A Metamodel for Context-Aware Component-Based Mashup Applications

Stefan Pietschmann, Vincent Tietz, Jan Reimann, Christian Liebing, Michèl Pohle, and Klaus Meißner
Technische Universität Dresden
01062 Dresden, Germany

ABSTRACT
Recently, mashups, i.e., composite web applications, have gained momentum in both the consumer and enterprise sector. However, they lack a structured development process which abstracts from specific platforms and enables a universal composition including the presentation layer. Moreover, support for context-awareness in such applications has been neglected so far. Yet, it becomes increasingly necessary due to the growing heterogeneity of users and devices.

We address these issues by proposing an open, extensible metamodel for component-based mashup applications. It defines their control flow, layout, and adaptivity while building on lessons learned from traditional web application models. This metamodel forms the basis for a structured, model-driven development process entailing reusability and independence from specific composition platforms. We implemented the metamodel and successfully proved its practicability with a number of exemplary applications deployed on different platforms.

Categories and Subject Descriptors
D.2.2 [Software]: Software Engineering, Design Tools and Techniques [Modules and Interfaces]

1. INTRODUCTION

While a number of component- and service-based integration paradigms for the data and application layer have been designed to face the complexity of software development [22], none of them has addressed integration at the user interface layer. Mashups have become a prominent approach for composing web applications on a higher level of abstraction. They build on conceptual and technological foundations of information and application integration, but introduce the user interface as integration layer. Thus, a mashup is developed by combining distributed resources providing data, application logic, and/or user interfaces.

While mashups are predominantly developed by manual, script-based integration, recent approaches from industry and academia provide easy-to-use, visual mashup tools like Yahoo! Pipes or the mashArt editor [4]. They allow for a visual composition of applications by dragging components on a canvas and wiring their individual inputs and outputs. Despite indisputable benefits of this agile development, these approaches suffer from a number of limitations.

The tools and platforms imply the use of specific technologies and proprietary models which hinder interoperability and extensibility. Neither applications nor components of different platforms can be integrated with each other. The lack of a generic, platform-independent component and composition model also impedes a structured development process for composite mashup applications which is inevitable to meet additional requirements, such as efficiency, robustness, security, and context-awareness [2].

As a matter of fact, several sophisticated concepts for the model-driven development of web applications exist already. However, we will show that none of them meets the specific requirements implied by modeling adaptive mashup applications following the paradigms of presentation integration [5] and universal composition [4]. Thereby, we denote a composite web or mashup application as a composition of uniform components encapsulating distributed web resources, i.e., services providing UI parts, business logic, and data.

The development of such applications comes with five fundamental requirements [5]: it necessitates a component and composition model including adequate inter-component communication styles, as well as mechanisms for discovery, binding, and visualization of components. In this paper we focus specifically on modeling the composition, which includes an underlying component model and communication styles. The overall design process, the role model implied, and run time aspects like dynamic discovery, binding, and visualization of components are discussed in [19].

To address the shortcomings of existing approaches, we present a generic, platform-independent metamodel that allows for modeling component-based mashup applications including the necessary means to describe component configuration, control flow, layout, as well as context-aware behavior. After outlining corresponding requirements for an application metamodel in the following, Section 3 discusses related modeling efforts with regard to these challenges. Section 4 presents our metamodel and the underlying component model. Afterwards, Section 5 outlines the typical modeling workflow, its support with automatic transformations.
as well as an exemplary use case. Then, Section 6 gives implementation details and clarifies the use of our approach with an existing composition infrastructure. Finally, in Section 7 we conclude this paper and outline future work.

2. REQUIREMENTS ANALYSIS

Designing an adequate metamodel for the universal composition of web applications entails a number of challenges. The most obvious requirement is the definition and integration of a uniform component model, i.e., a universal specification of back-end and front-end components and their application-specific configurations on a generic, technology-independent level. This implies a seamless integration of existing services using their native interface descriptors, e.g., WSDL, and mapping those to the model. As we envision the application of dynamic, service-oriented binding mechanisms to the UI layer, the model needs to abstract from concrete UI implementations (bindings) and support semantic classes of UI components. The assignment of UI component instances to classes used in the model is subject to the model. The next section discusses related work with regard to the above-mentioned requirements.

3. RELATED WORK

A prominent approach to service composition is BPEL [13], which defines executable orchestrations of SOAP-based web services. With regard to our requirements, BPEL and derived approaches [6, 3, 18] are targeted at skilled programmers and enable machine-to-machine communication – they do not take into account presentation integration [5]. While BPEL4People [14] specifies the integration of human activities into such business processes, it is not applicable to our requirements: We aim for the integration of services and processes by composition at the presentation layer, instead of merely integrating UI into a process description. Alternative approaches, e.g., IBM’s Shareable Code [10] or the Open Mashup Alliance [17], describe service compositions with means of domain specific languages (DSL). Yet again, the focus on the application and data layers and the complexity of above-mentioned concepts impair their applicability for component-based mashup applications.

A prominent and mature UI composition approach are web portals. They combine different applications and services by composing their individual portlets to an integrated user interface. However, inter-portlet communication is restricted, and despite standardization of the portlet model, several portal characteristics, e.g., definition of the layout, remain vendor-specific. Sophisticated concepts for a model-driven, platform-independent development and for the adaptation of portal applications are missing.

Mashups have become a lightweight alternative to portals for web-based composition and integration. While their majority is developed manually, few concepts are based on structured models. MashArt [4] introduces the Universal Composition Model which – similarly to our concept – aims at a uniform description and integration of UI components and services. Yet, the authors value model simplicity over completeness, so several aspects such as layout, screen flow, and context-awareness of compositions are not considered. As stressed before, simplicity in our approach is provided by adequate authoring tools instead of model restrictions. In ServFace [7], the Composite Application Model is used to represent relationships between service operations and UI elements. Hence, it does not support a universal composition, backed by dedicated UI components, but instead infers UIs from proprietary service annotations. Additionally, compositions are limited to SOAP services so far.

Finally, several “traditional” solutions from model-driven web engineering exist which are primarily targeted at skilled programmers and do not take into account mashup-specific requirements. For instance, the visual modeling language WebML [1] uses content units and pages to compose a web application, but lacks a uniform component model, flexible n:m communication, and mixes context-aware behavior with the content model [2]. UWE [9] relies on object-oriented modeling with UML, its stereotypes and associations, but it neither supports a universal composition nor provides platform-independent models for layout and presentation. The modeling concepts presented in [8] are also based on UML profiles, yet they restrict to (web) service composition and do not consider composition and adaptation at the presentation layer. Finally, OOHHLA [11] has been proposed with a focus on component-based, interactive web applications. Unfortunately, it applies platform-specific means to define presentation and orchestration – in fact it relies on the Google Web Toolkit.
4. A UNIVERSAL COMPOSITION MODEL FOR MASHUP APPLICATIONS

In this paper we present a platform-independent metamodel that serves as a basis for the model-driven development of composite mashup applications. It provides means to describe all necessary application aspects on a platform-independent level, such as its components, control and data flow, layout, as well as context-aware behavior. To distinguish from the predominant data-level mashups, we term compositions including a presentation layer user interface mashups. After introducing necessary information on the underlying component model and description, this section discusses the composition metamodel and its constituent parts in detail.

4.1 Component Model

Universal composition of applications is based on the fact that all its constituent parts on different application layers adhere to one universal component model. Thus, components are independent building blocks of an application providing data, business logic, user interface or combinations of these. Following the principles introduced with web services, we do not dictate any internal component structure or format. However, our model specifies fundamental characteristics of components’ external interfaces, which are relied upon for their integration and communication.

Figure 1: Universal Mashup Component Model

4.1.1 Model Abstractions

As illustrated by Figure 1, our model describes components on a non-technical level using three abstractions, namely Property, Event, and Operation.

The set of Properties of a component resembles its visible state. In contrast to traditional service-oriented systems using stateless web services, mashup components including Uls necessitate a state representation [4]. In our model, it is realized by name-value pairs which are defined by a component developer and contain any data relevant to the component’s surrounding. Property values adhere to a primitive or complex type, e.g., specified with XML Schema. Examples include a graph’s type (line, bar, pie), a map’s projection or type (normal, satellite, hybrid).

To publish state changes to other components or the composition environment, a component issues Events. They may be triggered by user interaction, internal component logic, or notifications from underlying services. In addition to a name, Events can contain data in the form of typed Parameters, i.e., name-value pairs similar to properties. Picking up the example from above, a typical map Event would be LocationSelected(GLatLng location).

Finally, Operations are methods of components which are triggered by Events. They can include any functionality foreseen by a component developer, such as internal state changes, calculations, or service requests. As an input, they consume Parameters of its trigger Events. An Operation get- CountryInfo(GLatLng location) could be triggered by the aforementioned map Event, which would result in a SOAP request to a web service providing information about countries. Its response would again be published as an Event to be consumable by other components.

4.1.2 Component Description

Component interfaces need to be described uniformly. In the underlying research project, we have developed the Mashup Component Description Language (MCBDL). This XML-based description defines metadata, signature (the component model explained above), as well as semantics of a component. MCDL for mashup components is comparable to WSDL for web services. If a component wraps service access, the corresponding WSDL or WADL descriptions can be directly mapped to MCDL. The integration of presentation components necessitates additional information, such as a screenshot, interaction metaphors, suitability for devices, etc. Therefore, we created the User Interface Service Description Language (UISDL) – an minor extension of MCDL. Yet, MCDL and UISDL are beyond the scope of this article.

In summary, our lightweight component model describes reusable building blocks for the composition of applications. It is specifically designed to support loose coupling of components and their synchronization across different application layers using events. Those form the basis for both control and data flow. Composition is based on uniform component descriptions, which are used by authoring tools at design time as well as dynamic selection and integration mechanisms at runtime, as discussed later in Section 6.2.

4.2 Composition Metamodel

To design a mashup application as a composition of uniform building blocks, we strive for a platform-independent model that covers all relevant aspects of an application, such as illustrated in Section 2. In summary, those are the components used, the communication between them, the overall page/screen flow as well as the layout of the application’s user interface and its adaptive behavior. We have developed an extensible metamodel – the UI Mashup Composition Model – to describe all of the above objectives.

The approach presented here is consistent with the Meta Object Facility (MOF) specification [15] introduced by the OMG. Our metamodel is located on MOF level M2 while

1Details and formal specification of the metamodel are available at http://mmt.inf.tu-dresden.de/cruise/model.
its instances on level M1 represent a concrete application model. With this conformity we ensure that all MOF-compliant systems can interpret and work with our metamodel. Figure 2 gives an overview of its first-level submodels, namely the Conceptual, Communication, Layout, Screen Flow, and Adaptivity Model, each of them describing a specific application aspect. Additionally, the dependencies among them are indicated. The following paragraphs will examine these models in more detail, yet, the figures are slightly simplified to maintain a certain level of clarity.

**Figure 2: Submodels of the Composition Metamodel**

### 4.2.1 Conceptual Model

The Conceptual Model contains application-wide concepts referenced and used by the other submodels. Therefore, it comprises four concept containers for Components, Data Types, Styles, and Environment (cf. Figure 3).

Most importantly, Components contains all building blocks of the application adhering to the aforementioned component model (cf. Section 4.1), colored blue in the following figures. In an application we can distinguish three different component types with regard to the application layer they apply to. On the topmost layer, UI Components (UIC) encapsulate visualization code and presentation logic. A popular example would be an interactive map, such as provided by Yahoo! or Google. To aid the interoperability events and operations, Logic Components (LC) enable data manipulation, e.g., filtering, conversion, and aggregation. Especially the use of complex event data, such as Person, necessitates means to make parts of it (surname, age, gender) processable by operations of other components. Exemplary LCs supported by the model support split, join, and iteration operations. They guarantee that events and operations “fit”, even in combinations unforeseen by component developers.

Finally, at the lowest application level, Service Components (SC) wrap access to services providing data or complex application logic, e.g., via SOAP or REST. In addition, the model supports hierarchical composition: existing compositions can be included into higher-level compositions as Composite Components. To the composer, they fall into one of types mentioned above, depending on the components included. Thus, in a nutshell, service components integrate arbitrary external services, their data being transformed by logic components and visualized by UI components.

Data and control flow between components relies on operations and events which entail typed parameters. Therefore, data type definitions (cf. Data Type in Figures 3 and 1) are parameterized, i.e., they have a signature of typed parameters. Hence, only those publishers and subscribers with a suitable signature can be registered at a channel.

![Diagram of the Conceptual Model](image)

**Figure 3: The Conceptual Model**

Finally, Environment encapsulates all external aspects of a composition. First, in Runtime arbitrary events and operations (triggers) that are triggered at application startup, e.g., to fill the application with initial data without any user interaction. Consequently, service components providing particular data are invoked automatically when the application starts. Second, access to the context model is defined, consistently following our component model. Therefore, Context is resembled as a component, providing properties and operations to request as well as events that are triggered at application startup, e.g., to fill the application with initial data. It plays a key role for adaptation, e.g., whenever variable context data are consumed by other components. Furthermore, context events are the primary triggers for context-aware behavior as specified by the Adaptivity Model discussed later.

Figure 3 illustrates our idea of a universal composition, as all application parts as well as the interface to external entities (higher-level compositions, composition platform, and context model) equally comply with the principles presented in Section 4.1.

### 4.2.2 Communication Model

The Communication Model (Figure 4) defines data and control flow of an application. It supports arbitrary communication paradigms, of which we currently only use Publish-Subscribe to support a loose coupling between components. Therein, Data Channels connect Publishers (events) with Subscribers (operations) of information. To achieve syntactical correctness as mentioned in Section 2, channels are parameterizable, i.e., they have a signature of typed parameters. Hence, only those publishers and subscribers with a suitable signature can be registered at a channel.
Since signatures do not always fit perfectly in real-world applications, composers can create Parameter Mappings of channel and component parameters. This comes in handy when operation and event parameters are semantically equal, but simply occur in a different order. Another typical case is operations which need only certain parts of the event data. Again, this problem can be solved with a mapping which maps only the necessary parameters to the operation.

4.2.3 Layout Model

The arrangement of UI Components on the canvas is defined by the Layout Model. As Figure 5 depicts, the former are referenced from Layout Elements. An author can use multiple well-known Layouts, such as Grid-, Tab-, or Fill-Layout. For instance, the former spans multiple rows and columns, each cell referencing a UI Component from the Conceptual Model. It is possible to nest layouts within each other to model any kind of desired arrangement. Layouts at the root of the hierarchy, i.e., not contained by others, form the foundation for the Screen Flow Model described in the next paragraph. On a side note, layouts can be decorated by Styles (Section 4.2.1) to achieve a consistent appearance.

4.2.4 Screen Flow Model

The Screen Flow Model (Figure 5) contains several Views of an application which are equivalent to page concepts in related approaches. Each one holds a reference to a top-level Layout representing a specific visual state with only certain visible UI Components. One View is marked to be initial, while many others can be defined. View Transitions resemble changes from one view to another, triggered by Events. With the help of Conditions, a fine-grained definition of trigger rules can be achieved, i.e., by binding the transition decision to particular event data. Events may be issued by components as a result of user interaction or application logic, or by the composition platform represented by the Runtime class in the Conceptual Model. All in all, the view concept addresses the demand for more structured, flexible UIs. It allows for modeling multi-step interfaces representing typical workflows.

4.2.5 Adaptivity Model

Finally, the Adaptivity Model (Figure 6) illustrates the extensibility of our metamodel, adding the notion of context-awareness to a composition. As shown in Figure 2, this aspect is kept completely decoupled from the other submodels. Developers can specify Adaptation Aspects crosscutting all of the above-mentioned models. Each aspect represents a model change and thus a dynamic manipulation of the application. Arbitrary Events from the model can be referenced as aspect triggers, hence adaptation can take place, for instance, upon user interaction or context changes. An adaptation can be subject to conditions (Query) related to Parameters of trigger events. Conditions are validated by the run time system and if they hold, the resulting model changes are characterized in by Weave Mappings. They mark elements of the model as adaptation subjects (Pointcut) and contain an adapted version (Advice).

Since almost all parts of a composition are adaptable (labeled by the IAdaptable interface), adaptation becomes very flexible and powerful. It can be applied to different application layers and levels of granularity: aspects can address layouts, data channels, components, and their properties. Thereby, manifold adaptation techniques can be modeled, such as component reconfiguration and exchange, dynamic layout changes, (un)subscriptions to data channels – even the alteration of adaptation aspects themselves with regard to context changes or other events within the application.
5. MODELING SUPPORT AND USE CASE

Following the presentation of our metamodel in the previous section, we now illustrate the typical modeling workflow and its support by model-to-model (M2M) transformations with the help of a sample application.

5.1 Modeling Support by Transformations

To support and simplify the modeling process, we specified and realized several model-to-model (M2M) transformations, whose order results from the typical composition workflow. They are universal, as they apply to application models on a platform-independent level, and can be integrated in dedicated authoring tools.

In a first step, the initial Conceptual Model is refined. Section 4.1 introduced MCDL/UISDL, WSDL, and WADL as basic component descriptors. At design time, these can be used to generate the component model: authors only specify the URL of a component’s description file with the attribute descLocation and invoke the refinement. The latter automatically adds the corresponding properties, events and operations to the composition model and imports necessary data type definitions. Consequently, modeling components becomes as simple as attaching their description files.

Once a composer has defined the Layout Model by placing above-mentioned UI Components in Layouts, a next transformation step allows the semi-automatic generation of the Screen Flow Model: Distinct parent layouts are identified and each referenced by a newly created View. View Transitions are generated as well, yet, authors still need to add the trigger Events and target Views manually.

In a final step, parts of the Communication Model can as well be generated automatically. Data Channels are created automatically for every Event and Operation signature found in the Conceptual Model. The corresponding Events and Operations are registered as publishers and subscribers to the channels. In ideal cases, the communication works instantly, yet, in more complex settings with manifold components, composers might need to change or extend the generated model accordingly.

By utilizing these transformations, the authoring process can be supported actively and becomes a time-saving alternative to the traditional, manual development. Once a composition model has been completed, it is converted into a platform-specific, executable representation using dedicated model-to-code (M2C) transformations (cf. Section 6.2).

5.2 Using the Metamodel – An Example

To validate the sufficiency and practicability of our concept, we (re-)designed a number of previously existing and new applications. In this section we exemplify the usage of our metamodel with the help of a simple use case.

In our scenario, Jane, an employee of a property administration business, needs to assign maintenance orders for buildings under her control to service providers (SP). Therefore, she fetches open maintenance calls from a local database. More sophisticated adaptations may be considered, however, their discussion is beyond the scope of this article.

6. IMPLEMENTATION AND PROJECT CONTEXT

In order to verify the concepts presented in the previous sections, we implemented the composition metamodel, a dedicated editor, and used it to design and deploy a number of exemplary composite applications within an existing composition platform. This section describes details of the realization including a brief discussion of the corresponding M2C transformations.

6.1 Metamodel Implementation

The composition metamodel was realized with the metamodel Ecore being part of the Eclipse Modeling Frame-
work\textsuperscript{2} (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 steps by using a number of existing languages from the Eclipse modeling project. We used operational QVT \cite{10} to implement the above-mentioned M2M transformations. Those incorporating information from outside the metamodel, e.g., the generation of the components from WSDL or WADL descriptions, were realized in Java as “black box libraries” for QVT. For modeling we use a customized tree editor with built-in validation support (based on OCL constraints within the model), which integrates seamlessly with Eclipse (cf. Figure 7) and makes composing a fairly easy task. Overall, Eclipse offers a powerful and flexible environment for future extensions, including the development of a visual editor which is currently underway.

![Figure 7: Composition model editor](image)

6.2 Model-Driven Development in CRUISe

The CRUISe\textsuperscript{3} project focuses on the development of novel concepts for the model-driven development and deployment of adaptive, composite web applications. Its central idea is to support a universal composition of services across all application levels, including services at the presentation layer. Guiding principles and architecture have been presented in \cite{19}. In the following, we clarify the conceptual role and use of the proposed metamodel within this approach.

While CRUISe initially used a DSL as a composition definition, it has been refined to a model-driven approach. As a result, an instance of the metamodel presented forms the starting point of application deployment. It is built with the tree or visual editor mentioned in the previous section, seamlessly integrating back end services (via WSDL and WADL) as well as UI services (via USDL).

A completed composition model is transformed into a platform-specific, executable web application – either statically at design time or dynamically triggered by client requests. In CRUISe, the metamodel has been tested with two different platforms: for the Thin-Server Runtime (TSR) \cite{20} we implemented M2C transformations which generate an executable, client-side web application from a model instance, while our server-side runtime platform dynamically interprets the model (serialized as XMI) directly.

We use Xpand\textsuperscript{4} for the automatic code generation based on polymorphic template invocation. Following the metamodel structure, a root template applies to the top-level UI Mashup Composition Model and invokes all underlying templates associated with the individual submodels. It creates the “skeleton” of the HTML, JavaScript and CSS documents required to execute a TSR application. These files are subsequently filled by the subtemplates. For instance, the most important one generates configuration placeholders for UI Components from the Conceptual Model and generates JavaScript for Logic and Service Components, respectively.

The template for the Communication Model generates code that creates the necessary channel and registers components’ events and operations, accordingly. Similarly, templates for the transformation of the Layout, Screen Flow and Adaptivity Model exist. Model checks and validation are based on the related language Check.

The execution of the M2C transformation is managed by a Maven life cycle, which (1) loads the TSR and all associated files, (2) loads and validates the composition model, (3) invokes the root template, (4) copies the generated files into the target folder and (5) generates a WAR archive to be deployed on an application server.

The resulting application is executed completely client-side. The TSR manages its control and data flow as defined in the model, binds back end services, and dynamically integrates those UI Components that best fit application requirements and context, such as user preferences, device capabilities, and the integration platform (for details see \cite{19, 20}). As mentioned before, an alternative runtime platform based on Eclipse RAP renders transformations unnecessary, as it interprets the model directly. Thus, we have shown the practicability and suitability of our metamodel to build component-based mashup applications, supporting a structured platform-independent development process following the idea of universal composition.

7. CONCLUSION AND FUTURE WORK

The development of composite web applications is usually carried out manually or with the help of proprietary platforms that lack a universal composition approach and thus the necessary interoperability and extensibility. With regard to the increasing complexity of development, e.g., implied by a growing heterogeneity of users and devices, a structured, model-driven development process is desirable.

In this paper we have presented a novel platform-independent metamodel to support the model-driven development of composite mashup applications. It is based on a uniform component model defining reusable parts of an application on a high level of abstraction by the means of Properties, Operations, and Events and thereby supporting the universal composition on all application layers.

The metamodel facilitates the universal definition of components as application constituents, the communication between them, the overall layout and screen flow of the application, as well as its adaptive behavior. Control and data flow between components are uniformly based on Events, which allows for the loose coupling of application parts. In contrast to prevalent efforts, the platform-independent and custom definition of layouts and multiple views is supported. Finally, the metamodel provides means to model context-

\textsuperscript{2}http://eclipse.org/modeling/emf/

\textsuperscript{3}http://mmt.inf.tu-dresden.de/cruise

\textsuperscript{4}http://www.eclipse.org/modeling/m2t/?project=xpand
aware behavior in an aspect-oriented fashion, completely decoupled from the actual application logics. On a side note, it can be easily extended with additional submodels to express needs such as QoS requirements and constraints between components.

Evaluation was based on an adequate implementation including OCL constraints, exemplary M2M and M2C transformations and an authoring tool to simplify the development process. A number of composite applications were (re-)designed using the metamodel, which proved its sufficiency and practicability. Consequently, the CRUISe infrastructure was extended in order to use the model as a basis for its development and deployment process.

We believe the presented concepts are a valuable contribution for the efficient, model-driven development of composite and context-sensitive applications, exploiting the benefits of platform-independence, interoperability, and reusability.

Future work includes improvements of the authoring process, e.g., with additional tooling support and concepts for business process integration. While our adaptivity model proved to be very powerful and sufficient, adaptation aspects quickly get complex and interactions, i.e., conflicts between them are not easily conceivable. We believe that more research should address this specific problem. Thus, we plan to simplify authoring of adaptive behavior by extending the Adaptivity Model with higher-level concerns, such as device independence and location-awareness as proposed in [12]. Finally, we intend to integrate QoS measures and constraints into the metamodel as well as corresponding tools to support quality assurance and testing of composite applications.

8. ACKNOWLEDGEMENTS

The CRUISe project is funded by the BMBF under promotional reference number 01IS08034-C. The work of V. Tietz is funded by the European Social Fund (080939514), the Free State of Saxony (Germany), and Saxonia Systems AG.

9. REFERENCES


