Application Composition at the Presentation Layer: Alternatives and Open Issues

Stefan Pietschmann
Technische Universität
Dresden, Germany
Stefan.Pietschmann@tu-dresden.de

Tobias Nestler
SAP Research Center
Dresden, Germany
Tobias.Nestler@sap.com

Florian Daniel
University of Trento
Italy
Daniel@disi.unitn.it

ABSTRACT

The concept of application composition at the presentation layer, i.e., the development of web applications with user interfaces (UI) starting from stand-alone, reusable components, is a relatively new research area. The recent advent of *web mashups* and component-based web applications has produced promising results, but we argue that there is still a lot of space for improvement. By looking at three advanced approaches in this area, we investigate the current solution space and consequently unveil challenges and problems still to be solved in order to turn presentation composition into common practice.

1. INTRODUCTION

Despite its beginnings as a static source of information, the web has become a platform for manifold applications. A paradigm which has made this trend possible is the so-called *Programmable Web*. Therein, data, application logic and user interfaces are made available in an open and reusable fashion via Web APIs and services. Mashups build on this foundation by combining services and contents, i. e., distributed resources, into new *composite* web applications. Composing applications from reusable parts has been subject to research for a long time, yet traditional approaches were typically limited to the integration on lower application layers – namely, the business logic and data layers. However, as mashups reusing pieces of UIs from other web sites show, the need for similar concepts at the presentation layer – UI integration and composition – has become evident [1].

Commonly, mashup applications are developed manually by skilled programmers. Yet, in order to assist also less skilled developers or web users, so-called *mashup platforms* (e. g., Yahoo! Pipes) have emerged. They aim at partially or fully assisting mashup development – an ambitious goal. Indeed, mashups can be very complex applications, requiring a variety of specific web development skills that cannot easily be hidden behind simple user interfaces, especially when they include extensive user interactions. Consider, for instance, the following development problem, which we will use throughout the paper as reference scenario:

John, an administrative employee of the plumbing company Acme Inc., wants to develop a small application to easily assign incoming service requests to the plumbers of Acme. His typical workflow consists of the following steps: (i) assessing the priority of service requests, (ii) checking customer details and service history, (iii) retrieving customer addresses, (iv) matching service requests with their geographically closest plumbers, and (v) notifying customers and plumbers of the planned service times.

John's work therefore includes up to five individual tasks, which are usually performed with the help of different, dedicated applications. Let's assume (i) that John can source service requests via a company-internal RSS feed, (ii) that there are services available that allow him to inspect customer and plumber details, (iii) that he uses a Google Map (it comes with a JavaScript interface) to assess the distance between customers and plumbers, (iv) that the company has an own email service, and (v) that there exist some visualization widgets (e. g., an RSS reader) available for John to display raw data. This list of ingredients shows that the development of the envisioned service request management application is anything but trivial, and it is evident that John as simple employee is not able build it without the help from expert programmers.

The goal of this paper is to look at current best practices for the composition of web applications like the one required by John to understand what is missing, in order to enable John to develop applications independently. The distinguishing strength of the approaches we consider is that they are able to abstract the previous scenario as a problem of composition at the presentation layer, which eases the understanding by end users. Specifically, in this paper:

- We introduce an *evaluation framework* that highlights what we think are the most important aspects in assisted composition at the presentation layer;
- We review *three composition approaches* that specifically focus on the problem of component-based development of web applications at the presentation layer: mashArt [2], ServFace [3], and CRUISe [4];
- We *compare* the three approaches and discuss what we think is still missing, outlining promising directions for future research.

In the next section we present related composition approaches targeting the presentation layer. After introducing

iiWAS2010 8-10 November, 2010, Paris, France

our evaluation framework in Section 3, Section 4 presents and discusses the three composition approaches mentioned accordingly. Finally, Section 5 summarizes the results of the evaluation, highlights open issues and challenges, and outlines directions for future work.

2. BACKGROUND AND RELATED WORK

Historically, composition on the web has focused on the problem of web service composition, e. g., in terms of BPEL [5], the composition standard by OASIS. The approaches discussed in this paper are inspired by what has been achieved in this regard, yet their focus on the presentation layer clearly distinguishes them from pure service composition.

The problem of integration, i. e., composition at the presentation layer has first be introduced in [1], where the authors concluded that UI composition is still scarcely supported: Desktop UI technologies such as .NET CAB (http://msdn.microsoft.com/en-us/library/aa480450.aspx) or Eclipse RCP (http://wiki.eclipse.org/index.php/RCP) 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 [6] or WSRP [7] represent a mature and web-friendly solution for the development of portal applications; portlets are however typically executed in an isolated fashion, and communication among portlets or with web services remains hard, as portals do not support service orchestration.

In the context of web mashups, some industrial tools already partly address the problem of composition. For instance: Yahoo Pipes (http://pipes.yahoo.com) focuses on data integration via RSS or Atom feeds; JackBe Presto (http://www.jackbe.com) adopts a Pipes-like approach for data mashups and also supports the visualization of their outputs; IBM QEDWiki (http://services.alphaworks.ibm.com/qedwiki) provides a wiki-based mechanism to glue together UI widgets; Intel Mash Maker (http://mashmaker.intel.com) is a browser plug-in for the personalization of web pages with UI widgets inside the web browser. In this paper, instead, we focus on projects that employ similar concepts and capabilities to the ones of the above approaches, but specifically focus on the presentation layer. Our direct involvement in them allows us to discuss details typically hidden or not supported in industrial tools.

3. PRESENTATION INTEGRATION: TECHNIQUES AND INGREDIENTS

To create the application in our reference scenario, several aspects have to be considered: Firstly, components as atomic parts of an application have to provide the required functionality. Secondly, they have to be connected to design the desired application. Finally, a tool environment should enable the user to accomplish all parts of the composition process in a user-friendly way.

Each of these aspects is typically supported in different ways by different composition approaches. In order to be able to systematically compare distinct approaches, we have developed an *evaluation framework* covering major functional aspects involved in integration at the presentation layer. The framework distinguishes three main *dimensions* of analysis: the components subject to composition, the composition model, and the corresponding environment for development and execution. The three dimensions can further be split into more detailed *sub-dimensions*, which in turn can assume a set of *values*:

Component models

- **Types:** What kind of technical resources do components represent? The most common types are SOAP and RESTful web services, RSS/Atom feeds, and JavaScript UI components.
- **Data format:** What specific data formats are supported by the approach? We distinguish XML and JSON (hierarchical structure), and parameter-value pairs (flat structure).
- **Description language:** How are components formally described? A descriptor may comply to WSDL, WADL, or a proprietary format.

Composition model

- **Internal component model:** How is a component internally described? Interactions with components might be specified in the native format of the component or be abstracted via suitable technology adapters.
- **Composition logic:** How are components glued together? Multiple techniques can be used conjunctively: static configurations of components, event exchanges, control flows, data flows, shared variables (blackboard approach).
- **Layout logic:** How is the composite layout defined? Typically there are either predefined HTML templates, customizable templates, or custom (user-provided) layouts. Additionally, multiple views (pages) can be supported.
- **Output type:** What kind of application results from the composition process? We distinguish data services, application services, and interactive web applications.

Design/execution environment

- **Target users:** What kind of users should be enabled to compose applications? The range spans from professional programmers to non-professional programmers, skilled web users, and unskilled end users.
- **Design paradigm:** What is the general paradigm for the application design? For instance, "live" composition at run-time, visual development (drag-and-drop, wiring), and textual coding.
- **Deployment/execution paradigm:** How are applications deployed and executed? Does the approach generate code or compile the composition into an executable format, or is the composition interpreted at runtime as is?
- **Additional features:** What are special features of the approach? Typical features include multi-platform-support, adaptivity (to varying runtime contexts), and the availability of component and composition repositories.

Below, we discuss mashArt, ServFace, and CRUISe as approaches for presentation integration, i. e., composition, and position them with respect to our evaluation framework.

4. PRESENTATION INTEGRATION IN PRACTICE

4.1 Universal Composition - mashArt

The mashArt [2] project conducted at the University of Trento, Italy, focuses on so-called *hosted universal composition* for the web. The aim of the project is to devise models, languages, paradigms and development instruments that allow one to abstract from low-level implementation details and to compose components that are characterized by heterogeneous technologies, ranging from simple feeds to complex web services and UI components, within one and the same development environment. Doing so requires conciliating the need for orchestration of process-oriented service composition with the need for synchronization of event-based UI development, a challenge the project addresses with a unique event- and data flow-based composition logic.

Component models

Types: One of the foundations of the mashArt approach is its universal composition model, which abstracts from the peculiarities of typical technologies used in the web and provides access to components through an event-based interface logic that describes components in terms of their internal state (a set of parameter-value pairs), events (that communicate state changes), and operations (that allow one to enact state changes). The abstraction is able to accommodate component types as varied as RSS/Atom feeds, XML resources, SOAP and RESTful web services, and JavaScript UI components. That is, mashArt's component model covers all the three layers of the typical application stack: data, application logic, and user interface.

In order to interact with components at runtime, mashArt natively supports JavaScript UI components that adhere to its event-based interface logic; interaction with these components occurs via JavaScript function calls. Interaction with RESTful and SOAP web services, instead, utilizes suitable protocol adapters, which mediate between the internal event-based logic and the components' own protocol. Access to RSS/Atom feeds or XML resources is simply handled by a REST adapter, as interacting with them actually means performing an HTTP-GET operation on their URL.

Data format: mashArt components provide access to data via simple parameter-value pairs. That is, data formats of integrated components are converted to or interpreted as "flat" data structure: in terms of XML, first-level elements represent parameter names, their nested contents represent parameter values. The transformation of JSON, RSS and Atom is similar, while interaction with JavaScript components expects parameter-value data as associative array.

Description language: The abstract description of native mashArt components is done via the *Mashart Description Language* (MDL), a lightweight XML dialect for the specification of events, operations, and state parameters. SOAP web services can be accessed by directly referencing their WSDL descriptor, while RESTful services require the manual setup of the respective adapter.

Composition model

Internal component logic: Internally to the design and runtime environment, components are treated as event-based MDL components, independently of their actual implementation, thanks to the technology adapters that are able to mask individual protocols. This means, for instance, that a standard request-response invocation of a SOAP service is interpreted as event-operation sequence.

Composition logic: Given the even-based component logic, mashArt's composition logic is equally event-based (so-called event listeners are first class composition objects). Yet, it is also data-flow-based, in that it allows the developer to connect events (outputs) to operations (inputs) of components, thereby realizing a flow of data from one component to another. Besides supporting the definition of listeners, the *Universal Composition Language* (UCL) comes with support for parallel splits, conjunctive and disjunctive joins, conditional executions; due to its data-flow-based nature, loops are currently not supported. Data is passed among components as payload of events, formatted as parameter-value pairs. UCL currently supports a simple parameter-parameter mapping in order to assign output parameters to input parameters; the default mapping is simply based on the position of the parameters.

Layout logic: The layout of a mashArt application is implemented as a predefined or a custom HTML template with placeholders (e. g., DIV or IFRAME elements) accessed through unique identifiers of the form `mashart-X` where `X` is a customizable name. Only UI components need to be associated to placeholders. At startup of an application, its layout template is loaded and UI components are instantiated in their respective placeholders, filling the final layout with the actual content of the application.

Output types: Not only in input, but also in output mashArt theoretically provides support for all the three application layers. Applications with own user interface are the standard output. Pure service compositions or data processing pipelines are supported by the composition logic, yet running these kinds of compositions still requires the interaction with a suitable UI component; work on exposing these kinds of compositions via simple URLs is ongoing.

Design/execution environment

Target users: The mashArt platform targets non-professional programmers and skilled web users that are familiar with the semantics of their individual application domain and understand the meaning of components. Programmers may use the platform to enhance their productivity.

Design paradigm: The development of universal compositions leverages a Yahoo!-Pipes-like, visual drag-and-drop paradigm. Users can choose components from a palette within the development environment and place them onto a modeling canvas. There they can configure the components by setting static configuration parameters and wire together events with operations by graphically connecting output with input connectors. Connected components are highlighted in green, while dangling components are shaded in gray. Also, the association of UI components with placeholders in the HTML layout template can be performed by dragging and dropping components onto their destination placeholders. For fast feedback and immediate testing, mashArt provides a preview, which allows designers to run the composite application inside the design environment and to check its correct functioning.

Deployment/execution paradigm: Complete compositions can be stored on the mashArt server and published as web-accessible applications; the latter produces a unique

URL to access and run the application. All applications are hosted on the server, without the need for client-side software. The actual runtime environment of mashArt is distributed over client and server: instantiation starts at the server, where all necessary files (layout template, MDL descriptors, style files, libraries) are automatically assembled and sent to the client. The latter instantiates and manages all UI components, while service invocations and external notifications toward the application (originating from remote web services) require server-side action.

Additional features: The platform comes with a dedicated, hosted registry for components, compositions and layout templates. Through its graphical user interface, it can easily be filled by a user with own objects. The platform's built-in access rights manager allows users to manage the visibility of his objects and to share components or compositions with other users.

In order to implement the application for our reference scenario, let's assume we can rely on the following components: the RSS feed with customer support requests, an RSS Reader UI component, a JavaScript UI component providing access to customer data, a JavaScript UI component with all plumbers of the company, a Google Maps UI component, a SOAP service returning the current GPS position of active plumbers, and a SOAP email service. Setting up the necessary integration logic in mashArt would then require: placing the above components onto the modeling canvas; connecting the output of the RSS feed to the operation of the RSS Reader that visualizes the items in the feed; connecting the RSS Reader's "item selected" event to the customer UI component; connecting its "customer selected" event to the Google Map and to the email service; connecting the "plumber selected" event of the plumbers UI component to the GPS service and the output of that service to the Google Map. Finally, connecting the "plumber selected" event to the email service and configuring the two input links as conjunctive join allows the delivery of emails with both customer and plumber details. For the layout of the application, a predefined HTML template with four placeholders can be used, containing the RSS reader, customer, plumber and Google Maps UI components. The application can be run in the preview tab of the composition editor and be published to the mashArt server for public access.

4.2 Service Frontend Composition - The ServFace Builder

The *ServFace Builder* is a web-based authoring tool developed in the frame of the EU project ServFace [3]. It enables service composition at the presentation layer by combining service frontends, rather than their application logic or data. The tool utilizes the advantages of *service annotations* [8] to visualize these frontends already during design time in order to give users an impression of the resulting UI at any time of the development process. In his role as a service composer and application designer, the end user creates the desired application in a kind of WYSIWYG (What you see is what you get) style. No technical knowledge about service composition is required, as applications are modeled and designed in a graphical way without writing any code.

Component models

Types: Components within the ServFace Builder are called *service frontends* and represent single operations of SOAP web services via form-based UIs. The frontends consist of a nested container structure, which includes UI-elements like text fields or combo boxes that are bound to the corresponding service operation parameters.

Data format: Data is exchanged via SOAP messages using XML as a universal data format, so all parameters adhere to the WSDL data types.

Description language: Web services that should be used within the ServFace Builder must be described with WSDL and enriched with UI-related service annotations. The latter are reusable information fragments formally defined in a meta-model that are linked to concrete elements within the WSDL. They cover aspects of the visual appearance of a service (e. g., *labels*, *grouped* parameters, *enumerations* for predefined vales) the behavior of UI elements (e. g., client-side *validation* rules, input *suggestions*) and relations between services (e. g., *semantic data type relation*) [8].

Composition model

Internal component logic: Each integrated web service is described with an internal service model that is automatically inferred from the WSDL and additionally contains the corresponding annotation model. This service model unites all necessary information about a service, serves as the foundation for the visualization of the frontend and holds a reference to the WSDL.

Composition logic: The service model is an integrated part of the internal application model, the *Composite Application Model* (CAM). The CAM defines the application structure visualized within the ServFace Builder. An application consists of several pages, which act as containers for frontends and represents a dialog visible on the screen. Pages can be connected to each other to model navigation flows. Adding a service operation to a page triggers the creation of UI elements for each particular operation parameter. Every user action (e. g., *add page*, *integrate frontend*, *define data flow*) is constantly synchronized with the CAM instance to ensure a valid model representation at any time of the design process.

Layout logic: General layout information is stored in predefined templates including information like screen size, device type and toolkit specifics. Composers can customize specific layout aspects like background color or frames for all pages within an application. Furthermore, they can arrange frontends freely on the pages and can thereby define the layout for each page individually.

Output types: A serialized instance of the internal application model (CAM) serves as input to a model-to-code transformation process in order to generate executable applications for different target platforms like web applications or mobile platforms (as described in [3]).

Design/execution environment

Target users: The ServFace Builder targets non-programmers and skilled web users familiar with the semantics of their individual application domain. They need to understand the meaning of provided service functionality without having a deep understanding of technical concepts like web services or service composition.

Design paradigm: The main idea of ServFace is to compose web services in a WYSIWYG manner, i. e., to directly interact with single UI elements, complete frontends or pages

in order to model the application. No other abstraction layer is needed to define data or control flow. To connect two service frontends, the target UI element of the frontend to be filled with data has to be selected. Using a context menu and selection of the source UI element the data flow is defined and visualized. The direct use of the UI elements of the frontends benefits from the fact, that form-based applications are well-known and most users are familiar with them. The definition of the control flow is supported visually as well: users simply connect two pages in order to create a automatically generated link for the page transition.

Deployment/execution paradigm: After the visual modeling, the CAM instance is serialized to an XMI representation that serves as the input for the fully automated generation process of a deployable application. Currently, support for hosted Microsoft Silverlight and a Google Android applications is under development.

Additional features: The ServFace Builder supports the development for different target devices (multi-platform-support). Depending on the selected platform, aspects like page size, supported toolkits and device-specific functionalities are considered accordingly during the design phase.

In order to implement the application described in the reference scenario, all required resources must be available in form of annotated web services. Customer requests can, for example, be provided by a web service (instead of an RSS feed) offered by the CRM (Customer Relationship Management) system of Acme Inc. Required customer data is already given by a SOAP service, as well as information on plumbers, and a service operation to assign a plumber to a specific customer request. Messaging services are also already available as SOAP web services. John can now use the ServFace Builder to create the application from these services. Firstly, he has to choose a target platform for his application. Let's assume, he would like to design a web application. Having made this decision, he can start the actual composition process by dragging the customer request service operation onto the modeling canvas of the first page. The tool automatically creates the corresponding frontend, which can be arranged on the page. In a second step, John integrates the service operation that provides the customer data onto the page. Now, he has to define a connection between the operations by combining the two frontends via their UI elements. Therefore, he selects a UI element (e. g., input field for the customer name) of the customer data frontend and defines the data flow by selecting the customer name UI element in the output of the customer request frontend. This task must be done for the other service operations as well to define all required data flows. John can place the frontends on several pages to create a multi-page application. All created pages are shown in a graph-like overview and can be connected to define the page flow. After finishing the design process, John can deploy his web application on a web server and access it via his browser.

4.3 Dynamic Context-Aware Composition - CRUISe

The main objective of the CRUISe [4] project is to develop and evaluate novel concepts for the model-driven development and deployment of composite applications. Its central idea is the extension of the service-oriented paradigm to the presentation layer to support a *universal composition* of context-aware applications. This is accomplished by providing reusable UI components in a distributed, service-oriented fashion, and their context-aware, dynamic invocation and integration with other mashup components.

Component models

Types: Different semantic types of components are distinguished in CRUISe. On the topmost application layer, *UI Components* encapsulate reusable UI parts with the corresponding presentation logic, such as an interactive map. Commonly, those are represented as JavaScript. *Logic Components* facilitate compatibility and efficient communication between components, as they provide means for data transformations. Finally, *Service Components* wrap heterogeneous back-end services (SOAP or REST) providing data or application logics.

Data format: Components communicate via events, that contain typed parameters. XML is used as a universal data format, so all values adhere to an XML Schema data type. Data is automatically transformed into the particular XML structure if necessary.

Description language: Components can be specified uniformly with the *Mashup Component Description Language* (MCDL) which falls in two parts: *Class* descriptions define generic interfaces with information on metadata, signature (operations and events) and semantics of a component; *Bindings* map concrete component implementations to a class and include platform-specific dependencies. WSDL or WADL files of back-end services can be directly mapped to MCDL, while UI components are described with UISDL which provides UI-specific extensions like "screenshots" and "interaction styles".

Composition model

Internal component logic: To the composition environment components are black boxes. However, they adhere to a universal component model that builds on three abstractions, namely *configuration*, *events*, and *operations*. The configuration represents a component's state by a set of predefined key-value-pairs (*properties*). State changes are published by events, which are triggered by user interaction (UI), component logic or notifications from external services (model). Operations are methods of a component triggered by events. They can include arbitrary functionality, such as state changes, calculations, or service requests. These abstractions facilitate the abstract definition and loose coupling of application components using events.

Composition logic: CRUISe uses a platform-independent composition model defining all relevant aspects of an application. It specifies components used, their configurations, data types, and visual styles. Control and data flow are modeled by connecting events with operations via *channels* that pass parameters from one component to another. Mappings can be defined on both sides of a channel to avoid naming or ordering conflicts. Component interoperability is ensured by *Logic Components* providing data manipulations, e. g., filtering, iteration, and aggregation of data. Besides layouts and screen flow, adaptive behavior can be specified using *aspects*. They define model manipulations with regard to context changes, thereby crosscutting all of the above-mentioned models.

Layout logic: The layout of a composite UI is defined in the model using common layouts that can be nested ar-

bitrarily, As an example, a *FlowLayout* allows components to be stacked horizontally or vertically. Additionally, multiple *views* can be defined, each representing a specific layout. Transitions between views – the so-called "screen flow" – are triggered by events issued from components or the platform.

Output types: The result of the model transformation/-interpretation varies with regard to the platform and client-server-distribution. Typical output is a web application with an adaptive user interface. However, if no UI components are included, service compositions can as well be defined and executed. The definition of complex constraints and dependencies between components is not supported, though.

Design/execution environment

Target users: The CRUISe composition model is targeted at completeness rather than simplicity. Thus, modeling requires knowledge of components and event-based communication, which restricts the target group to programmers and skilled web users familiar with concepts of component-based software. Consequently, CRUISe aims at simplifying the development of composite applications in IT departments, rather than supporting end user development.

Design paradigm: Modeling applications can be done using a text, tree, or visual editor. Certainly, the latter is most suitable, providing modeling abstractions, drag-and-drop composition and wiring as well as means to define layout and screen flow. As a result, IT departments are enabled to compose applications without writing a single line of source code.

Deployment/execution paradigm: Compositions can be transformed into web applications, which are made available on a web server. Execution is carried out by a runtime system which exists in different "flavors", e. g., based on Eclipse RAP, JSP, or completely browser-based. It controls the application's and components' life cycles and manages control and data flow. UI components are dynamically integrated using a dedicated service which selects, adapts, and provides components being most suitable for the current context and platform.

Additional features: UISDL descriptions are managed by a dedicated registry, which is invoked during the UI selection process. The runtime environment supports further adaptation techniques, such as layout changes, component reconfigurations and exchange. As a foundation, it comprises context monitors and uses an ontology-based context management service [9].

To develop our reference application with CRUISe, its back-end services need to feature an interface description, such as WADL (RSS feed), or WSDL (SOAP DB service). Furthermore, we assume the following UI components and UISDL descriptions are present: an RSS reader to visualize service requests, a generic list viewer to list plumbers, a map component to pinpoint locations, a details viewer listing request and customer details, and a messaging UI. Using the component descriptions, a member of the IT department, or even John himself, can compose the application using an editor. Therefore, he configures components with their description files and connects them with each other by linking their events and operations. For instance, the output of the feed component is connected to the RSS viewer's input. The viewer's output "selectedItem" is connected to the "getCustomerDetails" operations of the customer service component. Since event and operation signatures do not always fit, logic components can be added, e. g., to iterate over service requests from the feed, or to extract the customer's location from complex customer data. The location is then linked to the map's "setMarker" operation, and so forth. Alternatively to this low-level approach, previously modeled compositions, e. g., a feed, iterator and RSS viewer, can be included as composite components. Finally, UI components are arranged in layouts and may be divided into multiple views. Additional adaptation aspects can be defined to ensure that the application fits different device characteristics (available plug-ins, screen size) and adapts to user preferences (color schemes, favorite map services). The model is then either transformed into an executable application or directly interpreted, depending on the runtime system used.

4.4 Comparison and Discussion

Table 1 summarizes the details of the previous descriptions and highlights similarities and differences. A first fundamental difference between mashArt/CRUISe and ServFace can be seen by looking at the supported *component models*: ServFace does not employ (JavaScript) UI components but instead generates suitable UIs for SOAP services. In contrast, CRUISe supports additional (UI) components whose technologies depend on the particular integration platforms. Although all approaches focus on integration at the presentation layer, they do recognize the need for supporting "traditional" services. As the number of consumer- and enterprise-level web services is growing fast, developing modern applications requires being able to integrate them without serious effort. For this reason, each approach supports WSDL descriptors, alongside proprietary descriptors for their own component models. XML is commonly used as data format, being a de-facto standard in the web service domain. Yet, lightweight alternatives like JSON are gaining momentum. They become handy for UI synchronization, which typically does not require extensive data exchange.

Looking at the *composition model*, we note that all approaches support data flows. This does not come unexpectedly: the semantics of data flows is easy to understand as it naturally conciliates the control flow with data artifacts. Thus, a data flow connector implies both the activation of its destination plus the passing of data items. In traditional approaches like BPMN or BPEL, control flow and data artifacts are modeled independently of each other, which increases complexity. Consequently, most industrial mashup tools, such as Yahoo! Pipes or JackBe Presto, adopt data flows. As for the output types, all approaches support web applications (HTML); CRUISe also supports application services in case no UI components are part of the composition. However, the composition complexity supported is not comparable to dedicated languages and platforms, like BPEL. The layout of applications is generally based on templates, which is common practice in web development. In contrast to most mashup platforms, ServFace and CRUISe explicitly support multi-page applications, while mashArt is working toward that feature.

All *design/execution environments* presented target non-professional programmers and skilled web users. Yet, only ServFace can claim support for the latter, being backed by user studies. Development is assisted by visual drag-and-drop development and by wiring components – no need for manual coding. The sensible use of standard web technolo-

			mashArt	ServFace	CRUISe
Component model	Types	JavaScript	●	–	●
		SOAP	●	●	●
		REST	●	–	●
		RSS/Atom	●	○	○
		Additional	–	–	●
	Data format	XML	●	●	●
		JSON	○	–	○
		Parameter-value pairs	●	–	●
	Description	WSDL	●	●	●
		WADL	–	–	●
		Proprietary desc.	●	●	●
		No description	–	–	–
Composition model	Internal component model	Native model	–	–	–
		Abstract model	●	●	●
	Composition logic	Configuration	●	–	●
		Event-based	●	○	●
		Control flow	–	●	–
		Data flow	●	●	●
		Blackboard-based	–	–	–
	Layout logic	Predefined templates	●	●	●
		Customizable templates	–	●	●
		Custom layouts	●	●	●
		Screen/Page flow	○	●	●
	Output types	Data service	○	–	–
		Application service	○	–	○
		Web application	●	●	●
Design/execution environment	Target user	Prof. programmer	●	–	●
		Non-prof. progr.	●	●	●
		Skilled user	○	●	○
		Unskilled user	–	–	–
	Design paradigm	Live composition	–	–	–
		Drag-and-drop	●	●	○
		Wire	●	●	●
		Coding	–	–	–
	Deployment/execution	Code generation	–	●	●
		Interpretation	●	–	●
		Compilation	–	–	–
	Additional features	Multi-Platform-Supp.	–	●	●
		Adaptivity	○	–	●
		Component repository	●	●	●
		Composition repository	●	–	○

Table 1: Summary of the evaluation of the three composition approaches; "●" indicates supported features, "○" indicates features under development, "–" indicates unsupported features.

gies (e. g., XML and HTML) facilitates code generation or interpretation from completed compositions, rendering complicated compiler logics unnecessary.

Advanced features like adaptability and adaptivity are also partly supported. In this regard, CRUISe supports the dynamic, context-aware integration and reconfiguration of UI components, backed by information from a steadily-updated context management service.

While all approaches may be used to realize our reference scenario, there are still shortcomings. So far, mature support for JSON and WADL, two ingredients characterizing advanced RESTful services, is missing. The blackboard approach to exchange data among components is not considered suitable for presentation composition due to its event-based nature (however, approaches like BPEL or Intel Mash Maker use it). Though not explicitly discussed above, data mappings among components using different data formats are still a manual task. In this context, we regard the *accuracy of fit* between components as a crucial factor for presentation composition to succeed. Thus, research in concepts for (semi-)automated, semantically enhanced data and component mapping, i. e, matching, is desirable.

With regard to the growing importance of mashup solutions for the enterprise, the effect and application of the above-mentioned concepts for presentation integration on traditional, human-involved business processes is another challenging research direction. In this regard, both CRUISe and mashArt have started to investigate how their concepts may simplify the development and interaction with traditionally heavy-weight processes.

Finally, although all approaches restrain from manual coding, they do not achieve the goal of enabling average web users to compose their own applications. However, they successfully reduce development time and complexity for skilled users and professionals.

5. CONCLUSION AND OUTLOOK

This paper evaluates three alternative approaches for the composition of web applications at the presentation layer: mashArt, ServFace, and CRUISe. The analysis is based on three main dimensions (supported components, composition model and design/execution environment), articulated into sub-dimensions and possible values. The dimensions particularly aim at the evaluation of functional aspects, yet the

framework is generic and extensible, e. g., to also include non functional aspects such as the ones stemming from ISO 9126 or quality aspects as discussed in [10]. Extending the framework toward such non-functional properties will allow the inclusion of other evaluation criteria, such as the usability of compositions and the level of efficiency the tools provide – all aspects that heavily influence the success of a composition approach.

Our evaluation shows that mashArt and CRUISe follow a similar philosophy centered around the idea of event-based UI components, while ServFace distinguishes itself through its generation of form-based UIs from annotated SOAP services and its simplicity. All three composition platforms share an important peculiarity with the majority of industrial mashup platforms: they strongly focus on the *simplification of technologies*. The lessons learned from this are:

- There is a need for a *standard UI component technology* in terms of both interface logic and description language, similar to what we have for web services (e. g., SOAP and WSDL), to overcome dependencies and shortcomings of proprietary platforms.
- The benefit and the usability of presentation integration are determined by the availability, quality and reusability of individual components. Therefore, since the complexity of application development is shifted towards the component developer, elaborate *component design processes* and tools must support them.
- As the domain of lightweight composition on the web is still in its infancy, *non-functional properties* like security, reliability, performance, or quality are not yet adequately addressed.
- It is generally hard to find the right balance between *completeness* of features and *simplicity* of the composition approach. In this respect, tools and platforms must be aligned with its target users as well as application domain and requirements.
- State-of-the-art approaches aim at enabling average users to develop applications by *simplifying technology*, which is simply not enough. What users really lack is *development knowledge*, which thus needs to be provided somehow in order to really empower average users, e. g., by learning from expert programmers.

As mentioned above, in addition to simplifying technology, it is equally important to *support reuse* – not only in terms of implementation-level components, but also in terms of *domain knowledge* and, more importantly, in terms of *composition knowledge*. With regard to our reference scenario, the former implies that components presented to John should have an immediate meaning to him, and a platform should allow only compositions that make sense in the given domain. The latter could be achieved by recommending composition patterns that have been used successfully in the past, or by semi-automated application evolution derived from how other people improve their own compositions.

If we place the three discussed instruments into a simplification/reuse quadrant (Figure 1), we can easily see that reuse is so far limited to components only, while there is no support to tailor presentation composition platforms to specific domains or to reuse composition knowledge.

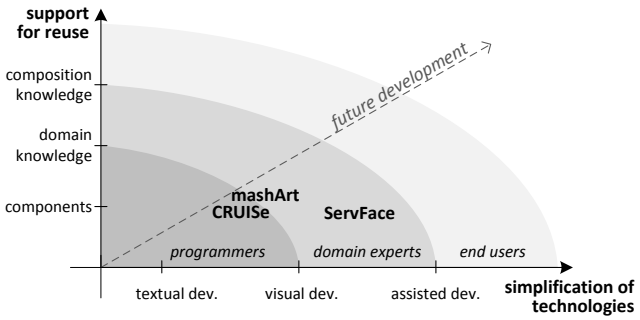

Figure 1: Simplification/reuse relation of presentation composition approaches

Moving into that direction will be the object of our future research and, hopefully, also of other researchers in this area, as doing so involves a number of challenges that can only be achieved by joint, interdisciplinary approaches.

Acknowledgments

This work is partially supported by the EU Research Project (FP7) ServFace. The CRUISe project is funded by the BMBF under promotional reference number 01IS08034-C.

6. REFERENCES

[1] Daniel, F., Yu, J., Benatallah, B., Casati, F., Matera, M., Saint-Paul, R.: Understanding ui integration: A survey of problems, technologies, and opportunities. Internet Computing **11**(3) (May/June 2007)

[2] Daniel, F., Casati, F., Benatallah, B., Shan, M.C.: Hosted universal composition: Models, languages and infrastructure in mashart. In: ER. (Nov. 2009)

[3] Feldmann, M., Nestler, T., Jugel, U., Muthmann, K., Hübsch, G., Schill, A.: Overview of an end user enabled model-driven development approach for interactive applications based on annotated services. In: Proc. of the 4th Workshop on Emerging Web Services Technology, ACM (2009)

[4] Pietschmann, S., Voigt, M., Rümpel, A., Meißner, K.: CRUISe: Composition of Rich User Interface Services. In: Proc. of the 9th Intl. Conf. on Web Engineering (ICWE 2009). (June 2009) 473–476

[5] OASIS: Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html (2007)

[6] Sun Microsystems: JSR-000168 Portlet Specification. http://jcp.org/aboutJava/communityprocess/final/jsr168/ (2003)

[7] OASIS: Web Services for Remote Portlets (WSRP). http://oasis-open.org/committees/wsrp (2003)

[8] Janeiro, J., Preußner, A., Springer, T., Schill, A., Wauer, M.: Improving the development of service based applications through service annotations. In: Proceedings of IADIS WWW/Internet 2009. (2009)

[9] Mitschick, A., Pietschmann, S., Meißner, K.: An ontology-based, cross-application context modeling and management service. Intl. Journal on Semantic Web and Information Systems (Feb.)

[10] Cappiello, C., Daniel, F., Matera, M., Pautasso, C.: Information Quality in Mashups. IEEE Internet Computing **14**(4) (July-August 2010) 14–22