Chapter 5: From Mashup Technologies to Universal Integration: Search Computing the Imperative Way

Florian Daniel, Stefano Soi, and Fabio Casati

University of Trento - Via Sommarive 14, 38123 Trento - Italy
`{daniel,soi,casati}@disi.unitn.it`

Abstract. Mashups, i.e., web applications that are developed by integrating data, application logic, and user interfaces sourced from the Web, represent one of the innovations that characterize Web 2.0. Novel content wrapping technologies, the availability of so-called web APIs (e.g., web services), and the increasing sophistication of mashup tools allow also the less skilled programmer (or even the average web user) to compose personal applications on the Web. In many cases, such applications also feature search capabilities, achieved by explicitly integrating search services, such as Google or Yahoo!, into the overall logic of the composite application.

In this chapter, we first overview the state of the art in mashup development by looking at which technologies a mashup developer should master and which instruments exist that facilitate the overall development process. Then we specifically focus on our own mashup platform, *mashArt*, and discuss its approach to what we call universal integration, i.e., integration at the data, application, and user interface layer inside one and the same mashup environment. To better explain the novel ideas of the platform and its value in the context of search computing, we discuss an example inspired by the idea of search computing.

1 Introduction

The advent of Web 2.0 led to the participation of the user into the content creation and application development processes, also thanks to the wealth of social web applications (e.g., wikis, blogs, photo sharing applications, etc.) that allow users to become an active contributor of content rather than just a passive consumer, and thanks to *web mashups* [1]. Mashup tools enable fairly sophisticated development tasks inside the web browser. They allow users to develop their own applications starting from existing content and functionality. Some applications focus on integrating RSS[1] or Atom[2] feeds, others on integrating RESTful services [20], others on simple UI widgets, etc. Mashup approaches are innovative especially in that they tackle integration at the user interface level and do not "just" focus on data and in that they aim at simplicity more than robustness or completeness of features (up to the point to enable also non-professional programmers to develop own mashups).

[1] `http://cyber.law.harvard.edu/rss/rss.html`
[2] `http://www.ietf.org/rfc/rfc4287.txt`

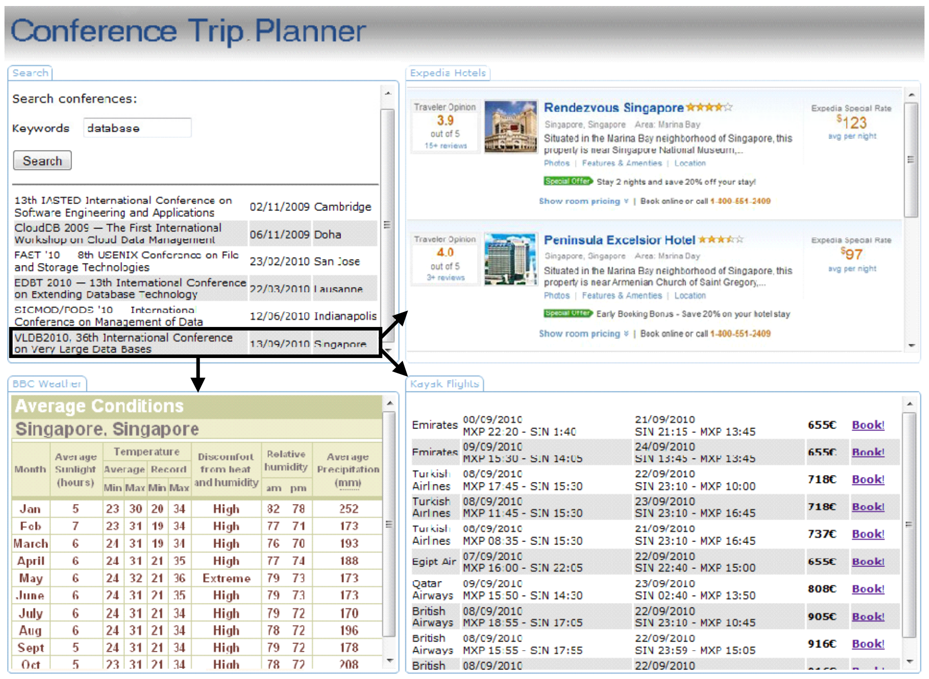
Integrating content and services from the Web also means integrating *search results* or *services*, which makes mashups a natural candidate for search computing applications, but also poses novel requirements in terms of composition features – especially as for what regards UIs.

Inspired by and building upon research in SOA and capturing the trends of Web 2.0 and mashups, this chapter introduces the concept of *universal integration*, that is, the creation of composite web applications that integrate data, application, and user interface (UI) components, effectively enabling the imperative development of advanced search computing applications. Our aim is to do what service composition has done for integrating services, but to do so at all layers, not just at the application layer, and remove some of the limitations that constrained a wider adoption of workflow/service composition technologies. Universal integration can be done (and is being done) today by joining the capabilities of multiple programming languages and techniques, but it requires significant efforts and professional programmers. In this chapter we provide abstractions, models and tools so that the development and deployment of universal compositions is greatly simplified, up to the extent that even non-professional programmers can do it in their web browser.

Scenario. As a reference scenario throughout this chapter, we reuse the conference search scenario described in [18], based on the search query "*find all database conferences in the next six months in locations where the average temperature is 28°C degrees and for which a cheap travel solution including a luxury accommodation exists*". Answering this request requires (i) finding interesting conferences; (ii) understanding whether the conference location is served by low-cost flights; (iii) finding luxury hotels close to the conference location with available rooms; and (iv) checking the expected average temperature of the location. Instead of automatically deriving a query plan to answer the request, in this chapter we focus on how the request can be answered through a composite application for the Web that interactively involves the user into the search process.

The screenshot in Figure 1 shows how such a *Conference Trip Planner* (CTP) application could look like. The application is composed of a variety of different components: In the upper left corner we have a *Conferences Search* component that allows the user of the application to specify a query string and to search for conferences that satisfy the query; retrieved results are displayed below the search form. This is a so-called UI component, as – besides supporting the conference search function – it also comes with its own UI, which is reused as-is by the composite application. Similarly, in the lower left corner, we have a *BBC Weather* UI component that shows the average weather conditions for a selected city, and in the upper right corner we have an *Expedia Hotel* UI component that provides a list of hotels given the name of a city. Finally, in the lower right corner, we have an *RSS Reader* UI component that displays a list of possible flight connections from Milano to the destination city.

The four UI components are synchronized via the *Conferences Search* component, which represents the entry point for the evaluation of the overall "search query", i.e., the content displayed by the UI components. Specifically, by selecting an event of interest from the retrieved conferences, the user synchronizes the content of the other UI components in the page, resulting in a re-computation of the weather, hotel and flight components. By clicking on the proposed hotels or flights, the user is directly forwarded to the respective booking pages, where he/she can conclude the booking.


Conference Trip Planner

Search

Search conferences:

Keywords database

Search

13th IASTED International Conference on Software Engineering and Applications	02/11/2009	Cambridge
CloudDB 2009 — The First International Workshop on Cloud Data Management	06/11/2009	Doha
FAST '10 8th USENIX Conference on File and Storage Technologies	23/02/2010	San Jose
EDBT 2010 — 13th International Conference on Extending Database Technology	22/03/2010	Lausanne
SIGMOD/PODS '10 International Conference on Management of Data	12/06/2010	Indianapolis
VLDB2010, 36th International Conference on Very Large Data Bases	13/09/2010	Singapore

Expedia Hotels

Traveler Opinion 3.9 out of 5 15+ reviews

Rendezvous Singapore ★★★★☆

Singapore, Singapore Area: Marina Bay

Situated in the Marina Bay neighborhood of Singapore, this property is near Singapore National Museum,...

Photos | Features & Amenties | Location

Special Offer Stay 2 nights and save 20% off your stay!

Show room pricing | Book online or call 1-800-551-2409

Expedia Special Rate $123 avg per night

Traveler Opinion 4.0 out of 5 3+ reviews

Peninsula Excelsior Hotel ★★★½☆

Singapore, Singapore Area: Marina Bay

Situated in the Marina Bay neighborhood of Singapore, this property is near Armenian Church of Saint Gregory,...

Photos | Features & Amenties | Location

Special Offer Early Booking Bonus - Save 20% on your hotel stay

Show room pricing | Book online or call 1-800-551-2409

Expedia Special Rate $97 avg per night

BBC Weather

Average Conditions

Singapore, Singapore

Month	Average Sunlight (hours)	Temperature				Discomfort from heat and humidity	Relative humidity		Average Precipitation (mm)
		Average		Record					
		Min	Max	Min	Max		am	pm	
Jan	5	23	30	20	34	High	82	78	252
Feb	7	23	31	19	34	High	77	71	173
March	6	24	31	19	34	High	76	70	193
April	6	24	31	21	35	High	77	74	188
May	6	24	32	21	36	Extreme	79	73	173
June	6	24	31	21	35	High	79	73	173
July	6	24	31	21	34	High	79	72	170
Aug	6	24	31	21	34	High	78	72	196
Sept	5	24	31	21	34	High	79	72	178
Oct	5	23	31	21	34	High	78	72	208

Kayak Flights

Emirates	00/09/2010 MXP 22:20 - SIN 1:40	21/09/2010 SIN 21:15 - MXP 13:45	655€	Book!
Emirates	09/09/2010 MXP 15:30 - SIN 14:05	24/09/2010 SIN 13:45 - MXP 13:45	655€	Book!
Turkish Airlines	08/09/2010 MXP 17:45 - SIN 15:30	22/09/2010 SIN 23:10 - MXP 10:00	718€	Book!
Turkish Airlines	08/09/2010 MXP 11:45 - SIN 15:30	23/09/2010 SIN 23:10 - MXP 16:45	718€	Book!
Turkish Airlines	08/09/2010 MXP 08:35 - SIN 15:30	21/09/2010 SIN 23:10 - MXP 16:45	737€	Book!
Egipt Air	07/09/2010 MXP 16:00 - SIN 22:05	22/09/2010 SIN 22:40 - MXP 15:00	655€	Book!
Qatar Airways	09/09/2010 MXP 15:50 - SIN 14:30	23/09/2010 SIN 02:40 - MXP 13:50	808€	Book!
British Airways	08/09/2010 MXP 18:55 - SIN 17:05	22/09/2010 SIN 23:10 - MXP 10:45	905€	Book!
British Airways	08/09/2010 MXP 15:55 - SIN 17:55	22/09/2010 SIN 23:59 - MXP 15:05	916€	Book!
British	08/09/2010	22/09/2010		

Fig. 1. Reference scenario: the conference trip planner application. Selecting a conference from the list aligns the content shown by the components in the page.

We assume that the *Conferences Search* component is implemented via a simple, generic search component in conjunction with an external conference search service; in our example, we use a Yahoo! Pipe[3] to search for conferences and filter them according to the user's query. Similarly, we use a standard *RSS Reader* component to visualize flights that are retrieved via the *kayak.com* search engine. For the *BBC Weather* and the *Expedia Hotel* components, instead, we assume that they are both provided as readily usable UI components by the respective companies.

The application in Figure 1 represents only one possible application able to answer the initial query. In fact, other combinations of components and services could be adopted, e.g., using *lufthansa.com* instead of *kayak.com* or switching the position of the weather and the hotel components, but in this chapter we are not interested in identifying the best combination of components (i.e., the best "query plan" using the terminology of [18]). The challenge we address is how to *enable the average web user to compose* an application like the one in Figure 1, relying on his/her own judgment of how components are best glued together.

Approach and Structure of the Chapter. In this chapter we focus on mashups and universal integration for the Web. We first offer an overview of the state of the art in traditional composition technologies (Section 2) and then specifically focus on the

[3] `http://pipes.yahoo.com/pipes`

recent trend of composition on the Web, i.e., mashups (Section 3). Next, we introduce the idea and principles of universal integration (Section 4). As an advanced case study and concrete implementation of the universal integration idea, in Section 5 we focus on mashArt. Specifically, we describe the conceptual and architectural aspects of mashArt, which constitute its innovative contributions in terms of *component* and *composition model*s as well as *development* and *runtime* infrastructure, and show mashArt at work. Section 6 concludes the chapter.

2 Traditional Composition and Development Approaches

Several areas of research are related to (lightweight) composition and mashups on the Web. In this section, we briefly survey the areas of *service composition*, *UI composition*, *computer-aided web engineering tools*, *web portals and portlets*, all areas we feel particularly related to universal composition for the Web. In the next section we then put some more focus on *mashups*.

2.1 Service Composition Approaches

A representative of service orchestration approaches is BPEL [6], a standard composition language by OASIS. BPEL is based on WSDL-SOAP web services, and BPEL processes are themselves exposed as web services. Control flows are expressed by means of structured activities and may include rather complex exception and transaction support. Data is passed among services via variables (Java style). So far, BPEL is the most widely accepted service composition language. Although BPEL has produced promising results that are certainly useful, it is primarily targeted at professional programmers like business process developers. Its complexity (reference [6] counts 264 pages) makes it hardly applicable for web mashups.

Many variations of BPEL have been developed, e.g., aiming at invocation of REST services [7] and at exposing BPEL processes as REST services [8]. In [9] the authors describe Bite, a BPEL-like lightweight composition language specifically developed for RESTful environments. IBM's Sharable Code platform [10] follows a different strategy for the composition of REST or SOAP services: a domain-specific programming language from which Ruby on Rails application code is generated, also comprising user interfaces for the Web. In [11], the authors combine techniques from declarative query languages and services composition to support multi-domain queries over multiple (search) services, while in [21] the authors follow a document-centric approach to service composition and propose the use of AXML for service mashups. All these approaches focus on the application and data layer; UIs can then be programmed on top of the service integration logic. mashArt features instead universal integration as a paradigm for the simple and seamless composition of UI, data, and application components. We argue that universal integration will provide benefits that are similar to those that SOA and process centric integration provided for simplifying the development of enterprise processes.

2.2 UI Composition Approaches

In 12] we discussed the problem of integration at the presentation layer and concluded that there are no real UI composition approaches readily available: Desktop UI component technologies such as .NET CAB [13] or Eclipse RCP [14] are highly technology-dependent and not ready for the Web. Browser plug-ins such as Java applets, Microsoft Silverlight, or Macromedia Flash can easily be embedded into HTML pages; communications among different technologies remain however cumbersome (e.g., via custom JavaScript). Java portlets [15] or WSRP [2] represent a mature and Web-friendly solution for the development of portal applications; portlets are however typically executed in an isolated fashion and communication or synchronization with other portlets or web services remains hard. Portals do not provide support for service orchestration logic.

2.3 Computer-Aided Web Engineering Tools

In order to aid the development of complex web applications, the web engineering community has so far typically focused on model-driven design approaches. Among the most notable and advanced model-driven web engineering tools we find, for instance, WebRatio [16] and VisualWade [17]. The former is based on a web-specific visual modeling language (WebML), the latter on an object-oriented modeling notation (OO-H). Similar, but less advanced, modeling tools are also available for web modeling languages/methods like Hera, OOHDM, and UWE. All these tools provide expert web programmers with modeling abstractions and automated code generation capabilities, which are however far beyond the capabilities of our target audience, i.e., advanced web users and not web programmers.

2.4 Portals and Portlets

Still in the context of web applications, portals and portlets represent a different approach to the UI integration problem on the Web. Their approach explicitly distinguishes between UI components (the portlets) and composite applications (the portals) and it is probably the most advanced approach to UI composition as of today (We use the term “portlets” taken from the JSR-168 portlet specification [15], but our considerations also hold for ASP.NET Web Parts). Portlets are full-fledged, pluggable Web application components that generate document markup fragments (e.g., (X)HTML) and facilitate content aggregation in a portal server. Portlets are conceptually very similar to servlets. The main difference between them consists in the fact that while a servlet generates a complete web page, portlets generates just a piece of page (commonly called fragment) that is designed to be included into a portal page. Hence, while a servlet can be reached through a specific URL, a portlet can only be reached through the URL of the whole portal page. A portlet has no direct communication with the web browser, but this communications are managed by the portal and the portlet container that allow the request-response flows and the communication between portlets. A portal server typically allows users to customize composite pages (e.g., to rearrange or show/hide portlets) and provide single sign-on and role-based personalization.

Today, there are several standards for portlets, JSR-168 being the original specification. JSR-286 introduced inter-portlet communication via a portlet container that manages a publish-subscribe infrastructure that can be used by the portlets. Finally, WSRP [2] also added support for accessing remote portlets as web services over the Web. The portlet model is powerful as for what regards the presentation integration part, yet portals do not naturally support interactions with generic web services or the specification of orchestration logics.

3 Web Mashups

Web mashups somehow address the above shortcomings. Web mashups are web applications that are developed by combining content, presentation, and application functionality from disparate Web sources [1]. The term mashup typically implies easy and fast integration based on open APIs and data sources, yielding applications that add value to the individual components of the application and thereby often use components in ways that differ from the actual reason that led to the original production of the raw sources.

Mashups are strongly related with the Web. The Web is the natural environment for publishing content and services today, and therefore it is the natural environment where to access and reuse them. Content and services are published in a variety of different forms and by using a multiplicity of different technologies; we can categorize the means to source content and services from the Web into three basic groups:

– *Data services* like RSS (Really Simple Syndication) or Atom feeds, JSON (JavaScript Object Notation) or XML resources, or simple text files. A typical example is newspapers and magazines that publish their news headers via RSS or Atom feeds that allow users to easily jump to the respective articles. These simple technologies are used to publish data on the Web that are meant for consumption by machines, not humans. In fact, they focus on the efficient distribution of content, rather than on the effective presentation of such contents to human users. Sourcing data via one of these technologies is typically very simple: it mostly requires accessing an online resource and processing the response. Data services to not have complex interaction patterns to be followed.
– *Web services or public APIs accessible over the Web*, such as SOAP (Simple Object Access Protocol) or RESTful (REpresentational State Transfer) web services or, to a lower degree, Java classes (accessed via the IIOP protocol) or similar. These technologies are used to publish application logic on the Web. Their goal is therefore not just to provide access to contents or data, but also to computing logic (e.g., the processing of an order for a book shop). Typically, the interaction with web services or APIs is ruled by so-called interaction protocols, which state which operations can be invoked, in which order, by which partners, etc. Not following the rules stated by the protocol may impede the correct functioning of the service or API.
– *User interface elements*, such as HTML clips or JavaScript APIs with own user interface (e.g., Google Maps), but also banners or advertisements. Content may also be represented by already formatted and graphically rendered data (typically in HTML). In many cases, accessing such kind of content means extracting them from a web page, as there is no equivalent data service available that can be used to

source the same data. Typically, this occurs without the provider of the contents actually knowing that there is someone extracting data from its web pages. In other cases, e.g., Google Maps, the provider of the contents explicitly publishes its data at the user interface level only.

The very innovative aspect of web mashups is that they integrate sources also at the UI layer, not only at the data and application logic layers. Integration at the data and application logic layers has been extensible studied in the past, while integration at all three layers is still a goal that put architects and programmers in front of important conceptual and technical problems.

Mashup development is still an ad-hoc and time-consuming process, requiring advanced programming skills (e.g., wrapping web services, extracting contents from web sites, interpreting third-party JavaScript code, etc). There are a variety of mashup tools available online, but, as we will see, only few of them adequately address the problem of integration at all its layers. In this section, we give an overview of the state of the art in the mashup world, spanning from manual development to semi-assisted and fully assisted development approaches.

3.1 Manual Development

Developing applications that aggregate data, application logic and UIs coming from diverse sources requires deep knowledge about technologies like: (X)HTML, dynamic HTML, AJAX (Asynchronous JavaScript and XML), RSS, Atom; XML specifications like DTD, XSD, XSLT; protocols like SOAP or HTTP for SOAP and RESTful web services; programming languages like JavaScript, PHP, Ruby, Java, C#, and so on; relational or object-oriented databases, etc. In addition, it might be necessary to master the business protocols of employed services and to have knowledge about how to compose services into service orchestrations. This long and not exhaustive list of technologies highlights how mashing up even a simple application, such as the one in our reference scenario, is a hard and time-consuming task that can only be completed by skilled programmers.

The development of our Conference Trip Planner requires, for instance, the following skills: First of all, the developer needs to understand well the dynamics behind and interaction logic of the *Yahoo! Pipes* and *Kayak* services and the *BBC Weather*, *Expedia Hotels* and *RSS Reader* UI components of the application. In the specific case, *Expedia Hotels* and *BBC Weather* expose JavaScript APIs that allow the developer to use and interact with their services; *Pipes* and *Kayak*, instead, return their output as RSS feeds, which need to be appropriately parsed to extract all the necessary information. While the UI components already come with their own UIs, for the conference and flight search results an ad-hoc user interface has to be developed in HTML. Next, the developer needs to implement the necessary synchronization logic among the *Conferences Search* component and the others, such that on the selection of a conference the other components will coherently update their content. In addition to invoking some JavaScript functions of the UI components, this also implies interacting with the remote search services upon the selection of a conference from the list. Finally, the developer needs to create a suitable layout for the composite application, which is able to accommodate the developed components and to render the final mashup application.

The described situation is already an ideal one: all components provide some kind of componentization. If, instead, we imagine that the developer also needs to develop the components to be mashed up, things get even worse. For instance, it could be necessary to implement a wrapper for the *BBC Weather* component that is able to automatically request weather forecasts for the correct city, to extract the HTML code of the average weather conditions, and to expose a JavaScript interface that allows the interaction with other components in the application. Similar operation would be necessary also for the other components of the application.

3.2 Semi-assisted Development

To speed up and simplify the development especially of components to be mashed up, some useful web tools and frameworks have been recently introduced. Typically, they address the problem of data extraction from web sites and the provisioning of such data in form of data services or re-usable user interface elements. In the following, we analyze two representative tools, i.e., Dapper[4] and Openkapow[5], which are very user-friendly.

Dapper is a free online instrument for the generation of data wrappers that extract data from well-structured web pages. Dapper is based on a point and click technique able to assist the user in the selection of the contents to be extracted and to infer suitable extraction rules (e.g., regular expressions). Specifically, data extraction leverages the structure of the HTML formatting to understand which elements to extract (e.g., the first cells of all the rows in a table). Once properly identified, extracted data fields can be named and structured and then published, for instance, as RSS or XML data services. Published services can easily be accessed via a unique URL and are processed each time the respective URL is accessed.

Openkapow is a similar open service platform based on the concept of extraction robot, that is, user-created wrappers. Users of Openkapow can build their own robots, expose their results via web services, and run them from openkapow.com for free. Robots are able to access web sites and support the extraction and reuse of data, functionality and even pieces of user interfaces. Robots are built through a visual development environment called RoboMaker. RoboMaker allows the user navigate inside the target web site and to define a series of simple steps, each one representing an event in the page, until the target data is reached. The extraction results can be exposed in two main ways: as a RESTful service or as an RSS feed, depending on the extracted content and on the expected use of it. After their publication on the Openkapow servers, robots are accessible through a public URL, which identifies the specific robot to run. So exposed services may also need some input values (e.g., user-id and password) that can be used to parameterize the services. Inputs can easily be passed by appending them to the service URL as name-value pairs, following the standard URL model.

To better understand how these tools can be used in the mashup context, let's refer again to the Conference Trip Planner example. Let us suppose that the *Kayak* flight search site does not have an RSS output for its search results. In this case, a data extraction service can be used to automatically extract the flight combinations from the

[4] `http://www.dapper.net/open/`
[5] `http://openkapow.com`

result page. With Dapper, for instance, a developer needs to load one or more example pages into the Dapper environment. The more example pages are loaded, the better the inferred rules. Then, the developer needs to identify the individual data items he/she wants to extract from the page by clicking on the respective HTML elements (e.g., airline, departure time, arrival time, price, intermediate stops, link to booking), to label them and to assemble the final output (e.g., an RSS feed). There is no need to write any own line of code, in order to publish the extraction results on the Web.

While this kind of tools undoubtedly speeds up the development of data extraction from existing web sites, the development effort regarding the composition of components into a new application remain in unchanged. Therefore, the developer still has to be familiar with the services and APIs to be integrated, to display sourced data in a suitable way, and to manage the communication and synchronization logic between the components. Even assuming that data extraction tools can be successfully used by non-programmers, the final mashup development therefore still remains the hard task that can be performed only by skilled programmers.

3.3 Fully-Assisted Development

The previous analyses and consideration show that mashup development is typically a knowledge-intensive work, involving a variety of technologies and components. In addition to simplifying the creation of data extraction instruments for web pages, which address the problem of developing *components* for mashups, it is important to also aid the actual *composition* of components into applications, which is as hard and time-consuming as developing components, if not properly supported. Mashup tools or mashup platforms address exactly this problem, each of them focusing on different composition aspects and following different mashup approaches. In the following, we analyze four of these tools, which we think are most representative for this kind of assisted mashup development: Yahoo! Pipes, JackBe Presto[6], Microsoft Popfly[7], and Intel Mash Maker[8]. There are also other tools like Google App Engine[9] or IBM's Lotus Mashups[10] and so on, but their discussion exceeds the scope of this chapter.

Yahoo! Pipes provides a simple and intuitive visual editor that allows one to design data-centric compositions. It takes data as input and provides data as output; the most important supported formats are RSS/Atom, XML, and JSON. A pipe is a data processing pipeline in which input data (coming from diverse data sources) are processed, manipulated and used as input for other processing steps, until the target transformation is completed. This pipeline-style process is implemented through an arbitrary number of intermediate operators, which manipulate data items inside the data feeds or provide features like loops, regular expressions or more advanced features like automatic location extraction or connection to external services. The set of operators are predefined and fixed; new functionality can be included in form of web services. Also, stored pipes can be reused as sources of another pipe.

[6] `http://www.jackbe.com/`

[7] `http://popflyteam.spaces.live.com` – MS Popfly has been discontinued since August 24, 2009.

[8] `http://mashmaker.intel.com/web`

[9] `http://code.google.com/intl/it-IT/appengine/`

[10] `http://www-01.ibm.com/software/lotus/products/mashups/`

Yahoo! Pipes' development environment is characterized by a simple and intuitive development paradigm that is however targeted at advanced web users or programmers. In fact, the level of abstraction of its operations (e.g., the regular expression component) and the characteristic data flow logic is only hardly understandable to non-programmers. Pipe's output is not meant for human consumption (RSS, Atom, JSon, etc.) but rather for integration in other applications. This limits both the variety of input sources that can be used and the accessibility of its output. In fact, the absence of any support for UIs prevents the direct use of Pipe's output by common web users. However, Pipes is a very popular data-mashup development tool, very likely due to its efficient and intuitive component placing and connection mechanism.

The development tool does not need any installation or plug-ins; it runs in any AJAX-enabled web browser. The development environment comes with a very efficient, integrated debugging tool that helps the developer during the design phase. Pipes are stored online and accessible via an own URL. When invoking a pipe, an execution process is started on the server side, relieving the client from the execution overhead. This characteristic could represent a problem under a scalability perspective: if a large number of simultaneous accesses to a pipe are made, performance and stability might suffer.

Considering our example application, with Yahoo Pipes it would be unfeasible to realize the application as described in the reference scenario, as there is no support for the user interface of the application. However, what we can do, for instance, is using Pipes to simplify the collection, aggregation and filtering of conferences sourced from different web sources, such as *conference-service.com* and *allconferences.com*. On top of this pipe, it is then necessary to provide a suitable user interface.

JackBe Presto is a robust and complete mashup platform which provides enterprise-level solutions. Presto gives the possibility to easily produce (design, test and deploy) mashups merging data coming from disparate sources. In particular it can be also connected to data sources very common in the business world (like Excel spreadsheets, Oracle data software, etc.), that most of mashup competitor's solutions cannot access. Simple mashup composition can be done, also by non-IT users, through the Presto Wires tool. More advanced composition can be obtained only by professional developers implementing them in EMML language with the support of the Presto Mashup Studio plug-in for Eclipse. This language is the main actor of the OMA (Open Mashup Alliance) project, which aims to define an open language allowing enterprise mashup interoperability and portability.

The development environment is constituted by several independent tools. Wires is a visual editor based on a simple and intuitive data pipeline composition approach. It allows one to merge data coming from disparate internal and external sources producing a final output that can be graphically displayed as a mashlet. Mashlets can be plugged into a dash-board like user interface or a portal, or they can be embedded into a regular web page. Mashlet development is assisted by the Presto Mashlet tool, while the Mashup Studio is an Eclipse plug-in providing Java programmers with complete control on the mashup development process. Connectors allow one to hook up Presto to diverse software, such as Microsoft Excel, web portals, any Oracle technology, and similar. Presto services can be accessed through APIs, available for main programming languages (Java, JavaScript, C#, Python, etc.).

The runtime server provides secure mechanisms to virtualize (abstract the user from actual implementation details) and normalize (put the service output into standard formats: JSON or XML) any kind of service or data (SOAP, REST, RSS, DB, Excel) and expose them in a secure and governed way. Presto is not a hosted service, like Yahoo! Pipes; it needs to be installed and configured in each company individually.

Let us briefly analyze the possibility to create our Conference Trip Planner application with Presto. Just like Yahoo! Pipes, Wires gives the opportunity to easily access, merge and filter the RSS channels of the conferences search services and the Kayak flights search service. Retrieved items can be displayed by means of two mashlets. The development of the other UI components in form of mashlets has to be done manually in Mashup Studio using a standard programming language like Java. At this point the produced mashlets can be put together inside one web page. However, this solution does not provide for the synchronization of the basic components in the application (the mashlets), so that the selection of a conference updates the data shown in the other components. There is not inter-mashlet communication.

Microsoft Popfly gained a great consensus in the mashup community and achieved good levels of popularity and usage. Although the Popfly project has been discontinued, we analyze this mashup tool because we consider it an interesting example for UI composition with peculiarities that cannot be found in other tools.

Popfly provides a visual development environment for the realization of mashups based on the concept of components, or *block* as they are called in Popfly. A composition is created by dragging and dropping blocks of interest onto a design canvas and by graphically connecting them to create the desired application logic. A block can take the role of connector to external services or it can represent some internal functionality (implemented through a JavaScript function). Each block provides input and output ports that enable its connection to other blocks. Blocks can also be used to provide a user interface that can display the result of some processing. Placing multiple visualization blocks into a same page allows one to define the overall layout of the page. The internal layout of blocks can be customized by inserting ad-hoc HTML, CSS or JavaScript code. Popfly has a wide collection of available blocks, offering functionalities like RSS readers, service connectors, map components based on Virtual Earth, etc. New blocks and compositions can be defined (in JavaScript), saved, shared and managed in a dedicated section of the platform.

At runtime, the communication flow is event-driven, that is, the activation of a certain component depends on the raising of some event by another component. There is no support for exception and transaction handling, but Popfly provides a section dedicated to the test and preview of the composition. Ready compositions are stored on the Popfly server, but the execution is done on the client – as many of the built-in blocks are based on the Silverlight platform. The client-side execution of mashups alleviates the server from heavy loads and limits scalability and performance.

Considering the Conference Trip Planner application, Popfly is the first tool that can be used to fully implement the application. We assume that skilled programmers already developed and published all blocks needed for the composition, especially the UI components *Conferences Search*, *Expedia Hotels* and *BBC Weather*, while the RSS reader necessary to display the output of the conference and flight search services already exists. At this point, the developer of the composition can drag and drop these components onto the modeling canvas and connect the blocks, also providing for the necessary mapping of the data parameters from outputs to inputs. In particular, the *Conferences Search* block must be connected to all the other blocks, in order to provide for the synchronization of the whole composition. Finally, the graphical appearance of the application's layout can be set up by including a custom CSS style sheet into the page. What is missing in Popfly is the possibility to define more complex, process-like service compositions, as could for example be needed to process the conference search results directly in Popfly.

Intel Mash Maker provides a completely different mashup approach: an environment for the integration of data from annotated source web pages based on a powerful, dedicated browser plug-in for the Firefox web browser. Rather than taking input from structured data sources such as RSS/Atom feeds or web services, Mash Maker allows users to reuse entire web pages and, if suitably annotated, to extract data from the pages. That is, the "components" that can be used in Mash Maker are standard web pages. If a page has been annotated in the past, it is possible to extract the annotated data from the page and share it with other components in the browser. If the page has not been annotated, it is possible reuse the page as is without however supporting any inter-page communication.

In order to annotate a page, Mash Maker allows developers and users to annotate the structure of web pages while browsing and to use such annotations to scrap contents from annotated pages. Advanced users may leverage the integrated Structure Editor to input XPath expressions with the help from FireBug's DOM Inspector (another plug-in for the Firefox web browser). Annotations are linked to target pages and stored on the Mash Maker server in order to share them with other users.

Composing mashups with Mash Maker occurs via a copy/paste paradigm, based on two modes of merging contents: *whole page merging*, where the content of one page is inserted as a header into another page; and *item-wise merging*, where contents from two pages are combined at row level, based on additional user annotations. The two techniques can be used to merge also more than two pages. Data exchange among components is achieved by means of a blackboard-like approach, where data of components integrated into an application are immediately available to all other components. Not only the development, but also the execution of mashups is entirely performed with the help from the browser plug-in at the client side; on the server side there are only the annotations for data extraction and the stored mashup definitions.

To build the Conference Trip Planner with MashMaker, first we need to devise the necessary components in form of annotated web pages. For instance, instead of using the RSS interface toward the conference search services or toward the flight search service, we need to navigate the respective web sites and annotate the data items that are necessary to answer our reference query. Similarly, we need to annotate the UI components of our application. Next, all these individual pieces of HTML markup and annotations must be joined following an item-wise merging strategy. It is possible to implement the needed synchronization mechanisms to coordinate the components of the application with each other by means of sophisticated merge operations. The whole development procedure is a non-trivial and time-consuming, it requires some non-intuitive skills to annotate, decompose, merge and reconstruct pages and web applications of arbitrary complexity. Without advanced programming skills it is hard to implement the synchronization of components upon selection of a conference.

4 Universal Composition: Guiding Principles

As highlighted above, although existing mashup approaches have produced promising results, techniques that cater for simple and universal integration of web components at all the three layers of the application stack are still missing. We think such techniques are necessary to transition Web 2.0 programming from elite types of computing environments to environments where users leverage simple abstractions to create composite web applications over potentially rich web components developed and maintained by professional programmers.

We aim at *universal integration*, and this has fundamental differences with respect to traditional composition. In particular, the fact that we aim at also integrating UI implies that:

(i) *Synchronization*, and not (only) orchestration a-la BPEL, should be adopted as interaction paradigm;
(ii) Components must be able to react to both *human user input* and *programmatic interaction*;
(iii) We must be able to design the *user interface* of the composite application, not just the behavior and interaction among the components.

This shows the need for a model based on state, events and synchronization more than on method calls and orchestration. We recognize in particular that events, operations, a notion of state and configuration properties are all we need to model a *universal component*.

On the data side, we realize that *data integration* on the Web may also require different models: for example RSS feeds are naturally managed via a pipe-oriented data flow/streaming model (a-la Yahoo Pipes) rather than a variable-based approach as done in conventional service composition.

Another dimension of universality lies in the *interaction protocols*. As there might be a variety of components and component implementations, we must be able to deal with multiple communication protocols at the same time. For instance, the most used protocols on the Web are REST/HTTP, SOAP, RSS, Atom, and JSON.

These requirements are often at odds with the other key design goal we have: *simplicity*. We want to enable advanced web users to create applications (an old dream of service composition languages which is still somewhat a far reaching objective). This means that the universal composition paradigm must be fundamentally simpler than programming languages and current composition languages. As an example, we target the complexity of creating web pages with a web page editor, or the complexity of building a pipe with Yahoo Pipes (something that can be learned in a matter of hours rather than weeks).

5 The mashArt Platform

To achieve simplicity in mashArt, we make three design decisions: First, mashArt aims at *hiding* the complexity of the specific protocol or data model supported by each component. That is, the goal is that from the perspective of the composer all these specificities are hidden – with the exceptions of the aspects that have a bearing on the composition (e.g., if a component is a feed, then we are aware that it operates, conceptually, by pushing content periodically or on the occurrence of certain events).

As a second decision, we keep the composition model *lightweight*: for example, there are no complex exception or transaction mechanisms, no BPEL-style structured activities or complex dead-path elimination semantics. This still allows a model that makes it simple to define fairly sophisticated applications. Complex requirements can still be implemented but this needs to be done in an "ad hoc" manner (e.g., through proper combinations of event listeners and component logic) but there are no specialized constructs for this. Such constructs may be added over time if we realize that the majority of applications need them.

The third decision is to focus on simplicity only *from the perspective of the user* of the components, that is, the designer of the composite applications. In complex applications, complexity must reside somewhere, and we believe that as much as possible it needs to be inside the components. Components usually provide core functionalities and are reused over and over (that's one of the main goals of components).Thus, it makes sense to have professional programmers develop and maintain components. We believe this is necessary for the mashup paradigm to really take off. For example, issues such as interaction protocols (e.g., SOAP vs. REST or others) or initialization of interactions with components (e.g., message exchanges for client authentication) must be embedded in the components.

In the following, we describe in more detail the component model and the composition model enabling universal integration and the implementation of the mashArt platform with its design-time and runtime support.

5.1 The mashArt Component Model

The first step toward the universal composition model is the definition of a component model. *MashArt* components wrap UI, application, and data services and expose their features/functionalities according to the mashArt component model. The model described here extends our initial UI-only component model presented in 3] to cater for universal components. The model is based on four abstractions: state, events, operations, and properties:

– The *state* is represented as a set of name-value pairs. What the state exactly contains and its level of abstraction is decided by the component developer, but in general it should be such that its change represents something relevant and significant for the other components to know. For example, in our *Conference Search* component we can change the search string of the query and re-compute the list of pertaining conferences; this component-internal activity is irrelevant for the other components who are not interested in such low level of detail. Instead, clicking on (selecting) a specific conference expresses an information that may lead other components to show related information or application services to perform actions (e.g., query for flights). This is a state change we want to capture. In our case study, the state for the *Conference Search* component is the set of conferences being displayed plus the selected conference.

  Modeling state for application components is something debatable as services are normally used in a stateless fashion. This is also why WSDL does not have a notion of state. However, while implementations can be stateless, from a modeling perspective it can be useful to model the state, and we believe that its omission from WSDL and WS-* standards was a mistake (with many partial attempts to correct it by introducing state machines that can be attached to service models). Although not discussed here, the state is a natural bridge between application services and data-oriented services (services that essentially manipulate a data object).

- Events communicate state changes and other information to the composition environment, also as name-value pairs. External notifications by SOAP services, callbacks from RESTful services, and events from UI components can be mapped to events. When events represent state changes, initiated either by the user by clicking on the component's UI or by programmatic requests (through operations, discussed below), the event data includes the new state. Other components subscribe to these events so that they can change their state appropriately (i.e., they synchronize). For instance, when selecting a conference in the Conference Search component, an event is generated that carries details (e.g., name, city, start/end date) about the performed selection.
- Operations are the dual of events. They are the methods invoked as a result of events, and often represent state change requests. For example, the ConferenceSearch component has a state change operation ShowConferences that can be used to display retrieved conferences. In this case, the operation parameters include the necessary information about the state to which the component must evolve (the list of conferences). In general, operations consume arbitrary parameters, which, as for events, are expressed as name-value pairs to keep the model simple. Request-response operations also return a set of name-value pairs – the same format as the call – and allow the mapping of request-response operations of SOAP services, Get and Post requests of RESTful services, and Get requests of feeds. One-way operations allow the mapping of one-way operations of SOAP services, Put and Delete requests of RESTful services, and operations of UI components. The linkage between events and operations, as we will see, is done in the composition model. We found the combination of (application-specific) states, events, and operations to be a very convenient and easy to understand programming paradigm for modeling all situations that require synchronization among UI, application, or data components.
- Finally, configuration properties include arbitrary component setup information. For example, UI components may include layout parameters, while service components may need configuration parameters, such as the username and password for login. The semantics of these properties is entirely component-specific: no "standard" is prescribed by the component model.

In addition to the characteristics described above, components have aspects that are *internal*, meaning that they are not of concern to the composition designer, but only to the programmer who creates the component. In particular, a component might need to handle the invocation of a service, both in terms of mapping between the (possibly complex) data structure that the service supports and the flat data structure of mashArt (name-value pairs), and also in terms of invocation protocol (e.g., SOAP over HTTP). There are two options for this: The first is to develop ad hoc logic in form of a wrapper. The wrapper takes the mashArt component invocation parameters, and with arbitrary logic and using arbitrary libraries, builds the message and invokes the service as appropriate. The second is to use the built-in mashArt bindings. In this case, the component description includes component bindings such as *component/http*, *component/SOAP*, *component/RSS*, or *component/Atom*. Given a component binding, the runtime environment is able to mediate protocols and formats by means of default mapping semantics. In summary, the mashArt model accommodates component models such as UI components, SOAP and RESTful services, RSS and Atom feeds.

In Figure 2(a) we introduce our graphical modeling notation for mashArt components that captures the previously discussed characteristics of components, i.e., state, events, operations, and UI. *Stateless* components are represented by circles, *stateful* components by rectangular boxes. Components with *UI* are explicitly labeled as such. We use arrows to model *data flows*, which in turn allow us to express events and operations: arrows going out from a component are *events*; arrows coming in to a component are *operations*. There might be multiple events and operations associated with one component. Depending on the particular type of operation or event of a stateless service, there might be only one incoming data flow (for one-way operations), an incoming and an outgoing data flow (for request-response operations), or only an outgoing data flow (for events). Operations and events are bound to their component by means of a simple dot-notation: *component.(operation|event)*.

The actual model of a specific component is specified by means of an abstract component *descriptor*, formulated in the *mashArt Description Language* (MDL) a simple, XML-based interface description language. MDL is for mashArt components what WSDL is for web services.

5.2 Universal Composition Model

Since we target universal composition with both stateful and stateless components, as well as UI composition, which requires synchronization, and service composition, which is more orchestrational in nature, the resulting model combines features from *event-based* composition with *flow-based* composition. As we will see, these can naturally coexist without making the model overly complex.

In essence, composition is defined by linking events (or operation replies) that one component emits with operation invocations of another component. In terms of flow control, the model offers conditions on operations and split/join constructs, defined by tagging operations as optional or mandatory. Data is transferred between components following a pipe/data flow approach, rather than the variables-based approach typical of BPEL or of programming languages. The choice of the data flow model is motivated by the fact that while variables work very well for programs and are well understood by programmers, data flows appear to be easier to understand for non-programmers as they can focus on the communication between a pair of components. This is also why frameworks such as Yahoo Pipes can be used by non-programmers.

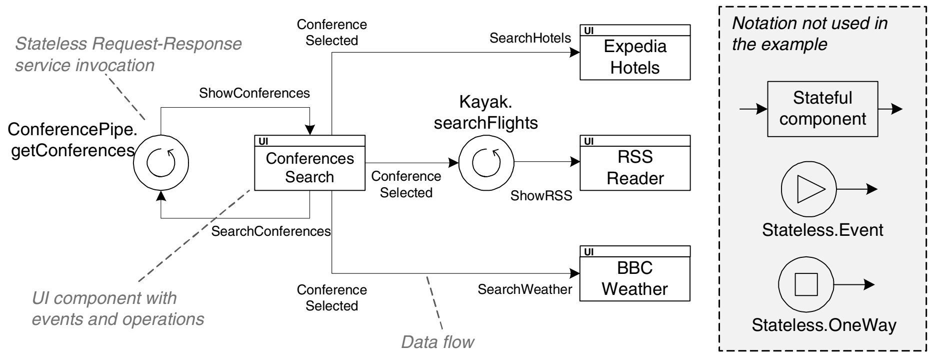
The universal composition model is defined in the Universal Composition Language (UCL), which operates on MDL descriptors only. UCL is for universal compositions what BPEL is for web service compositions (but again, simpler and for universal compositions). A universal composition is characterized by:

- *Component declarations*: Here we declare the components used in the composition and provide references to the MDL descriptor of each component and set possible constructor parameters.
- *Listeners*: Listeners are the core concept of the universal composition approach. They associate events with operations, effectively implementing simple publish-subscribe logics. Events produce parameters; operations consume them (static parameter values may be specified in the composition). Inside a listener, inputs and outputs can be arbitrarily connected (by referring to the respective IDs and parameter names) resulting into the definition of *data flows* among components. An optional condition may restrict the execution of operations; conditional statements are XPath statements expressed over the operation's input parameters. Only if the condition holds, the operation is executed.
- *Type definitions*: As for mashArt components, the structures of complex parameter values can be specified via dedicated data types.

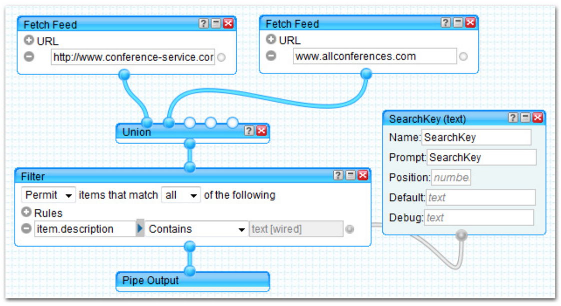
We are now ready to compose our Conference Trip Planner. Composing an application means connecting events and operations via data flows, and, if necessary, specifying conditions constraining the execution of operations. The graphical model in Figure 2(a) represents, for instance, the "implementation" of the reference scenario described in the introduction. We can see the four UI components *Conferences Search*, *Expedia Hotels*, *RSS Reader* and *BBC Weather* and the two stateless service components *ConferencePipe* and *Kayak*. The composition has four listeners:

1. If a user enters a conference search string and starts the search (*SearchConference* event), the *ConferencePipe* service is invoked by processing a Yahoo! pipe that queries two other services: *conference-service.com* and *allconferences.com*. The internals of the pipe are shown in Figure 3(b). The pipe joins the results coming from the two services and applies the filter condition provided by the user; the result is passed back to the mashArt composition by invoking the *ShowConferences* operation of the *Conferences Search* UI component.

   Note that similar operators and feed processing logics as shown in Figure 3(b) could easily be implemented also directly in mashArt, but we prefer reusing Yahoo! Pipes to show an example of how mashup platforms can interoperate.
2. If a user selects a conference from the list of retrieved conferences (*ConferenceSelected* event), three listeners reacting to the same event are activated. The first listener propagates the selected conference location and dates to the *Expedia Hotel* service that retrieves a list of available hotels from the Expedia repository.
3. The second listener activated after the selection of a conference searches for matching flights and visualizes them in the *RSS Reader*. The flights are retrieved by invoking a *kayak.com* flight search service and delivering its results as RSS feed. Such feed is provided as input to the *RSS Reader* via the *ShowRSS* operation.
4. Finally, the last listener activated upon selection of a conference aligns the data shown in the *BBC Weather* component by forwarding the name of the city the conference is located in through the *SearchWeather* operation. This causes the component to visualize the average weather conditions for the selected city.



(a) The mashArt composition model for the example scenario plus the notation not used in the example


Fetch Feed
URL: http://www.conference-service.cor

Fetch Feed
URL: www.allconferences.com

Union

SearchKey (text)
Name: SearchKey
Prompt: SearchKey
Position: numbe
Default: text
Debug: text

Filter
Permit items that match all of the following
Rules
item.description Contains text [wired]

Pipe Output

(b) The internals of the conference search aggregation and filtering pipe

Fig. 2. Composition model for the Conference Trip Planner application

In the model, stateful components handle multiple invocations during their lifetime; stateless components represent single invocations. The *ConferencePipe* service is invoked each time a user inputs a new search query, while the *Conferences Search* component is instantiated only once and handles multiple events and operations.

Regarding the semantics of the three data flows leaving the *Conferences Search* component upon a *ConferenceSelected* event, it is worth noting that we allow the association of *conditions* operations. A *condition* is a Boolean expression over the operation's input (e.g., simple expressions over name-value pairs like in *SQL where* clauses) that constrains the execution of the operation. The three data flows in Figure 2(a) represent a *parallel branch* (conjunctive semantics); if conditions where associated with either *SearchHotel*, *ShowRSS* or *SearchWeather* the flows would represent a *conditional branch* (disjunctive semantics). A similar logic applies to operations with multiple incoming flows that can be used to model *join* constructs. Inputs may be *optional* if they are not required for the execution of the operation. If only mandatory inputs are used, the semantics is conjunctive; otherwise, the semantics is disjunctive.

Data transformations can be defined via either (i) simple parameter mappings as described above; (ii) inline scripting, e.g., for the computation of aggregated or combined values; (iii) runtime XSLT transformations; or (iv) dedicated data transformation services that take a data flow in input and transform it, producing a new output.

5.3 Implementing and Provisioning Universal Compositions

Development Environment. In line with the idea of the Web as integration platform, the mashArt editor runs inside the client browser; no installation of software is required. The screenshot in Figure 3 shows how the universal composition of Figure 2(a) can be modeled in the editor. The modeling formalism of the editor slightly differs from the one introduced earlier, as in the editor we can also leverage interactive program features to enhance user experience (e.g., users can interactively choose events and operations from respective drop-down panels). But the expressive power of the editor is the same as discussed above.

The *list of available components* on the left hand side of the screenshot shows the components and services the user has access to in the online registry (e.g., the *Conferences Search* or the *BBC Weather* component). The *modeling canvas* at the right hand side hosts the composition logic represented by *UI components* (the boxes), *service components* (the circles), and *listeners* (the connectors). A click on a listener allows the user to map outputs to inputs and to specify optional input parameters.

In the lower part of the screenshot, tabs allow users to switch between different views on the same composition: visual model vs. textual UCL, interactive layout vs. textual HTML, and application preview. The layout of an application is based on standard HTML templates; we provide some default layouts, own templates can easily by uploaded. The preview panel allows the user to run the composition and test its correctness. Compositions can be stored on the mashArt server.

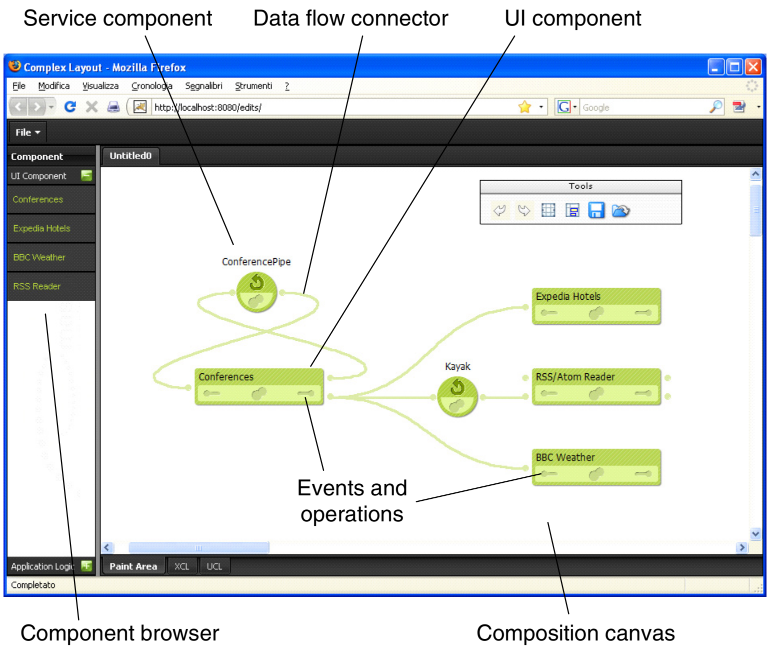


Fig. 3. The mashArt editor

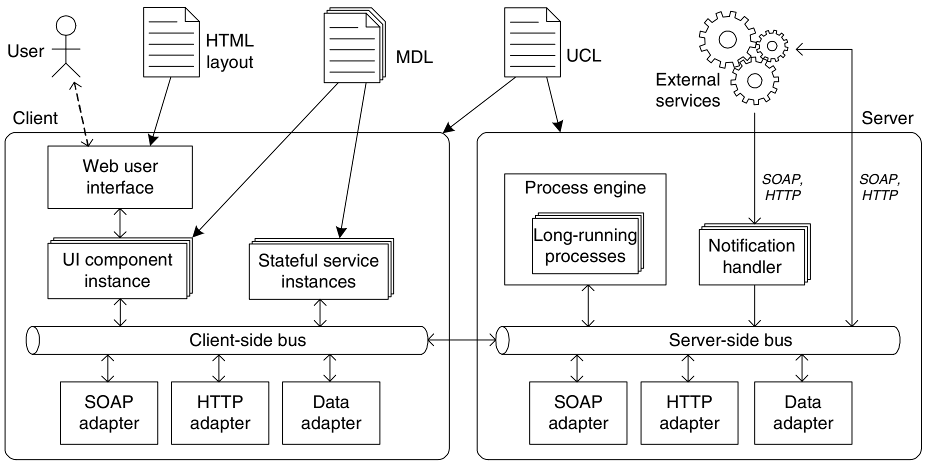


Fig. 4. Universal execution framework

The implementation of the editor is based on JavaScript and the Open-jACOB Draw2D library (http://draw2d.org/draw2d/) for the graphical composition logic and AJAX for the communication between client and server. The registry on the server side, used to load components and services and to store compositions, is implemented as a RESTful web service in Java. The platform runs on Apache Tomcat.

Execution Environment. Developing the mashArt execution environment requires solving issues like (i) the seamless integration of stateful and stateless components and of UI and service components, (ii) the conciliation of short-lived and long-lasting business process logics in one homogeneous environment, (iii) the consistent distribution of actual execution tasks over client and server, and (iv) the transparent handling of multiple communication protocols [19].

Figure 4 illustrates the functional architecture of our execution environment. The environment is divided into a client- and a server-side part, which exchange events via a synchronization channel. On the client side, the user interacts with the application via its UI, i.e., its UI components, and thereby generates events that are intercepted by the client-side event bus. The bus implements the listeners that are executed on the client side and manage the data and SOAP-HTTP adapters. The data adapter performs data transformations, the SOAP-HTTP adapters allow the environment to communicate with external services. Stateful service instances might also use the SOAP-HTTP adapters for communication purposes.

The server-side part is structured similarly, with the difference that the handling of external notifications is done via dedicated notification handlers, and long-lasting process logics that can be isolated from the client-side listeners and executed independently can be delegated to a conventional process engine (e.g., a BPEL engine).

The whole framework, i.e., UI components, listeners, data adapters, SOAP-HTTP adapters, and notification handlers are instantiated when parsing the UCL composition at application startup. The internal configuration of how to handle the individual components is achieved by parsing each component's MDL descriptor (e.g., to understand whether a component is a UI or a service component). The composite layout of the application is instantiated from the HTML template filled with the rendering of the application's UI components.

6 Conclusion

In this chapter, we have considered a novel approach to UI and service composition on the Web, i.e., *universal composition*. This composition approach is the foundation of the mashArt project, which aims at enabling even non-professional programmers (or Web users) to perform complex UI, application, and data integration tasks online and in a hosted fashion (integration as a service). Accessibility and ease of use of the composition instruments is facilitated by the simple composition logic and implemented by the intuitive graphical editor and the hosted execution environment. The platform comes with an online registry for components and compositions and will provide tools for monitoring and analysis of hosted compositions.

Throughout the chapter, we have constantly kept an eye on the connection between universal composition and *search computing*. The Conference Trip Planner tool implemented using the mashArt instruments and languages shows that it is indeed possible to develop a component-based application that provides answers to the conference search problem, provided that the necessary basic components are readily available. The application's integration logic is achieved by means of an imperative drag-and-drop composition paradigm that allows the users of the mashArt platform to compose applications according to their own knowledge about which components are needed and about how to glue them together. There exist many alternative solutions to the implementation of the same application; yet, unlike in [18], where an optimal query plan is identified automatically, in mashArt it is up to the developer to decide which solution fits best his/her individual needs.

In terms of output of the composition, it is interesting to note that while in the traditional search scenario the output is a set of result tuples, the output in mashArt is rather represented by the whole application, i.e., the individual components and their interconnection. Given the search query introduced in the introduction of this chapter, its answer is therefore represented by the screenshot in Figure 1, which naturally combines simple search outputs with sophisticated UI components.

References

[1] Yu, J., Benatallah, B., Casati, F., Daniel, F.: Understanding Mashup Development and its Differences with Traditional Integration. Internet Computing 12(5), 44–52 (2008)

[2] OASIS. Web Services for Remote Portlets (August 2003), `http://www.oasis-open.org/committees/wsrp`

[3] Yu, J., Benatallah, B., Saint-Paul, R., Casati, F., Daniel, F., Matera, M.: A Framework for Rapid Integration of Presentation Components. In: WWW 2007, pp. 923–932 (2007)

[4] Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer, Heidelberg (2003)

[5] Dustdar, S., Schreiner, W.: A survey on web services composition. Int. J. Web Grid Services 1(1), 1–30 (2005)

[6] OASIS. Web Services Business Process Execution Language Version 2.0 (April 2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
[7] Pautasso, C.: BPEL for REST. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 278–293. Springer, Heidelberg (2008)
[8] van Lessen, T., Leymann, F., Mietzner, R., Nitzsche, J., Schleicher, D.: A Management Framework for WS-BPEL. In: ECoWS 2008, Dublin (2008)
[9] Curbera, F., Duftler, M.J., Khalaf, R., Lovell, D.: Bite: Workflow Composition for the Web. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 94–106. Springer, Heidelberg (2007)
[10] Maximilien, E.M., Ranabahu, A., Gomadam, K.: An Online Platform for Web APIs and Service Mashups. Internet Computing 12(5), 32–43 (2008)
[11] Braga, D., Ceri, S., Daniel, F., Martinenghi, D.: Optimization of Multi-Domain Queries on the Web. In: VLDB 2008, Auckland, pp. 562–573 (2008)
[12] Daniel, F., Yu, J., Benatallah, B., Casati, F., Matera, M., Saint-Paul, R.: Understanding UI Integration - A Survey of Problems, Technologies, and Opportunities. IEEE Internet Computing, 59–66 (May 2007)
[13] Microsoft Corporation. Smart Client - Composite UI Application Block (December 2005), http://msdn.microsoft.com/en-us/library/aa480450.aspx
[14] The Eclipse Foundation. Rich Client Platform (October 2008), http://wiki.eclipse.org/index.php/RCP
[15] Sun Microsystems. JSR-000168 Portlet Specification (October 2003), http://jcp.org/aboutJava/communityprocess/final/jsr168/
[16] Acerbis, R., Bongio, A., Brambilla, M., Butti, S., Ceri, S., Fraternali, P.: Web Applications Design and Development with WebML and WebRatio 5.0. TOOLS (46), 392–411 (2008)
[17] Gómez, J., Bia, A., Parraga, A.: Tool Support for Model-Driven Development of Web Applications. In: Ngu, A.H.H., Kitsuregawa, M., Neuhold, E.J., Chung, J.-Y., Sheng, Q.Z. (eds.) WISE 2005. LNCS, vol. 3806, pp. 721–730. Springer, Heidelberg (2005)
[18] Braga, D., Ceri, S., Daniel, F., Martinenghi, D.: Optimization of Multi-Domain Queries on the Web. In: VLDB 2008, Auckland, New Zealand, August 2008, pp. 562–573 (2008)
[19] Daniel, F., Casati, F., Benatallah, B., Shan, M.-C.: Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In: ER 2009 (November 2009)
[20] Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine, Dissertation (2000), http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
[21] Abiteboul, S., Manolescu, I., Zoupanos, S.: OptimAX: efficient support for data-intensive mash-ups. In: ICDE 2008, pp. 1564–1567 (2008)