Rich Communication Patterns for Mashups

Stefan Pietschmann, Martin Voigt, and Klaus Meißner

Technische Universität Dresden
01062 Dresden, Germany
{Stefan.Pietschmann,Martin.Voigt,Klaus.Meissner}@tu-dresden.de

Abstract. Mashups imply the lightweight combination of distributed web resources – a paradigm which can be also applied to the presentation layer to build interactive web applications. However, current solutions are limited to very basic composition patterns and do not reflect the coordination needs of the user interface. To tackle this problem, we propose a novel approach for modeling rich communication patterns as part of a mashup composition model, which supports the synchronization between widgets, asynchronous data requests to backend services, and interaction techniques like drag-and-drop. The concepts were realized and validated with a number of sample applications.

1 Introduction

Mashups have become a prominent approach for building web applications from distributed web resources, which has resulted in a multitude of mashup platforms. Recently, research has addressed both formal, platform-independent models and the integration of user interface (UI) parts as first-class citizens into mashups, e. g. in mashArt [1] or CRUISe [5]. However, the current solutions are very limited when it comes to the "glue", i. e., the means to connect the resources. The latter are typically loosely coupled by "wiring" their outputs and inputs with unidirectional links mapped on a publish/subscribe system, which is supposed to offer the highest flexibility [2]. This results in a "fire-and-forget" communication, which is simple at the first sight, but leads to more complex models when data requests and synchronization between components are needed.

In the light of "universal composition" approaches, which equally integrate backend and frontend components, new communication and coordination requirements arise: The seamless integration of backend services as well as the synchronization within the presentation layer are just two examples, which are hard to realize with prevalent solutions.

To emphasize the **requirements**, we introduce a use case which serves as a **reference scenario** throughout this paper. The application *StockMash* shown in Fig. 1 gives an overview of stock indexes ① ②, allows for comparing stock performance ④ and managing a personal depot ③ ⑤. The most basic coordination need is resembled by the green arrows: *unidirectional* connections, e. g., to notify ② when the stock index in ① changes. Further, the stock selection in ① may serve as input for different components, e. g., for comparison using ④.

M. Brambilla, T. Tokuda, and R. Tolksdorf (Eds.): ICWE 2012, LNCS 7387, pp. 315–322, 2012.
© Springer-Verlag Berlin Heidelberg 2012

As the “target” depends on the context, the user needs to decide where to direct the data (blue arrows). This can be achieved by platform-specific techniques, like drag-and-drop. As the data is supplied by backend services, UI components must be able to actively *request* it and receive *asynchronous updates* (to prevent extensive polling). Finally, the stock comparison using ④ underlines the need for the *synchronization* of components, e. g., to adjust the time frame in both views (brown arrow).

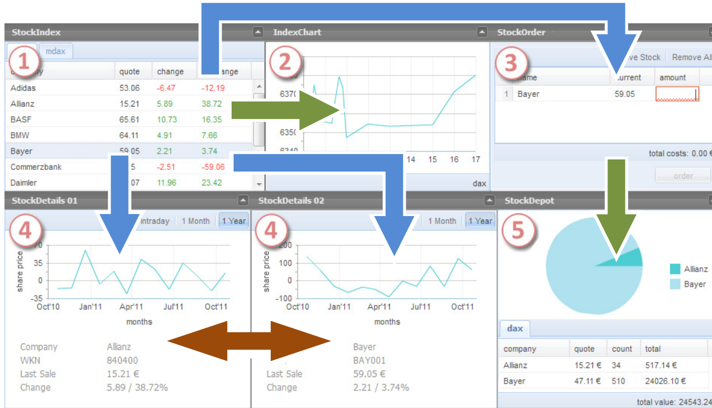

Fig. 1. Coordination relations in the use case *StockMash*

The **contributions** of this paper are twofold. First, we present an advanced communication model comprising different coordination types, which are modeled as extensions of an event-based composition model. Second, we discuss its interpretation and application within an existing composition infrastructure, and show how drag-and-drop interaction can be mapped to this model.

Our paper is structured as follows: In Sect. 2, we introduce the design-time concepts to model rich communication. Then, Sect. 3 presents the runtime concepts and realization, including the communication system and extensions towards drag-and-drop, with the help of our reference scenario. In Sect. 4 we conclude this paper and give an outlook on future work.

2 Modeling Rich Coordination in Mashup Applications

Our solution builds on the universal composition approach of CRUISe, which facilitates the model-driven development and deployment of adaptive, composite web applications. Therefore, we rely on its event-based component and composition models described in [5]. Therein, components of a mashup are described declaratively with the *Semantic Mashup Component Description Language* (SMCDL) using three abstractions, namely *Property*, *Event*, and *Operation*. Based on these abstractions, developers can compose interactive mashup applications using the *Mashup Composition Model* (MCM).

In the following, we present an extension of these concepts to foster the above-mentioned coordination needs.

2.1 Modeling Static Communication Patterns with Links

Communication and coordination in mashup systems are usually expressed by "wiring" inputs and outputs of components. In our solution, those wires are called *links* which connect n events with m operations in case their parameters are semantically compatible. Thus, when one component issues an event indicating a state change, all operations registered at the same link are invoked with the event data. Since the link acts as a mediator between events and operations, all components remain loosely-coupled.

To support *unidirectional*, *bidirectional* and *synchronization* connections, we introduce different *link types*, which are discussed in the next few paragraphs.

Links represent the basic type of **unidirectional** communication as supported by the majority of composition approaches. They allow for connecting n events with m operations, so the data of an event is published to all registered operations. A response is not expected, thus, a **one-way** communication is established. Every link is implicitly typed by the data, i. e., the parameters it carries. Hence, only events and operations whose parameter signatures are semantically equal can be linked. The MCM does offer means for manipulating and mapping parameter signatures, yet, those concepts are out of the scope of this paper.

BackLinks represent **bidirectional**, i. e., request-response connections between components, as usually required for data requests to backend components. As illustrated by Fig. 2-1, BackLinks indicate an implicit callback (link) created at runtime, which returns requested information to the publisher of the initial link. To avoid ambiguities between the returning messages, BackLinks are established between n requesters (events) and only *one* replier (operation).

As soon as an event is published on a BackLink, the target operation is invoked, just as with a Link. However, upon completion, it issues a *CallbackEvent* with a return message, which is routed to invoke the *CallbackOperation* of the

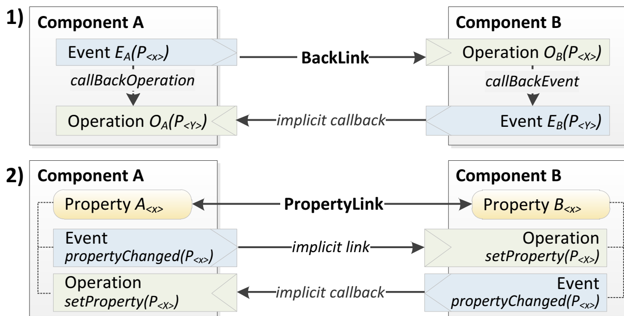

Fig. 2. Communication using Back- or PropertyLinks

initial event. Consequently, to model such a connection, the information about callback events and operations must be available at design time. Therefore, we extended the SMCDL with these two cross references, indicating the implicit connections between incoming and outgoing messages of a component.

By default, the implicit response channel is closed after the callback message has been sent (*single pull*). However, it can be kept open by the replier to facilitate asynchronous updates (*active push*). This needs to be explicitly supported by the replying component as expressed by the operation attribute (*syncable*) within its SMCDL. Using the attribute *syncThreshold* of the BackLink, model authors may set the minimum time interval (in s) between new updates, e. g., to limit the communication and performance overhead.

PropertyLinks allow for the **synchronization** of the stateful components by connecting their properties. This is especially useful for multiple views on shared data, which necessitates their filters to be synchronized. In our reference scenario, this is exemplified by the stock details component showing different stocks, yet in a synchronized time interval.

As illustrated in Fig. 2-2, PropertyLinks can be seen as an abstraction layer on top of the event model. They connect properties – something end-users can more easily understand – but are actually mapped to the corresponding change events and setter operations automatically. The synchronization works mutually, so all participants of this pattern are uniformly modeled as *SyncTargets*.

The introduction of PropertyLinks does not only reduce the complexity and redundancy in the MCM by replacing $n!$ Links with only one PropertyLink with n SyncTargets. Even more importantly, it allows runtime platforms to handle cycles, which would result from using normal Links ($A \Rightarrow B \Rightarrow A$).

2.2 On-Demand Coordination

If the static definition of a mashup's internal data flow is either not desirable or not possible, on-demand coordination becomes necessary. With regard to our reference scenario, this is the case for the stock details components ④. As the stock list ① has only one output – the selected stock – it cannot be connected with both detail views without both of them showing the same data. Thus, the choice, which stock to show in which view, is to be made on-demand at runtime by the user, e. g., by drag-and-drop or other techniques available.

To support this, components need to specify a *dataSource* as part of their interface description. As with any property, it comprises a number of semantically typed parameters representing the data to be shared. Potential targets of an interaction are already specified in the form of operations, as they define which data can be consumed by components, regardless of how it is invoked.

The basic idea of on-demand coordination is: If a dataSource of a component is active, e. g., upon a drag or voice command, any compatible operation within the mashup may act as data sink. The detection of the trigger as well as the coordination between the data source and sink is up to the platform, which hides components from the peculiarity of specific interaction techniques.

2.3 Modeling the Reference Scenario

To prove the feasibility and practicability of our model and composition system, we built several composite applications, one of which represents the use case introduced in Sect. 1. Fig. 3 illustrates the coordination of its stock details components ④ as modeled in the MCM. As soon as a stock has been selected, they request its data via a BackLink and from this point receive updated values every 5 s. Further, a PropertyLink connects the *interval* property of both instances, so that the time span shown in both views is always the same.

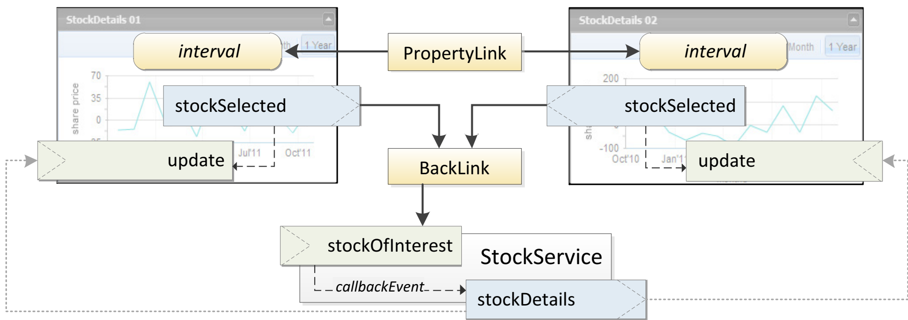

Fig. 3. Coordination links within the sample application

This practical example also illustrates the simplification of the MCM: In comparison with a traditional version, where requests and synchronizations were mapped to normal Links, the novel model allowed for a reduction of connections from 19 to 8, not even considering the additional possibilities of permanent, asynchronous updates and the handling of cycles.

3 Supporting Rich Communication Patterns at Runtime

As mashup components are developed independently and can be composed dynamically, realizing the above-mentioned communication patterns is a challenging task. In the following, we sketch the basic principles of a corresponding communication system, which was implemented as extension of the CRUISe platform.

3.1 Message Format

To realize the data exchange between components developed by different vendors with different technologies a uniform vocabulary is needed. Therefore, all components must adhere to a platform-independent message format. The universal composition implies the communication between components of different application layers and complexity, so the format we have developed is generic, simple,

yet extensible. It is divided in control information and the actual payload. The latter forms the *body* of a message and contains the actual data represented by event and operation parameters. The control information in the *header* includes – among others – the following elements:

Status indicates the success or failure of message distribution and data transfer. To this end, we adopt the HTTP status codes.

Name equals the event name which, combined with the component name, results in a unique ID is used to resolve link subscriptions for the message.

CallbackID is an identifier for a certain bidirectional connection. It is automatically added by the MRE, forwarded by replying components and thus used to identify associated messages and subscribers on the BackLink.

SyncThreshold is an optional parameter which defines a threshold for permanent updates (cf. Sect. 2.1). For unidirectional messages, this field is left blank, while any other numeric value defines the minimum time interval between two updates to be enforced by the channel.

3.2 Link Interpretation and Realization

To support the different link types at runtime, we employ the broker pattern [3], which nicely fits with the event-based nature of the model. The **Event Broker** – a module of the mashup platform – is responsible for managing all the channels specified in the MCM. Further, it offers an API to create, configure and send messages. Thereby, components can easily create and send messages, including life-cycle events, change-events for properties, as well as "ordinary" events specified in the SMCDL. Apart from message and type validations, the Event Broker's main responsibility lies in distributing messages according to the links in the MCM, as discussed in the following.

Links can be mapped directly to the existing infrastructure: Components simply create a new message and publish it, using the broker's API. Using the combination of event and component name as unique ID, the message can be assigned to all the Links it is part in. Following optional mediation steps (cf. [4]), it is then used to invoke all the operations registered with the Link. Upon completion, the broker returns a message to the publishing component, which includes a status code to indicate the success of the data transfer. It is up to component developers if and how this information is interpreted.

BackLinks pose additional challenges to the communication architecture. While the initialization and distribution of messages follow the workflow described for Links, the handling of the response requires additional steps. This is, where the CallbackID from Sect. 3.1 comes into play. It is added by the broker to every message published on a BackLink and forwarded by the replier in the Callback-Event. Thereby, returning messages can be distinguished from "ordinary" ones published by a component and can be forwarded to the CallbackOperation of the initial requester (cf. Fig. 2-1). Asynchronous updates published via CallbackEvents are handled likewise; however, the broker additionally enforces the *syncThreshold* defined in the MCM.

Overall, the BackLink is a simple yet effective mechanism to realize data requests between UI and backend components. In our use case scenario, both the automatic update of stock and index information as well as of the depot data can be realized this way. Instead of polling the information, the data is permanently updated from the backend by keeping the channel open.

PropertyLinks offer an abstraction to the event model and are thus harder to interpret. Basically, they are mapped to Links between change events and setter operations of the corresponding properties. However, the following challenges must be handled: (1) The synchronization affects all participants of the PropertyLink, including the trigger component. Thus, in order to prevent the distribution of a state change to the originator, the latter must be filtered out by the broker dynamically. (2) Once the new property value is set for all registered components, they send change events in return. To prevent communication overhead, the broker caches the state of a PropertyLink, i. e., the current property value. If the payload of an update message equals the cached state, the distribution of this message is skipped. 3) Finally, property changes may overlap, which may lead to data loss if change events are issued while the previous synchronization has not been finished. Thus, the Event Broker employs a FIFO queue, which saves incoming updates and delays them, until the current update is finished.

With regard to our reference scenario, PropertyLinks can be used to realize the synchronization of the time interval (property) in the stock detail components ④ to improve the comparability of the stocks charts.

3.3 Supporting On-Demand Coordination

Realizing on-demand coordination using device- or platform-specific interaction techniques poses additional challenges. On the one hand, the trigger interaction is generally recognized by the source component itself. However, as it has no knowledge of the surrounding platform and components, the platform needs to handle the coordination by mapping the interaction to the link model, so that the target component remains independent from any technological peculiarities. This mediation is carried out as follows:

Starting point of an on-demand coordination is a dataSource (Sect. 2.2) which is defined in the SMCDL and, thus, represented in the MCM. If a component detects the trigger interaction, e. g., dragging of data, it publishes a corresponding event. The message contains a reference to the dataSource and its typed parameters. As a result, the MRE creates invisible *data sinks*, e. g. drop zones. Sinks are only enabled for such components that comprise at least one operation compatible with the dataSource. Finally, when the end of the interaction is detected, e. g., a tangible has been placed, the platform usually receives a corresponding system event. If the target component offers more than one appropriate operation, the user may select the action to take. Then, the data sinks are removed and the platform realizes the data flow: Therefore, it requests the data in question from the source component and invokes the selected operation of the target component.

4 Conclusion and Future Work

Recently, mashup development has moved towards the presentation layer, resulting in universal mashups which enable the lightweight combination of distributed backend and frontend resources. However, current solutions are limited to basic communication patterns and do not support the coordination needs implied by the UI, e. g., data requests to backend services, synchronization of widgets, and on-demand coordination using interaction techniques, such as drag-and-drop.

In this paper, we have introduced a novel concept for modeling advanced communication patterns as part of a universal composition model. In contrast to prevalent solutions, it supports active pull and push connections as well as component synchronization. Further, we have shown, how to support on-demand coordination, e. g., using drag-and-drop, and how all these types of coordination can be mapped to common event-based coordination mechanisms.

By realizing the use case scenario, among others, the solutions could be validated and proved to be feasible and practicable. They both simplify the modeling effort and allow unleashing the full potential of universal composition, as they enhance its coordination capabilities with respect to the needs implied by the interactivity of the applications.

Currently, we are working on mechanisms to support end-users in dynamically establishing coordination, i. e., property links, themselves. After that, we plan to conduct extensive user studies, which include the on-demand coordination concepts described here.

Acknowledgments. The work of Martin Voigt is funded by the German Federal Ministry of Education and Research under promotional reference number 01IA09001C. Further, we would like to thank our student Robert Wende for his valuable contributions to this work.

References

1. Daniel, F., Casati, F., Benatallah, B., Shan, M.-C.: Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In: Laender, A.H.F., Castano, S., Dayal, U., Casati, F., de Oliveira, J.P.M. (eds.) ER 2009. LNCS, vol. 5829, pp. 428–443. Springer, Heidelberg (2009)
2. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. 35, 114–131 (2003)
3. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (March 1995)
4. Pietschmann, S., Radeck, C., Meißner, K.: Semantics-based discovery, selection and mediation for presentation-oriented mashups. In: Proc. of the 5th Intl. WS on Web APIs and Service Mashups. ACM (September 2011)
5. Pietschmann, S., Tietz, V., Reimann, J., Liebing, C., Pohle, M., Meißner, K.: A metamodel for context-aware component-based mashup applications. In: Proc. of the 12th Intl. Conf. on Information Integration and Web-based Applications & Services. ACM (November 2010)