Helping Non-programmers to Understand the Functionality of Composite Web Applications

Carsten Radeck and Klaus Meißner
Faculty of Computer Science, Technische Universität Dresden, Dresden, Germany

Keywords: Mashup, Awareness, Inter Widget Communication, Generated Tutorials, End User Development, Assistance.

Abstract: The mashup paradigm allows end users to build their own web applications by combining components in order to fulfill specific needs. Mashup development and usage are still cumbersome tasks for non-programmers, for instance, when it comes to understanding the composite nature of mashups and their functionality. Non-programmers may struggle to use components as intended, especially if the latter provide capabilities in combination, and may lack awareness for inter-widget communication (IWC). Prevalent mashup approaches provide no or limited concepts for these aspects, resulting in more or less successful trial and error strategies of users. In this paper, we present our proposal for assisting non-programmers to understand and leverage the functionality of components and their interplay in a mashup. Based on annotated component descriptions, interactive explanations and step-wise instructions are generated and presented directly in context of components' user interface (UI). In addition, active IWC is visualized to foster awareness of users. We describe the iterative design which led us from early approaches towards our current solution. The concepts are implemented in our mashup platform and evaluated by means of a user study. The results indicate that our solutions help non-programmers to better understand the functionality of composite web application (CWA).

1 INTRODUCTION

Building up on the increasing number of web resources and application programming interfaces, the mashup paradigm allows re-use and lose coupling of components in a broad variety of application scenarios and use-cases. Universal composition approaches enable platform-independent modeling of mashups. They apply uniform description and composition of components spanning all application layers, ranging from data and logic services to UI widgets.

The mashup paradigm and end-user development complement each other quite well, and together allow to better meet the long tail of user needs. However, it is still cumbersome for end-users to develop and even use CWA, especially as non-programmers are the target group. There are several challenging tasks for non-programmers in CWA development and usage (Radeck et al., 2016; Chudnovskyy et al., 2013). In this paper we specifically focus on the following two: ① understanding what single components are capable of and what functionality they provide in interplay, as well as ② being aware of IWC.

Our platform adheres to universal composition and aims at enabling non-programmers to build and use situation-specific CWA. Non-programmers can manipulate an application while it is executed , and thus get instant feedback on their actions. Further, they are guided by recommendations on composition patterns. Components are semantically annotated with the capabilities they provide. Based on this, the capabilities of arbitrary composition fragments are automatically derived (Radeck et al., 2016). This allows our mashup environment to offer a set of assistance features with which we tackle the challenges mentioned above. With the help of our **reference scenario**, we illustrate both challenges in detail now.

Non-programmer Bob uses an existing mashup for travel planning recommended by a friend. It consists of two maps, a route calculator, a weather widget and two widgets for searching points of interest and hotels. Since he is neither familiar with the overall application nor the single components utilized, Bob faces several understanding problems of which functionality the mashup provides and which not and how to achieve it. For instance, Bob is not sure why there are two maps, if the location in a map has effect in other components, and if so, which kind of effect, and how to find hotels near the target location. While normally he would have to explore the mashup manually

Radeck, C. and Meißner, K.
Helping Non-programmers to Understand the Functionality of Composite Web Applications.
DOI: 10.5220/0006362301490159
In *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST 2017)*, pages 149-159
ISBN: 978-989-758-246-2

in a trial and error style, the platform supports Bob in gaining insight. First, there is an overview panel displaying the mashup functionality, possibly composed of several sub-functionalities. It allows Bob to inspect what tasks he can solve with the application at hand and which components partake in certain functionalities. This way, Bob understands that one map serves for selecting the start location, while the other is used to select the target location for the route. Since Bob has no clue about how to see a list of routes, he activates an explanation mode, which shows interactive animations of necessary steps and interactions he has to perform. It highlights elements of the component UI, indicates sequential and parallel processes and supplements this with textual descriptions of what to do. Such explanations not only work for component interplay, but for capabilities of single components and how these are reflected on the component UI, too. This helps Bob understand that he can move a marker or type the location name in an input field of the map in order to select a location. Bob did not yet notice the communication link between map and weather widget, because these are positioned far away from each other on the screen. The platform offers a feature that animates the data flow when it occurs, utilizing the same visualization techniques as in the explanation mode. This way, Bob gets aware of data transfer between map and weather widget.

In order to implement the reference scenario and to tackle the challenges stated above, there are at least the following foundational **requirements**:

- Mashup functionalities have to be explained to non-programmers. While for single components tutorials can be statically provided by developers, this is an issue for CWA since end-users are the developers and components are combined in unforeseen ways. Explanations should be optically consistent for all CWA. Thus, it seems inappropriate to depend on component-provided tutorials and combine them. Therefore, explanations have to be provided automatically and have to support arbitrary mashups. They should be interactive and directly presented in the UI of components.
- There need to be mechanisms to foster awareness for IWC when it occurs. They have to support arbitrary communication channels. As pointed out in (Tschudnowsky et al., 2014), such mechanisms should directly be presented in the UI of components, as this actually can help users.

While most mashup approaches support users with visual composition paradigms and recommendations, assisting the understanding of a CWA at hand and making users aware of IWC are neglected or very limited so far. Traditional approaches like static help pages and forums lack suitability, too. Thus, as a main **contribution** of this paper, we introduce a set of generic techniques for exploration, explanation and awareness of the capabilities of arbitrary CWA. They allow non-programmers to investigate the functionality of single components and of the IWC of a mashup. Based on component annotations, step-by-step tutorials are generated, which textually and graphically present instructions on how to achieve capabilities in the component UI. Furthermore, we propose concepts for awareness of IWC at runtime. The techniques are especially valuable since they work for unforeseen combinations of black box components. We evaluate our concepts with the help of a user study.

The remaining paper is structured as follows. In Section 2 we discuss related work. Next, we briefly outline our overall assisted end user development (EUD) approach in Section 3. A requirements analysis based on evaluation results of our early approaches is subject of Section 4. Based on this, we introduce our concepts for explanation and awareness of IWC in Section 5. Results of a user study we conducted for evaluation are presented in Section 6. Finally, Section 7 concludes the paper and outlines future work.

2 RELATED WORK

According to (Gery, 1995), three types of performance support can be distinguished. *Intrinsic support* is inherent to the system and thus seamlessly integrated with the UI and behavior of the system. As an advantage there is no break in the workflow of users. But such techniques are limited in extent since they compete with the normal application for the UI space. *Extrinsic support* is directly integrated with the system as well, yet not in the primary workspace. It is often contextualized regarding the user task and typically requires the user to invoke or accept it (Gery, 1995). Examples for this category include explanations, demonstrations and wizards. Our approach can be assigned to this category. *External support* is not integrated with the workspace itself, for example, forums, tutorials and help pages on the web. Due to missing context specificity, users have to transfer the knowledge to their current task context ad hoc. External support is suited for extensive developer-provided content, like components in our case, and introductions to static content. However, in the case of short-termed, situation-specific CWA with unforeseen combinations of components and dynamic application context, they lack practicability and suitability.

The *Idea Garden* (Cao, 2013) aims at supporting users to overcome design barriers. To this end, strate-

gic suggestions for problem solving are visualized in an extrinsic manner. Suggestion also include stepwise instructions on how to perform a strategy. However, suggestions are often generic and context-free and there is no direct link to the component UI.

The *Whyline* (Ko and Myers, 2004) supports users to debug their programs. In order to do so, developers construct "why did" and "why did not" questions using menus and referencing objects involved in an algorithm. Answers are presented using a graph visualization of actions, which happened at runtime, and arrows that indicate causality. This approach operates on a more source code oriented level than our concept and there is no visualization of active data transfer.

In the mashup domain, most EUD platforms provide at least external support. As we explicate in the following, intrinsic and extrinsic techniques are missing or very limited so far. Some composition paradigms and metaphors can also be used to explain what should go on in a mashup. Usually a derivation of "if-then" is employed, e. g., in *IFTTT*[1], *CapView* (Radeck et al., 2013), *PEUDOM* (Picozzi, 2013) and *NaturalMash* (Aghaee and Pautasso, 2014). In simple scenarios, the resulting composition logic is suitable to also explain functionality. However, a link to the actual UI needs to be provided and missing in *IFTTT* and *CapView*. In *PEUDOM* communication channels are created or edited in a dedicated dialog by selecting an event and an operation. Based on this, a short sentence reflects the causal relation. However, there is no means for explaining the overall functionality. In a widget editor, the link between data schema and UI can be established graphically. However, techniques are not used to explain widget functionality later on. *NaturalMash* (Aghaee and Pautasso, 2014) uses restricted natural language to define composition logic. Conceptually, when selecting text fragments, corresponding UI elements are highlighted. However, details and an evaluation on this are unavailable. We also utilize generated labels, but further support animations for complex composition fragments directly within the UI of a CWA. In *FAST-Wirecloud* (Lizcano et al., 2016) semantically annotated widgets are composed. *Behaviors* represent functionality parts of a mashup and feature natural language descriptions. They are used to ease the composition process, but not to explain a mashup at runtime.

Within the *OMELETTE* project, a live development mashup environment has been created. A distinct composition paradigm is followed, where per default all possible communication channels between widgets are established. This leads to challenges in awareness and control of users (Chudnovskyy et al., 2013). Thus, features fostering awareness for IWC were proposed (Tschudnowsky et al., 2014). Communication relations of widgets are shown on the canvas and data transfer is indicated by highlighting the involved widgets. However, the authors point out, that suitable visualization techniques are still an open research question. In addition, there are no explanations of the functionality communication relations provide.

The *MashupEditor* presented in (Ghiani et al., 2016) highlights UI elements of widgets and visualizes arrows to indicate communication relations. We use similar base techniques, but also present animations and textual explanations to the user.

In summary, none of the presented approaches fulfills all requirements. Thus, we propose generic techniques for mashup platforms that explain CWA functionality in a manner adequate for non-programmers.

3 THE CRUISE PLATFORM FOR EUD OF MASHUPS

Now we briefly outline the core characteristics and necessary foundations of the CRUISE mashup platform (Radeck et al., 2016) shown in Figure 1 and relate the concepts we present in this paper to it.

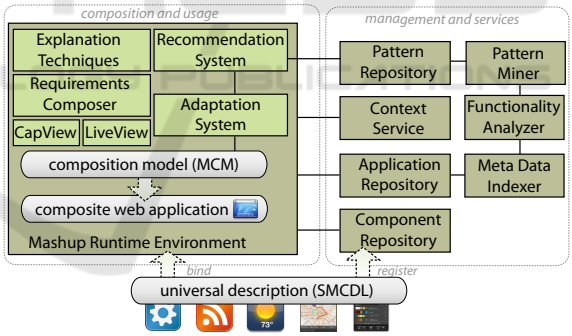


Figure 1: Architectural overview of our platform.

The CRUISE platform follows a model-driven approach to create and execute mashups. Web resources and services can be encapsulated as components. In line with universal composition, components of all application layers are uniformly treated as black-boxes and share a generic *component model*. The latter characterizes components by means of several abstractions: events and operations with typed parameters, typed properties, and capabilities. The declarative Semantic Mashup Component Description Language (SMCDL) provides a concrete syntax for the component model. It features semantic annotations to clarify the meaning of component interfaces and capabilities (Radeck et al., 2013). All aspects of a CWA

[1]https://ifttt.com/

are modeled with the help of the *Mashup Composition Model (MCM)*. This includes the components to be integrated, application screens and their layout, and the event-based communication.

Capabilities allow to model functional and behavioral semantics of composition fragments, i. e., of components, applications as well as patterns. Capabilities describe what a composition fragment is able to do or which functionality it provides, like displaying a location or searching hotels. To this end, capabilities essentially are tuples (*activity*, *entity*) – denoted `activity entity` from now on – and express which activity or task is performed on or with which domain object, e. g. `search hotel`. References to semantic concepts described in ontologies back the description with formal semantics providing domain-specific knowledge and allow for reasoning. To achieve a capability, it may be necessary for the user to partake and interact with the UI or not. Therefore, UI and system capabilities are distinguished. To establish hierarchical structures, capabilities can be *composite*. The relation of children of a composite capability is defined by a *connective*, e. g., parallel and sequential. In case of sequences, capabilities are chained to define the order using *next* and *previous*. As an example, the capability `search route` can be described as a sequence of `select start`, `select destination`, `search route` and `display route`.

View bindings are a concept particular for UI capabilities. They link the semantic layer and the UI of a component. Basically, a view binding comprises interaction steps, modeled as atomic, parallel or sequential operations. Atomic operations point to DOM elements, using a selector language, e. g. CSS selectors, can name the elements and define interaction techniques, like click and sweep (see lines 3 and 6 in Listing 1). To reduce complexity of annotating and algorithmic processing, composite operations are restricted to one layer of atomic operations. Each UI capability can be equipped with multiple view bindings, which are considered alternatives then, e. g., if it is possible to select a location via typing something in a text field or double clicking a map (lines 2 and 5).

```
1 <capability activity="Select" entity="Location">
2   <viewbinding>
3     <atomicoperation element="input[id$='
          mapTextField']" interaction="type" />
4   </viewbinding>
5   <viewbinding>
6     <atomicoperation element="div[id$='
          locationMarker']" elementName="map
          marker" interaction="dragAndDrop" />
7   </viewbinding>
8 </capability> ...
```

Listing 1: Excerpt of a SMCDL descriptor.

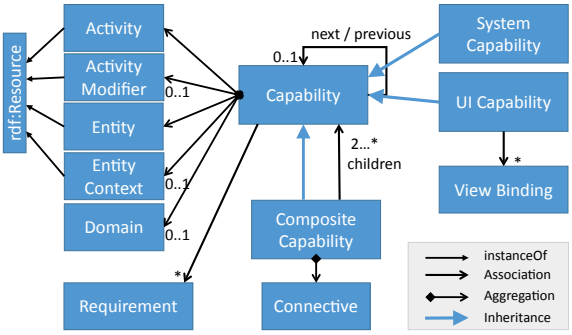


Figure 2: The capability meta-model.

The CRUISE mashup platform strives to enable end users to build custom CWA. Thereby, we specifically address domain experts without programming knowledge. They typically know their problem and possible solutions in terms of domain tasks to perform, but fail to map such solutions on technical requirements or mashup compositions. A fundamental feature of our approach is that run time and development time of a CWA are strongly interwoven. Users can seemingly switch between editing and using an application. Composition model concepts and other technical terminology are hidden to a high extent. Instead, communication with users takes place on capability level and necessary mappings of composition steps to composition model changes are handled transparently by the platform.

A mashup runtime environment (MRE) is equipped with a set of tools and mechanisms. For example, a recommender system gives advice on composition fragments to users. Deriving recommendations is based on patterns, which represent composition knowledge. An MRE provides different views on the current CWA: The live view is mainly intended for usage and thus only component UIs are visible to the user. Complementing this, there are overlay views, like the CapView (Radeck et al., 2013), that display component and composition model details and mainly serve for development purposes. Tools like the aforementioned CapView and our novel explanation techniques, which are subject to this paper, explain the capabilities originating from components and their interplay in a textual and visual manner.

Components, composition models and patterns are managed by server-side repositories and analyzed by several modules. For instance, the functionality analyzer calculates the capabilities of composition fragments (Radeck et al., 2016). The concepts we present in this paper build up on the derived capability models. Regarding CapView, we re-use its label generator, but provide explanation techniques directly in the component UI rather than in an abstract view.

4 FIRST APPROACHES AND THE LESSONS LEARNED

As described in (Radeck et al., 2013), the **CapView** not only allows to perform development tasks like adding communication channels. It additionally provides basic means for inspection and navigation of component capabilities. CapView is overlaying the live view of CWA and positions capability representations above the components that provide them (see Figure 3). It visualizes connection lines between capabilities in case there are communication channels. Based on a set of rules and the ontology concepts referred to in capabilities, labels are generated. After a capability is selected, the labels of all other capabilities, with which it can be connected, are adapted to form short sentences. This way, the functionality provided by communication channels is explained to users. Non-UI components are treated specially. In CapView an icon representing the service is attached to connections where it is involved. The live view features a dedicated **non-ui component panel**. In both cases, generated labels describe the functionality.

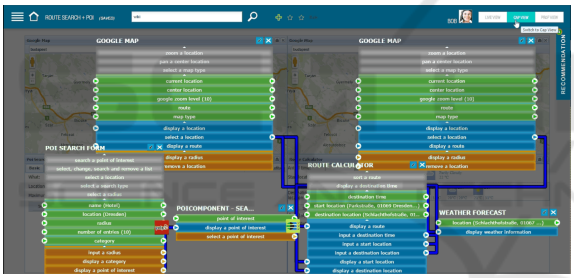


Figure 3: CapView prototype in a rather complex CWA.

To evaluate this first iteration of concepts, we conducted several small user studies. We found in the user study presented in (Radeck et al., 2013), that this approach, especially adaptive labels and positioning capabilities above components, has potential. However, we also pointed out some drawbacks. For instance, since CapView is designed as an overlay and abstracts from instance data and component UI, users had problems to understand how to achieve capabilities. We conducted another small user study, in order to deepen our problem understanding by identifying conceptual and usability issues users face using our composition environment. Five non-programmers participated. They were briefly introduced to the platform and then asked to solve several tasks. Using our prototype, different CWA, e. g., a route search mashup with two maps and a route calculator, were presented to the users. The tasks aimed at testing if users understand component and application functionality and if the existing tools are accepted and usable. For instance, the participants were asked to name the overall mashup capabilities, to identify components contributing to the `route search` capability, to list interacting components and the resulting functionality, and to describe how to `search points of interest`. Finally, the participants were encouraged to freely use a third CWA and thereby mention understanding problems and suggest improvements.

The results of this user study were manifold. First, we confirmed some known and identified new issues with CapView as a means to understand CWA. Main problems we identified were switching into a dedicated view and thus leaving the live-view, and users struggling with realizing the connection between CapView and corresponding elements in the component UI. Consequently, users experimented in the live view rather than using CapView. Few users were unable to cope with CapView at all. Generally, it became evident that even in rather simple applications, users may face understanding problems when using components or applications they are unfamiliar with or which do not match with their expectations. Some users misinterpreted the purpose components serve for, since they expected that it is determined by positioning on the canvas or the temporal sequence they interact with components. Moreover users were sometimes unaware of IWC, e. g., between map and weather widget. In this point, we can confirm the results of (Chudnovskyy et al., 2013). Also directed data flow caused confusion when a bidirectional synchronization was assumed, e. g., between maps and route calculator. The non-ui component panel was often ignored or misinterpreted as advertisement.

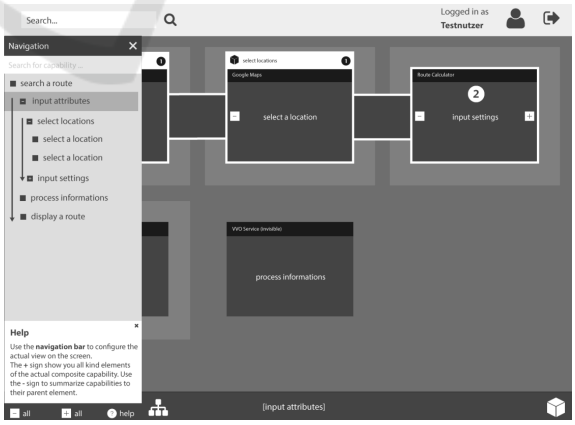


Figure 4: The discontinued CompCapView approach.

As another limitation, CapView does not give hints about overall capabilities of a CWA at hand. Thus, we developed **CompCapView** as an extension that supports composite capabilities, see Figure 4. Basic principles are kept, e. g., it overlays the live view,

and capability representations are drawn above the components, which offer these capabilities. Depending on the current position within the capability hierarchy, representations of child capabilities are drawn and connected to each other. In case of a sequential relation of the capabilities, their order is denoted by numbers. Initially, all root capabilities are displayed. It is possible to zoom in and out within the hierarchy. In addition, a **panel displaying CWA capabilities** in a tree-like manner was created. As a main feature, it is synchronized with the current view. In case of CompCapView it propagates the currently selected capability which is focused and highlighted subsequently. Within the live view, the components providing or partaking at the selected capability are highlighted.

In order to evaluate these concepts, we conducted a small user study. The evaluation was based on a prototype of CompCapView and the capability panel within our client-server MRE. Six persons participated, which were in average 23 years old and all had no knowledge about mashups and programming. They worked or studied in different sectors, like economy, medical engineering, social pedagogy and retail.

After a short introduction to the principles of CompCapView, participants were asked to solve several tasks that aimed at the core features of the CompCapView. For example, they were asked to name the overall capability of a given application and to describe in which sequence components are employed to fulfill this functionality. Furthermore, we were for instance interested in whether the navigation between layers is understandable and usable and if participants comprehend the meaning of connections.

All participants were able to determine the overall mashup functionality and 5/6 stated correctly which components actually contribute to mashup functionality. Most participants understood the layering concept (4/6) as well as the relation of composite capability representations and the actual mashup components (5/6). However, only half of the participants identified sequential or parallel capabilities correctly. Several remarks and comments address issues of the prototype rather than conceptual ones. A crucial conceptual problem was that connections of composite capabilities were misinterpreted as an indicator for order, which was further complicated by assumed reading directions. The study revealed, that specially the navigation panel was considered useful since it provides quick access to capabilities of a CWA. An average system usability scale (SUS) score of 66 indicates acceptable, yet limited usability. In addition, the results of the Task Load Index questionnaire show medium mental demand and effort. Although the participants considered time demand low and frustration low to medium and were satisfied with their performance, this underpins the finding, that the chosen approach has potential but lacks suitability for non-programmers.

In summary, from these findings and observations, we amongst others conclude that:

- explanations techniques should operate directly within the UI of components of the CWA at hand;
- visual connections of capabilities must carry an intuitive semantics; and
- composite capabilities should be accessible via an overview panel.

Next, we present our current approach that incorporates the lessons learned.

5 CURRENT APPROACH

In this section we go into details on our concepts for assisting users to understand what functionality a CWA provides and how to use it. This subsumes several features, which serve different purposes, like giving an overview on capabilities, providing detailed stepwise instructions and making users aware of IWC.

5.1 Preliminaries

As described in Section 3, components are annotated with capabilities. Each UI capability should carry *view bindings* in SMCDL.

Capabilities of components serve as a foundation for capabilities of arbitrary composition fragments. Their connections defined both within components and by IWC can be analyzed to derive a capability graph as presented in (Radeck et al., 2016). Such a graph consists of capability links as edges and capabilities as nodes. Capability links represent inter-component communication channels and intra-component relations as defined in SMCDL. Isolated subgraphs which are coherent in themselves are called capability chains. Starting from the capability graph, a hierarchy graph is calculated per capability chain. Thereby, composite capabilities representing higher level functionalities provided by a CWA are derived. In order to identify the interaction steps for a composite capability from the hierarchy graph, the hierarchy has to be recursed until reaching atomic capabilities. This leads to the corresponding capability chains, allowing to apply the cases introduced above.

In light of the presented modeling concepts, several constellations of capabilities and view bindings, summarized in Figure 5, have to be considered by explanation techniques based on capability graphs.

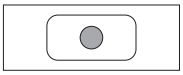
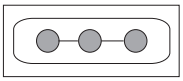
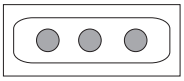
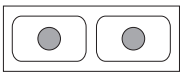
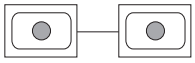
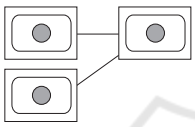
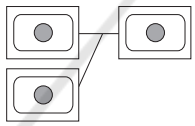
Case ID	Description	Schematic representation
Single UI Capability		
1	**One view binding** with **one atomic operation**	
2	**One view binding** with **one sequential operation** consisting of several atomic operations, which have to be performed in a specific order.	
3	**One view binding** with **one parallel operation** expressing that several atomic opereations have to be performed without specific order.	
4	**Several view bindings** representing alternatives. Cases 1-3 apply as well.	
Capability Chains		
5	Single capability link between two Capabililties. Cases 1-4 apply as well.	
6	Multiple ingoing capability links (**OR Pattern**): Data for a target capability are provided by several capabilities. Cases 1-4 apply as well.	
7	Multiple ingoing capability links (**AND Pattern**): Several capabilities are required in conjunction for a target capability. Cases 1-4 apply as well.	

Figure 5: Main constellations to be considered.

5.2 Core Visualization Concepts

Explanation techniques directly operate within the UI of components rather than in abstract views. To do so in context of our black-box component model, view bindings declaratively establish links between UI and capabilities. To cover all cases outlined in Figure 5, there are three basic visual element types, which we consistently use within the explanation techniques.

View binding frames highlight UI elements referenced by view bindings.

Arrows connecting view binding frames represent capability links and indicate data flow direction.

Description boxes are placed near view binding frames or arrows and display textual explanations of capabilities or interaction steps. The text is context-sensitively generated utilizing capability and view binding annotations.

Per default, a view binding frame is rendered for all DOM elements referenced by the selectors of the atomic operations of a capability c. If c is no UI capability or has no view binding, the whole component panel is framed. In the special case of non-UI components a description box is rendered and framed. Further challenges are posed by UI elements referenced by view bindings, which are invisible at the time needed, e. g., if a component uses tabs. Although it would be possible to make such elements visible, e. g., via declarative instructions in SMCDL or imperative implementations that are invoked by the MRE. However, this results in high effort for component developers and it is questionable whether such solutions work generically. Thus, we employ a simplified, generic solution. If a referenced element is invisible, the whole component panel is framed and a textual hint is shown in description boxes.

5.3 Assistance Features

Based on the capabilities described in SMCDL, the capability and hierarchy graphs of a CWA, and the visual element types introduced above, we propose a set of generic explanation techniques. These are: *Capability panels* listing capabilities of mashups and components; a *recommendation panel* that shows recommended patterns emphasizing their capabilities; the *inspection mode* for exploring capabilities and capability links in the UI; the *tutorial mode* which provides step-by-step instructions; as well as the *awareness mode* indicating data flow at runtime.

The **inspection mode** allows users to explore the functionality of components and CWA. Its main purpose is to present capability chains of a mashup, describe their functionality and indicate relevant UI elements. This way, the inspection mode can be considered as a CapView integrated within the live view.

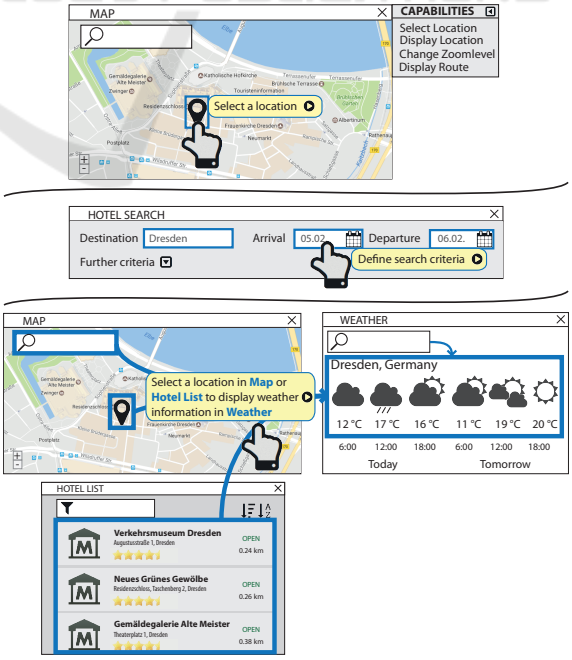


Figure 6: Exemplified inspection mode.

From a dedicated menu button the inspection mode can be activated and deactivated. Once it is active, all inter-component capability links are visualized by drawing view binding frames and arrows. Intra-component links are hidden initially if they are not transitively connected to inter-component links, and are shown on demand. The rationale for this is to avoid confusing appearance and that component capabilities are often known to users.

As soon as the user explores a component's UI, its capability panel appears. Further, view binding frames become visible when the corresponding UI elements are hovered as illustrated in the upper part of Figure 6. In cases 2 and 3, all other elements referenced in sister atomic operations are highlighted as well, see middle part of Figure 6. Additionally, a description box appears. It provides access to the tutorial mode for that capability, indicated by a "play button", and hosts a text which is derived from capabilities using the concepts from (Radeck et al., 2013). If the capability whose view binding is hovered is part of a capability chain, all corresponding arrows and frames are emphasized, respectively intra-component chains become visible. Further, the description text mentions all capabilities of a link. It explains cause and effect by forming sentences based on the capabilities and communication channels involved, paying attention to the link direction and the hovered capability. We extended the label generation facilities to also include component names and to connect several capability labels by "and" (case 7) and "or" (case 6). Hovering special text fragments, like component names and capability labels, leads to highlighting of component panel or view binding frame.

When selecting an arrow, it and all other arrows and view binding frames belonging to that capability link are emphasized and a description box appears as shown in the lower part of Figure 6. Again, the user can start a tutorial to learn in detail what to do.

The **tutorial mode** focuses on explaining interaction steps rather than the full detail of capabilities. Therefore, UI capabilities are of main interest and serve to generate stepwise instructions. This mode is accessible via buttons in capability panels and in description boxes of the inspection mode.

When activated for a capability or a capability chain, the tutorial mode shows and highlights the view binding frames of the first interaction step by graying out the remaining CWA in order to direct user attention, cf. Figure 7. A description box fades in, featuring a generated text and a navigation bar. The latter has controls to auto-play, close, pause, step forward, step back in the tutorial. Generated description texts consist of the interaction technique required, the

Figure 7: Exemplified tutorial mode.

element name, the capability achieved and the component name. Again, the sentence structure differs slightly depending on the position of the capability in a link in order to reflect cause and effect.

Examples for the cases 1 and 4 can be seen in the upper part of Figure 7. In case 2 one tutorial step per atomic operation exists, see middle part of Figure 7, while in case 3 there is a single step and a text mentioning all atomic operations in conjunction. Similarly, in case 6 all source capabilities of a link are explained in one step, with a description box per source capability. In contrast, a constellation like in case 7 results in separate steps per source capability.

Tutorials incorporate non-UI components. To this end, an additional step is included, which visualizes a description box displaying the component's name, icon and capabilities provided in context of the capability chain to be explained. For an example, please refer to the lower part of Figure 7.

The purpose of the **awareness mode** is to make non-programmers aware of IWC when it occurs at run time of an arbitrary CWA. To this end, it builds

upon the presented visualization concepts for inter-component capability links. Via a dedicated button in the menu bar, the mode can be activated.

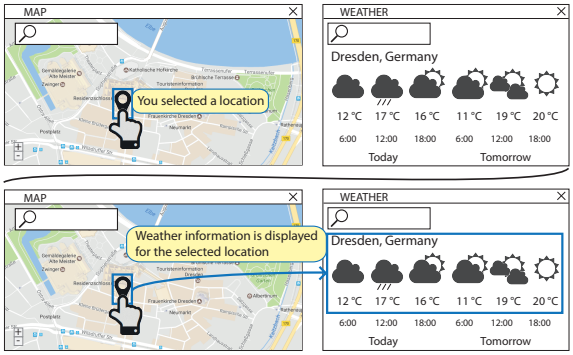


Figure 8: Exemplified awareness mode for two instants.

As soon as an event is fired on a communication channel, the corresponding capability c is determined and set as start capability c_s. In case c is no UI capability and is linked with preceding capabilities, c_s is set to the first UI capability from the predecessors if there is such. Analogously c_t is obtained at the target side of a capability link. An animation of configurable duration is started then, first displaying the view binding frames of c_s as illustrated in the upper part of Figure 8. A description box with generated text is presented to indicate the action. Next, an arrow is shown, followed by the view binding frames of c_t and a description box, see lower part of Figure 8. Case 7 requires particular consideration, where several interactions are necessary in order to join all data before c_t can actually be performed. Thus, the last animation step is postponed and instead the view binding frames of the required other capabilities are displayed for a while, each complemented with a description box pointing out necessary user actions. When all input is provided, the animation continues with the frames and description boxes of c_t.

A **capability panel** provides insight on the capabilities of the overall mashup or single components. It represents the tree-like hierarchy graphs of a CWA or lists the capabilities of components, as indicated in Figure 6. Capabilities are textually described by generated labels. When selecting a capability, all partaking components are highlighted. In addition, the tutorial mode can be started for that capability. Synchronization also takes place with the inspection mode: When a user hovers a view binding there, affected capability panels highlight the corresponding capability.

Figure 9 depicts the **recommendation panel** which lists recommended composition steps. A novel two-step approach for visualizing recommendations is applied and largely builds up on the capabilities of the underlying patterns. As a prerequisite, clusters of

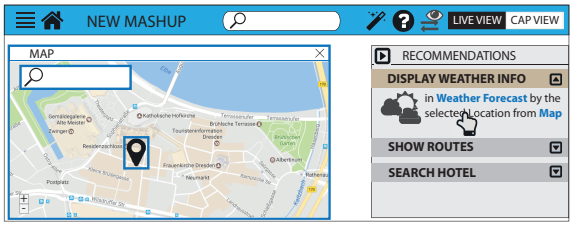


Figure 9: The recommendation menu.

recommended patterns according to their capabilities are calculated. In a first step, all clusters are presented to the user with the help of generated labels, e. g., `display weather info`. After the user made his choice, descriptions for each pattern from that cluster are shown. Therein, besides component icons, additional text fragments complement the cluster label to form a short sentence containing component names and the corresponding capabilities. When hovering these text parts, the component is highlighted or the view binding frame appears. In the previous example, a pattern-specific text extension may be "`in Weather Forecast by the location selected in Map`".

In summary, we propose generic explanation techniques which enable users to explore, learn how to utilize and become aware of CWA capabilities.

6 EVALUATION

In this section, we go into detail on the prototype we developed and how we validated our concepts.

6.1 Implementation

Our prototype extends the client-server runtime environment of the CRUISE platform, mainly its client-side part, which is implemented using technologies like HTML5 and CSS. Several JavaScript frameworks are employed. We utilize Bootstrap Popovers[2] for rendering description boxes and jsPlumb[3] for drawing arrows based on SVG. Bootstrap Tour[4] offers the necessary features to implement step-by-step tutorials. The capability model of the current CWA, which is derived by a dedicated web service and managed by the MRE as JSON object, is analyzed in order to setup corresponding data structures for the explanation techniques. Thereby, capability chains are processed in several ways: Intermediate system capabilities, e. g. in case of non-UI components, are skipped

[2]http://getbootstrap.com/javascript/
[3]https://jsplumbtoolkit.com/
[4]http://bootstraptour.com/

and information about composite capabilities, multiple view bindings of a capability and parallel or sequential operations are used to derive all interaction steps. Based on this, Tour objects are instantiated. Once activated, the prototype of all three modes creates a transparent overlay div element per component. It serves as canvas for rendering view binding frames, description boxes and arrows. We also implemented the CWA capability and recommendation panels.

6.2 User Study

Methodology. To validate the concepts presented in Section 5, we conducted a user study. Ten non-programmers with an average age of 31, ranging from 16 to 56, and from very different background and sectors, e. g., forestry, mechanical engineering, photography, economics, sale and medicine, participated.

We split the users in two groups of five each. Group A used our mashup platform without, the other group B with the explanation techniques. In line with the think-aloud method, our participants were asked to individually solve a set of tasks and thereby express their thoughts about next necessary steps, expected system behavior and how the actual system response matches their expectations. The participants had to solve multiple tasks of increasing complexity in context of different CWA. In the simplest case, component capabilities had to be identified and achieved. Further, we tested if the users were able to identify and perform capabilities provided by component interplay, like searching a route or points of interest, and if they are aware of IWC during mashup usage.

Besides creating the think-aloud protocols, we measured the number of successfully solved tasks per participant. In order to get a widely accepted and comparable usability measure, we asked each person to fill out a SUS questionnaire. Finally, users were encouraged to comment on things they liked or disliked.

Our main goal was to show that the explanation techniques help participants to gain insight about given CWA and thus better perform the tasks, indicated by the number of solved tasks. In order to do so, we compared the results of both groups then.

Results and Discussion. Tables 1 and 2 illustrate the resulting SUS score and the percentage of solved tasks T_s for group A respectively group B. As can be seen, T_s differs between both groups. Without the explanation techniques, the participants of group A were in average able to solve 50% of the tasks, ranging from 36% to 59%. Group B was more successful, as they reached an average of 79% with a minimum of 64% and maximum of 100%. Regarding the usability of our prototype we calculated the following SUS scores: Group 1 in average 57, group 2 in average 74.

Table 1: Results for group A (plain mashup platform).

	P1	P2	P3	P4	P5	Avg.
T_s	55%	36%	55%	55%	59%	50%
SUS	60	48	58	65	53	57

Table 2: Results for group B (using our concept).

	P6	P7	P8	P9	P10	Avg.
T_s	73%	64%	100%	73%	86%	79%
SUS	73	73	78	75	70	74

We interpret both as an indication, that our approach seems useful and actually helps non-programmers to better understand CWA they are unfamiliar with. A SUS score of 74 attests a decent degree of usability, which we consider as a good result with regard to the preliminary status of our prototype. From the comments and think-aloud protocols we found potential for improvement especially of our prototype. For instance, members of group B repeatedly tried to directly perform steps when they were shown in a tutorial, which however is impossible due to the invisible `div` element. Further, there were minor usability problems, like inadequate icons. Not surprisingly, users did not pay much attention to the capability list of components they are familiar with. In addition, some users identified the limited support of parallel sequences in our implementation. Such shortcomings are subject to future work.

In summary, the results of our study are promising and show that the presented concepts can improve users' understanding of CWA functionality and awareness of IWC. However, there are threads to validity. For instance, the rather small number of users is suitable to identify usability problems (Nielsen, 2000), but larger-scale studies are needed to confirm our results. In addition there may be uninvestigated side-effects from other platform features that may especially have hindered the participants of group A. The presented concepts show limitations, e. g., our solution for hidden elements is rather simplistic. Additionally, annotations have to be provided, which may be a demanding task, also due to different styles allowed (amount and granularity of capabilities and view bindings) leading to a multitude of options to describe circumstances. However, in our opinion the proposed approaches offer a lot of benefits. They work for black box components and arbitrary CWA. The functionality of mashups can now be explored by users within the component UI in a systematic and assisted way, rather than in a trial and error style. Tutorials are generated for CWA, lowering the effort for component developers and resulting in a consistent appearance for all components. In addition, tutori-

als provide contextualized instructions and operate in component UI. As our study indicates, the proposed explanation techniques help users to be aware of and to understand the capabilities of mashups.

7 CONCLUSIONS

Development and usage of CWA are still cumbersome tasks for non-programmers, for example, when it comes to awareness of IWC and to understanding the composite nature of the functionality of unfamiliar CWA. Prevalent mashup platforms offer no or very limited assistance in this regard, resulting in trial and error strategies of non-programmers.

Our model-driven mashup platform is characterized by interwoven runtime and development time and a palette of EUD tools that aim to overcome limitations of current mashup platforms. In this paper, we describe the iterative design process of our novel explanation and awareness techniques. It is based on scenarios and insights gained from early concepts as well as corresponding prototypes and user studies. As a result, we introduce our generic approach for explaining the functionality of arbitrary CWA. Utilizing component capabilities and their view bindings, we visualize directly within the component UI what single components are capable of and which interactions users have to perform. This is the foundation for explaining the capabilities of a mashup, which are determined by component interplay through IWC. Besides an overview panel of mashup capabilities, step-wise tutorials are generated and are interactively and animatedly presented to non-programmers. In addition, we propose to re-use the visualization and interaction techniques to make users aware of IWC when it occurs at run time, and to allow them to investigate the effects of recommendations. As we discussed, there are limitations to our concepts. However, our evaluation indicates that the proposed approach can actually help non-programmers in understanding CWA.

Future work includes performing the next cycle in our user-centered design process, i. e., to incorporate feedback and lessons learned in the current prototype and evaluate the next iteration on a larger scale. In addition, we strive to use the live view for composition tasks, utilizing basic visualization concepts presented in this paper. Future research directions include a question answering system supporting non-programmers to debug a composite web application.

ACKNOWLEDGEMENTS

The work of Carsten Radeck is funded by the European Union and the Free State of Saxony within the EFRE program. Thanks to Johannes Pflugmacher and Konrad Michalik for their valuable contributions.

REFERENCES

Aghaee, S. and Pautasso, C. (2014). End-user development of mashups with naturalmash. *Journal of Visual Languages & Computing*, 25(4):414 – 432.

Cao, C. (2013). *Helping End-User Programmers Help Themselves – The Idea Garden Approach*. PhD thesis, Oregon State University.

Chudnovskyy, O., Pietschmann, S., Niederhausen, M., Chepegin, V., Griffiths, D., and Gaedke, M. (2013). Awareness and control for inter-widget communication: Challenges and solutions. In *Web Engineering*, volume 7977 of *LNCS*, pages 114–122. Springer.

Gery, G. (1995). Attributes and behaviors of performance-centered systems. *Performance Improvement Quarterly*, 8(1):47–93.

Ghiani, G., Paternò, F., Spano, L. D., and Pintori, G. (2016). An environment for end-user development of web mashups. *International Journal of Human-Computer Studies*, 87:38 – 64.

Ko, A. J. and Myers, B. A. (2004). Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158. ACM.

Lizcano, D., Lóez, G., Soriano, J., and Lloret, J. (2016). Implementation of end-user development success factors in mashup development environments. *Computer Standards & Interfaces*, 47:1 – 18.

Nielsen, J. (2000). Why you only need to test with 5 users. https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/. Last accessed 02.01.2017.

Picozzi, M. (2013). *End-User Development of Mashups: Models, Compositition Paradigms and Tools*. PhD thesis, Politecnico di Milano.

Radeck, C., Blichmann, G., and Meißner, K. (2013). Capview – functionality-aware visual mashup development for non-programmers. In *Web Engineering*, volume 7977 of *LNCS*, pages 140–155. Springer.

Radeck, C., Blichmann, G., and Meißner, K. (2016). Estimating the functionality of mashup applications for assisted, capability-centered end user development. In *Proceedings of the 12th International Conference on Web Information Systems and Technologies (WEBIST 2016)*, pages 109–120.

Tschudnowsky, A., Pietschmann, S., Niederhausen, M., Hertel, M., and Gaedke, M. (2014). From choreographed to hybrid user interface mashups: A generic transformation approach. In *Web Engineering*, volume 8541 of *LNCS*, pages 145–162. Springer.