A Model-Driven Development Process and Runtime Platform for Adaptive Composite Web Applications

Stefan Pietschmann

Technische Universität Dresden
Faculty of Computer Science, Chair of Multimedia Technology
01062 Dresden, Germany
Stefan.Pietschmann@tu-dresden.de

***Abstract*—So far, little research has addressed composition and integration at the presentation layer of web applications. Service-oriented architectures provide uniform models for encapsulation and reuse of data and application logic in the form of web services, but this paradigm has not yet been applied to the presentation layer, impeding a *universal* composition of web applications. Thus, UIs are usually handcrafted, lack flexibility and reusability, resulting in an expensive and onerous development process. We address these issues with a model-driven development process and a corresponding runtime architecture facilitating the universal, dynamic composition of web applications. Therein, user interface parts are as well provided "as a service" and can thus be selected, customized and exchanged with respect to the current context. We validated our approach using a prototypical implementation and a number of sample applications.**

***Keywords*-web engineering, model-driven development, composite applications, mashups, user interfaces, adaptive hypermedia**

I. Introduction and Motivation

The WWW has evolved into a rather stable, universal software platform. Numerous applications are provided as "Software as a Service" (SaaS) over the Internet, leveraging the location- and time-independent access as well as new business models like pay-per-use.

An enabling paradigm for this trend is the so-called *Programmable Web* [1]. Therein, data and application logic are provided in a decoupled and technology-independent fashion via generic service interfaces or APIs. Web-based applications can and will increasingly be composed from such distributed and reusable parts or services. A seminal[1] type of such *composite applications* are *mashups*, which create added value by combining web resources, e. g., data and logic from local or remote services. They allow for a shorter development cycle and thereby more situational applications that foster the *Long Tail* [4] of software needs. In this context, component-based concepts from academia for a more structured design process based on a "universal composition" have been proposed [5]. We build on this idea of composite applications and use the term synonymously to "user interface (UI) mashup" throughout this paper.

[1]According to Gartner, by 2010 mashups will be the predominant model (80%) for the development of composite enterprise applications [2]. Other institutes, e. g., Forrester, also underline their growing importance [3].

The underlying, service-oriented approach has simplified the integration at the data and application layers through standardization efforts and frameworks. Web services allow for the technology-independent encapsulation and deployment of functionality, which facilitates flexibility of the business logic by their exchange and custom configuration. However, *presentation integration* has not yet been addressed by research adequately [6], so there is a lack of comparable efforts for the presentation layer. Current concepts and technologies lack proper reuse mechanisms and interoperability.

Due to mobile and decentralized access to web applications, mashup developers face the problem of the heterogeneity of users and devices. To fully exploit the advantage of time-, location- and device-independent access, web applications need to adapt to the current situation, i. e., context (location, screen resolution, etc.), while preserving usability standards. This has dramatically complicated UI development. However, research in this field is still largely restricted to basic hypermedia systems and suffers from the "open corpus problem" [7]: Those approaches only work well for closed systems with predefined structures and preindexed or annotated documents, but fail when it comes to context-aware Rich Internet Applications (RIA) and unforeseeable, dynamic content. Thus, users are yet again facing "one-size-fits-all" UI solutions, which seems like a step backwards from the achievements of the "Adaptive Hypermedia" community over the last decade.

Additionally, UI developers are confronted with a myriad of (not necessarily new) programming languages, web frameworks and technologies to choose from. These offer a high level of UI individualization as opposed to classical desktop applications relying on uniform window-based UI libraries. This degree of freedom is an advantage, but it often results in inconsistent interaction metaphors between different applications, in low usability and thus confused users. Overall, development and maintenance of user interfaces still add up to about 70% of the overall software development [8] – a problem which is further intensified by challenge of context-awareness described above.

To address the above-mentioned problems, we strive for a model-driven, platform-independent development of composite web applications, and their context-aware execution by a service-oriented UI integration and composition system [9], [10]. By extending the service-oriented approach from the business layer to the presentation layer, we facilitate reusability and flexibility therein and thus simplify the development of context-aware rich web applications.

This article is structured as follows. In Section II we discuss relevant work related to web-based UI composition models and systems. Section III describes our model-driven approach to develop mashup applications, providing details on the underlying component and composition models. The concept of the related composition infrastructure, including the deployment process and run time integration of UI components, is presented in Section IV. After a brief discussion of implementation details, Section V illustrates the practicability of our approach by means of two sample applications. Section VI concludes this article and outlines future research directions.

II. Related Work

As described in the last section, the Web lacks uniform models for web-based components as well as open models and systems for their composition. We therefore present and discuss related composition and integration efforts.

Research in the field of composite web applications with focus on *presentation integration*, faces five fundamental challenges [11]. These comprise the development of a *component model*, a *composition model*, adequate inter-component *communication styles*, as well as mechanisms for *discovery and binding* and *visualization* of the UI components. In this article we focus on the first three issues, since run time discovery of user interface components is still ongoing work and visualization is usually carried out by the browser.

There already exist numerous component and composition models for the Web. The problem with both client-side (JavaScript frameworks, Applets, ActiveX Controls, Flash, etc.) as well as server-side (Portlets, ASP.NET Web Parts, etc.) solutions is that they all imply their very own interfaces, communication models, and, moreover, technological platforms. We aim for a uniform approach wrapping technology-specifics behind a generic component interface, comparable to web services. Also, with regards to popular UI frameworks such as the YUI library, we provide more complex, high-level components with integrated presentation and application logic.

As an example, *Portlets* are one of the oldest and most mature models for UI integration on the web. By composing them within a *Web Portal*, users are presented a consistent interface of several integrated UI parts forming a "Single Point of Access" for different back end services [12]. Thus, a Web Portal constitutes both a composition framework and common presentation layer of Service-Oriented Architectures [13], [14]. However, the use of portals comes with a number of disadvantages [15]. In contrast to more modern approaches they are limited to the server-side aggregation and communication of portlets. Despite the standardization of portlets, other portal characteristics, e. g., the layout, remain vendor-specific. Furthermore, with respect to our requirements, sophisticated concepts for a model-driven, platform-independent development and for the adaptation of portal applications are missing.

As already mentioned in the last section, *mashups* are an emerging, lightweight trend for the development of composite applications. However, the majority of current tools and platforms are still in their infancy [16] and concentrate on the integration of data and application logic, thus overlapping with composition systems like portals [17]. Integration is usually based on the *Piping* style [18] and supported by (often visual) composition languages and tools (Yahoo Pipes[2], JOpera [19]). User interface development is not [19] or insufficiently supported, as in Microsoft Popfly[3]. Enterprise-oriented platforms equally require authors to program the UI traditionally with the help of JavaScript libraries (JackBe Presto[4]) or WYSIWYG editors (Serena Business Mashups[5]).

Lately, an alternative SOA composition principle to portals has been proposed for the presentation layer [14]. Therein, so-called *widgets* or *mashlets* [15] are combined, each representing a self-contained application with its own user interface. Standardization of such "web parts" is currently pushed, e. g., by the W3C [20] and Google [21].

Few scientific approaches have addressed the challenges of *presentation integration* [11], one of the first being the "Presentation Integration Framework" *Mixup* [22]. Using a *Composition Middleware*, heterogeneous presentation components are assembled based on a declarative composition description and platform-specific adapters. Rather than adapters, our approach uses a generic wrapper that provides platform-specific UI components as a service. Thereby, components can be distributed and exchanged at run time, while in [22], [23] only design time composition of locally available components is supported. Such components are loosely coupled by dividing the component and composition models, and by using *publish-subscribe* mechanism for communication. As the *Mashup Component Model* presented in [24], Mixup only composes portlet-like components that constitute full applications. Thus, the approach lacks a separation of the traditional application layers and impedes UI flexibility, i. e., adaptivity, at run time.

[2]http://pipes.yahoo.com/
[3]http://www.popfly.ms/
[4]http:/www.jackbe.com/
[5]http://www.serena.com/geo/de/products/business-mashups/

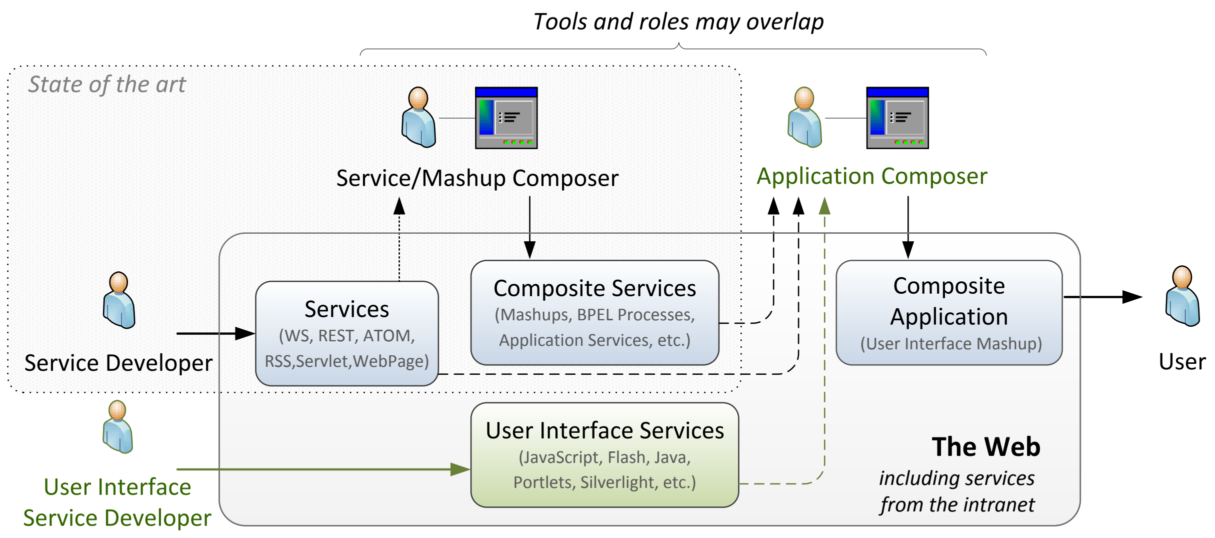

Tools and roles may overlap
State of the art
Service/Mashup Composer
Application Composer
Services
(WS, REST, ATOM, RSS,Servlet,WebPage)
Composite Services
(Mashups, BPEL Processes, Application Services, etc.)
Composite Application
(User Interface Mashup)
User
Service Developer
User Interface Services
(JavaScript, Flash, Java, Portlets, Silverlight, etc.)
User Interface Service Developer
The Web
including services from the intranet

Figure 1. Role model of the UI mashup development

Within the follow-up project mashArt [5] the component model was extended to be “universal” for UI, application and data building blocks. A fixed client-server infrastructure is provided, combining the client-side Mixup platform with a server-side part, whose necessity can be seen as an disadvantage [25]. Means for specifying control flow and adaptivity of an application as well as their support by the runtime environment are missing. Most importantly, in contrast to our approach, both Mixup and mashArt do not offer a platform-independent, model-driven development.

An alternative approach for service composition at the presentation layer is proposed by *ServFace* [26]. Herein, the user interface is generated from UI annotations of the orchestrated services beneath at design time. Dynamic context-awareness is not considered in this generation process, though. Moreover, such a transformation usually results in simplistic, form-based UIs, while we aim for the integration of rich, potentially multi-modal UI components that can undergo functional and usability tests prior to deployment.

As can be seen, there exist promising concepts for the integration and composition of web-based services and resources. However, due to the lack of (de-facto) standards for the description of components and compositions, those approaches suffer from interoperability problems. Compositions are usually based on proprietary models and lack support of desirable and increasingly necessary aspects like dynamic configuration and composition [16], control flow, and adaptivity of such applications. Traditional, model-driven concepts for the development of web applications, such as WebML [27] and OO-H [28] can not be directly applied to the mashup domain since they are too complex [5] and document-centric. As a result, the development of adaptive, context-aware mashup applications remains highly time- and money-consuming.

III. Model-Driven Development of Context-Aware, Composite Web Applications

To overcome the restrictions discussed above, we present a novel concept for the model-driven development and deployment of composite applications. Its central idea is the application of the service-oriented paradigm to the mashup presentation layer to support a universal composition on all application layers (data, application logic, and UI) and to simplify the overall development of context-aware, composite web applications. By using services to compose a web-based user interface, we facilitate reuse, customizability and technology-independence. We do this by (1) the encapsulation of generic, reusable web UI components (UIC), (2) their distributed deployment as so-called *User Interface Services* (UIS) and (3) their context-aware, dynamic invocation, configuration and integration with other mashup components, resulting in a fully service-oriented, composite web application.

Figure 1 gives an overview of the roles participating in the development of such a universal composite application. Traditional service developers (on the left) provide application- or domain-specific data as well as application logic via services, which are combined with the help of existing composition tools and frameworks, e. g., data mashup platforms and BPEL engines. Our concept of *User Interface Services* introduces reusable, service-based UI components that can be composed together with traditional services to a composite application, i. e., *user interface mashup*.

In this section we provide details on the underlying, generic component model for the encapsulation of reusable parts of a mashup, on their description and service-oriented provision, and on the composition model that defines an application as an arrangement of such components.

A. Component Model

To allow for a universal composition of an application, its constituent parts need to adhere to a generic component model. It defines the basic structure and interface of application building blocks, being either user interface parts, application logic or data providers. The model discussed in the following is a successor of our previous results presented in [9]. Following the principles introduced with web services, it does not dictate any internal component structure or format. However, it specifies fundamental characteristics of component interfaces, which are relied upon for their integration and the communication among themselves.

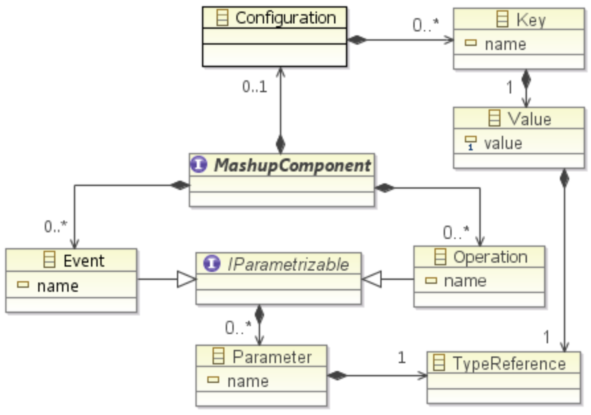


Figure 2. Universal Mashup Component Model

As illustrated by Figure 2, our model characterizes mashup components by three abstractions, namely configuration, event, and operation.

The *configuration* of a component resembles its visible state, which differs somehow from the service-oriented paradigm mentioned: While (web) services are loosely coupled and stateless by design, there is a special need for modeling a state of mashup components, as they form “a natural bridge between application services and data-oriented services” [5]. In our model, the state is represented by arbitrary key-value pairs, which are defined by the component developer and may contain any data that seems relevant to the component’s surrounding. Examples include a graph’s type (line, bar, pie) and a map’s projection or type (normal, satellite, hybrid). In contrast to prevalent solutions, e. g., mashArt [5], our model supports complex value lists and trees by allowing XML-schema complex types to be able to map complex component-internal data structures more naturally to external properties.

To publish state changes to other components or the composition environment, a component can issue *events*. They may be triggered by user interaction (UI), time (logic) or notifications from external services (model). Besides a name, events can contain data in the form of a number of typed parameters, i. e., key-value pairs. Picking up the example from above, a map could issue an event *Location-Selected(String countryCode)* upon user interaction.

Operations are methods of a component which are triggered by events. They can include any functionality foreseen by the developer, such as state changes, calculations, or service requests. As an input, they consume parameters provided by the trigger events. As an example, an operation *getCountryInfo(String countryCode)* could be triggered by the aforementioned map event, which would result in a SOAP request by the component to a web information service providing information about the country. Its response would again be published as an event to be consumable by other components. As event and operation parameters don’t always fit as nicely as in our example, we facilitate the definition of a mapping, so that parameter names and order become irrelevant to the wiring.

The component model presented here is specifically designed to support the loose coupling of application components based on a publish-subscribe mechanism (cf. Section III-B). Thus, events and operations form the basis of all application-internal communication.

Within a user interface mashup, we can distinguish different component types. Although they all comply with the component model presented above, they differ in the semantics, i. e., in the application layers they apply to. In the composition model discussed later, we differentiate between four types: At the topmost layer, *User Interface Components* (UIC) encapsulate parts of an application UI with corresponding presentation logic. A popular example would be an interactive map, as provided by Google or Yahoo. To support an efficient communication between components, we can employ *Logic Components* (LC). They provide means for data manipulations, e. g., transformation, filtering, or aggregation, so that parameters of events and operations fit, even in combinations unforeseen by their developers. Especially the use of complex data types, such as `Person`, necessitates data transformations to make parts of it (surname, age, gender) processable by operations of other components. Finally, at the lowest application level, *Service Components* wrap access to services providing data or complex application logics, e. g., via SOAP or REST.

Since we aim for flexible, context-aware user interfaces, deployment and description of UI components differ slightly from other component types. As already mentioned, **User Interface Services** (UIS) form an integral part of our concept. They facilitate the distributed deployment, technology-independent provision, and integration of above-mentioned UI components via a public service interface – something already common to back end services. A trend towards such services for the presentation layer can already be witnessed, prominent examples being Google’s Maps or Visualization APIs, that offer the integration of configurable, interactive

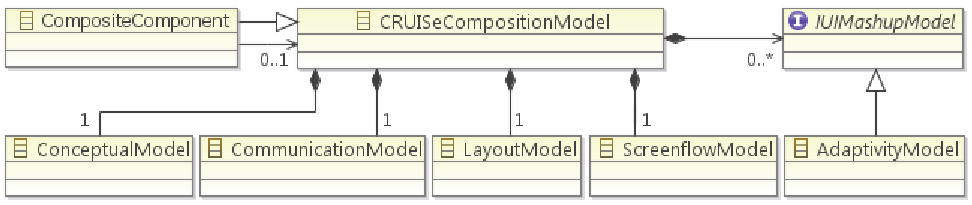


Figure 3. Main parts of the Composition Metamodel

maps and charts from a remote server. We generalize such techniques and propose a concept, in which the whole web application UI results from the integration and composition of UIS, or, more precisely, the UI components provided by them. UIS imply that back end development and UI design can be completely decoupled by using services, whose combination eventually results in a composite application.

Following this approach, we can equally bind web and UI services at runtime. We can even link the UIS selection and integration process with contextual information, such as user preferences, device capabilities, and the integration platform. This allows for context-aware, composite application UIs, but also implies that we need some kind of UIS description.

To dynamically select a UI service, an open interface description is needed to specify the characteristics of the corresponding UIC, including its semantics (purpose) and signature (operations and events). Moreover, since – opposed to traditional services – UI services provide components to be integrated (not executed remotely), those need to be wrapped with respect to the integration platform, i e., the technology and framework used. Consequently, a UI description needs to provide information on the compatibility and language bindings of a component.

Thus, similarly to service components described by WSDL or WADL, the interface of UIS is specified with the help of a *User Interface Service Description Language* (UISDL). It is an XML-based description of all information needed to select, integrate and use a UI component provided by the corresponding UIS. Therefore, it consists of two parts: the UISDL *class* description defines the generic interface of a component, i. e., its name, semantic concepts, license, and signature (properties, operations, events); the UISDL *binding* describes the mapping of a platform-specific component implementation to a class, including constructor information, references to required libraries, etc. UISDL metadata is stored in a *Component Registry* (cf. Figure 4) and used to dynamically match application requirements and context data with available UIS – a process which is discussed later in Section IV-B. Details on the UISDL are out of focus of this article and will be published separately.

In summary, the lightweight component model presented here describes reusable building blocks for the composition of mashup applications and supports synchronization on all its layers. Service components allow for the integration of arbitrary external services, their data being transformed by logic components and visualized by UI components. The latter are distributed and provided in a service-oriented fashion, the uniform component interface and the declarative interface description language UISDL hiding specific APIs and technologies. This approach enables interoperability and run time exchangeability of user interface parts and thus forms a basis for context-aware mashup user interfaces.

B. *Composition Model*

To compose web applications from components as discussed in the last section, a platform-independent model is needed to define all relevant aspects of an application, including the components used, the communication among them, the overall control flow as well as the layout of the user interface. Therefore, we have developed an extensible metamodel to describe all of the above aspects. Figure 3 provides a simplified overview of the *CRUISeCompositionModel* and its submodels, each describing a specific application aspect. As can be seen, any composition model also represents a component itself (*CompositeComponent*) and can thus be included in higher level compositions, either by reference or direct inclusion. The interface *IUIMashup-Model* facilitates extensibility to model additional aspects, as exemplified with the *Adaptivity Model*. In the following, we give a brief overview of the main parts, i. e., submodels of our composition model. Further details are beyond the scope of this article and will be published separately.

The *Conceptual Model* contains all application-wide concepts. Most importantly, these comprise the components of an application – the different types presented above including specific configurations. Since their events and operations contain typed parameters, data type definitions (XML Schema) of all complex types used are included or referenced there. Additionally, reusable style classes can be modeled and applied to different UI components – comparable to CSS – to achieve a homogeneous *look and feel* of an application. Finally, two more *special* components round off the Conceptual Model: one allows for accessing context variables which are provided by a dedicated service at run time. Those context parameters can be connected with other application components, e. g., to constantly feed a map with the current location of a user. The other special component defines the external interface and execution of a composition. It includes application-wide variables, events to be issued at application initialization, and references to those events and operations of the composition that shall be accessible externally by a higher-level composition.

The layout of the composite application is described by the *Layout Model*. A developer can use multiple predefined layouts, following layout managers as common in several UI libraries. As an example, a *Grid Layout* with multiple rows and columns can be defined, each cell containing a certain mashup component (or another layout). Layouts can be hierarchically nested to achieve any desired arrangement of UI components.

Layouts are utilized within the *Screenflow Model* to describe several views (pages) of an application. Each view is represented by a specific layout, i. e., a visual state of the mashup with certain UI components visible. One view is the marked as initial, but plenty others can be defined. Transitions between views are triggered by events issued from components or the environment (the runtime system). This allows for very flexible and multi-step user interfaces.

The *Communication Model* supports arbitrary communication paradigms. We currently model communication based a *publish-subscribe*-mechanism, wherein channels connect publishers (events) with subscribers of information (operations). Developers can create manual mappings of channel and operation parameters. This is useful, when semantically different parameters differ in their names, or when event and operation parameters simply occur in a different order.

Finally, the *Adaptation Model* illustrates the extensibility of our metamodel. It was added to facilitate the definition of adaptation *aspects* crosscutting all of the above-mentioned models. Each aspect is triggered by an arbitrary event within the application, which leads to the validation of a condition defined by the composer. An aspect specifies both the part of the model to adapt, and an action which defines exactly what to do. As an example, it could change a layout, insert an additional component's configuration parameter, or exchange one component with another. With the help of this model, we can specify adaptations of a composite application at very different levels of granularity.

A complete composition model is transformed into an executable web application in a multi-step process, including a number of model-to-model and model-to-code transformations. They can be triggered dynamically by a client request, or statically during design. Alternatively, the model can be directly interpreted by dedicated runtime systems.

In conclusion, the composition metamodel presented provides the means to model all necessary aspects describing a service-oriented, composite application in a platform-independent fashion. Therefore, generic components (cf. Section III-A) at different application layers are integrated and linked by communication channels. The composition paradigm supports a seamless integration of service-oriented UI components, similarly to the integration of service-oriented back end logic and data. With the help of an adaptation model, any aspect of an application can be altered with respect to a particular context, resulting in highly flexible, self-adaptive compositions.

IV. CRUISe: A System Architecture for Adaptive User Interface Mashups

While the modeling approach presented in the previous section specifically focuses on design time of adaptive composite applications, this section presents an open and flexible system architecture for their dynamic composition and execution, called CRUISe. After a brief architectural overview, we present our concept in detail. First, we explain how an application is initialized including the dynamic integration and composition of remote UI components to an application UI. Then, we highlight details on its runtime adaptation.

A. Architectural Overview

Figure 4 gives an overview of the overall conceptual infrastructure of CRUISe. Its central concept is the use of distributed services for the dynamic composition of web applications to exploit the advantages of service-oriented architectures, like reusability, customizability and technology-independence at all application tiers, including the presentation layer. As mentioned in Section III-A, we do this by dynamically selecting, configuring, and integrating generic, reusable UIS into an application UI, and binding them to application-specific logic and service components.

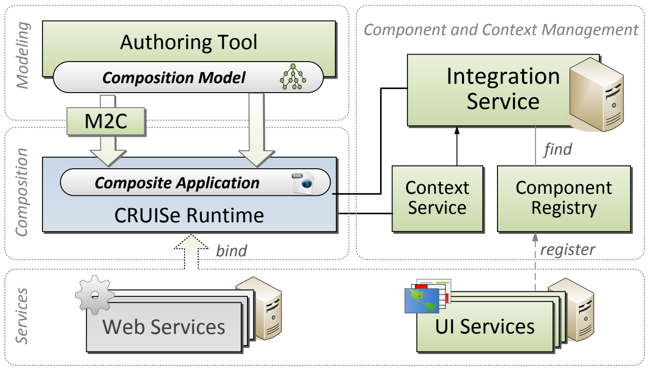


Figure 4. Architectural overview of the CRUISe infrastructure

On the left, Figure 4 illustrates the model-driven nature of our approach: Initially, a composite application is defined with the help of dedicated visual authoring tools and by means of the platform-independent *Composition Model*, presented in the last section. The latter is transformed to a platform-specific, executable application, either statically at design time, or triggered by a client request at run

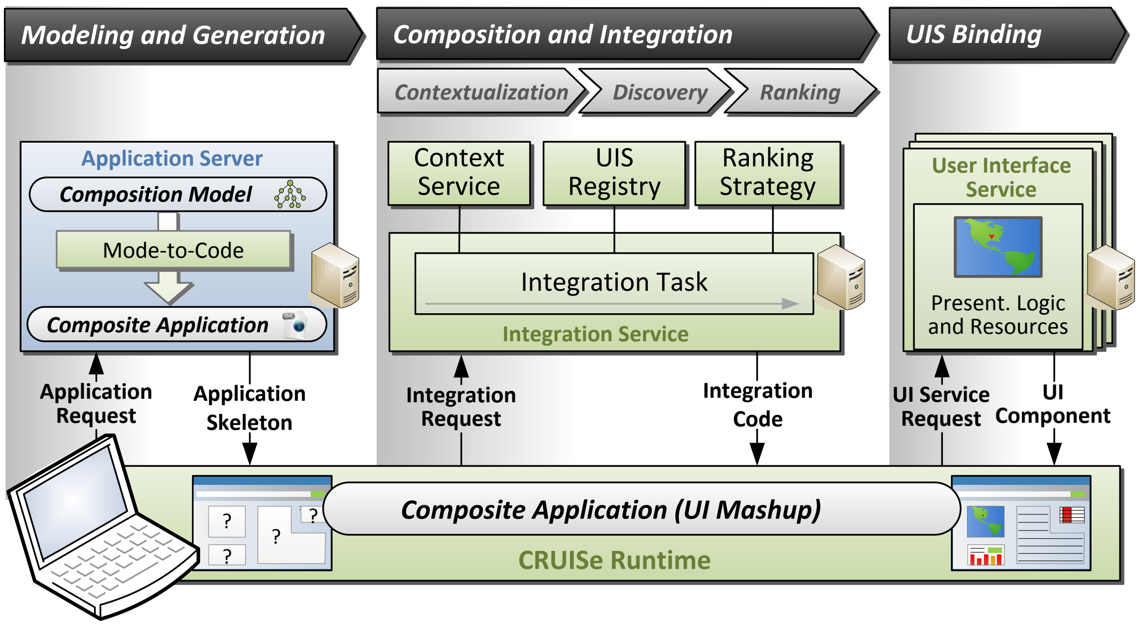


Figure 5. UI composition process

time. Alternatively, the model can be directly interpreted by dedicated run time environments.

The resulting *Composite Application* is executed by a runtime platform which provides all necessary means for controlling the application aspects specified in the composition model (event, data and control flow, service access, etc.) – the *CRUISe Runtime*. During application initialization, it is specifically responsible for invoking the *Integration Service*, which provides UI components "as a service", matching given application requirements and context information with available UIS listed in the *Component Registry*. Once the integration of UI components into an application is finished, the *Runtime* controls its execution. Additionally, it monitors context data and sends it to a context management service. There, it is processed, refined, and later provided to be used in the discovery and ranking process of UIS, as well as for the dynamic adaptation applications.

With our first prototype [9] we gained some useful knowledge on web-based UI integration and – as an outcome – decided to keep the place of integration conceptually flexible. Thus, the *Runtime* can reside both on the server or on the client-side, depending on the application requirements. For instance, service authentication in an enterprise setting necessitates a server-side part. Alternatively, the *Thin-Server Runtime* [29] allows for a completely browser-based composition and execution of consumer-oriented UI mashups, conforming to the SOFEA architectural style [25].

The next sections provide some more insight into the composition and integration process, as well as into dynamic adaptation mechanisms included with the *Runtime*.

B. Dynamic, Context-Aware Composition

The composition process outlined in the last section is illustrated by Figure 5. As can be seen, the provision of a service-oriented user interface for a mashup application is based on an integration work flow, which consists of three subsequent steps, namely (1) application generation, (2) UIS integration and (3) UIS binding. In this section we focus on steps 2 and 3.

The generation process results in a so-called *skeleton*, containing placeholders instead of actual component instances so that UI components can be "bound" at runtime just like back end services. Thus, at application startup, the *Runtime* is responsible for integrating UI components and for initializing them together with the other components included within the composition. Therefore, it sends a request containing application- and context-dependent requirements for each component to be integrated to the *Integration Service*. Subsequently, the latter starts an *Integration Task*, illustrated by Figure 6, which consists of different modules each responsible for a certain integration aspect. Its purpose is to find those UIS in the *Component Registry* that match given application requirements and context, to rank them by their accuracy of fit, and to return the platform-specific *binding* (cf. Section III-A) for the best match to the *Runtime*.

Since the *Integration Service* has an open service interface for requesting platform-specific components bindings, it can be used by different kinds of integration platforms – both client- and server-side. This even allows for the integration into independent solutions, e. g., into the JSP compilation process [9], into human tasks included in a business process [10], but also directly within the browser [29].

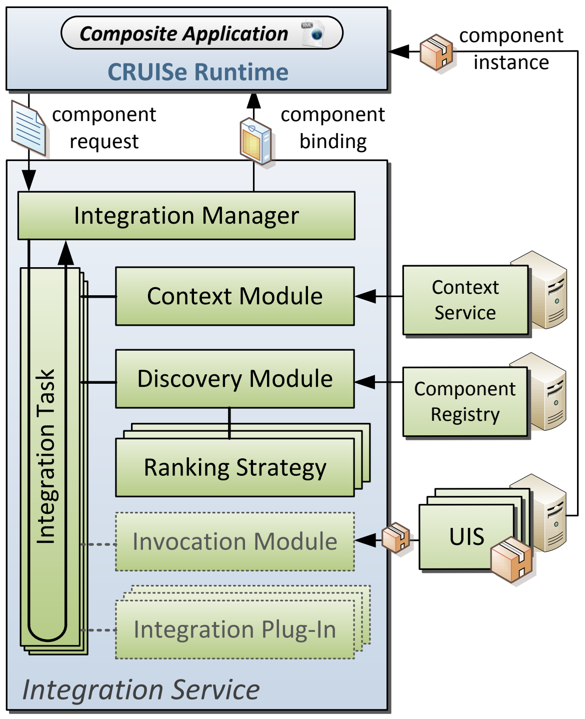


Figure 6. Internal structure of the *Integration Service*

The following modules are involved in the matching and integration process:

Integration Manager: This module handles all incoming component requests. Thus, it marks the remote entry point to the integration system and manages its internal data flow with the help of so-called *Integration Tasks*. For every request, a new task is started, which comprises a chain of actions (represented by individual modules) eventually returning a platform-specific component binding that best meets the given application requirements and context.

Context Module: Since the selection and configuration of UIS are based on contextual data, this module resolves references to context information being part of the transferred requirements into actual data, and evaluates context conditions. The quality and amount of this data heavily depends on the underlying context monitoring and modeling system. Therefore, we suggest the use of a sophisticated, service-oriented solution as described below.

Context Service: Modeling context places high demands on adaptive systems, including consistency checks, validation, and reasoning of information. In [30] we have presented a suitable, service-based solution called CroCo, which allows for the cross-application, ontology-based context management and reasoning, covering the above-mentioned requirements. Arbitrary context providers, such as the *Runtime* itself, other applications on the user's device, or hardware sensors, can send information to CroCo. Likewise, the *Context Module* and the *Runtime* request context data from the service, either synchronously or asynchronously by using a callback interface. More details on CroCo are discussed in [30].

Discovery Module: This module requests suitable component descriptions from the **Component Registry** – comparable to UDDI. Currently, we focus our work on the discovery and integration of UI components. Discovery within the registry is based on component *class*, e. g., "Map". In response, a result set of UISDL bindings is returned, which has to be ranked afterwards to determine the most adequate UIS. Therefore, the *Discovery Module* passes the set to a *Ranking Strategy*.

Ranking Strategy: In this step, the list of UIS in question is sorted with regard to predefined, possibly context-dependent criteria. Different ranking algorithms, or "strategies", may exist. They can be exchanged dynamically to support domain- or application-specific weightings of ranking criteria. Hence, the discovery process is divided into a class-based, functional matching carried out by the registry, and an application-specific, context-aware ranking performed by the strategy within the *Integration Service*.

Invocation Module: The necessity of a server-side invocation of UIS depends on whether there exists a suitable UISDL *binding* for the particular integration platform. As mentioned in Section III-A, it describes, how a component is integrated into a specific technology, i. e., how it can be initialized and how the interface signature maps to internal parameters and methods of the component. If, for example, a UI component should be integrated on the client-side into a JavaScript environment, a corresponding JavaScript-based component can be integrated directly on the client by loading the remote script in the browser. However, if technologies differ, UI components need to be wrapped in platform-specific code by the *Integration Plug-In* and are thus loaded from the UIS beforehand.

Integration Plug-In: In the ideal case, this module only extracts the necessary integration code from the UISDL *binding*. This includes initialization code, e. g., the constructor, as well as dependencies to other libraries. If no suitable binding exists, the component is wrapped with code, specific to the integration platform. Thus, for every runtime platform, there exists a corresponding *Integration Plug-In*. As an example, we have developed a plug-in, which integrates JavaScript-based components into Eclipse RAP[6] applications by automatically creating the corresponding server-side life cycle classes and by dynamically integrating them with the help of the *RAP Runtime*.

Once the *Integration Task* is finished, the *Integration Service* returns a platform-specific *binding* or wrapped component to the *Runtime*. It is interpreted or embedded into the composite application and executed, eventually. This can be referred to as *UIS binding* as shown in Figure 5. Of course,

[6]http://eclipse.org/rap/

this process involves a number of additional tasks, such as error handling and the provision of distinct namespaces to assure unique identifiers for each component included. However, those actions are not discussed in detail here.

After an application has been successfully initialized, the *Runtime* controls the event and data flow between its components as specified in the composition model. It also serves as a homogeneous access layer for various back end services. Furthermore, it monitors context data on the client, like user interactions and device capabilities, and sends them to the *Context Service*. In the end, it also carries out dynamic adaptations of the composite application, which is discussed in the following section.

C. Dynamic Adaptation of the Composite Application

As motivated in the beginning, situational awareness becomes increasingly important for web applications and poses additional challenges for web developers. Web applications need to adapt to different end users (characteristics, preferences, roles), devices (screen size, resolution, plug-ins) and situations (time, location, etc.). In CRUISe, context-awareness can be attained in different ways.

First and foremost, since UI components are selected and configured dynamically at run time, this process can be influenced by arbitrary context data, as discussed in the last section. For instance, the availability of necessary plug-ins on the client (e. g., Flash) can be taken into account when deciding which UI component to integrate.

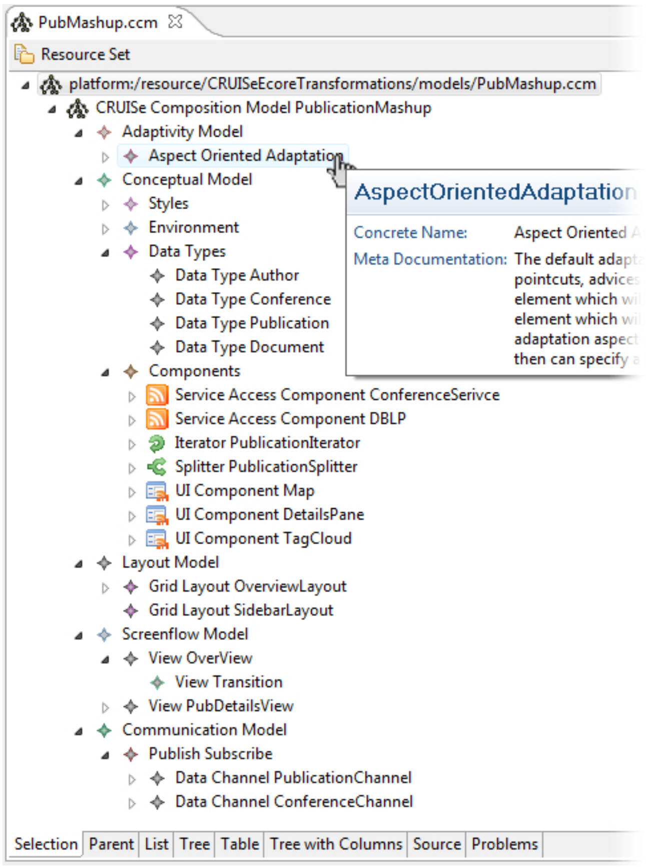
Second, context parameters may directly influence the configuration and state of a mashup component. This can be achieved by wiring contextual events from the context component (cf. Section III-B) with other components of an application. As an example, a location-aware map can be configured in such a way, that events from the context component providing the current geolocation trigger the map operation that updates its marker accordingly. Similarly, context parameters can be referenced within initialization events of an application, which results in a context-aware component configuration.

Finally, the *Runtime* contains an adaptation infrastructure, which dynamically evaluates context conditions specified in the *Adaptivity Model*, and carries out adaptations accordingly. They include component reconfiguration and exchange, adaptive layout and communication, as well as migration of components between client and server. Adaptation within the *Runtime* is defined with the help of rules, which define comporise context events, corresponding conditions, component references, and adaptation actions. Context data is requested or actively pushed from an external context service, e. g., CroCo. Details on adaptation rules, techniques, and context management are beyond the scope of this paper and will be published separately.

V. Implementation

To verify the concepts presented in the previous sections we implemented the composition model and the CRUISe infrastructure, and tested them with the help of different, exemplary composite applications.

The composition metamodel discussed in Section III-B was realized based on the meta-metamodel Ecore being part of the Eclipse Modeling Framework[7] (EMF). By using EMF, application models can be serialized in XMI and hence become exchangeable and tool-independent. Furthermore, an API can easily be created from the metamodel, simplifying the subsequent transformation step by using a number of already existing languages, such as QVT [31]. We also generated a tree editor, which integrates seamlessly with Eclipse (cf. Figure 7), and makes modeling composite applications rather easy with integrated validation support. Overall, Eclipse offers a powerful and flexible environment for future extensions, including the development of a corresponding visual editor.



```
PubMashup.ccm
Resource Set
platform:/resource/CRUISeEcoreTransformations/models/PubMashup.ccm
  CRUISe Composition Model PublicationMashup
    Adaptivity Model
      Aspect Oriented Adaptatio
    Conceptual Model
      Styles
      Environment
      Data Types
        Data Type Author
        Data Type Conference
        Data Type Publication
        Data Type Document
      Components
        Service Access Component ConferenceSerivce
        Service Access Component DBLP
        Iterator PublicationIterator
        Splitter PublicationSplitter
        UI Component Map
        UI Component DetailsPane
        UI Component TagCloud
    Layout Model
      Grid Layout OverviewLayout
      Grid Layout SidebarLayout
    Screenflow Model
      View OverView
        View Transition
      View PubDetailsView
    Communication Model
      Publish Subscribe
        Data Channel PublicationChannel
        Data Channel ConferenceChannel

AspectOrientedAdaption
Concrete Name:        Aspect Oriented A
Meta Documentation:   The default adapt
                      pointcuts, advices
                      element which wi
                      element which wi
                      adaptation aspect
                      then can specify a

Selection | Parent | List | Tree | Table | Tree with Columns | Source | Problems
```

Figure 7. Composition model tree editor

To validate the flexibility of the integration process, we put into practice different *Runtimes*, both server- and client-side. Our first prototype used a *Client-Server Runtime*, in

[7]http://eclipse.org/modeling/emf/

which UI integration takes place on the server as part of the JSP compilation process. The integration is HTML-based, utilizing the jMaki[8] widget framework, which manages component interaction on the client side. To assure uniqueness of class and object identifiers of the integrated components (with respect to the composite application), we use so-called *Universally Unique IDentifiers* [32], which are assigned during the integration process.

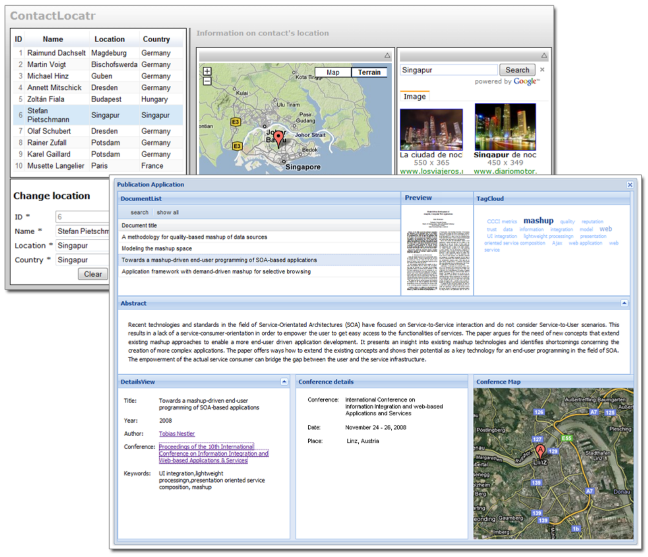
In [10] we presented the dynamic integration of UI components into a BPEL-environment based on the BPEL4People and WS-HumanTask (WSHT) standards. Therein, our *Runtime* is a Java-based extension of the ActiveBPEL[9] *Task List Client*, which lists and presents all human-involved tasks to users. Our extension includes two components: a *Parser* interprets the serialized composition model, which is embedded in the Human Task description, and *Bridge* component handles all the communication with the *Integration Service* and dynamically embeds UI components into the task UI.

As part of our latest work, we developed a *Thin-Server Runtime* (TSR) [29] which runs completely within the browser, following the SOFEA architectural style [25]. Therefore, we extended the JavaScript framework Ext[10]. The resulting architecture manages the components' dynamic integration, life cycles and communication, and provides a homogeneous service access layer which redirects external service access to a proxy provided by the *Integration Service*. This is needed to bypass the client-side "same origin policy" [33] which prevents access to services outside the original application domain. All *Runtimes* feature adequate error handling strategies for different faults, e. g., during component request, integration, and initialization. Typically, the action is repeated first, before discovery of alternative components is started. Additional run time security mechanisms are in the working.

The *Integration Service* is realized in Java according to the architecture illustrated in Figure 6 with additional functionality like local resource management and caching. It features both a lightweight REST and a SOAP interface. Functionality of the latter is largely based on Apache Axis2[11] and thus benefits from steady improvements and comprehensive standards support. The *Component Registry* builds on the WSMO framework [34] and internally models components uses the Web Service Modeling Language (WSML).

For demonstration and testing purposes, several prototypical, composite web applications were designed and built. To this end, a number of UIS were developed, encapsulating typical UI components, such as maps (Google Map, Yahoo Map), charts (Google Visualization API), an image browser, a feed viewer, a tag cloud, etc. Technologically, these components range from simple HTML and JavaScript (Google Maps, Dojo) to Flash (Flex) and Google GWT.

Figure 8 shows two prototypes. The rear one is one of the first sample applications built upon our approach (cf. [9]). It allows for the management of contacts and provides additional information on their current location. Users may edit their data with the help of a form or by changing their location directly on a map. The application in the front lets users browse publications listed in a digital library. They can see details on the papers and check the corresponding conferences for related work. Different REST and SOAP services are utilized by this mashup, providing information on the authors and conferences which are visualized accordingly (e. g., keywords in a tag cloud, conferences on a map). The underlying composition model is partly shown in Figure 7.



ID	Name	Location	Country
1	Raimund Dachselt	Magdeburg	Germany
2	Martin Voigt	Bischofswerda	Germany
3	Michael Hinz	Guben	Germany
4	Annett Mitschick	Dresden	Germany
5	Zoltán Fiala	Budapest	Hungary
6	Stefan Pietschmann	Singapur	Singapur
7	Olaf Schubert	Dresden	Germany
8	Rainer Zufall	Potsdam	Germany
9	Karel Gaillard	Potsdam	Germany
10	Musette Langelier	Paris	France

ID *	6
Name *	Stefan Pietschm
Location *	Singapur
Country *	Singapur

Document title
A methodology for quality-based mashup of data sources
Modeling the mashup space
Towards a mashup-driven end-user programming of SOA-based applications
Application framework with demand-driven mashup for selective browsing

Recent technologies and standards in the field of Service-Orientated Architectures (SOA) have focused on Service-to-Service interaction and do not consider Service-to-User scenarios. This results in a lack of a service-consumer-orientation in order to empower the user to get easy access to the functionalities of services. The paper argues for the need of new concepts that extend existing mashup approaches to enable a more end-user driven application development. It presents an insight into existing mashup technologies and identifies shortcomings concerning the creation of more complex applications. The paper offers ways how to extend the existing concepts and shows their potential as a key technology for an end-user programming in the field of SOA. The empowerment of the actual service consumer can bridge the gap between the user and the service infrastructure.

Title:	Towards a mashup-driven end-user programming of SOA-based applications
Year:	2008
Author:	Tobias Nestler
Conference:	Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services
Keywords:	UI integration,lightweight processingn,presentation oriented service composition, mashup

Conference:	International Conference on Information Integration and web-based Applications and Services
Date:	November 24 - 26, 2008
Place:	Linz, Austria

Figure 8. Two mashup applications integrating several UIS

With the help of these prototypes we could prove the feasibility of our model-driven development process and architecture for different application scenarios and platforms. They exemplify the dynamic integration of UI components into a mashup composition process, the event-based synchronization among components on all application layers, and the technology-independence of our approach.

Our modeling tools support the easy, platform-independent composition of applications from services and UIS at low authoring cost. At run time, the server-side integration and composition seems expedient especially for enterprise scenarios, because the quality and authenticity of back end and front end services can be ensured by the application server. As a downside, this negatively affects performance when the number of services, composite applications, and users grows. Thus, for a large part of use cases, a client-side composition and execution seems favorable, considering ongoing standardization efforts to

[8]https://ajax.dev.java.net/
[9]http://www.activebpel.org
[10]http://www.extjs.com/products/extcore/
[11]http://ws.apache.org/axis2/

resolve issues with the "same origin policy". Obviously, performance within a thin-server setting heavily depends on the browser used. In our tests, Google Chrome, Mozilla Firefox, and Opera offered a reasonable performance.

Of course, the proposed abstraction at the presentation layer implies a certain overhead at both design and run time. As application complexity can not be eluded, our concept basically shifts it from the application composer to the component developer. Reliable, elaborate component implementations and accurate descriptions are key factors for our concept to succeed. Given this, composition with our tools and models simplifies and shortens overall development while yielding platform-independence and context-awareness. Run time overhead of the dynamic component discovery and integration is less of a problem, since our infrastructure proves to scale rather well using caching and redundant services.

VI. Conclusion and Future Work

The development of composite web applications with rich user interfaces is a time- and money-consuming task, as current approaches lack a universal composition approach and are limited to integration at the data and application layers. Providing context-aware UIs for such applications poses additional challenges for developers. To address these issues, we have presented a model-driven development process for composite applications based on a universal, platform-independent composition metamodel, and a corresponding execution platform and infrastructure.

Our concept proposes a generic component model defining configurable, reusable parts of an application, and a novel, service-based deployment method for UI components. It implies the dynamic, context-aware composition of a mashup UI from so-called *User Interface Services* (UIS). In this context, a corresponding component description language (UISDL) has been developed. Composite applications are described with a platform-independent composition model, which defines all components used, their configuration, communication, and layout, as well as screen flow and adaptive behavior. On a higher level of abstraction, it specifies the coupling of UI services with back end services.

We have designed and tested the composition infrastructure CRUISe, which provides the necessary means for application composition and execution. This includes the homogeneous binding of back end services and the dynamic, context-aware selection, configuration, and integration of UIS. Moreover, the architecture supports dynamic adaptivity and adaptability of the composite application by means of component reconfiguration, exchange, adaptive layout, etc. To validate our approach, we built several prototypes illustrating the dynamic composition of context-aware mashup applications for different usage scenarios and platforms.

To our knowledge, CRUISe marks the first model-driven and fully service-oriented approach to a universal composition: It greatly simplifies platform-independent development, reuse and maintenance of composite web applications with context-aware UIs by deploying UI components "as a service" – comparable to service-oriented back end logic. Besides, it enables novel business models in the form of potentially commercial UIS, which may offer visualization and interaction at a higher level than standard interfaces.

Currently, we are working on more sophisticated, context-aware selection mechanisms being part of the *Component Registry* and the *Integration Service*. To this end, we are building a semantic classification of UIS to support semantic run time matching. In parallel, we are developing a sandboxing concept to improve security and privacy mechanisms within our *Runtime*s, so that a certain level of stability and information quality can be guaranteed, regardless of the services used by applications. Additionally, we are working on a visual composition tool to further simplify the development process.

Future work includes the extension of component descriptors and the *Adaptation Model* with regard to adaptation: Components' ability to self-adaptation and adaptability must be described to form the basis for defining higher-level *adaptation concerns* [35] ("device independence", "location-awareness", etc.) in the composition model. Moreover, we plan to extend our infrastructure to manage and dynamically include all types of components, including compositions themselves.

Acknowledgements

The CRUISe project is funded by the German Federal Ministry of Education and Research under promotional reference number 01IS08034-C.

References

[1] E. M. Maximilien, A. Ranabahu, and S. Tai, "Swashup: Situational Web Applications Mashups," in *Companion to the 22nd Conf. on Object-Oriented Programming Systems and Applications (OOPSLA'07)*. New York, NY, USA: ACM, 2007, pp. 797–798.

[2] "Gartner Identifies the Top 10 Strategic Technologies for 2008," Gartner Inc., Tech. Rep., October 2007, http://www.gartner.com/it/page.jsp?id=530109.

[3] O. Young, E. Daley, M. Gualtieri, H. Lo, and M. Ashour, "The Mashup Opportunity," Forrester, Tech. Rep., May 2008.

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

[5] F. Daniel, F. Casati, B. Benatallah, and M.-C. Shan, "Hosted Universal Composition: Models, Languages and Infrastructure in mashArt," in *Proc. of the 28th Intl. Conf. on Conceptual Modeling*, November 2009.

[6] B. Benatallah and H. Nezhad, "Service Oriented Architecture: Overview and Directions," *Advances in Software Engineering: Lipari Summer School*, vol. 5316, pp. 116–130, 2007.

[7] P. Brusilovsky and N. Henze, "Open Corpus Adaptive Educational Hypermedia," in *Adaptive Web: Methods and Strategies of Web Personalization*, vol. 4321, 2007, pp. 671–696.

[8] A. Kleshchev and V. Gribovy, "From an Ontology-Oriented Approach Conception to User Interface Development," *Information Theories & Applications*, vol. 10, no. 1, pp. 87–94, 2003.

[9] S. Pietschmann, M. Voigt, and K. Meißner, "Dynamic Composition of Service-Oriented Web User Interfaces," in *Proc. of the 4th Intl. Conf. on Internet and Web Applications and Services (ICIW 2009)*. Mestre/Venice, Italy: IEEE CPS, May 2009, pp. 217–222.

[10] ——, "Adaptive Rich User Interfaces for Human Interaction in Business Processes," in *Proc. of the 10th Intl. Conf. on Web Information Systems Engineering (WISE 2009)*, WISE. Springer LNCS, October 2009.

[11] F. Daniel, J. Yu, B. Benatallah, F. Casati, M. Matera, and R. Saint-Paul, "Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities," *IEEE Internet Computing*, vol. 11, no. 3, pp. 59–66, May/June 2007.

[12] O. Diaz and J. J. Rodriguez, "Portlets as Web Components: An Introduction," *Journal of Universal Computer Science*, vol. 10, no. 4, pp. 454–472, April 2004.

[13] W. Martin and R. Nußdorfer, "Role of Portals in a Service-Oriented Architecture (SOA)," S.A.R.L. Martin, CSA Consulting GmbH, Whitepaper, March 2006.

[14] M. Steger and C. Kappert, "User-facing SOA," *Java Magazin*, pp. 65–77, March 2008.

[15] S. Abiteboul, O. Greenshpan, and T. Milo, "Modeling the Mashup Space," in *Proc. of the 10th Intl. Workshop on Web Information and Data Management (WIDM)*. Napa Valley, CA, USA: ACM, October 2008.

[16] D. Benslimane, S. Dustdar, and A. Sheth, "Service Mashups," *IEEE Internet Computing*, vol. 12, no. 5, pp. 13–15, 2008.

[17] V. Hoyer and M. Fischer, "Market Overview of Enterprise Mashup Tools," in *Proc. of the 6th Intl. Conf. on Service Oriented Computing (ICSOC)*, vol. 5364. Springer-Verlag, 2008, pp. 708–721.

[18] V. Hoyer, K. Stanoesvka-Slabeva, T. Janner, and C. Schroth, "Enterprise Mashups: Design Principles towards the Long Tail of User Needs," in *Proc. of the Intl. Conf. on Services Computing*, vol. 2. IEEE, 2008, pp. 601–602.

[19] C. Pautasso, "Composing RESTful Services with JOpera," in *Proc. of the 8th Intl. Conf. on Software Composition*, ser. LNCS, no. 5634. Springer, 2009, pp. 142–159.

[20] M. Caceres, "Widgets 1.0: Packaging and Configuration," W3C Working Draft, April 2008. [Online]. Available: http://www.w3.org/TR/widgets/

[21] Google Inc., *Gadgets Specification*, http://code.google.com/apis/gadgets/docs/spec.html, Std.

[22] J. Yu, B. Benatallah, F. Casati, F. Daniel, M. Matera, and R. Saint-Paul, "Mixup: A Development and Runtime Environment for Integration at the Presentation Layer," in *Proc. of the 7th Intl. Conf. on Web Engineering (ICWE'07)*, ser. LNCS 4607, Como, Italy, July 2007, pp. 479–484.

[23] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera, "A Framework for Rapid Integration of Presentation Components," in *WWW '07: Proc. of the 16th Intl. Conf. on World Wide Web*, 2007, pp. 923–932.

[24] X. Liu, Y. Hui, W. Sun, and H. Liang, "Towards Service Composition Based on Mashup," in *IEEE Congress on Services*, 2007, pp. 332–339.

[25] G. Prasad, R. Taneja, and V. Todankar, "Life above the Service Tier," October 2007.

[26] T. Nestler, M. Feldmann, A. Preußner, and A. Schill, "Service Composition at the Presentation Layer using Web Service Annotations," in *Proc. of the 1st Intl. Workshop on Lightweight Integration on the Web (ComposableWeb'09)*, June 2009.

[27] R. Acerbis, A. Bongio, M. Brambilla, S. Butti, S. Ceri, and P. Fraternali, "Web Applications Design and Development with WebML and WebRatio 5.0," *Objects, Components, Models and Patterns*, pp. 392–411, 2008.

[28] J. Gómez, C. Cachero, and O. Pastor, "On Conceptual Modeling of Device-Independent Web Applications: Towards a Web-Engineering Approach," *IEEE Multimedia*, vol. 8, no. 2, pp. 20–32, 2001.

[29] S. Pietschmann, J. Waltsgott, and K. Meißner, "A Thin-Server Runtime Platform for Composite Web Applications," in *Proc. of the 5th Intl. Conf. on Internet and Web Applications and Services (ICIW 2010)*. Barcelona, Spain: IEEE, May 2010.

[30] S. Pietschmann, A. Mitschick, R. Winkler, and K. Meißner, "CroCo: Ontology-Based, Cross-Application Context Management," in *Proc. of the 3rd Intl. Workshop on Semantic Media Adaptation and Personalization*, December 2008.

[31] *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, Object Management Group Std., April 2008. [Online]. Available: http://www.omg.org/spec/QVT/1.0/

[32] *RFC4122: A Universally Unique IDentifier (UUID) URN Namespace*, Internet Engineering Task Force (IETF) Std., July 2005. [Online]. Available: http://tools.ietf.org/html/rfc4122/

[33] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell, "Protecting Browser State from Web Privacy Attacks," in *Proc. of the 15th Intl. Conf. on World Wide Web (WWW 2006)*. Edinburgh, UK: ACM, 2006, pp. 737–744.

[34] R. Herzog, H. Lausen, D. Roman, M. Stollberg, and P. Zugmann, *WSMO Registry*, WSMO Working Draft, Std., Rev. 0.1, April 2004, http://www.wsmo.org/2004/d10/v0.1/.

[35] M. Niederhausen, K. van der Sluijs, J. Hidders, E. Leonardi, G.-J. Houben, and K. Meißner, "Harnessing the Power of Semantics-based, Aspect-Oriented Adaptation for AMACONT," in *Proc. of the 9th Intl. Conf. on Web Engineering (ICWE'09)*, ser. Edition 5648, M. Gaedke, M. Grossniklaus, and O. Díaz, Eds., San Sebastian, Spain, Juni 2009.