# CapView – Functionality-Aware Visual Mashup Development for Non-programmers

Carsten Radeck, Gregor Blichmann, and Klaus Meißner

Technische Universität Dresden, Germany
{carsten.radeck,gregor.blichmann,klaus.meissner}@tu-dresden.de

**Abstract.** Building mashup applications from existing web resources becomes increasingly popular, and, in theory, accessible even for end users without programming skills. Current proposals for end user development of mashups mainly focus on visual wiring of component interfaces supplemented by recommendations on composition steps and a certain degree of automation. However, it is still a major challenge to provide an appropriate level of functional abstraction in order to visualize the functionality of a mashup and its components, and for composing on a functional level instead of merely assembling structural units. This becomes crucial, especially when non-programmers are the intended target group. In this paper, we propose CapView, a novel functionality-aware development view on running composite applications. CapView is part of the EDYRA platform and provides a functional overview of the mashup by abstracting from interface and wiring details. It enables users to understand mashup development as an assembly process that is centered on the capabilities of components and mashup fragments. We evaluate the concepts in a user study and present lessons learned.

**Keywords:** mashup, end user development, non-programmers.

## 1 Introduction

Powered by the growth of available web resources and application programming interfaces, the emerging mashup paradigm enables loosely coupled components to be reused in a broad variety of application scenarios to fulfil the long tail of user needs. Thus, mashups and end user development (EUD) complement each other quite well. However, when supporting non-programmers, their limited understanding of technical concepts and experience on development practices have to be considered. In addition, it is hard for non-programmers to map their problem, for which they probably know a solution in terms of necessary tasks or activities, to a composition of components.

In order to empower non-programmers to build applications on their own, EUD tools have to fulfil several essential requirements as pointed out in the literature, e. g., [9,4]. Technical details, concepts and terminology have to be hidden from the user. Furthermore, there is a need for user guidance and automation throughout the composition procedure, for instance, recommendations on composition steps and support for correctly connecting components when solving heterogeneity issues. In addition, there should be immediate feedback on a user's composition actions and, as proposed by [9], task-oriented user interfaces should be applied instead of technology-led ad hoc visualizations.

However, prevalent mashup solutions mostly build up on purely wiring component interfaces. With respect to the requirements above, we argue, that this technical view is still too complicated for end users without programming knowledge. Although wirings allow to retrace data flow, understanding what actually happens in a mashup, or what functionality recommendations offer, requires manual investigation by the user or depends on community-provided documentation. Therefore, a more abstract way of building mashup applications is required, which focuses on the functionality to achieve rather than the technical solution in terms of component interfaces and composition glue.

Thus, we propose **CapView**, a novel functionality-aware development view on (running) composite applications. It is part of the EDYRA platform, which extends CRUISe [12] concepts and allows for live sophistication of mashups. Thereby, the mashup runtime environment becomes the authoring tool, seamlessly interweaving mashup design and usage, to provide for instant feedback for end user's development actions [15]. CapView provides a functional overview of the mashup abstracting from interface and wiring details.

The CapView essentially helps non-programmers to (1) realize "components" as task-solving entities, (2) investigate functionalities provided by a mashup, by its components and by recommendations, and (3) to manipulate a mashup through visually composing component functionalities.

The **contribution** of this work are manifold:

- We present capabilities, a semantic description of component functionality, and define generic rules for deriving natural language labels from capabilities.
- We introduce the CapView supporting non-programmers with a functional abstraction of composition details when developing a mashup independently.
- We evaluate the efficiency of the CapView via a user study.

The remaining paper is structured as follows. First, we discuss related approaches in Sect. 2. Then, Sect. 3 describes the conceptual foundation of our work. We introduce CapView in Sect. 4 and show the results of our evaluation in Sect. 5. Finally, Sect. 6 summarizes the paper and outlines future work.

## 2 Related Work

Similar to our approach, Yahoo Pipes[1] uses a mainly data flow oriented visual wiring paradigm via drag&drop in conjunction with highlighting possible connections while creating a pipe. However, there is a hard break between development and usage, and the user has to understand data structures and technical

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

concepts. IBM mashup center[2] allows to combine building blocks, including widgets, and to use the mashup while developing it. Connections are created through dialogues and are shown in a dedicated view, but functionality provided by components or compositions cannot be explored. Similarly, in Jackbe Presto wiring takes place via drag&drop, but the user is not supported in establishing correct or even useful connections, and in identifying transitive connections. A drawback is the required knowledge about technical interfaces of blocks and data types.

In academia, several projects have addressed some of the identified challenges of EUD for non-programmers. Similar to our approach, mashart [7] utilizes universal composition and a component model, but neglects semantic annotations. At development time, the components, their event-based composition and layout can be defined using drag&drop metaphors. Despite a preview, there is a separation between development time and run time, and the user is not supported by recommendations. ResEval Mash [8] is a mashup platform dedicated to the research evaluation domain. In a data flow oriented way, components of different type like sources or visualizations are coupled. A domain-specific appearance of those types on the modelling canvas may indicate implicit functionality, but there is no activity-based abstraction. The ServFace Builder [10] enables users to visually compose web services. Thereby, form-based front ends are generated from service descriptions. The data flow can be defined using drag&drop at development time, and is visible to the user. Again, only a technical view on the resulting application is provided. In MyCocktail[3] forms serve for configuration of components at design time, but there is no support regarding correctness while establishing connections. Understanding the functional interplay of the components is impeded by missing visualization of connections. Similar to our conceptual foundation, the FAST platform [2] utilizes semantically described components which are assembled to gadgets, so called screens. The latter can again be combined to screenflows using input/outputs. The functionality of gadgets is expressed by pre- and postconditions, rather concerning input and output than the activities provided. Yet, it lacks a smooth transition between run time and design time and the user has to be familiar with interface concepts. The Omlette Live Environment [17] provides interwoven runtime and development time. The user is supported with advice on patterns mined from existing mashups, and by automated integration of selected patterns. However, there is nothing comparable to our abstracting view and the recommended pattern are mainly visualized by the incorporated components. In line with our approach, DashMash [5] allows for manipulating a mashup during usage, and addresses similar users. However, to understand the data flow of a mashup, users have to inspect a dialog listing connections for a certain component, and establishing connections takes place on interface level. EnglishMash [1] shares a similar basic idea: abstracting from technical details through natural language. However, it provides restricted means for exploration of components' functionalities and no formal component model. DEMISA [16] proposes a task-oriented methodology to

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

[3] `http://www.ict-romulus.eu/web/mycocktail/home`

develop mashups. A task model has to be defined first which is semi-automatically transformed to an executable mashup composition. However, due to the top-down approach, there is a hard break between development and usage. Recently several proposals focus on semantic annotations, e. g. [3], and mediation, e. g. [11], for mashups. Therein, recommendations can be achieved based on semantic matching of annotations. Further, community-driven recommender systems, e. g. [6], have been proposed. We build up on similar techniques, but focus explicitly on functionality-centered visualization of recommendations.

To sum up, in current proposals, understanding the provided functionality of recommendations, components or even the mashup highly depends on meaningful naming and descriptions of interfaces, and is further impeded by missing highlighting of connections. Current tooling lacks task-oriented visualization and composition metaphors for running mashups. It is too interface-oriented, and, thus, features no sufficient level of abstraction for non-programmers.

## 3 Preliminaries

Within the EDYRA project we adhere to universal composition, which allows for platform and technology independent composition of arbitrary web resources and services [12]. This section outlines the underlying concepts of the CapView.

A declarative **composition model** defines all aspects of a mashup: components or templates for context-sensitive selection of matching components, their configuration, event-based communication, and layout [12]. For inter-component data exchange, several types of communication patterns are applicable: fire-and-forget according to the publish-subscribe paradigm via *Links*, request-response via *BackLinks*, and synchronization of properties using *PropertyLinks* [13].

In our **semantic component model**, web resources are encapsulated by black-box components. Furthermore, components with or without a UI (service components) are characterized by three abstractions, namely parametrized operations and events as well as properties. As a declarative descriptor implementing the component model, we use the Semantic Mashup Component Description Language (SMCDL) [11]. SMCDL covers non-functional properties, like quality aspects and authors, and the public component interface consisting of the abstractions mentioned above, see Fig. 1. In order to specify data semantics of those interface parts, references to ontology concepts, like classes and object-properties, are annotated. Thereby, we can leverage semantic matching and mediation techniques. To describe functional semantics in a formal, yet simple way, we have extended SMCDL with **Capabilities**. Basically, a capability is a tuple

$$<\textit{activity, entity, requiresInteraction}>$$

and can be defined at component and operation/event level. In the latter case, capabilities complement the input and output parameters. *Activity* and *entity* refer to ontology concepts, for example `act:Contact foaf:Person` and, combined, express which action is performed on a domain object. If the user is involved in the activity, this is stated by *requiresInteraction*. This way we get

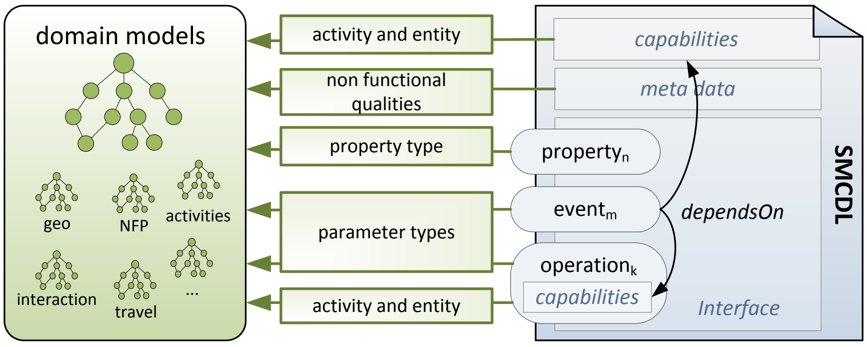

**Fig. 1.** Overview of the SMCDL

a tag-like descriptor of component functionality, backed by clear semantics to overcome ambiguity of tagging approaches. While the concepts referenced by entity and activity are typically domain-specific, we build up on an upper ontology for activities including generic concepts like *Calculate*, *Create*, and *Display*.

To express intra-component functional dependencies, events reference the capabilities that cause their occurrence using *dependsOn*. For sake of simplicity, or-semantics applies in case of several referenced capabilities. For instance, an event publishing results of an operation call refers to the operation's capability.

Two capabilities are **connectible** if the parameters of the underlying interface parts are semantically compatible. This either means that annotated concepts match perfectly (e. g. location $\rightarrow$ location) or can be mediated (e. g. latitude + longitude $\rightarrow$ location). Suitable mediation techniques are not in the scope of this paper, but we extend our work proposed earlier [11]. Connectibility of properties is more restrictive and requires equal or identical semantic types.

## 4 A Capability-Centered View for Non-programmers

The overall architecture of our platform is illustrated by Fig. 2. There are several repositories, depicted on the right. Components are managed by their SMCDL, whole mashups are represented by composition models. Certain composition fragments are mined from existing mashups or determined on the fly based on semantic annotations. Such composition fragments, like a coupling of two components or a more complex part of a composition model, are reactively or proactively queried and filtered by the recommendation manager, as part of the runtime environment [14]. Then, the fragments are presented to the end user, and, after selection, woven into the running mashup by the adaptation system.

There are different views on the mashup. The *LiveView* presents the running UI components integrated in the mashup while channels are hidden. For users with the necessary skills, the *ProfessionalView* provides a state-of-the-art wiring view which overlaps the first. The novel CapView is focused on capabilities in order to abstract from technical details. As with all views, CapView incorporates recommendations. Providing proper functionality-centered presentation for recommendations and for composition logic, CapView utilizes a *label generator*, which derives descriptions from semantic annotations of components and, thus, composition fragments. Every necessary mapping to the technical implementation in terms of the composition model is automatically performed by the runtime environment and completely hidden from the user.

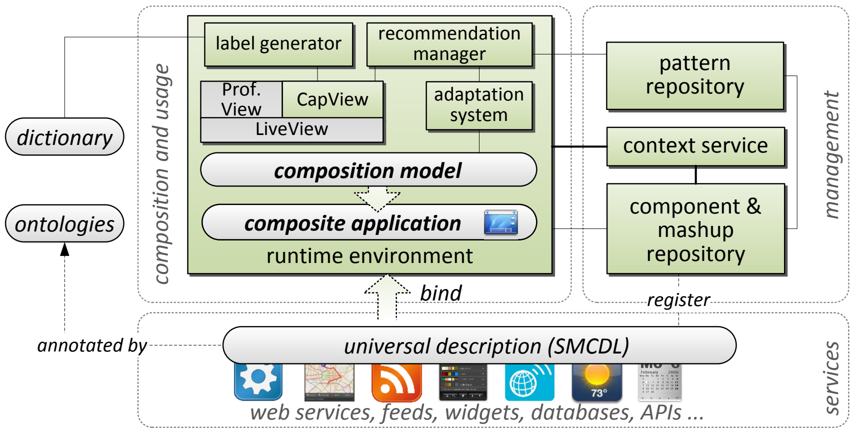

**Fig. 2.** Architectural overview of the EDYRA platform

## 4.1 Overview

In contrast to the ProfessionalView, the CapView does not explicitly display the operations and events of components. Instead, based on annotated capabilities, tasks that can be fulfilled using the component are clustered and visualized. To ease the correlation between LiveView and CapView, those tasks overlay the corresponding components. Our basic assumption is, that a mashup and its components are offering a set of functionalities. For execution, these functionalities may require inputs or produce outputs, which can be provided or consumed by other functionalities in a data flow based manner. This reflects the underlying component model (c. f. Sect. 3) and leads to tuples $<<A, E, iR>, P_{in}, P_{out}>$ where $<A, E, iR>$ denotes a capability as defined in Sect. 3. $P_{in|out}$ are optional sets of the parameters of the operation or event, which correspond to the capability. Besides capabilities, components' properties are part of CapView, since we argue, that it is intuitive that objects are characterized by attributes.

An overview is shown in Fig. 3. The main part shows the overlaying CapView that lists capabilities and properties of components in the mashup, as well as connections. On the right is the **recommendation menu**, giving advice on composition fragments represented by the capabilities they offer.

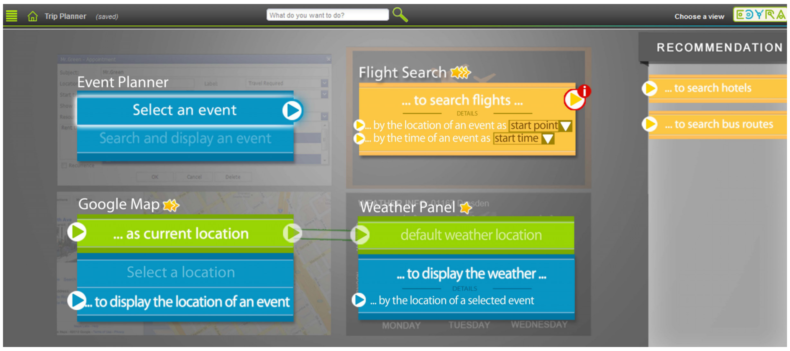

**Fig. 3.** Exemplified overview of the CapView

## 4.2 Visual Exploration of a Mashup's Functionality

As exemplified in Fig. 3, we conceptually utilize several colors in order to distinguish the *representation* of capabilities and properties. Representations for a certain component can be collapsed and expanded by the user. Further, to couple components implicitly by connecting representations, the latter can have ports as interaction elements corresponding to the inputs required and/or outputs provided. In addition, natural language labels are provided throughout the CapView, derived from semantic annotations. Thereby, several generic rules apply in order to label capabilities and properties. We go in more detail on the rules in Sect. 4.3 including examples.

*Capabilities at component level* are grayed out and carry no connection ports if no event refers to them via *dependsOn*. This way, the user is aware of the component's capability and the fact that it cannot be coupled. Otherwise the capability is colored according to the *requiresInteraction* (blue in Fig. 3 if *true*, else orange) and has an output port. *Capabilities at operation level* adhere to the same coloring scheme. Input ports are always visible, and their counterparts appear if at least one event refers to the capability. *Properties* are always colored uniformly (green in Fig. 3) and can have input and output ports.

A user can select a capability or property, denoted representation $r_0$. The **1-layer** $L$ is defined as the set of all connectible capabilities or properties in the CapView that can directly provide input ($S \subseteq L$) for or handle output ($T \subseteq L$) of a representation. In order to avoid cycles, we assume that a certain representation cannot be in $S$ and $T$ of $r_0$. Self-connections of a component are prohibited, i. e., there is no $r$ belonging to its own 1-layer. Established connections between representations are visualized, too. The appearance differs depending on $r_0$.

When selecting $r_0$, all $r_i \in L_0$ are highlighted and renamed, see Fig. 3 where $r_0$ is Select an event. The renamed labels are visually highlighted as well for several seconds to allow for awareness. Channels not connected to $r_0$ are grayed out in order to improve clarity. In addition to direct channels, indirect connections are highlighted as well in a less bright appearance. This way, a user can follow transitive data flow easier. For instance, if a compatible $r_j \in T_0$ is not yet connected with $r_0$ but with an $r_q \in T_j$, transitive highlighting $r_j \rightarrow r_q$ also applies. However, label adaptation exclusively changes the 1-layer.

Highlighting possible ports can be understood as a seamless visualization of recommendations. Besides this inline presentation, the recommendation menu lists capabilities of components not part of the mashup yet. In any case, we utilize stars to emphasize the three best rated recommendations.

## 4.3 Context-Sensitive Label Generation

How labels for representations are derived is subject of this section. The notion used thereby incorporates functions and indices, which we briefly explain now.

- *dLabel*() and *aLabel*() return a human readable description of an annotated property type or capability entity respectively activity. Thereby, either the name of the concept, extracted from its URI, or its `rdfs:label` is used.
- *art*() inserts a correct article
- index *pp* denotes the past participle, queried from the dictionary
- index *norm* indicates the "normalized" concept, i. e. `rdfs:range` of an object property, else the concept itself
- As part of recommended composition fragments, there is always a **mapping definition** $map_{P_1 \rightarrow P_2}$ that defines how interfaces have to be coupled possibly incorporating mediation techniques. For more complex fragments, there can of course be multiple definitions, one per channel in the fragment.

The generation process distinguishes essentially two cases. First, the basic case where nothing is selected by the user. The basic configuration for *properties* leverages the label or the name of the ontology concept annotated as type. To ease understanding for the user, the current value of the property is shown as well if it is set. A capability *<A, E, iR>* is displayed utilizing the human-readable labels given for $A$ and $E$, e. g. search a route, following the scheme:

$$aLabel(A)\ art()\ dLabel(E)$$

Secondly, labels are adapted to the user selection which serves for clarifying cause and effect. The algorithm built upon a generic rule set takes the 1-layer of $r_0$ and determines the label for representations on the 1-layer rather than $r_0$ itself. Thereby, $r_j \in S$ and $r_i \in T$ are treated differently. Further, dots are appended or prefixed to clarify the reading direction.

**A Property Is Focused.** When selecting $r_0$, re-labelling the 1-layer of a property distinguishes properties and capabilities.

*$r_j \in S$ or $r_i \in T$ is a property.* When connecting two properties is possible, it depends on whether $r_0$ is the target or source of the connection. In the first case a $r_j \in S$ is renamed according to the scheme

$$\text{Use } dLabel(Val^j_{prop}|Type^j_{prop}) \text{ as} \ldots$$

where $Type_{prop}$ denotes the type concept and $Val_{prop}$ the currently set value of a property. In the other case, the rule slightly differs, and a $r_i \in T$ is labeled:

$$\text{Use } dLabel(Val^0_{prop}|Type^0_{prop}) \text{ as } dLabel(Type^i_{prop})$$

For illustration, consider the examples listed in the following table.

| $r_j \in S$ | $r_0$ | $r_i \in T$ |
|---|---|---|
| $Type_{prop} = Location$ | $Type_{prop} = hasCenter$ | $Type_{prop} = Location$ |
| Use location (Dresden) as ... | Center | Use center as location |

*$r_i \in T$ is a capability.* The selected property $r_0$ can serve as input for a representation $r_i = << A^i, E^i, iR^i >, P^i_{in}, P^i_{out} >$, where $map_{Type_{prop} \rightarrow P^i_{in}}$ is a injection and $Type_{param}$ denotes the single matched parameter's type. Hereby, we distinguish whether the normalized entity and the normalized type of the single parameter are mediable. If not and if the $Type_{param,norm}$ is equal to or superconcept of $Type_{prop,norm}$, we utilize the following scheme:

$$aLabel(A^i)\ art()\ dLabel(E^i) \text{ using } art()\ dLabel(Type_{prop})$$

Another option is, that $Type_{param}$ is part of the concept $Type_{prop,norm}$ and can be queried from instance data at runtime (**SplitRule**).

$$aLabel(A^i)\ art()\ dLabel(E^i) \text{ using } art()\ dLabel(Type_{param}) \text{ of } art()\ dLabel(Type_{prop})$$

| $r_0$ | $r_i \in T$ |
|---|---|
| $Type_{prop} = hasCurrentLocation$ | <<Search, Hotel, $\perp$ >, {Location, Time} > |
| Current location | Search a hotel using the current location |
| $Type_{prop} = Event$ | <<Display, Hotel, $\top$ >, {Location, Time} > |
| Event | Search a hotel using the location of the event |

Contrary, if entity and parameter are mediable, a shorter rule applies to provide more compact labels (**CompRule**). Analogously, the SplitRule is used.

$$aLabel(A^i)\ art()\ dLabel(Type_{prop})$$

| $r_0$ | $r_i \in T$ |
|---|---|
| $Type_{prop} = hasCenter$ | <<Display, Location, $\top$ >, Location > |
| Center location | Display the center location |
| $Type_{prop} = Event$ | <<Display, Location, $\top$ >, Location > |
| Event | Display the location of the event |

If the capability represented by $r_i$ offers multiple parameters ($|P^i_{in}| > 1$), there may of course be several possible mappings between the property and those parameters. Then, the options are declared via the suffix (**SuffixRule**):

$$\text{as } dLabel(Type_{param})$$

| $r_0$ | $r_i \in T$ |
|---|---|
| $Type_{prop} = hasCenter$ | <<Search, Route, $\perp$ >, {hasStart, hasDest} > |
| Center | Search a route using the center as start |
|  | Search a route using the center as destination |

*$r_j \in S$ is a capability.* The selected property $r_0$ can consume the output of a representation $r_j = << A^j, E^j, iR^j >, P^j_{in}, P^j_{out} >$, where $map_{P^j_{out} \rightarrow Type_{prop}}$ is a injection and $Type_{param}$ denotes the single matched parameter's type.

$$\text{Use } art()\ aLabel(A^j)_{pp}\ dLabel(E^j) \text{ as } art() \ldots$$

$$\text{Use } art()\ dLabel(Type_{prop}) \text{ of } art()\ aLabel(A^j)_{pp}\ dLabel(E^j) \text{ as } art() \ldots$$

| $r_j \in S$ | $r_0$ |
|---|---|
| <<Select, Location, $\top$ >, Location > | $Type_{prop} = hasCenter$ |
| Use the selected location as the ... | center location |
| <<Select, Event, $\top$ >, Event > | $Type_{prop} = hasCenter$ |
| Use the location of the selected event as the ... | center location |

It is possible that not all $p \in P_{\{in|out\}}$ of a representation are covered. However, this is not subject to the label generation, but visualized via an exclamation-mark at the connection after the latter has been established.

**A Capability Is Focused.** If a $r_j \in S$ or a $r_i \in T$ is a property, the rules presented previously apply. Thus, only the case of coupling two capabilities is discussed in detail now. As with properties, we check whether the condition for CompRule holds. To this end, both entities have to be equal, identical or in inheritance relation. The same condition is checked for the mapped parameters' types. The resulting pattern is:

$$\ldots \text{to } aLabel(A^i)\ art()\ aLabel(A^0)_{pp}\ dLabel(E^0)$$

| $r_0$ | $r_i \in T$ |
|---|---|
| <<Select, Location, $\top$ >, Location > | <<Display, Location, $\top$ >, Location > |
| Select a location | ... to display the selected location |

In the case CompRule is not applicable, the scheme shown below is used, and SplitRule and SuffixRule are utilized as required. In principle, depending on the mapping definition, there may be $n$ by- and $k$ as-parts (the latter are omitted if $k = 1$) in the resulting label, where $n$ and $k$ are the number of matched parameters in $P^0_{out}$ respectively in $P^i_{in}$.

$$\ldots \text{to } aLabel(A^i)\ art()\ dLabel(E^i) \Bigg( \text{by } art() \Big[ dLabel(Type^{0,n}_{param}) \text{ of } art() \Big] \\ aLabel(A^0)_{pp}\ dLabel(E^0) \Big[ \text{as } dLabel(Type^{i,k}_{param}) \Big]_k \Bigg)_n$$

For some examples, consider the following table.

| $r_0$ | $r_i \in T$ |
|---|---|
| <<Select, Location, $\top$ >, Location > | <<Search, Hotel, $\perp$ >, Location > |
| Select a location | . . . to search a hotel by the selected location |
| <<Select, Event, $\top$ >, Event > | <<Search, Hotel, $\perp$ >, Location > |
| Select an event | . . . to search a hotel by the location of the selected event |
| <<Select, Location, $\top$ >, Location > | <<Search, Route, $\perp$ >, {hasStart, hasDest} > |
| Select a location | . . . to search a route by the selected location as start |

Similarly, capability representations providing input for $r_0$ are handled, where $n$ and $k$ are the number of matched parameters in $P_{out,j}$ respectively in $P_{in,0}$.

$$\Bigg( \ldots\text{by } art() \left[ dLabel(Type^{j,n}_{param}) \text{ of } art() \right] aLabel(A^j)_{pp}\ dLabel(E^j) \left[ \text{as } dLabel(Type^{0,k}_{param}) \right]_k \Bigg)_n$$

| $r_j \in S$ | $r_0$ |
|---|---|
| <<Select, Location, $\top$ >, Location > | <<Display, Location, $\top$ >, Location > |
| . . . by a selected location | Display location |
| <<Select, Event, $\top$ >, Location > | <<Search, Hotel, $\perp$ >, Location > |
| . . . by the location of a selected event | Search a hotel |

Analogously to property rules, if there are parameters of the underlying interface part that are not covered yet, a hint is shown to the user.

## 4.4 Interaction Mechanisms to Establish Connections

Creating connections between two representations requires an active selection $r_0$. The procedure can be started by selecting the input or the output port to activate it. For convenience, if there is exactly one port, it is directly activated. Then, recommended connections are further restricted according to $S$ or $T$ of $r_0$.

Since there may exist several similar capabilities whose only difference is the parameter signature of the underlying operation or event, **clustering** takes place. Thereby, all representations for a particular component with the same activity and entity are grouped. The weather panel, see Fig. 3, offers two operations annotated with capability *<Display, Weather,* $\top$*>*. One requires a location parameter and the second an additional date. Different outputs, i. e., events of a clustered capability are transparently handled for the user and are not shown explicitly. When determining recommendations the events are investigated separately, and the correct one is chosen before implementing the channel in the mashup. A major advantage of our approach is this possibility to abstract from interface details, like heterogeneous signatures and overloaded operations.

When clustering is required, the user has to choose the alternative he desires. Furthermore, in case there is no unambiguous parameter mapping possible, the user has to confirm or adapt it. To this end, as introduced in Sect 4.3, different by- and as-parts are determined by the label generator. Those options are displayed and set to a probable configuration as delivered by a recommendation. However, due to space limitations and in order to preserve spatial correlation, instead of revealing the complete label, representations hide details in a collapsed state at first. Therein, only the essential part of the label is shown, and on mouse-over, the representation expands. Consider the represented capability with label ... to search flights... in Fig. 3, where the expanded state is illustrated. It shows several ports for each matching parameter of $r_0$, and options for the corresponding parameters of the target capability. In the collapsed state, the label would be a concise ... to search flights and no options would be visible. Similarly, capability representations $r_j \in S$ are handled.

After clicking the desired port or via drag&drop, the data flow oriented connection is created. Subsequently, the platform checks whether the new connection is "sufficient". If, for example, a parameter of the underlying operation is not assigned, an exclamation mark appears (c. f. Fig. 3) and provides general hints and the possibility to request recommendations. When cancelling the current selection, all labels are reset to their base configuration.

## 5 Evaluation

### 5.1 Methodology

We conducted a user study utilizing the think aloud protocol. 10 users in the age of 22 – 37 participated and were asked to fill a questionnaire to gather demographic and skill-related data. They are students from different fields like mechanical or electrical engineering, media and computer science, and logic. The participants had no or very basic knowledge about mashups, but frequently use web applications. 5 users described their programming skills as average, so that we could evaluate the suitability of CapView not only for non-programmers.

After a short introduction to mashups and the CapView, two scenarios of increasing complexity in the travel planning domain, each comprising five tasks, were presented by the interviewer. Based on a click prototype similar to Fig. 3 covering core UI and interaction concepts, each scenario includes a mashup application with UI and service components. In the first scenario, comprising four UI and one service component, the basic understanding of the exploration and interaction mechanisms was checked. Thereby, task like identifying which components can help to search flights, and connecting capabilities so that it is possible to search and book hotels had to be solved. The second scenario focused on the concepts for creating and manipulating connections using parameter mappings and extends the first scenario to confront the participants with a non-trivial mashup. Participants were asked to extend the mashup to be able to find events in the target location, search public transportation from the hotel to the event location and display the weather. Thereby, users had to reconfigure connections and handle multiple parameters. According to the think aloud protocol, while task solving, they were encouraged to express what they are doing and why, and what system behavior they expect. The interviewer observed and supported them if necessary. We were interested in whether participants are able to solve the tasks. Additionally, after completing their tasks, users were asked to fill out a questionnaire about their perceived task load and their assessment of the CapView's suitability using the System Usability Scale (SUS). Further, users were encouraged to comment on things they liked or disliked.

### 5.2 Results

As an important result, all participants were able to solve the tasks. Speed and efficiency differs depending on the user's background. In general, key concepts of CapView were perceived very positive. The basic idea of CapView to provide a functional abstraction for non-programmers was approved by all participants. Since CapView overlays the LiveView, the spatial correlation to live components is facilitated and eased the understanding of component functionality.

Further, natural language labels of capabilities were considered sufficiently intuitive (70%) to understand the functionality of components and to realize a mashup as a task-solving entity. 80% of the participants found highlighting connectible ports very helpful and even stated that they would not have been able to succeed without it. In line with this, the proposed context-sensitive adaptation of labels supported the understanding of connectibility. The combination of all exploration means (focusing, highlighting, label generation, appending or prefixing dots to build sentences, component name) eased the hurdles significantly. Due to reduced complexity and to improved clarity, 80% of the participants liked the overview and detail metaphor of expanding and collapsing representations.

We observed, that users used both approaches the concept offers to establish connections, i. e., starting with input respectively output ports. This underlines the necessity to provide both approaches to not constraint the user.

However, we discovered that users repeatedly faced the following difficulties. First, it is hard for non-programmers to understand the concept of service components. This lead to misinterpretations of capabilities, for example, a user assumed that search flights (see Fig. 3) directly displays the results as well. Thus, those details should be appropriately abstracted as well in future work.

In addition, it became evident that the expectations on components' functionalities are highly influenced by the users' experience with web applications like Google Maps. As a consequence, meaningful capabilities have to be provided. In this regard, we found that users interpreted input and output ports differently: In a more human-centered perspective, they expected that, for example, select an event only has an input. This partly contradicted with the system-oriented perspective we used when annotating components, where select an event provides output. However, after a short time they grew familiar with our perspective. Few users had problems to realize that CapView abstracts from instance data. Thus, LiveView and CapView should be stronger interwoven.

To get a widely-accepted evaluation scale, we additionally surveyed the SUS score as well as the Task Load Index. The average SUS score equals 78.5, with a maximum of 92.5 and a minimum of 70. We consider this as a good result with regard to the preliminary status of our prototype. In more detail, 80% of the participants would like to use the system frequently in their daily life. The system's complexity was stated as low by 90%. 90% found it easy to use, too, and 70% attested a quick learnability. The positive user feedback was confirmed by the Task Load Index. For instance, mental demand and effort was assessed between low and medium and the frustration level as very low. With regard to their overall performance, users were very content. Due to the nature of our study, physical and temporal demand have limited significance.

## 6 Conclusion and Future Work

Today, the mashup paradigm is widely-accepted as promising approach for end user development of web applications. However, prevalent solutions only partly meet the strict requirements of non-programmers. Mostly, interface-oriented wiring is used, requiring technical understanding of the application. In this paper, we propose CapView, a novel functionality-aware development view on running mashups. It provides an overview of the capabilities and properties of components and recommended composition fragments. CapView abstracts from composition and implementation details.

Natural language labels for capabilities are derived and adapted with respect to the current selection of the user. Thereby, short sentences are formed in order to emphasize the functional interplay of components. This approach causes stronger dependency on useful annotations, and may be less precise than a extensive textual description. However, in our opinion the main advantage is genericity. Even unforeseen constellations of components can comprehensively be covered, which is important to meet the long tail of user needs. Further, the dependency on human-provided documentation of every component or recommendation lessens since ontological knowledge can be reused.

Consequently, non-programmers are empowered to explore the functionality of a mashup and its building blocks, and to manipulate the mashup through visually composing capabilities. We evaluate the CapView via a user study, and outline the results as well as identified future challenges.

Based on the concepts introduced in this paper and the lessons learned from the user study, we are working on stronger interweaving CapView and LiveView. Using the CapView's level of abstraction, we strive for an intuitive way for the user to express his/her goal. Further, we want to derive capabilities and their relationships from more complex composition fragments, and elaborate functionality-based visualization of recommendations in LiveView and CapView.

Finally, after finishing the integration of the CapView in our demo prototype, we plan to conduct an extensive user study to evaluate the overall platform in comparison to existing mashup platforms.

**Acknowledgments.** Funding for the EDYRA project is provided by the Free State of Saxony and the European Union within the European Social Funds program (ESF-080951805).

## References

1. Aghaee, S., Pautasso, C.: Englishmash: Usability design for a natural mashup composition environment. In: Grossniklaus, M., Wimmer, M. (eds.) ICWE Workshops 2012. LNCS, vol. 7703, pp. 109–120. Springer, Heidelberg (2012)
2. Alonso, F., Lizcano, D., Lopez, G., Soriano, J.: End-user development success factors and their application to composite web development environments. In: Sixth Intl. Conf. on Systems (ICONS 2011) (2011)
3. Bianchini, D., Antonellis, V.D., Melchiori, M.: A recommendation system for semantic mashup design. In: DEXA Workshops, pp. 159–163. IEEE (2010)
4. Cappiello, C., Daniel, F., Matera, M., Picozzi, M., Weiss, M.: Enabling end user development through mashups: Requirements, abstractions and innovation toolkits. In: Piccinno, A. (ed.) IS-EUD 2011. LNCS, vol. 6654, pp. 9–24. Springer, Heidelberg (2011)
5. Cappiello, C., Matera, M., Picozzi, M., Sprega, G., Barbagallo, D., Francalanci, C.: Dashmash: A mashup environment for end user development. In: Auer, S., Díaz, O., Papadopoulos, G.A. (eds.) ICWE 2011. LNCS, vol. 6757, pp. 152–166. Springer, Heidelberg (2011)
6. Roy Chowdhury, S., Daniel, F., Casati, F.: Efficient, interactive recommendation of mashup composition knowledge. In: Kappel, G., Maamar, Z., Motahari-Nezhad, H.R. (eds.) Service Oriented Computing. LNCS, vol. 7084, pp. 374–388. Springer, Heidelberg (2011)
7. 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)
8. Imran, M., Kling, F., Soi, S., Daniel, F., Casati, F., Marchese, M.: Reseval mash: a mashup tool for advanced research evaluation. In: 21st Intl. Conf. companion on World Wide Web (WWW 2012), pp. 361–364. ACM (2012)
9. Namoun, A., Wajid, U., Mehandjiev, N.: Service composition for everyone: A study of risks and benefits. In: Dan, A., Gittler, F., Toumani, F. (eds.) ICSOC/ServiceWave 2009. LNCS, vol. 6275, pp. 550–559. Springer, Heidelberg (2010)
10. Nestler, T., Feldmann, M., Hübsch, G., Preußner, A., Jugel, U.: The servface builder - a wysiwyg approach for building service-based applications. In: Benatallah, B., Casati, F., Kappel, G., Rossi, G. (eds.) ICWE 2010. LNCS, vol. 6189, pp. 498–501. Springer, Heidelberg (2010)
11. Pietschmann, S., Radeck, C., Meißner, K.: Semantics-based discovery, selection and mediation for presentation-oriented mashups. In: 5th Intl. Workshop on Web APIs and Service Mashups (Mashups), pp. 1–8. ACM (2011)
12. Pietschmann, S., Tietz, V., Reimann, J., Liebing, C., Pohle, M., Meißner, K.: A metamodel for context-aware component-based mashup applications. In: iiWAS 2010, pp. 413–420. ACM (2010)
13. Pietschmann, S., Voigt, M., Meißner, K.: Rich communication patterns and end-user coordination for mashups. In: Brambilla, M., Tokuda, T., Tolksdorf, R. (eds.) ICWE 2012. LNCS, vol. 7387, pp. 315–322. Springer, Heidelberg (2012)

14. Radeck, C., Lorz, A., Blichmann, G., Meißner, K.: Hybrid recommendation of composition knowledge for end user development of mashups. In: ICIW 2012, pp. 30–33. XPS (2012)
15. Rümpel, A., Radeck, C., Blichmann, G., Lorz, A., Meißner, K.: Towards do-it-yourself development of composite web applications. In: Proceedings of the Intl. Conf. on Internet Technologies & Society 2011 (ITS 2011), pp. 231–235 (2011)
16. Tietz, V., Pietschmann, S., Blichmann, G., Meißner, K., Casall, A., Grams, B.: Towards task-based development of enterprise mashups. In: iiWAS 2011, pp. 325–328. ACM (2011)
17. Wilson, S.: D3.3 prototype implementation of the omelette live environment: Phase 1. Tech. rep., ICT Omelette (2012)