forms, resources (operators, web services, etc.) conditions and connectors	Based on folksonomies, keywords, rec-ommendations and SEO	Drag-and-drop components guided by visual support	Yes	Top-down

the necessary components. To give a better idea of these limitations, Table 1 compares the major features of each tool. This table also summarizes the features of the framework proposed in this article, which are detailed in the following sections. This should give a general and precise idea of the framework for comparison with the other analysed EUD tools.

We analyse the features listed in Table 1 with reference to the above problem statement and highlight the difficulties that end users encounter using each tool to solve the problem:

- Yahoo! tools offer very good support for managing RSS feeds. In the example, the user would have to create an operator to convert his personal organizer into a properly labelled RSS feed where each personal organizer event would be an RSS item. The part of the statement regarding the invocation of web service searches is the most troublesome. The user would have to create an operator to iterate through and decompose items and then send the right part of the item to a web service with a Yahoo! Pipes API in a HTTP request header (for example, the destination should be sent separately to the Google Maps API, and the date and destination to the flight search engine, etc.). When a data item is useful for invoking more than one service, the user will have to use an additional operator to replicate this data item. The user would have to enact the process bottom up, which would be quite demanding because he would have to be familiar with XML and know something about MIME codes and the use of variables (string, integers, etc.).
- iGoogle provides a simple mechanism for creating a mashup of several widgets, like a map, a flight search engine, a travel information search engine, etc. The problem is that the user will have to enter the input data for each widget manually at run time, as the platform regards the widgets as separate elements which are not easy to link at design time. In order to connect them up to form an integrated information system, the user would have to create a publication/subscription channel for each data type with an element acting as message-oriented middleware (MOM). This element has to be parameterized, the form of the message has to be defined (in JMI format) and a Java API has to be managed. There are very comprehensive tutorials, but an aptitude for object-oriented programming and an understanding of this programming paradigm will be required.
- PopFly is useful for creating a visual mashup of personal organizer support tools, and will offer compatibility with Outlook and ISS management systems. The problem is that it offers a functional abstraction for services invocation and the user will have to have an aptitude for recursive calls. For example, if he decides to search for flights to a destination offered by more than one company, he will have to program a flight search function (for which there is plenty of documentation) and prepare a time variable to be incorporated into the calls to save the best price found to date and the name of the company offering this price. Additionally, the user will have to check manually (or by trial and error) that the invocations satisfy a set of preconditions defined in pseudocode. To do this, the user will require an understanding of complex data types (vectors, hash tables, DTDs, etc.).
- OpenKapow is based on screen scraping and ports part of the target code offered by the portal in HTML div tags. The user will have to substitute any server-side code (php, servlets, etc.) used by the service for pseudocode in the form of an activity or flow diagram. In order to send the destination stored in his personal organizer to several web portals (map, flight, hotel and information search engines), for example, the user will have to codify a dataflow between what OpenKapow terms "robots" and program those robots: he will have to access data from forms with PHP-like code and inspect and use the names of each field, understand their data types, etc. The tool offers support for these tasks, but this task is likely to be unintelligible for a non-programmer user.
- Apple Dashboards uses operating system tools, like document viewers, agenda, contacts and shell commands. In order to solve the problem, the user will have to access his personal organizer appointment data and prepare the invocation using curl in order to send HTTP requests to remote resources and display the map or invoke a service. This command will require knowledge of HTTP verbs and the parameters for preparing headers, as well as the implementation of control flows based on functional diagram-like charts. This can be very hard for non-programmers to do insofar as their needs are completely unrelated to the examples explained in the tool tutorials.

The main problem with the above tools is that they do not have a compositional model suited for end users (Assmann, 2003). Their weak points are that they do not provide help for discovering composable services or user-centred mechanisms for invoking services. Also service orchestration is troublesome, and there are user interface and presentation logic problems. The framework presented here uses Web 2.0- and EUD-based ideas to exploit the strengths of existing tools and solve the above problems and weaknesses in order to provide a satisfactory response to problems like the one stated in the example. The strength of this framework is that it is composed, as already mentioned, of a component- and connector-based development model, port type-driven composition techniques that guide users through the selection, definition and organization of components and connectors, and a visual language that is useful for solving the problem. In the following sections, we detail work related with each of these three parts.

2.2. User-centred component models

Building block models and their relationships (Myers, 1990) have been successfully used for many different purposes (e.g., programming, user interaction and visualization) (Pautasso, 2004). They attempt to provide an effective, graphical, non-linear representation that has been successfully applied to modelling (e.g., UML), parallel computing, laboratory simulation, image processing, workflow description, hypertext design, and even object-oriented programming. We regard software composition as a potentially good application domain for a graph-based, visual notation. Instead of focusing on typical composition issues like how the "spatial" architecture of a software system can be specified in terms of components and connectors, we describe how services should be composed in "time" (Govindaraju et al., 2003). Apart from describing the dataflow structure of the interaction between different services, we have also included a separate description of their control flow dependencies in the component model, an idea already reported elsewhere (Pautasso and Alonso, 2003; Fukunaga et al., 1993).

2.3. User-centred composition techniques

As regards data-driven composition techniques, we have analysed existing tools and looked at how end users go about solving a problem. There are many studies analysing how users expect to use and build their own applications, but there are no studies investigating which composition technique most closely emulates how non-programmer users think.

There are not many composition techniques that do the job that we consider to be necessary in the EUD field. Spreadsheet formula techniques for applying off-the-shelf functions by entering cells with the right input types are perhaps the closest example. However, as far as we know, there are no techniques capable of generating the compositional process that an end user should

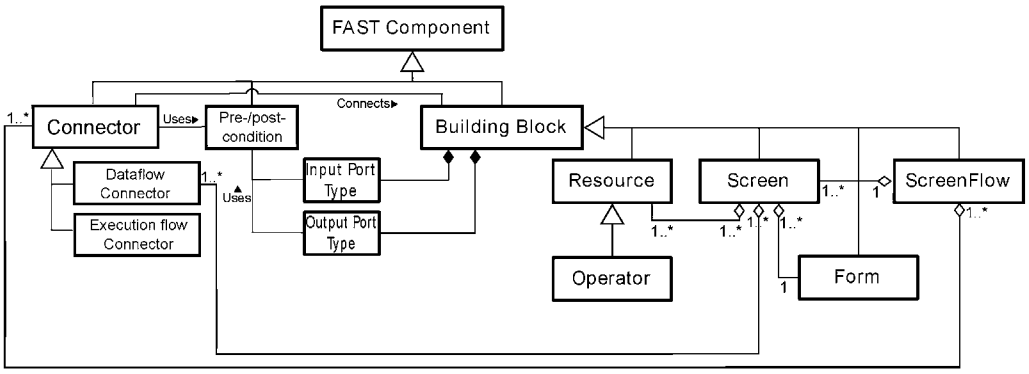

Fig. 2. FAST component model.

enact to build a web application from requirements by establishing dataflows among components published in a catalogue.

As users do not know how to program a solution, the development techniques have to be based on existing elements, which users will see as black boxes that serve a particular purpose. Additionally, users will be able to state their objective but will not know how to achieve their goal so they will not be able to use imperative techniques to compose these elements. Therefore, users will have to use a declarative technique detailing what they expect to achieve and the screen and dataflow that they expect to take place. This is equivalent to what users would do if they had to access each service, resource or portal.

2.4. User-centred visual composition languages

As far as visual languages are concerned, many graphical formalisms have in the past also been developed in this area. Some contributions that have been applied to workflow modelling, such as state charts and Petri Nets, are formalisms that have a natural visual representation. This gives users a good overview of the partial order of services invocation.

A limitation of a control flow-based visual language applied to service composition, however, is that there is no visual notation for specifying adaptations between mismatching service interfaces (Aragao and Fernandes, 2003). We have taken a design-by-contract approach whereby input and output port types are used to drive composition based on the target and source data. This solution overcomes this weakness (Lizcano et al., 2008).

In conclusion, the proposed FAST framework draws on previous research to create a compositional model based on the above three elements in order to help non-programmer users to build their own composite web applications. In the following, we present the component model, composition techniques and languages formulated in our research.

3. FAST component model

The FAST component model defines the main elements that users use to develop their solutions: building blocks, pre/postconditions which building blocks use to interact with each other, and connectors that are used to create either a dataflow among components or an execution flow among visual elements subject to their pre/postconditions. Fig. 2 shows the FAST components and their relationships, which are detailed in the following.

3.1. Building block

FAST building blocks are visual elements that abstract any part of a composite application. They are perceived by users as black boxes that process inputs to produce outputs. All end users know is what inputs they need and what outputs they generate. They do not have to program their operation which is pre-programmed or established by the finer-grained building blocks of which they are composed. These elements are published in catalogues to which users have access at three levels of abstraction. FAST includes four element types: screenflows, screens, forms and resources, which can be operators (for modifying the internal screen dataflow) or wrapped back-end services (service wrappers and low-level web resources for adapting services to the visual framework). FAST makes all these elements available for users who apply a top-down approach to develop a new solution. First, it displays ready-made high-level screenflows (built by software suppliers or end users that have developed and published their solution for use by other users). These screenflows can often solve all or a large part of use cases. If none of the screenflows match their needs, users can create their own based on off-the-shelf screens that they connect in an execution flow using connectors based on the pre/postconditions of the elements to be connected, as explained below. These screens are catalogued in the same manner as the screenflows. If users cannot find the screen they require, they can build it from atomic resources that appear at the third level of abstraction of the catalogue. Unless these resources already exist, end users will be unable to solve the problem, because they are incapable of doing what for programmers would be the rather straightforward task of building these resources.

Each building block type is detailed below in the top-down order in which they will be tackled by end users in a development process using FAST.

3.1.1. Screenflow

A *screenflow* is a meaningful aggregation of *screens* endowed with business logic. Business logic comes from the combination of each *screen*'s inner logic plus the composition logic. For users, they are a flow of visual elements, like forms, which can be used to carry out a functionality, like select and purchase a product from a supplier. This functionality involves visualizing the available items on screen, selecting the item, visualizing item data, adding the item to a shopping cart and paying for the item.

Following on with the running example, an end user will first search for a screenflow to book a flight to a destination, a screenflow to book a hotel, a screenflow to display tourist information about the destination, etc. If these screenflows are already available in the catalogue, all the user has to do is add the screenflow to his or her design, prepare their data inputs (using connectors) to satisfy their preconditions and analyse their postconditions in order to use the final outputs produced by the screenflow, if necessary. This applies to the screenflow for booking airline tickets, already available in many catalogues from suppliers like Expedia, see Fig. 3.

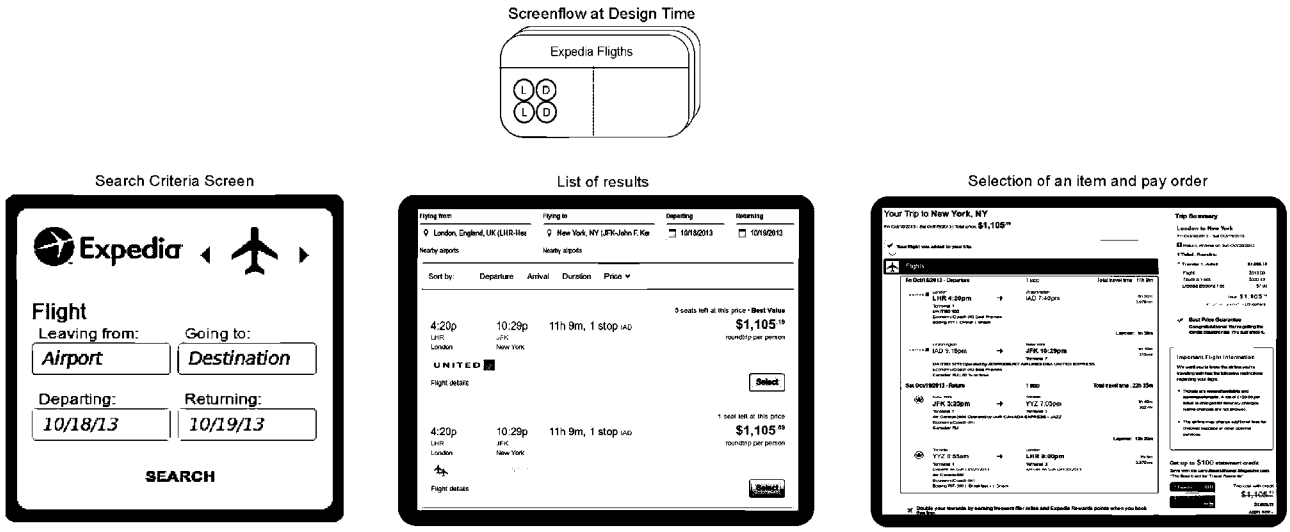

Fig. 3. Example of off-the-shelf screenflow to book a flight. (For interpretation of the references to colour in the text, the reader is referred to the web version of this article.)

These off-the-shelf components make the end user's job easier and are described in natural language in a catalogue, specifying their inputs, outputs and functionality. At run time, they provide a screenflow within a canvas positioned and sized to the users' liking. At design time, users manage the screenflow by means of a visual icon, as illustrated at the top of Fig. 3, which colour codes screenflow reachability (whether or not its inputs have been connected to a data source with the specified data types), and the expected input types and output types. The screenflow in the example requires two "location" (L) data types and two "date" (D) data types.

If users cannot find the screenflow that they need, they have to define the prototype building block that they require (required input types, expected output types), which they will build based on screens lower down in the catalogue. FAST will suggest screens that manage the data types defined in the prototype to expedite the keyword search of the catalogue for screens. In order to create a new screenflow, users will have to search for its constituent screens, create a dataflow between the screens and set up an execution flow in order to create the transitions between the visual elements on screen. Fig. 4 shows the creation of the screenflow for booking a hotel based on its constituent screens: initial search (product search), screen with details (product details) if the user selects a particular hotel from the list (product list), purchase order and booking confirmation.

The input and output types of these screens are connected by means of dataflow connectors and have an execution order based on the execution flow connectors used. The execution order is not necessarily related to the dataflow between screens. In the example, the end user has almost fully defined the data and execution flow.

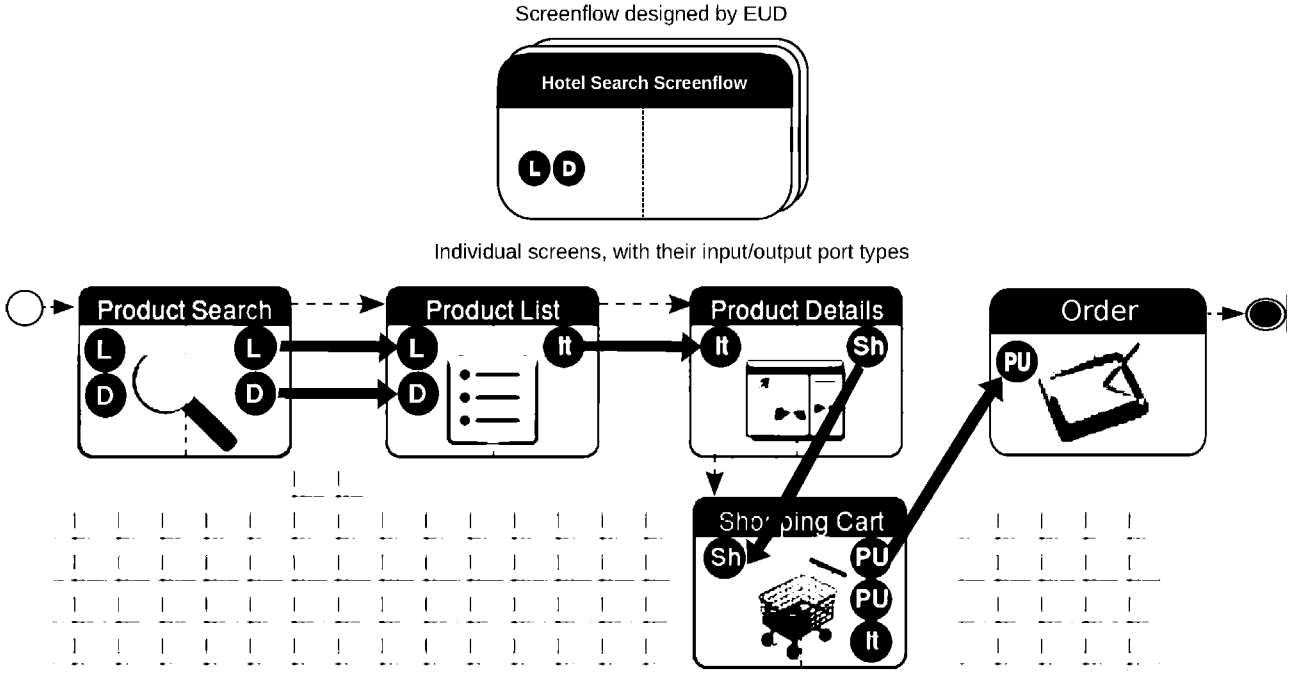

Fig. 4. Example of end-user designed screenflow to book a hotel. (For interpretation of the references to colour in the text, the reader is referred to the web version of this article.)

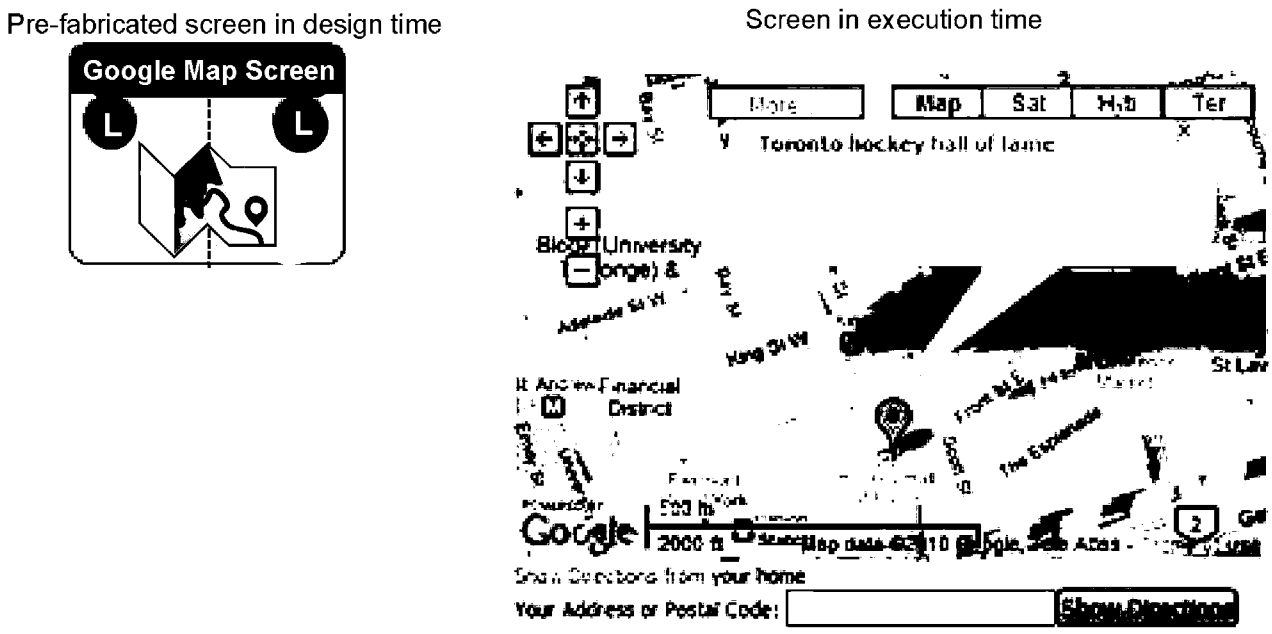

Fig. 5. Example of off-the-shelf screen.

The approach to *screen* integration in a screenflow is driven by *pre/postconditions*. Both *input* and *output port types* are used as *constraints* to drive the dataflow between screens and the transitions during *screenflow* execution. This technique will be explained in Section 4.

3.1.2. Screen

Screens are probably the most important component of our visual composition model. They are the smallest visual functional blocks that can be executed independently. They include both business logic and graphical user interfaces interconnected with each other by dataflow connectors. *Screens* have a *pre/postcondition*-based visual interface. This interface will play a key role in their composition to create *screenflows*, as discussed in the previous section. Bearing these constraints in mind, *screens* can be created in two different ways: (a) by linking several resources, operators and a form together in compliance with the FAST composition technique, or (b) by developing a monolithic and *ad hoc* piece of code on the condition that it conforms to the screen interface. Only option (a) is open end users, whereas software suppliers have two alternatives ((a) and (b)) for developing a screen for publication in the EUD catalogue.

Screens can be standalone or part of a larger screenflow. In the running example, the Google Map is a screen that is published in the catalogue and has standalone functionality. This screen displays a standard map of a location (input as a text string) (see Fig. 5). It can, if necessary, also extract the location of a point identified by a mouse click or by entering an address for use by other screens.

Users may often need to use screens that are not in the catalogue to build a screenflow. In such cases, FAST has a screen design tool for building screens from their constituent building blocks: a form and resources (operators and wrapped services) to create the required functionality. In the running example, users have to create a screen to search and list hotels provided by a web service in response to a search criterion based on the destination and dates specified on a form filled on an earlier screen. This screen is called product list in Fig. 4. This screen will be part of the screenflow necessary for booking a hotel. If it does not exist, it will have to be designed based on its constituent resources: a form that lists the results, a wrapped web

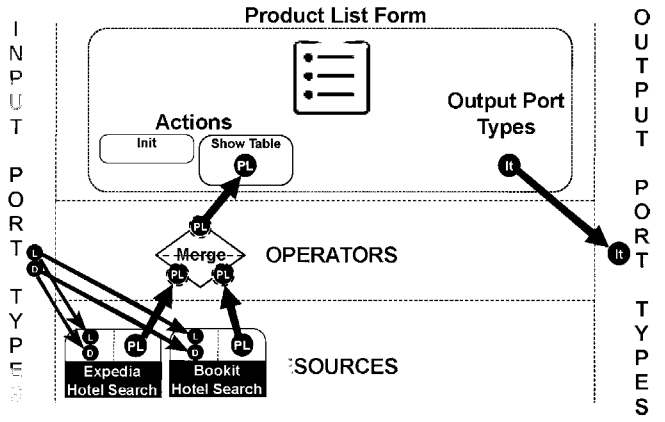

Fig. 6. Example of end-user design of the Product List screen.

service for searching the target agency, and the necessary operators, for example, an operator to sort the results by price. Fig. 6 shows how this screen would be designed using FAST.

Two remote resources, wrapped web services for searching hotels on Expedia.com and Bookit.com, are used in Fig. 6. The input of both services is a search criterion that is provided by the product search screen. The above criterion is a hotel location and an arrival date. These data types are used as input for the searches in both agencies, and the lists ("PL", product list) are merged using a binary operator called merge. This operator can also be parameterized to create a merged list sorted by price, for example. This list is connected to a form, which is used to select a particular item, producing a data type called item ("IT") for use on later screens. Each input/output port type is linked by dataflow connectors. These connectors and the input and output port types are explained later in this section.

3.1.3. Form

A *form* can be seen as a generic graphical user interface acting as a service front end. It is responsible for establishing visual communication with the end user at run time. In our proposed component

model, *forms* contain both view and presentation logic (i.e., event management or rendering operations). The forms can perform certain actions. Each action has a set of preconditions. In response to user-triggered events (like a click on a list item, submit, etc.), the forms output particular data types described as form postconditions. *Forms* are considered as black-box components and can be developed in any (web) technology. Note, however, that they should be designed as generically as possible to promote reusability across different application domains. FAST includes a wide range of these generic forms.

Remember that the *forms* will be the interface that end users will see and interact with, and it is important to offer the best user experience. But this is not easy to do with just a generic interface. This is where component parameterization comes in. Parameterization is useful for delivering customized forms (i.e., enabling internationalization or tailoring general-purpose interfaces to specific domains, etc.). In some cases, a customized generic form will not meet user requirements either. To solve this problem, the model supports domain-specific forms.

An example of a form for the running example would be a table that lists a set of items offered by a remote web resource enabling the user to select an item and output a dataflow with the information on this item. This is the case of the form displayed on the Product List screen in Fig. 4.

3.1.4. Resource

In the context of FAST, web services can be regarded as components that can and should be composed into larger systems (Papazoglou and Georgakopoulus, 2003). However, web services are only a particular case of invocable resources for composition.

One of the main advantages of our proposal is that adaptation is not tied to traditional SOAP-based web services. We are open to all kinds of back-end services, such as databases, legacy systems and even REST-compliant resources from a ROA (resource-oriented architecture Fielding, 2000). Our policy is to wrap all these back-end elements as resources that are integrated with other components by means of dataflow connectors. This creates a dataflow that can solve the business integration problems caused by back-end heterogeneity (Anon., 2001). This is a mashup rather than a SOA composition technology, as it is designed to wrap all types of resources, including non-dataflow SOA services. In our approach these resources are converted into dataflow feeds that are easy for users to use in their mashups.

Bearing this in mind, the component model defines a *resource* as the key component required to wrap or adapt services for subsequent composition. On the one hand, *resources* can be seen as an abstraction of an invocable method (i.e., a web service-specific method, a POST method for a POX-RPC service or any other type of back-end resource conforming to this concept). On the other hand, we propose to model resource inputs and outputs as *input* and *output port types* using semantic technologies to create dataflows between components, as explained later.

In order to wrap any web service or resource for use in FAST, it is necessary to set up a URI to manage the resource and create a façade, using Java EE for example, which provides a CRUD API in response to traditional HTTP verbs (GET, POST, PUT and DELETE) invoked on the selected URI (see Fig. 7). Additionally, it is necessary to create an XML representation according to an XML Schema defined in FAST, which provides the pre- and postconditions of the resource as wrapper metainformation. FAST will send a GET request to the URI of the resource to get the resource metadata and identify their inputs and outputs. Each resource functionality will be represented as a list of functionalities within the XML template. Each functionality will have a "URI_ global\ functionality" type URL schema on which the POSTs whose request header includes the data necessary for this functionality will be executed. This request must generate

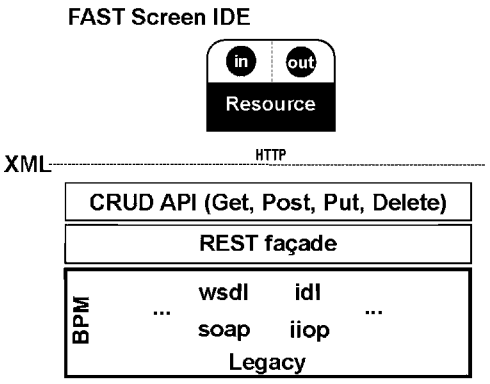

Fig. 7. How to wrap a data source or remote service for inclusion in the EUD.

a response whose body includes the resulting data. The generated CRUD façade must add the input data to a service-compliant request or external web resource (by preparing a SOAP invocation, another REST call, etc.), and process and prepare the response to this POST. The façade will also be responsible for processing any castings using the input or output data types.

The hotel search web service is an example of a resource that launches an internal search and lists the available options in response to a location and arrival date that the user enters on a query form. This resource is duplicated in the example illustrated in Fig. 4, as two resources are used to search hotels on two different web portals. Other examples of screenflow resources necessary for booking the hotel are the payment gateway, which manages end user payment given an item and its price, and the resource that actually books the hotel room given a list item provided by the hotel search web service and the OK provided by the payment gateway.

3.1.5. Operator

An *operator* is a subclass of FAST *resources*. *Operators* are meant to transform and/or modify dataflow data according to a process akin to piping. For example, an operator might be a filter that removes data items that do not meet a condition from a dataset, an element that performs a mathematical operation on an input data list, or an element that merges more than one list into one or splits one list into more than one depending on a criterion. No constraints have been placed on *operators* in FAST, and they are invoked through a common interface as if they were simple adapted services. FAST defines several instances of general-purpose operators, such as aggregators, filters, selectors or iterators. The designed interface will also provide the option of extending *operators* on demand.

In the running example, a "merge" operator, as illustrated in Fig. 6, can be used to merge the result of the items output by searching hotels offered by several agencies.

3.2. Pre/postconditions

Scientifically speaking, a *pre/postcondition* is *an objective and verifiable observation about a matter.* From the FAST standpoint, *conditions* are relevant assertions characterizing an instance of a domain concept, that is, a condition met by a specific data item. For example, an input is of a certain type has a specific length, is not empty or null, is in a repository known to the resource, is an instance/subsumption of an ontological concept or has a particular syntax. Until users enter the data at run time, there is no way of knowing whether or not particular preconditions hold, that is, whether or not the data item has particular characteristics.

Examples of preconditions are "the input data item is a card number", "a Google Map location", "an integer", "the 'card number'

text field is not empty", "the user is correctly logged in", etc. Examples of postconditions are "the output is an Expedia Hotel item", "a location", etc. Each FAST building block has a set of preconditions and a set of postconditions. They generally inform users about the needs and functionality of this building block and FAST about how this component can participate in a dataflow.

Conditions enable assisted semantic composition at design time to help users to enact processes: the components have input and output port types, expressed/defined in terms of pre- or postconditions. These conditions can be used to recommend valid dataflow connections among components that users should check and implement using a dataflow connector. The recommendations, which are checked using RETE, are explained later.

Input port types specify the set of input types (i.e., data syntax and characteristics) required for components to execute correctly according to their preconditions, which must be satisfied to guarantee execution. Examples of input port types for the "the input is a card number" and "the input is a Google Map location" preconditions are "card number" and "location". They represent the expected input data types, whereas the specific data values will not be available until run time.

Output port types are sets of output types that are produced by the execution of the component, that is, they are a set of assertions handled at design time about the outputs of the component at run time. An example of the output port types of a component that has "the user has logged on to the system" and "the component produces a location on a map" postconditions are "Boolean" and "location". Again they are the expected output data types of a building block, and the specific data value will not be available until run time.

For example, Figs. 3 and 4 show screenflows with "L" and "D" as input port types (L is a location and D is a date). In the next section we will see how *input* and *output port types* play an important role in the FAST composition technique.

At run time, *pre/postconditions* can be stored in any type of knowledge base or relational database, which can store the application state.

3.3. Connector

Connectors are elements that are used to integrate different resources with each other, establish dataflows among resources or define the user-defined component execution sequence at run time. There are two types of connectors: dataflow connectors and execution flow connectors.

Dataflow connectors are elements that are used to specify at design time which component outputs are directed at the inputs of another component. These connectors are used to generate a dataflow graph among components, which is often synthesized by matching input and output port types via FAST. Users receive recommendations on and decide which connections to set up in their application, as explained in Section 6. There are several connectors of this type in our running example, which, as discussed later, are depicted as a coloured arrow: dataflow connectors are used in Fig. 4 to define the internal behaviour of the screenflow for booking hotels. The output of the Product Search screen is used as the input of the Product List screen. If a user clicks on a listed product, an item data type is output, which is used as input for the Product Details screen, etc. The dataflow connectors are also used to create the internal dataflow of a screen, as shown in Fig. 6, where the inputs defined for this screen (a data type "D", date, and data type "L", location) are redirected to two wrapped web services. The web service outputs are directed to a binary operator and then passed to a form that displays the product list in table format, etc.

Execution flow connectors are useful for establishing a temporal execution order between screen resources to generate a screenflow. As explained later, screen execution usually generates a set of screen outputs, which match the types described by their output port types. Any screen that has access to the input data that it requires to execute (whose types will have been defined by its input port types) could be the next component to be executed in the application. Users use these connectors at design time in order to put together the screenflow.

In our running example, the execution flow connectors for the screenflow for searching and booking a hotel are illustrated as dashed directed lines in Fig. 4. The Product Search screen starts the execution of this screenflow. When this screen has finished running (the postconditions, product of a user-triggered event at run time, are satisfied), the Product List screen will execute. When the user selects an item that would satisfy the postcondition of the Product List screen, it will stop running, and the canvas will display the Product Details screen and so on. These types of connectors are used exclusively to create a screenflow from its constituent screens and not to design a particular screen.

There are special connectors for establishing complex iterations among screens, such as loops. These connectors operate like a loop with a stop condition. If the stop condition is met, the execution flow stops and an output is produced. If the condition is not met, the execution flow will be iterated up to n times until the stop condition is met. Unlike building blocks, these connectors do not have pre/postconditions; they just have a stop condition composed of the port type used and the expected value for stopping iteration. We found, however, that these types of connectors were not very often used to build the EUD applications developed in our experiments, because end users generate the iterations by manually performing operations on the lists of target items.

4. FAST composition technique

The aim of the *FAST composition technique* is to define how the components described in Section 3 can actually be composed. They are composed at two levels: by creating *screens* from existing resource, form and/or operator *building blocks*, or by composing *screenflows* from existing *screens*.

In this section we present the FAST composition technique that handles both processes. FAST includes visual support that tells end users which components they can connect with each other based on their input and output types. This visual support is very similar to the visual aids offered by all the studied tools, and is based on visually highlighting the port types that can be connected to other port types previously selected by users. Matching port types are coloured green, whereas port types that do not match but share the same data types (string, integer, double, float, array of strings, and so on) in the internal programming language are coloured yellow. For example, one port type may be a location and another keyword. They are not the same, so they will not be marked green to indicate that they are connectable. However, both port types are stored as a basic type string, and a valid dataflow could be generated if specified by the user. In these cases, port types are coloured yellow. These visual annotations rely on a *pre/postcondition*-based *input/output port typing mechanism*.

4.1. Input/output port typing mechanism

All FAST building blocks have a common interface whose inputs and outputs are defined at design time in terms of *pre/postconditions* which are implemented as a set of input and output data types. Building blocks use *input port types* to specify which data types they require to execute. The composition technique is based on dataflow connectors that link the output port types of particular components to the input port types of others with a

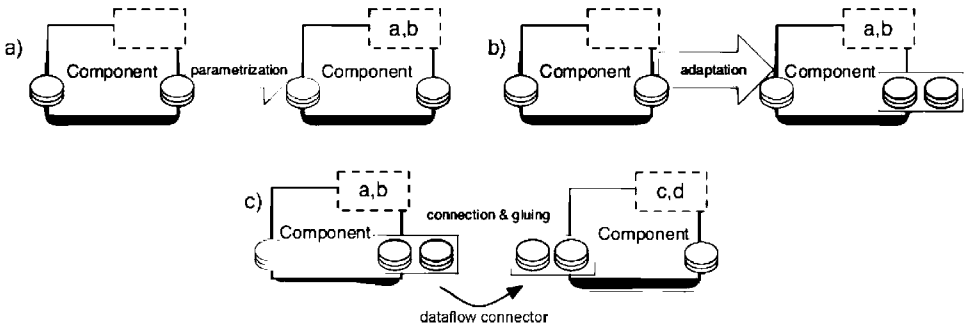

Fig. 8. Component parameterization and adaptation.

compatible data type. FAST uses pre/postconditions to recommend valid dataflows among components, and end users use dataflow connectors to establish these flows at design time. Building blocks that will not execute properly without a particular data type will be highlighted at design time until they receive the right type of information via a dataflow connector. Building blocks will not execute until they have received all the data that they require.

At run time, executing building blocks usually generate output data of a particular type. The *output port types* define these types at design time. In *screen composition*, the *output port types* will be propagated by dataflow connectors, which are used to establish a dataflow among components. They propagate output port types (the expected type or data semantics) at design time and the actual data at run time to the next building block in the composition chain. In *screenflow composition*, on the other hand, users can use execution flow connectors to establish an execution order among screens. If users fail to define an execution flow at design time or the data that any of the screens require to execute the defined flow are missing, FAST has an inference engine that will list the *screen(s)* to which the *output port type* could be delivered in order to eventually trigger a *screen* transition. If this happens, the end user will have to select the preferred screen from the list.

4.1.1. Parameterization and adaptation mechanism

Both the component model and the *input/output port typing mechanism* play a role in property parameterization and adaptation. First, components should be parameterized. To do this, a component is instantiated by entering specific internal attribute values (see Fig. 8a). Then, the component has to be adapted to our execution context by tailoring its interfaces, appearance, etc., to output the target *output port types* (see Fig. 8b). Finally, *pre/postconditions* are connected by dataflow connectors, thus enabling gluing, that is, the establishment of a valid dataflow between two or more components (see Fig. 8c). Parameterization and adaptation are supported at design time by visual aids (similar to the visual support offered by the other studied EUD tools), which help end users to generate the targeted outputs from the existing components.

An example of component parameterization and adaptation is the use of general-purpose forms: basic, general-purpose HTML forms that can be visually adapted by adding or removing fields, changing names, expected data types, etc. In our example, we already mentioned a hotel search screen (Fig. 4). We will take a general-purpose form for establishing search criteria on a screen called Product Search. The user will parameterize the form to include the search fields that they require (search location and date). Then they will adapt the form output port types, specifying Search Criteria as the output type. Finally, the Search Criteria will be glued to several wrapped search services, using a dataflow connector between the Product Search and Product List screens. Another example of form parameterization is the Product List screen shown in Fig. 6, where we can define the visual appearance of the hotel hit list table. The behaviour, appearance or functionality of almost any building block can be parameterized. The "merge" operator shown in Fig. 6 can be parameterized to use a sort criterion to be met by the output of this operator after having merged several different data sources.

It is the building block's creators that define the possible parameterizations, which are generally confined to visual appearance, data listing criteria, options for changing the effect of an input data item on the internal behaviour of a component (for example, a filter that outputs items including an input keyword or, alternatively, removes the items containing that keyword), etc. Visual descriptions of how this parameter affects the building block are offered to end users, with a default value that the user can alter by means of a selection from a pull-down list or a text field entry.

4.2. Screen composition

The bottom level of the FAST composition technique refers to the composition of existing *building blocks* to create *screens*. Fig. 6 shows how a screen for searching hotels can be created using this type of composition. We are now going to show how to create a somewhat more complex screen including more elements, namely, a screen that can gather tourist information (for example, from the TripAdvisor portal) on a specific destination. This screen, called Tourist Information List *screen* is illustrated in Fig. 9. In this case, we explain generally how to set up dataflows between several building blocks in order to illustrate how this composition technique works. This screen uses a search criterion output by previous screens and the user profile in order to search a wrapped web service provided by TripAdvisor for points of interest (POIs). In our example, users can also refine the product search by logging on and using user preferences specified in their TripAdvisor profile.

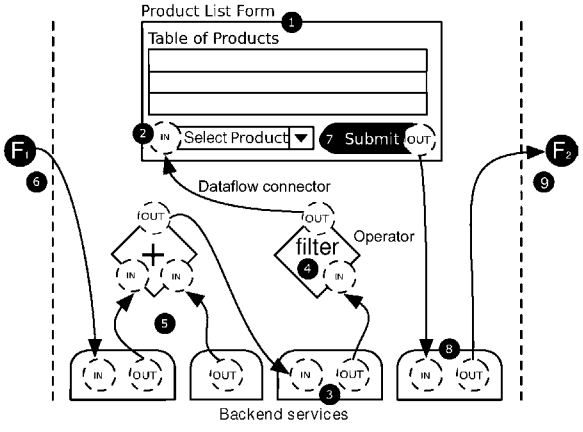

Fig. 9. FAST screen view.

To get the *screen* composition started, end users usually begin by selecting a *form* that meets their needs (1). In this case, the user selects a form displaying a table listing the POIs for the target destination. Users can select one of the table items and gather further information about the item (tourist sights, transport hubs, eating places, etc.). The user uses this form to parameterize the component (e.g., by choosing the sort criterion), adapt the component (e.g., specifying that the form output is an identifier of the selected item) and create the dataflow with the other components. The selected *form* may have to satisfy an *input port type* (2). In this case, the input port type is "a list of items with name and description". The end user is looking for a back-end *resource* that has the same *output port type* as the above *input port type*, such as a back-end resource that receives a "text string as a search criterion" and outputs an "item list" (3). TripAdvisor provides this resource, and, given a text string specifying a location, outputs a list of POIs at that location. Suppose that, apart from the name and description, the list resource (3) has countless other data about each item. Although the user has not managed to find exactly the right *resource*, because the component does not generate just a list of names and descriptions, he has found another resource that generates an output that is quite like what he is looking for. In this case, he might want to use a filter (an *operator*) (4) to adapt the data to the specific *input port type*. In the example, the filter should remove all the fields that are neither the item name nor description. *Dataflow connectors* will link back-end *resources*, *operators* and the *form*. These connectors will *semantically match* data port types to guarantee the validity of the data and their data type.

Thanks to the *input/output port typing mechanism*, back-end *resources* have their own *input port types*. These *input port types* also have to be satisfied in order to get the *screen* working. If they can be satisfied by means of other *resource output port types*, the user will connect these *resources* (5). Imagine that the precondition (F1) for the tourist information screen is a location, entered by the user as a keyword (6), and that the search criterion is used to search POIs using the TripAdvisor web service. Imagine a TripAdvisor back-end resource that outputs terms that the user has already searched depending on his or her profile and preferences. The two lists can be joined by a concatenation operator to produce a "a comma-separated text string as a search criterion", and this output port type enables the resource to execute (3). Any unresolved *input port types* within a screen (6) will have to be solved by executing a previous screen in the application screenflow.

Even if the *form input port types* have been satisfied, the *form* may not, depending on the type of *form* that the user has selected (interactive or otherwise), be able to execute (7) without a UI event. *Form* execution should create some output data (like an *output port type*, that is, "TripAdvisor item identifier" in this case). Depending on the screen business logic, the *output port type* could be propagated to a back-end *resource* to validate or manage its data. In this case, a TripAdvisor resource is accessed to validate the selected data item (8). Finally, if the execution is error free, the screen *output port type* will satisfy the *postcondition* (9), that is, in this case, "the output is a TripAdvisor item identifier" (F2).

The above-mentioned visual aids supporting the composition technique direct and recommend all interconnections among components, and are similar to the visual support offered by existing EUD tools.

At run time, users draw on their experience to enter specific inputs. Suppose that the value of the input port type "string as a keyword" is "New York", which, after executing the first resource, is converted into an output on the list "New York City, United States of America". This is then supplemented with the user preferences, for example, "Italian restaurants", "ICT conferences", "3*, 4*, 5* hotels" are added as these are search terms that the user has used recently. The resulting table lists POIs returned by searching TripAdvisor for these preferences. If the user selects any listed POI, the screen output will be "POI TripAdvisor item id", whose details will be displayed on the next screen of the screenflow. This is very like the Product Details screen used in the screenflow for booking hotels.

When the port types of the different building blocks match, FAST will visually highlight the match in green, recommending end users to set up a dataflow between the building blocks. If the port types do not match, but the data types representing the items in FAST's internal programming language are the same (strings, integers, etc.), the user will receive a visual cue highlighted in yellow, indicating that these types could be connected to set up a valid dataflow, even if the components do not appear to have been catalogued as collaborative. If the basic types do not match, the user will not be able to set up a dataflow and will have to use some other intermediate operator to make the data types compatible.

4.3. Screenflow composition

Screenflow composition is the top level of the FAST composition technique. It generates a fully functional composite application. As expected, every *screen* has a set of attached *input* and *output port types* that will be used to drive the transition from one screen to another through a set of *output port types* during *screenflow* execution. This way, a *screen* has two possible states—*reachable* and *unreachable*. If all the *input port types* of a *screen* are satisfied by the *conditions* output during the *screenflow* execution, the *screen* will be reachable. Otherwise, it will be unreachable.

End users can explicitly state the target screen execution order, using *execution flow connectors* that simply serve to establish the chronological order of *screen* execution. If execution flow connectors fail to define the execution order or the order cannot be followed because it includes unreachable screens, the FAST platform *input/output port typing mechanism* creates a list of the screens that can be rendered at any time (since their preconditions are satisfied). This list is displayed for end users to select which screen they want to execute next.

FAST displays a list with all possible execution flows for the screens that the users are using (depending on their inputs and outputs). In this way, users can either create a valid flow using execution connectors or leave it to the system to create a list of reachable screens from which they manually select the next screen to be executed.

Thanks to the *input/output port typing mechanism*, there should be no obstacle to adding a *screen* whose *input port types* are matched by the current data types present in the *screenflow*.

5. FAST composition languages

To meet all the composition language requirements, we propose two different representations with different aims:

- *FAST Visual Composition Language (FVCL).* FVCL is a visual language enabling end users and programmers to intuitively and productively compose applications.
- *FAST Modelling Format (FMF).* FMF is a way of defining an application persistence format using markup languages like JSON or XML. It is used to define the intermediate storage of FAST compositions. This representation is designed for processing and transmission purposes not for use directly by users.

As composite applications have to be compiled to executable languages for deployment on different execution platforms, another *execution language* is required on top of FVCL and FMF. This language is described in Section 6.

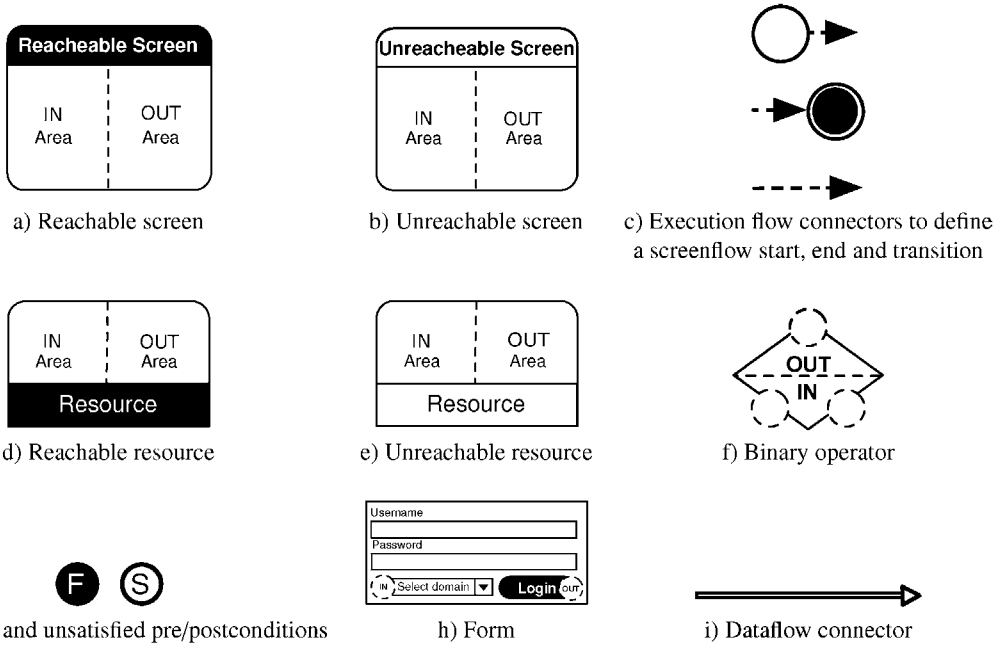

Fig. 10. Visual representation of FAST components. (For interpretation of the references to colour in the text, the reader is referred to the web version of this article.)

The composite applications design process will include a set of *model transformations* (Sendall and Kozaczynski, 2003) from the visual language to the execution language. However, these transformations will be automated, and end users will deal with the visual language (FVCL) only. In the following we detail these languages and the transformations.

5.1. FAST Visual Composition Language (FVCL)

FAST Visual Composition Language is the language for visually composing the different FAST components. There are lots of visual languages in the literature (Rumbaugh et al., 2004; Ceri et al., 2007). Some even describe how services are composed (Pautasso and Alonso, 2003; Nestler et al., 2009). However, these languages do not usually target non-programmer users. We have developed a new language based on the above composition technique. It has been designed to be visually simple to improve learnability for users.

One of the main issues when defining visual languages is how to describe the type of representations that the language uses. This language deals with *what* is to be represented, *how* it is to be represented, and how to *associate* the representation with what it represents (Narayanan and Hbscher, 1997).

5.1.1. Visual representation of FAST components

In the following, we describe how FVCL represents the FAST component model and explain the graphics used.

Screen. During *screenflow* design, *screens* are represented as rounded corner boxes. They are divided into three areas as illustrated in Fig. 10. There is a caption at the top. *Input* and *output port types* are both entered in separate areas, IN area and OUT area, respectively. Reachable and unreachable screens are coloured green (Fig. 10a) and red (Fig. 10b), respectively.

If we are composing other FAST components to create a *screen*, the screen view will be very similar to the illustration shown in Fig. 9.

Resources. *Resources* are represented in very much the same way as *screens*. In fact, both components have a caption and *input* and *output port type* areas. However, the layout is different as illustrated in Fig. 10d and e. As for *screens*, *resource* colour depends on their reachability.

Operators. Fig. 10f illustrates the visual syntax for a binary *operator*. As shown, *operators* are represented as a diamond divided into two halves. The top half contains *output port types*, whereas the bottom part contains the set of *input port types*.

Pre/postconditions. The visual syntax of a *pre/postcondition* is a small circle (see Fig. 10g). The circle will contain either the initial letter of its associated concept or a more elaborate acronym. For preconditions, a solid (and green) circle means that the precondition is satisfied, whereas a (red) outline indicates that the precondition is not satisfied. A precondition is satisfied by using dataflow connectors that supply the building block with the data types necessary for its execution. For postconditions, a (red) outline indicates that the building block still has unsatisfied preconditions and is, therefore, unable to execute and meet its postcondition. When all the preconditions of a building block are satisfied, this building block can execute, and its postcondition is also represented as a solid (and green) circle.

Forms. *Forms* are needed to create *screens*. As illustrated in Fig. 10h, they are represented as a rectangle whose background shows a thumbnail of the associated user interface and some *input* and *output port types*.

Execution flow connectors. At screenflow composition level, FVCL defines three execution flow connectors just in case the user wants to set the first or last screen of the screenflow or fix a transition between two particular *screens*. Fig. 10c illustrates both the start, end and transition connector symbols. The execution flow connectors are used to define the chronological order in which the screenflow screens are executed, as if it were an activity diagram.

Dataflow connector. At screen composition level we have a connector to create dataflows among components called *dataflow connector*. A dataflow connector is represented by a single arrow as shown in Fig. 10i. It is these connectors that drive component execution depending on the dataflow generated in the composite application.

5.1.2. FVCL views

Although all composition information could be displayed in one diagram, it would not be at all user friendly (Shneiderman, 2003). Without special sectional diagrams, called *views*, focusing on smaller parts of the composite application, users would be able

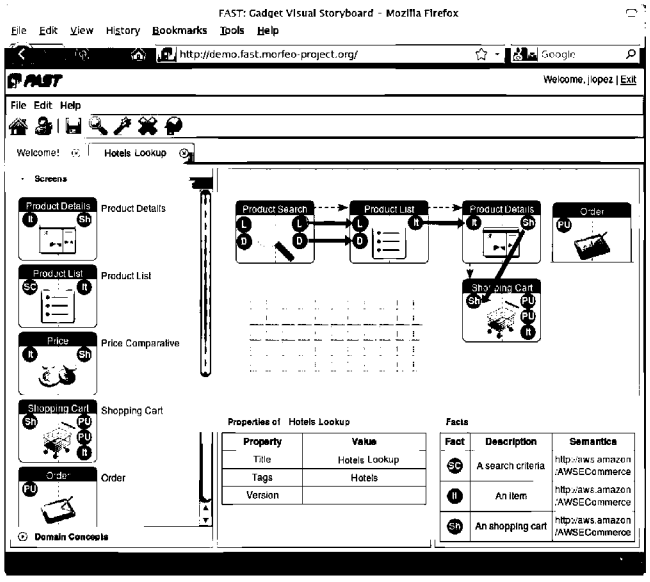

Fig. 11. Screenshot of the FAST tool's screenflow design.

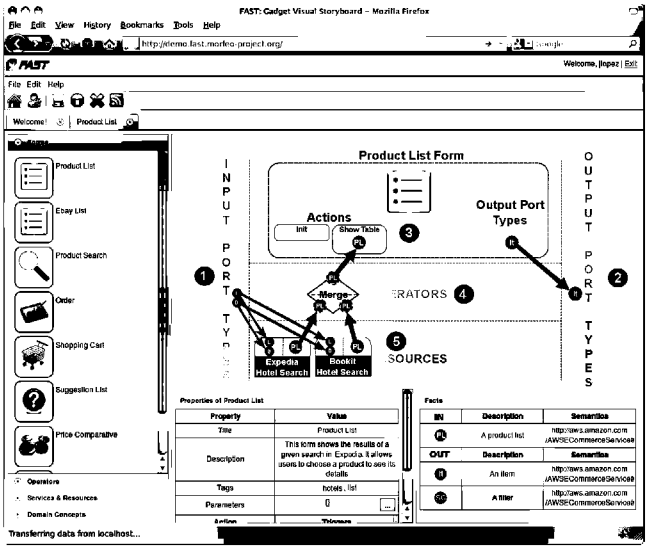

Fig. 12. Screenshot of the FAST tool's screen design view.

to understand only very straightforward composite applications. These views should deal with cohesive rather than arbitrary subsets of the application that are coupled with the rest of the application as loosely as possible.

FVCL offers several views depending on whether the user is composing at *screenflow* or *screen* level. These views are illustrated in the screenshots for the running example of a user developing an application to search travel agency web sites for travel options.

Screenflow view. This view shows the *screens* that are part of the composite application under development and the execution flow connectors that decide their execution order. Fig. 11 shows the compositional framework generating a screenflow called Hotel Lookup. It depicts a top-down design process where the user has created a screenflow to show details of hotels supplied by several travel agency services. It includes the screen view described below. The user uses Product Search to enter the location and date he is looking for, Product List to list hotel search results, Product Details to show details of any item selected by the user from the list, and Suggestion List to provide suggestions possibly of interest to the user based on his preferences (for this purpose, the user will first have to have logged in).

Screen input and *output port types* define an implicit screenflow that is constrained by execution flow connectors. For instance, a begin symbol attached to a login screen will guarantee that this is the first screen to be executed.

Screen view. In the screen view, the main area is partitioned into five different regions, as shown in Fig. 12 illustrating the visual composition of the screen listing hotels for a particular location and date (L and D):

- *Input port type area* (1). This area is on the left and contains preconditions modelling the input port types for the illustrated *screen*. In this case, the port types are L and D, which means the precondition "string as location" and "string as date".
- *Output port type area* (2). This area is on the right and contains the postconditions satisfied during the execution of the component. In this case, the output port type is IT, which means the postcondition "hotel catalogue item information".
- *Form area* (3). This area, located at the top of the main area must contain exactly one *form*. This component provides the user interface for the whole *screen*, and is therefore mandatory. In this case, the form is a table listing products. This form is adaptable and parameterizable for use on this screen. The form input port type is set to PL "the input is a string containing a list of products with name and description". The form output port type is set to IT, D and L "the output is a hotel item identifier, a string as location and a string as date". The results will be listed according to the preferences that users use to parameterize the form.
- *Operators area* (4). This area contains data operators, such as filters applicable to lists, operators for merging items provided by different sources to set up a single list containing the merged items, operators for ordering a list according to a criterion, arithmetic operators, etc. There is only one operator in the example used to merge the results from two different travel agencies and create a common product list.
- *Resources area* (5). Located at the bottom of the main area, the resources area stores one or more wrapped resources providing uniform access to business back-end services, web services and remote resources, etc. They will be invoked when their *input port types* are satisfied. As a result, their *output port types* can propagate through dataflow connectors, eventually triggering additional invocations. In this case, wrapped web resources from two different travel agencies are used. The resource input should be a location and an arrival date, and its output should be a "string with a list of products with name and description".

5.1.3. *Visual scaling*

Additional auxiliary views are provided for the sake of visual scaling. One illustrative example is when an *input* or *output port type* area of a *screen* contains too many port types. There is no limit on how many port types a screen can contain. Due to size constraints, the port types can be stacked and displayed for users by means of a menu. This is illustrated in Fig. 13.

Other possible views are a screen properties table, input/output port type inspector and description pop-ups. These auxiliary views are intuitive, self-explanatory and they pop up on demand (are user-event triggered).

5.2. *FAST Modelling Format (FMF)*

FVCL is intended for use by human beings through visual representations of the components and connectors making up this proposal. These visual representations do not necessarily all have to be the same (geometric forms or colours may vary) and they must be serialized for computer processing. Therefore, the FAST platform

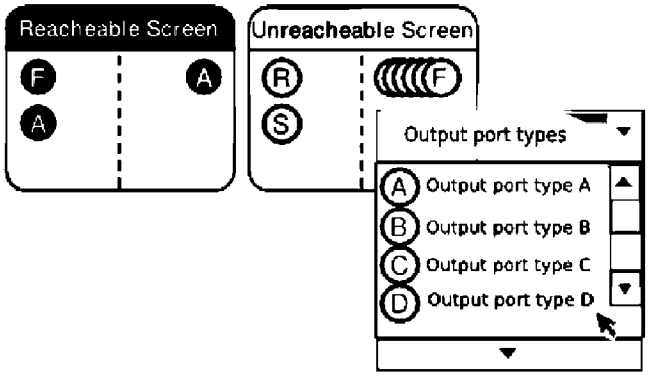

Fig. 13. Screen representation of many input/output port types.

requires a machine-friendly representation of the building block created during design. Apart from serializing the visual composition built from components and connectors, the JSON-based FMF must meet the following requirements.

On the one hand, FMF must be expressive enough to losslessly represent the component and connector model being designed. Not only must it ensure model persistence but it must also support social sharing mechanisms, that is, mechanisms for end users to use to publish their developments in a catalogue of EUD solutions for a particular domain, which will eventually ease the work of other end users tackling the same problem. On the other hand, resulting artefacts must conform to an unambiguous representation capturing all the modelling information and, at the same time, making provision for their translation into executable objects. Unlike FVCL, the focus moves from user-friendliness to computability and execution performance. This raises different design constraints.

Whenever a user visually adds a new component or connector to a composition or creates a new dataflow with a dataflow connector between two components, this visual manipulation has an effect on the model that has a counterpart in the FMF representation. Both representations are kept consistent throughout all the development steps, since there is a true correspondence which the tool can manage.

Two representations are used because they each serve a different purpose. The FVCL language is a visual language designed to enable non-programmer users to visually compose a component- and connector-based application generating a dataflow. And the FMF aims to describe in a markup language (JSON, XML, or other) the resulting composite application specified in FVCL, including the service infrastructure and technology required to get the application to run on multiple mashup platforms. It separates the runtime component or application from the visual tool used in the process. As a result, other users can use or reuse part of the application on other platforms.

FMF-modelled documents are intended to represent the following groups of information about a FAST component design:

- *Non-functional properties.* These properties describe the component that is being designed and include component metadata, such as author, creation date, semantically enriched tags and so on. Their purpose is to better identify and describe the component, and they are kept separate from the behaviour that is covered in more detail and more formally in the remainder of the document.
- *Executable components library.* Executable components described in FMF are defined in terms of lower level binary elements. When a user visually adds a new component or connector, the library is updated with the URI that identifies the executable component and its instantiation data accounting for parameterization, if any, and renders information for further editing.
- *Relationships.* The document shows how the components relate to each other. For instance, a dataflow connection will lead to the creation of new objects in the document describing the related entities and relationship type.
- *Referenced executable objects.* Lowest level components are executable binary components that are also described by FMF documents, where the components library and the relationship information is replaced by a reference to the actual implementation.
- *Input and output port types.* The boundaries of the component as a whole are defined in the form of input and output port types specified as a subset of SPARQL expressions (Prud'Hommeaux et al., 2006), plus a human-readable description. This kind of representation enables semantic-based recommendations during design and semantic component matching.

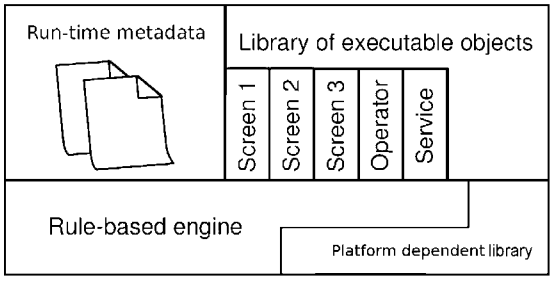

Fig. 14. Screenflow as an executable object.

FMF documents could be represented in any hierarchical object-oriented syntax, thus a variety of markup languages, such as XML or RDF, are suitable. However, the JSON object notation (Crockford, 2006) stands out in terms of interoperability and client-server round-trip communication. For the working draft definition of the language, see (Reyes et al., 2010).

The FMF representation of a screenflow, plus the linked representation of its components and the executable objects of the lowest level components, contains all the required information to build the composite application. The next section discusses this process.

6. Execution language

On top of the composition languages, we have need of a language to execute the application defined in FMF. The main objective of the FAST composition system is to produce an executable composite application that is able to run on several existing mashup platforms. The complete application built using FAST will be tailored to the available underlying technologies of the composite applications (i.e., JavaScript, HTML and CSS). These well-known technologies are the foundation of the execution language and are commonplace in web development.

As Fig. 14 shows, a compiled screenflow, which is the basis of the composite application, is made up of the following components.

- *A rule-based engine.* This engine manages the screen execution flow. It consists of a knowledge base that represents the current status of an executing screenflow and contains all the instances of concepts produced at any time, and a set of rules, each representing a screen or a connector between two screens. The engine relies on the well-known RETE algorithm (Forgy, 1982). RETE can be used because the input and output port types are modelled as rules and conditions. Other contract checking mechanisms, such as OCL or JML, could be used, but RETE returns good

performance results. RETE is able to check the underlying types of each port type and build two types of recommendation rules for the end user: strong recommendations if the port types match, conditional recommendations if the port types do not match but the programming language types do. The engine is tailored to the platform as follows: the engine checks the type of each generated mashup platform data item, notified by its API, and runs the production rules that contain the generated data type in their input port type to model the application dataflow at run time. These rules can then lead some components to output new data and so on. This is achieved by adding a few additional lines of code to the executable component in the object language run by the platform in question.

The rule-based engine is largely mashup-platform agnostic but some key and optional functionalities are separated in a platform-dependent module. Depending on the target platform, a suitable module will be plugged in.

- *Platform-dependent library.* This library conforms to a predefined API, providing common features such as AJAX calls or user preference management. Mashup platforms implement the above functionality in a non-standard fashion, calling for target platform-specific libraries.
- *Library of executable objects.* This library compiles all precompiled code for all the screens and other screenflow building blocks. The screen compilation process will be described below.
- *Run-time metadata.* The metadata include all the information necessary to instantiate the different executable screenflow components and an executable rule set used by the rule-based engine. None of the non-functional properties are needed at this stage, however.

Rules are independent of the full-blown semantic reasoner, as they include static mediation information.

On the other hand, screen compilation takes an FMF-based screen document and produces the executable component for integration into a library of executable objects. To do this, the following components must be compiled:

- *Interface characterization.* This metainformation is relevant and will be exploited during screenflow compilation and execution. It states the *screen input* and *output port types.*
- *Building block run-time dependencies.* List of references to executable components needed for screen execution that can be run through recursively in order to output a transitive dependency list. The combined list of dependencies is output as part of the screenflow compilation and comprises the form conveying the user interface, back-end services providing both data and functionality and operators that transform the data.
- *Instantiation metadata.* Executable objects to be instantiated at run time can be appropriately parameterized with user information held as part of the instantiation metadata.
- *Dataflow information.* A dataflow connection states that the output port of one component is connected to the input port of another component so that the types are matched at design time in order to assure a valid dataflow between binary components at run time. Therefore, the execution of a given executable component is dependent on the reception, along their respective dataflow connections, of all the inputs in compliance with the constraints established at design time.

This is equivalent to telling an executable component that cannot run without a given data input when these data are output by a component to which it is connected by a dataflow. If all the preconditions of an executable component are satisfied, the associated action is launched.

The screen compilation process will add a new screen to the catalogue. This *screen* can be consumed to create new *screenflows.*

6.1. Platform independence

The composite application development process ends with an execution language that is tailorable to different mashup platforms. This way, the executable components or applications built are platform independent. Most platforms on the market use the JavaScript, HTML and CSS execution languages. So, the executable components are based on pieces of HTML code with JavaScript scripts that receive inputs, perform functions and generate outputs. All mashup platforms also have a template containing component metainformation, stating the required inputs, the required outputs, their types and semantics. The proposed composition process wraps the components with generic templates based on the details described in FMF (using the JSON language). Thanks to its features, metainformation generated by the JSON language used in FMF is very easy to tailor to a particular template schema proper to a commercial mashup platform. FAST can now export the composite application as an EzWeb, Netvibes, iGoogle, Yahoo! Pipes or JackBe (Lizcano et al., 2008) widget or executable component. Tailoring the executable component to other platforms with templates that conform to other XML Schemas merely entails adapting the execution language tag syntax to a new naming schema. XSLT (Extensible Stylesheet Language Transformations) or manual customization of the generated template are solutions (Kovse and Harder, 2002). As the visual composition by the user using the FVCL language is translated to the intermediate FMF, there is a full list of components, connectors, and input and output port types for each dataflow node. From these, it is possible to generate a composite application template that is then easily convertible to other templates demanded by specific platforms to execute or publish components.

7. Evaluation of the composition model

We have evaluated the use of the FAST component- and connector-based composition framework presented here. The evaluation demonstrates that our premise of enabling non-programmer end users to build their own composite applications is feasible and true. The FAST composition framework is publicly available at http://conwet.fi.upm.es/fast_blog. FAST evaluation aims to test whether end users find the functionality and performance of this framework, which uses the user-centred composition model presented in this paper, satisfactory. To do this, we have run an experiment that aims to answer the following research questions:

- RQ1: Does FAST enable non-programmer users to create a composite web application to solve a real-world problem?
- RQ2: Does FAST outperform other compositional tools in the formulated scenario?
- RQ3: Does FAST meet end-user expectations with respect to functionality and performance?

To answer these research questions, we recruited 180 end users to solve a real problem using EUD tools. The 180 users were recruited via a web portal set up for the purpose by the international consortium participating in the FP7 FAST project development. Of over 300 users interested in participating in the experiment, we selected a sample of 180 users in order to form six groups of 30 users that were unbiased with respect to age, sex, training, professional experience, etc. The statistical studies conducted to validate the sample and check that all six groups were unbiased are reported later in this section. In order to answer RQ2, the sample of 180 users

Table 2
Sample characterization.

Characterization	End users (180)	Group 1 (30)	Group 2 (30)	Group 3 (30)	Group 4 (30)	Group 5 (30)	Group 6 (30)
Gender							
Male	95	15	15	17	15	17	16
Female	85	15	15	13	15	13	14
Age							
Under 20 years	31	6	5	5	4	6	5
21–34 years	47	7	7	8	8	8	9
35–49 years	42	7	7	7	8	6	7
50–64 years	36	6	7	6	6	6	5
Over 65 years	24	4	4	4	4	4	4
Educational attainment							
Secondary School	43	8	8	7	7	7	6
Vocational Training	47	7	8	8	9	7	8
Bachelor's Degree	42	7	7	7	6	8	7
Master's Degree	48	8	7	8	8	8	9
Employment							
Student	47	8	8	7	9	7	8
Researcher	48	8	8	8	8	9	7
Employee	85	14	14	15	13	14	15
Experience and previous knowledge							
Mashup Platforms	6	1	1	1	1	1	1
Web Services (SOAP, ESB, BPEL, etc.)	0	0	0	0	0	0	0
JavaScript, HTML, CSS, AJAX	0	0	0	0	0	0	0
Java, J2EE	0	0	0	0	0	0	0
Php, ASP	0	0	0	0	0	0	0
OO Programming	0	0	0	0	0	0	0
C, C++, C#	0	0	0	0	0	0	0
Haskell, Prolog	0	0	0	0	0	0	0

was divided into six groups: five groups worked on five major EUD tools (Yahoo! Pipes and Dapper, iGoogle, PopFly, OpenKapow and Apple Dashboard), and the sixth group used the FAST tool under evaluation. These tools were selected on the grounds of popularity and widespread use. Accordingly, each group specialized in a particular tool to solve the stated problem. None of the groups had tackled the problem before or had any previous experience of the tool that they were using. This guaranteed the validity of the results and the comparability of the results across all groups. The size of each group is statistically representative, and normality tests can be run. In this type of study, it is essential to assure that all six groups are homogeneous and that the allocation of the 180 individuals to their respective groups was not biased.

The user sample is shown in Table 2. This table also shows the division of users into different groups.

Only six of the users have programming skills, and are knowledgeable about mashup tools (in this case, iGoogle). These users were distributed across the groups to assure that their previous knowledge did not alter the results of any group.

In order to validate this division of users into different groups, we ran an analysis of covariance (ANCOVA) using the group to which each end user was allocated as the study variable and the user characteristics as the explanatory variables. This study builds a regression model to explain the study variable with respect to the other qualitative and quantitative variables. If the ANCOVA study were to return a well-fitted regression model, then the division would not be valid, as the groups would be biased with respect to the most influential explanatory characteristics in the fitted model. Table 3 shows the results of the ANCOVA study, which suggest that the model fit is extremely poor. This validates the selected sample and its distribution.

Looking at Table 3, we find that the coefficient of determination R^2 is very low (0.015). This suggests that there is a high percentage of variability in the modelled mean variable so that gender, age, educational attainment, employment and previous experience (the quantitative and qualitative variables for each individual) appear to explain only 1.5% of the division of users into the six groups. The other values are due to other unknown variables. The R^2 and adjusted R^2 values suggest that the group to which each end user was allocated is largely (98.5%) independent of user characteristics. The model error values, MSE (mean squared error) and MAPE (mean absolute percentage error), are very high (well above the ideal value 0), again suggesting that the model does not precisely explain the behaviour of the variable under study in the sample. Additionally, DW (Durbin-Watson statistic) values are not close to 0. This implies that there is no autocorrelation among the qualitative variables. If there were, the study would not be valid. Finally, Cp (Mallows' Cp statistic) suggests that the model is able to exactly explain the group to which only one (see df value in the model) of the 180 individuals was allocated. We have conducted a Type I and Type III sum of squares analysis. Type I (sequential) analysis provides an incremental improvement in the sum of squared errors as each effect is added to the model, and Type III (orthogonal) analysis is able to reduce the sum of squared errors by adding the term after all other terms have been added to the model. Their combined use means that we do not have to be concerned about the order in which the factors were added to the regression model. Taken together, the model results validate the sample, indicating that there is no bias related to the qualitative and quantitative variables characterizing the users and their recruitment for the study. Looking at the $Pr > F$ values of the ANCOVA model, we find that the characteristic that is most related to the allocation of a user to one group or another is education (the greatest $Pr > F$ in the study, equal to 0.477). We examined user education and found no statistical evidence of a direct correlation between education and division into groups.

The validated sample was analysed as follows. In response to RQ1 and RQ2, all six groups were asked to solve the same problem as illustrated in the running example, each using one of the following tools: Yahoo! Pipes and Dapper, iGoogle, PopFly,

Table 3
ANCOVA study to validate the sample recruitment and division into groups.

Goodness of fit statistics

Observations	Sum of weights	df	R^2	Adjusted R^2	MSE	MAPE	DW	Cp
180	180	64	0.015	0.025	5.142	4.462	1.157	2

Analysis of variance

Source	df	Sum of squares	Mean squares	F	$Pr > F$
Model	1	5.932	0.169	1.196	0.264
Error	178	9.072	0.142		
Corrected total	179	15.004			

Computed against model = mean (Y)

Source	df	Sum of squares	Mean squares	F	$Pr > F$
Type I sum of squares analysis					
2. Gender	1	0.042	0.042	0.294	0.319
3. Age	1	0.134	0.134	0.943	0.335
4.1. Education	3	0.752	0.251	0.968	0.362
4.2. Employment	2	0.163	0.081	0.575	0.266
5. Experience and previous knowledge	28	4.387	0.199	1.407	0.146
Type III sum of squares analysis					
2. Gender	1	0.524	0.175	1.232	0.305
3. Age	1	0.084	0.084	0.595	0.243
4.1. Education	3	0.212	0.106	0.949	0.477
4.2. Employment	2	4.041	0.184	1.296	0.209
5. Experience and previous knowledge	28	0.445	0.074	0.823	0.289

OpenKapow, Apple Dashboards and FAST. The problem statement is also described at http://apolo.ls.fi.upm.es/eud/problems_description.pdf (see Problem 0). Each tool provides a different problem-solving approach as outlined in Section 2.

The requested application requires the use of from 22 to 24 components (including screens, screenflows, forms, connectors, operators, back-end services, etc.), and their assembly requires the creation of approximately 20 dataflow connections among components. The problem was carefully defined to assure that all six tools under evaluation have all the components, composition and dataflow creation techniques necessary to be able to solve the problem. Before we conducted the study, we personally solved the problem using each tool to check that the task was feasible. Additionally, we also set up a catalogue of components, resources and operators for each tool before running the experiment. These catalogues included all the components and elements necessary to solve the problem, as well as general-purpose components that were no use for the problem at hand. All six catalogues contained around 650 components of different levels of abstraction. Therefore, this is not a straightforward development. For a full and detailed description of these six catalogues of components and connectors and the development processes enacted by the sample of users, see http://apolo.ls.fi.upm.es/eud/solution_development_process.pdf.

Once the tools and equivalent component catalogues for each group had been set up, the end user then received basic training via video tutorials on the tool that they were to use. Each group was separately given the same number of training sessions. The schedule for each group was:

- Theory session (four hours): introduction and familiarization with component- and connector-based development and mashup technology for the respective tool that the group was to use.
- Practical session (four hours): basic practical exercises set for each user group: Yahoo! Dapper and Pipes, iGoogle, PopFly, OpenKapow, Apple Dashboards and FAST platforms. Two short videos (see http://www.youtube.com/watch?v=qFt2LBlxkwU and http://www.youtube.com/watch?v=dpoRhnF8_1A) were used to describe and introduce the available components and the tool in question.
- Hands-on-workshop (two hours): each user was asked to solve the stated problem individually.

The training sessions focused on explaining how use each tool to solve problems akin to the stated problem, explaining the components to be used and the composition techniques provided by each tool. Accordingly, the design of the sessions for each tool was similar.

We unintrusively supervised the problem-solving workshop, analysing times taken, problems encountered, sources of conflict for users, etc. There was at least one supervisor for every 10 users at all times in order to provide a qualitative and quantitative analysis of this two-hour workshop. Table 4 shows the statistical data on how many users managed to build a valid solution that met all the set requirements in each study group, and how long it took them to do so.

Of the 30 users in the respective groups only two Yahoo! Pipes and Dapper users, three iGoogle users, three PopFly users, five OpenKapow users and six Apple Dashboards users managed to find a solution. These outcomes contrast with the results for the framework described in this paper, as 17 users achieved the goal using FAST. This experiment clearly reveals the superiority of FAST compared with the other analysed tools, but even so only 17 out of 30 users completed the task despite having received training before the trial. This is a clear indication that tools like these are often not suited for use by non-programmer end users, and a wizard needs to be built into the environment in order to provide guidance on the use of the different composition and development techniques and help users to build software to meet specific requirements. We consider this to be a significant lesson learned from this investigation,

Table 4
Development time taken using FAST, Yahoo!, iGoogle, Microsoft, Kapow and Apple visual languages.

Tool	*N*	Mean of time	Std. dev.	Std. error	95% lower b.	95% upper b.	Min.	Max.
FAST	17	37.84	0.97	0.0541	36.30	39.90	31.00	49.00
Yahoo! Pipes and Dapper	2	45.50	6.42	0.1469	40.60	50.54	30.00	67.00
iGoogle	3	63.20	8.79	0.2340	55.35	72.20	35.00	72.00
PopFly	3	56.40	11.30	0.1868	43.98	65.50	28.00	69.50
OpenKapow and RoboMaker	5	49.69	7.32	0.1072	40.50	58.50	39.00	71.00
Apple Dashboards	6	67.50	14.18	0.0579	45.50	70.25	40.00	87.00

which is discussed as a future line of research at the end of this article.

These results have been subject to several statistical studies, such as analysing which qualitative and quantitative variables describe the user characteristics that have most impact on the study variable, in this case, a Boolean variable indicating whether or not the user solved the problem. As an additional explanatory variable, we included the user group (and therefore the tool they used). An ANCOVA of these qualitative and quantitative variables suggests that the only factor that appears to affect the success or failure of the practical workshop is the tool used. This ANCOVA is reported at http://apolo.ls.fi.upm.es/eud/eud_paradigm_evaluation.pdf and is not reproduced here for reasons of space.

The data show that, statistically speaking, FAST is more successful than the other languages and tools. From the qualitative analyses and observations that we made during the experiment, we can say that this success is due largely to the levels of abstraction of the FAST components. They make the job of analysing a problem and developing a solution much easier because users start off with very high-level components. These coarse-grained parts do not necessarily have to be defined at this stage, and users can gradually adapt each component. This facilitates top-down development, where users can focus exclusively on the components that they know how to develop from the components of the catalogue that they have consulted. All the components of the other tools have the same level of abstraction, and users have to use very specific detailed components to perform the task. This can stump end users who are initially unable to devise a bottom-up solution from concrete catalogue elements. End users find it very hard to single out these components from all the other elements and are easily deterred.

Note that we analysed all the compositions that were shown by the study to be successful solutions to the stated problem internally. We inspected the composition languages and low-level code developed for each application. We ran white-box tests and found that applications met the set requirements, and all had similar computational efficiency, response times and robustness, irrespective of the visual language used. These tests and validation are reported in Hoyer et al. (2010). In some cases, the compositions had dataflows that the users had added to solve aspects that were not specified in the statement or were optional. These dataflows led to longer response times, albeit not to problems of robustness.

As regards the times taken to develop a solution with each tool, we found that all groups took from 45 to 67 min to solve the problem, except the group using FAST, which took less than 40 min on average. Again we consider that the use of abstract screens (instead of low-level elements) at the start of the design helps the user to make faster progress at the early development stages. As the successful samples for the other tools are very small (the number of users that achieved the goal are nowhere near 30), we cannot use ANOVA or any other traditional statistical tool to check whether the time differences are statistically significant. Therefore, we have used a valid descriptive technique for small data samples, known as box plotting, shown in Fig. 15.

Fig. 15 shows that the box plots charting the data for each of the existing tools and FAST do not overlap, meaning that there is a latent statistical difference in the development times taken using each tool. The workload was largest for Apple Dashboards, followed by iGoogle, PopFly, OpenKapow and finally Yahoo! Dapper and Pipes. The framework that took the least development time was FAST, which had a 4-min advantage over the best of the existing tools. This, together with the fact that FAST substantially increased the number of users that built a successful solution, is a sign of the efficiency of the proposed approach.

Apart from examining which users did or did not come up with a solution, how long it took them and how they did it (observing their work during the experiment), end users were surveyed about their impressions, opinions and experience. We designed the questionnaire illustrated in Table 5, which each member of each group completed at the end of the experiment, stating their individual opinion and impression of the EUD tool used by their work group. They rated each question on a five-point Likert scale indicating user satisfaction with the respective statement. We conducted a questionnaire consistency study to assure that the survey was valid. This study is reported in http://apolo.ls.fi.upm.es/eud/survey_justification.pdf and is not reproduced here for reasons of space. This study justified the use of each item and replicated questions with different statements to check whether responses are mediated or contradictory, etc. The

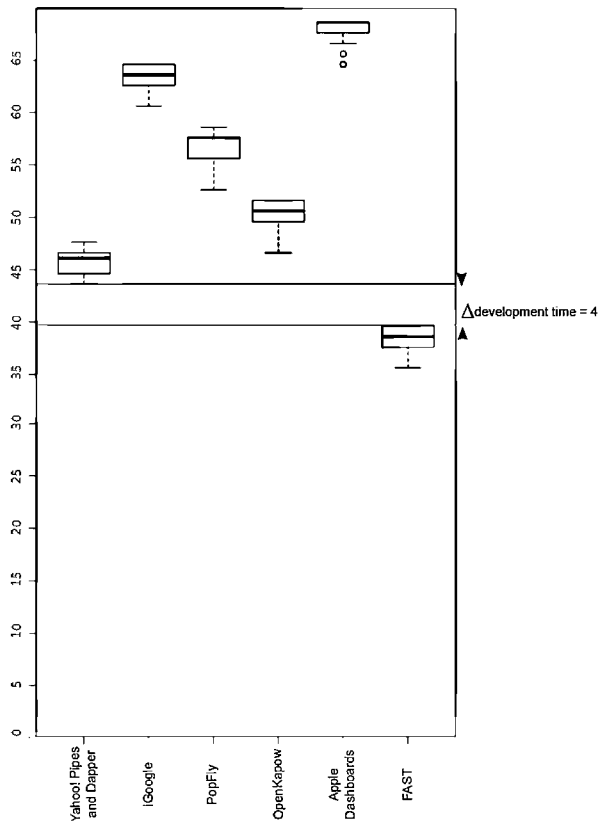

Fig. 15. Development time spent with each language.

Table 5
Main questionnaire questions.

No.	Question	Mean FAST	Mean Yahoo!	Mean iGoogle	Mean PopFly	Mean Kapow	Mean Apple
Usability							
Q1	The tool was very easy to use first time round	4.18	2.21	3.01	3.40	2.05	3.22
Q2	I would imagine that most people would learn to use this tool quickly	4.15	3.18	3.00	3.35	2.50	3.16
Q3	I felt confident using the tool	4.13	2.75	3.00	2.18	1.20	2.50
Q4	I didn't need to do a lot of learning before I could use the tool effectively	4.42	3.20	4.00	3.80	2.30	1.50
Functionality							
Q6	The relevant visual components were easy to find	4.02	3.75	2.80	3.60	2.07	3.20
Q7	The screenflow of a composite application was easy to model	4.12	2.00	1.20	2.05	1.00	2.35
Q8	Inputs and outputs were easy to define	4.18	2.36	3.19	3.05	2.79	3.05
Q9	The designed composite applications were easy to publish	4.06	3.90	3.80	4.50	4.20	3.90
Performance							
Q14	The system quickly responded to inputs	4.08	4.20	4.00	4.20	4.10	3.80
Q15	The system was stable	4.72	4.07	4.70	4.60	4.71	3.70
General							
Q16	I was able to create the relevant screens for the problem statement	4.01	3.20	3.10	2.28	3.12	2.00
Q17	The task was easy	4.30	1.50	1.30	2.00	1.90	1.50

mean score for each question on the scale of 1–5 is stated at the side.

The results reveal that all, even the unsuccessful, users have a positive and better impression of the FAST system than of all the other tools on the examined points. End users give very positive responses to all questions about FAST (which they mostly rated from four to five points), whereas the other tools were rated worse.

FAST usability was rated positively because participants located the components that they needed and found FAST easy to use. Users of other tools had the impression that they were not knowledgeable enough to solve the problem using the provided component model and techniques, as detailed in the responses to the open questions analysed later.

From a functionality perspective, respondents also rated FAST higher than the other tools, although they sometimes found it hard to find the right screen to use. Additionally, most users were satisfied with input and output port definitions. Then again, users found the procedure for publishing designed composite applications on a target platform easy. For the other tools, functionality was rated slightly negatively. Users stated that the components to which they had access were either useless or technically too sophisticated for them to use.

Regarding performance, most tools received a positive rating. FAST was stable without any critical exception throughout the entire evaluation time frame. Participants also felt at ease with the FAST terminology. This means that our composition system and its visual language are both intuitive. No stability failures, execution errors or similar were reported for the other tools. Users stated that the solutions were hard to create from the fine-grained components used in the existing EUD approaches. Noteworthy is the fact that users completed many exercises during the eight hours of training that each group received, and were therefore acquainted with the respective tools.

Apart from the questions listed in Table 5, qualitative analyses of each work group were conducted: users were asked open questions about the problems that they had with the tool they used, stumbling blocks that they were unable to negotiate, etc. The responses to these open questions are documented at http://apolo.ls.fi.upm.es/eud. The conclusions of a detailed analysis of these open questions are as follows:

- Regarding Yahoo! tools, 88% of the group that used the tool stated that they had great difficulty interconnecting Yahoo! Dapper widgets with each other, whereas 76% found it very hard to compose 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. The analysis carried out by the project supervisors revealed that the component inputs and outputs were based on MIME and XSD, which are completely foreign to users. Therefore, the 28 users that failed to develop a solution did so because they were unable to interconnect resources with each other or even build a working screen of use for problem solving.
- Regarding iGoogle, 90% of the group that used the tool stated that they had difficulty establishing a correct dataflow among widgets. iGoogle widgets produce events which can be consumed by particular slots of other widgets. The problem is that, to do this, it has need of a basic component which it uses in the background. This component acts like a data-sharing blackboard, message-based middleware that has to be configured and parameterized to manage the intercommunication. A total of 27 users did not manage to properly parameterize this MOM in order to interconnect the widgets that they required to solve the problem.
- Regarding PopFly, 90% of the group that used this tool criticized the fact that they had trouble finding the right elements for this problem in the catalogues. A total of 27 users were unable to enact the bottom-up process proposed by PopFly for developing EUD. The tool builds data integration schemas based on the low-level components, which are necessarily the starting point. Unless users know which operators to use, the schemas do not provide the necessary solutions to the problem.
- Regarding OpenKapow and RoboMaker (an auxiliary tool supporting OpenKapow), over 80% of the sample found that Kapow component linking and tailoring mechanisms were not handy (required programming knowledge), whereas 85% found that the component search, location, parameterization and recommendation mechanisms were hard to use and understand. The experiment monitors found that the 25 end users that failed to solve the problem had difficulty with the specification of the component control flow using simple pseudocode. Although the code is very basic, users are unfamiliar with loop control structures, stop conditions, etc., and this was why the 25 users were unable to find a solution.
- Regarding Apple Dashboards, 80% of the group that used the tool stated that the visual composition interface should not be confined merely to linking visual elements, as it is not possible to parameterize or adapt the components to new situations or problems. Additionally, 75% of the sample found it impossible to establish the correct dataflow among the different components

for the problem. Because they had to internally modify a visual component and add other operators and basic components to alter the component behaviour, 24 users were unable to go ahead with their design. They viewed each component as an end product rather than a mere prototype, as they were regarded by the tool. On this ground, they were unable to redesign the components using other more basic components.
- Regarding FAST, users were also asked about their impressions and experiences. All 13 users that did not manage to build a successful solution using FAST referred to the same problem: they were unable to find certain components in order to generate the necessary dataflows. Although users received visual guidance to create a dataflow, the monitors found that it is very hard for users to relate specific requirements to components in the catalogue. Once they have selected the components, users find it easy to get on with the design, but this choice should be facilitated by an analysis wizard.

Analysing these results, it appears that Yahoo! Pipes and Dapper targets programmers with basic knowledge of web programming, and non-programmer users will come up against insurmountable obstacles for creating a dataflow among more complex operators. iGoogle offers specific visual widgets that end users can relate to given functional requirements, but it is very hard for non-programmer users to create a dataflow between the widgets because this requires additional parameterization. PopFly relies on a preliminary bottom-up design which is used to build a solution based on its constituent components. This approach has proved to be far from straightforward for non-programmer users who find that the catalogue contains too many similar components or unfamiliar components that they do not know how to use (filters, operators, RSS, etc.). OpenKapow tools require knowledge of algorithms, and, although they use pseudocode, users need to know what a loop is, what a stop condition is, what a variable is and how to iterate based on a variable value. Development using Apple Dashboard is based on iterative and incremental prototyping. However, the prototypes cannot be adapted visually, and XML and J2EE servlet container operations are required. This is all very abstract and complex knowledge for end users.

The results show statistically that FAST component- and connector-based visual composition framework usability, functionality and performance are high. The overall impression is that end users rate FAST positively as returning satisfactory results, suggesting that the composition system has the potential to enable non-programmer end users to develop composite web applications. We can conclude that the composition model implemented in FAST (and which includes the component model, composition techniques and languages described here) has two advantages over other existing EUD tools: many more non-programmer users using FAST can successfully create a solution to a problem, which they do faster than the users of other tools.

8. Conclusions, limitations and future trends

In this paper, we present a visual composition framework. End users with problem expertise but without previous programming knowledge can use this framework to leverage existing web services and resources in order to build their own composite solutions to their problems.

Our research has revealed that our approach, implemented in FAST, is more effective (more users are able to solve their problem) and more efficient (users solve the problem faster) than existing tools. It empowers non-programmer users to build their own applications. Such applications are valuable since they leverage user domain expertise in a short development time and with low development costs.

The proposed approach has some limitations that should be considered in the future with a view to achieving better results as to software solution development by end users:

- The creation of this visual development tool is only the first step towards the solution of a broader problem (Anon., 2003): how to provide full support to help non-programmer users solve complex problems requiring the use of existing IoS services. This toolkit should be supported by a *formalized groundwork* to validate and standardize the development environment, its implementation process and the produced resources (Wulf et al., 2008). Formalization could lead to a common conceptualization of the development process and composition techniques (Obrenovic and Gasevic, 2009). Ultimately, this specification is the first step towards a global standardization process that could enable, through proper interconnection, the joint management of resources created by different tools and IDEs, no matter what their source.
- Another limitation is that although the composite application built using our approach is platform independent, the full software or mashup generated from this application on a mashup platform in order to solve a more complex problem cannot be exported to other platforms. In our example, the composite application for searching and booking of hotels can be used on EzWeb, Netvibes, iGoogle, etc. But if the user generates the full travel management application on any of these platforms, the generated RIA will only be executable in the environment for which it was designed. This is because each platform has a different API and programming resources and technologies. All manufacturers set out to sell their own solution by making software built on their platform incompatible with competitor platforms.

With respect to future work, note that 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 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 (Anon., 2009). The more components there are, the more likely users are to make use of this type of tools. A future line of work is to study how to trigger this escalation of mutual interests, and how to fill existing catalogues with more and more components, using an automatic component adaptor.

Also, we are working on the definition of a taxonomy of forms and operators. Our goal is to find a set of common visual patterns present in both the actual web application UIs and the service front ends. These patterns would be a great seed for building a repository of visual UI components that could be exploited to tailor any service front end to any requirements through reuse and connection.

Acknowledgements

This work was partially supported by the European Commission under the first call of its Seventh Framework Programme (FAST STREP Project, grant INFSO-ICT-216048) and by the European Social Fund and UPM under their researcher training programmes.

References

Anderson, C., 2006. The Long Tail: Why the Future of Business Is Selling Less of More. Hyperion, New York, USA.

Anon., April 2001. Enterprise Applications – Adoption of E-Business and Document Technologies: 2000–2001 North America Executive Summary. Tech. Rep. AIIM BookStore and Gartner.

Anon., 2003. Web Services Composite Application Framework (WS-CAF) TC, http://www.oasis-open.orghttp://www.oasis-open.org.

Anon., August 2006. Hype Cycle for Software as a Service. Gartner Research, Gartner Inc., Stamford, USA.

Anon., 2008. Services Sciences, Management and Engineering. IBM North America, Faculty Press, New York, USA http://www.research.ibm.com/ssme/

Anon., May 2009. Building the front end of the future internet of services. Technical Report. Service Front End Open Alliance, Switzerland.

Aragao, V.R., Fernandes, A.A.,2003. Conflict resolution in web service federations. In: Proceedings of the International Conference on Web Services (ICWS-Europe 2003), vol. 2853 of LNCS. Springer, pp. 109–122.

Assmann, U., 2003. Invasive Software Composition. Springer-Verlag New York Inc., New York, USA.

Ceri, S., Daniel, F., Matera, M., Facca, F., 2007. Model-driven development of context-aware web applications. ACM Trans. Internet Technol. 7 (1).

Crockford, D., 2006. The application/json media type for JavaScript object notation (JSON).

Davenport, T.H., 2005. Thinking for a Living: How to Get Better Performance and Results from Knowledge Workers. Harvard Business Press, Boston, MA.

Fielding, R.T., 2000. Architectural styles and the design of network-based software architectures. University of California, Irvine (Ph.D. thesis).

Forgy, C., 1982. Rete: a fast algorithm for the many pattern/many object pattern matching problem. Artif. Intell. 19 (1), 17–37.

Fukunaga, A., Pree, W., Kimura, T.D., 1993. Functions as objects in a data flow based visual language. In: Proceedings of the 1993 ACM Conference on Computer Science, pp. 215–220.

Govindaraju, M., et al., 2003. Merging the CCA component model with the OGSI framework. In: CCGrid03 Proceedings, vol. 5(8), pp. 182–189.

Hoyer, V., Fuchsloch, A., Kramer, S., Moller, K., López, J., February 2010. Evaluation of the implementation. Tech. Rep. D6.4.1, FAST Consortium. https://files.morfeo-project.org/fast/public/M24/D6.4.1_ScenarioEvaluation_M24_Final.pdf

Kovse, J., Harder, T., 2002. Generic XMI-based UML model transformations. In: Bellahsene, Z., Patel, D., Rolland, C. (Eds.), Object-Oriented Information Systems, vol. 2425 of Lecture Notes in Computer Science. Springer Berlin, Heidelberg, pp. 183–190.

Lieberman, H., Paternò, F., Wulf, V. (Eds.), 2006. End User Development. Springer, Berlin, Germany.

Lizcano, D., Soriano, J., Reyes, M., Hierro, J.J.,2008. EzWeb/FAST: reporting on a successful mashup-based solution for developing and deploying composite applications in the upcoming web of services. In: ACM Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services, iiWAS 2008. ACM, pp. 15–24, ISBN 978-1-60558-349-5.

Lizcano, D., Jiménez, M., Soriano, J., Cantera, J.M., Reyes, M., Hierro, J.J., Garijo, F., Tsouroulas, N., 2008. Leveraging the upcoming internet of services through an open user-service front-end framework. In: Towards a Service-Based Internet. Proceedings of the ServiceWave 2008 Conference, vol. 5377 of Lecture Notes in Computer Science, iSSN 0302-9743, ISBN 10 3-540-89896-4.

Lizcano, D., Jiménez, M., Soriano, J., Hierro, J.J., Martínez, A.L., 2009. The morfeo open source community: building technologies of the future web through open innovation. In: Proceedings of the 18th International World Wide Web Conference.

Myers, B.A., 1990. Taxonomies of visual programming and program visualization. J. Vis. Lang. Comput. 1 (1), 97–123.

Narayanan, N., Hbscher, R., 1997. Visual language theory: towards a human-computer interaction perspective. In: Marriot, K., Meyer, B. (Eds.), Visual Language Theory. Springer-Verlag, Berlin, Germany, pp. 85–127.

Nestler, T., Feldmann, M., Preussner, A., Schill, A., 2009. Service composition at the presentation layer using web service annotations. In: First International Workshop on Lightweight Integration on the Web (ComposableWeb 2009), pp. 63–68.

Obrenovic, Z., Gasevic, D., 2009. Mashing up oil and water: combining heterogeneous services for diverse users. IEEE Internet Comput. 13 (6), 56–64, http://dx.doi.org/10.1109/MIC.2009.97.

Papazoglou, M.P., Georgakopoulus, D., 2003. Service-oriented computing. Commun. ACM 46 (10), 25–28.

Pautasso, C., Alonso, G., 2003. Visual composition of web services. In: Proceedings of the Twelfth International World Wide Web Conference, pp. 92–99.

Pautasso, C., 2004. A flexible system for visual service composition. Swiss Federal Institute of Technology Zurich (Ph.D. thesis).

Prud'Hommeaux, E., Seaborne, A., et al., 2006. SPARQL query language for RDF, W3C working draft 4.

Reyes, M., García, F., López, J., Fuchsloch, A., February 2010. Fast complex gadget architecture. Tech. Rep. D3.1.2, FAST Consortium. https://files.morfeo-project.org/fast/public/M24/D3.1.2_FASTComplexGadgetArchitecture.pdf

Rumbaugh, J., Jacobson, I., Booch, G., 2004. Unified Modeling Language Reference Manual. The Addison-Wesley Object Technology Series, 2nd edition. Addison-Wesley Professional, Boston, USA.

Scaffidi, C., Shaw, M., Myers, B.A.,2005. Estimating the numbers of end users and end user programmers. In: VLHCC'05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing. IEEE Computer Society, Washington, DC, USA, pp. 207–214, http://dx.doi.org/10.1109/VLHCC.2005.34.

Schneider, J.G., 1999. Components, scripts, and glue. Universität Bern (Ph.D. thesis).

Sendall, S., Kozaczynski, W., 2003. Model transformation: the heart and soul of model-driven software development. IEEE Software, 42–45.

Shneiderman, B.,2003. Promoting universal usability with multi-layer interface design. In: CUU'03: Proceedings of the 2003 Conference on Universal Usability. ACM, New York, NY, USA, pp. 1–8, http://dx.doi.org/10.1145/957205.957206.

Wong, J., Hong, J.I.,2007. Making mashups with marmite: towards end-user programming for the web. In: CHI'07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, New York, NY, USA, pp. 1435–1444, http://dx.doi.org/10.1145/1240624.1240842.

Wulf, V., Pipek, V., Won, M., 2008. Component-based tailorability: enabling highly flexible software applications. Int. J. Hum. Comput. Stud. 66 (1), 1–22.