Estimating the Functionality of Mashup Applications for Assisted, Capability-centered End User Development

Carsten Radeck, Gregor Blichmann and Klaus Meißner
Faculty of Computer Science, Technische Universität Dresden, Dresden, Germany
{*carsten.radeck, gregor.blichmann, klaus.meissner*}*@tu-dresden.de*

Keywords: Mashup, End User Development, Capability, Capability Estimation, Assistance.

Abstract: The mashup paradigm allows end users to build their own web applications consisting of several components in order to fulfill specific needs. Thereby, communicating on a non-technical level with non-programmers as end users is crucial. It is also necessary to assist them, for instance, by explaining inter-widget communication and by helping to understand a mashup's functionality. However, prevalent mashup approaches provide no or limited concepts for these aspects. In this paper, we present our proposal for estimating and formalizing the functionality of mashup compositions based on capabilities of components and their communication links. It is the foundation for our end-user-development approach comprising several assistance mechanisms, like presenting the functionality of mashups and recommended composition steps. The concepts are implemented and evaluated by means of example applications and an expert evaluation.

1 INTRODUCTION

Powered by the growth of available web resources and application programming interfaces, the mashup paradigm enables loosely coupled components to be re-used in a broad variety of application scenarios to fulfill the long tail of user needs. Recently, universal composition approaches allow for platform-independent modeling of composite web application (CWA) and uniformly describing and composing components spanning all application layers, ranging from data and logic services to user interface widgets.

The mashup paradigm and end-user development complement each other quite well. It is, however, still very cumbersome for end-users, especially as non-programmers are the target group, to develop and even use CWA. Challenging tasks in CWA development and usage, posing tough requirements to mashup platforms, are amongst others: (1) expressing goals or requirements towards the mashup in a non-technical manner, (2) understanding what single components are capable of and what functionality they provide in interplay, (3) being aware of inter-widget communication, as shown by (Chudnovskyy et al., 2013), (4) adding or removing whole "functional blocks" rather than several technical elements like components and connections, and (5) understanding what functionality recommendations will provide in context of the current task.

Our platform adheres to universal composition and strives for enabling domain experts without programming skills to build and use situation-specific mashups. Non-programmers can extend or manipulate a running application to get instant feedback on their actions and are guided by recommendations on composition patterns (Radeck et al., 2012). We semantically annotate components with the functionality they provide in terms of capabilities. Based on this, the capabilities of whole composition models are estimated. This allows our mashup environment to offer a set of assistance features which we illustrate with the help of two **scenarios**.

Scenario 1: 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 components utilized, Bob faces several understanding problems of what the mashup provides and what not. 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 in a try&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.

Radeck, C., Blichmann, G. and Meißner, K.
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) - Volume 2*, pages 109-120
ISBN: 978-989-758-186-1

It allows Bob to inspect what tasks he can solve with the application and gets aware of the components that partake. 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. Furthermore, Bob can start animations explaining necessary steps and interactions he has to perform, e. g., to see a list of routes. Bob activates a mode animating the actual data flow. Thus, Bob gets aware of data transfer between map and weather widget, which are positioned far away from each other on the screen. Additionally, Bob is assisted in identifying capabilities of a component and how these are reflected on the component UI. So, Bob understands that he can move a marker or type the location name in an input field of the map in order to select a location. After using the mashup for a while, the platform recommends extensions useful regarding the functionality it already provides. All recommendations are visualized by displaying the functionality they would offer.

Scenario 2: Knowledge-worker Alice has good domain knowledge, but no programming skills, and requires an enterprise search CWA for finding experts within her company for a certain topic. In a wizard-style dialog Alice is asked by the mashup platform to answer questions or define criteria in order to derive her goals in form of domains concepts and activities or tasks to be performed on those. Thereby, Alice gets advice on existing, similar, alternative and complementary concepts as well as tasks. During this iterative procedure, mashups that semantically match her requirements at least partially are identified based on a classification of the provided functionality and are previewed to her. Since a hierarchical functionality description is supported, mashups that offer "search experts" on highest level can be considered possible candidates although on lower levels of the capability model and especially comparing the underlying composition models there may be differences. Facilitating this, Alice can decide which optional functionalities she needs or does not need, implicitly selecting a candidate. After finishing a certain subtask in the selected mashup, she removes it from the application. Necessary changes according to the technical composition model are done transparently. Finally, she shares another sub-task with the responsible colleague Horst.

In order to implement the scenarios, to provide the mentioned features, and to tackle the challenges stated above, there are at least the following foundational **requirements**:

- The functionality of composition fragments has to be described. The notion composition fragment refers to arbitrary partial composition models like components, patterns and whole applications. In order to allow for automation at least some formalism is required. To further ease understanding, there should be a link between capabilities and actual UI-parts which serve to provide them.
- While capabilities of components can be statically defined, it is far from trivial to estimate the functionality of component interplay in an arbitrary composition fragment. Such a description should be derived semi-automatically, i. e., automatic estimation complemented with learning techniques and feedback for validation to increase quality.

While most mashup approaches support users with recommendations, assisting the understanding of the mashup at hand or presenting recommendations by the functionality they provide is neglected so far. Estimating which functionality a user wants to achieve with his current mashup is out of scope, too. In order to allow for such features, basic concepts like a proper model and derivation algorithms are currently missing.

Thus, the **contributions** of this paper are twofold. First, we introduce a model for light-weight functional semantics – *capabilities* – of composition fragments, which also allows to establish a link between semantic and UI layer. Based on this, we present, evaluate and show the practicability of an algorithm for estimating a composition fragment's capabilities.

These concepts are the basis for our capability-centered End User Development (EUD) approach. Several development and assistance tools and mechanisms rely on knowledge about a composition fragment's capabilities, provided by component capabilities and inter-component communication, and their relation to component UIs, e. g., in order to calculate and present recommendations and to explain the application functionality.

The remaining paper is structured as follows. In Section 2 we discuss related work. Next, we introduce our overall approach for assisted CWA development and usage in Section 3. Modeling foundations of our concepts are subject of Section 4. Based on this, an algorithm for estimating a composition fragment's functionality is described in Section 5. Then we evaluate our concepts in Section 6. Finally, Section 7 concludes the paper and outlines future work.

2 RELATED WORK

In the mashup domain, recent approaches feature a tightly interwoven development and usage as a commonality with capability-centered mashup EUD. Within the OMELETTE project (Chudnovskyy et al.,

2012) a live development mashup environment has been created, which features a recommender system and a user assistant for expressing goals. Patterns reflect composition knowledge and recommendations are based on patterns and are visualized by incorporated components and textual description which has to be provided manually. There is no model for functional semantics. Similarly, PEUDOM (Matera et al., 2013) allows to manipulate mashups during usage and offers a recommender system, but there is nothing similar to our capability model and algorithm. SMASHAKER (Bianchini et al., 2010) utilizes semantic component annotation and based on this a recommender system. Part of those annotations are categories which describe functionality, however, less expressive than capabilities. And deriving category annotations of whole mashups is not supported. In NaturalMash (Aghaee and Pautasso, 2014) restricted natural language is used to describe and define mashup functionality and a link between text fragments and corresponding UI parts is provided, too. We utilize a formal, semantic model to describe functionality, which we also use to derive natural language sentences. For instance, we previously developed CapView (Radeck et al., 2013), an overlay view that allows to explore and manipulate the mashup's functionality, and abstract the composition procedure to coupling capabilities. However, CapView does not support composite capabilities, for which we provide the foundation in this article. DEMISA (Tietz et al., 2013) proposes a top-down procedure to build mashups. Mashup developers first define a semantic task model, which is transformed semi-automatically into a CWA then. As already stated, our capability model is influenced by task models, but dedicated to CWA. Further, we also enable the bottom-up approach, i. e. from CWAs to capability graphs, with our algorithm.

A tagging-based approach to annotate components and discover and compose them to applications is described in (Bouillet et al., 2008). Tag taxonomies are utilized to avoid unambiguity and allow more flexible matching. However, the model is less formal and expressive. Further, the application functionality equals all annotations of a flow, while we estimate subordinate capabilities, i. e., a hierarchical structure.

(Bai et al., 2012) describe an ontology-based model of mashups and their functionality. It shares some similarity with ours, however, functionality is not semantically backed but rather free-text. Further, an algorithm to instantiate such models from existing mashups is provided. It uses lexical analysis of functionality only, and no hierarchical structuring and sub-sequencing takes place.

3 OVERALL APPROACH

Now we briefly outline our overall approach and relate the concepts we describe in this paper to it.

Adhering to universal composition, the CRUISE platform follows a model-driven composition approach to create and execute presentation-oriented CWA. Thereby, components of the data, business logic and UI layer are basically 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 Semantic Mashup Component Description Language (SMCDL) serves as a declarative language implementing the component model. It features semantic annotations to clarify the meaning of component interfaces and capabilities (Radeck et al., 2013). Based on the component model, the declarative Mashup Composition Model (MCM) describes all aspects of a CWA, like components to be integrated, views with their layout and transitions between them, and event-based communication including mediation techniques to resolve interface heterogeneity.

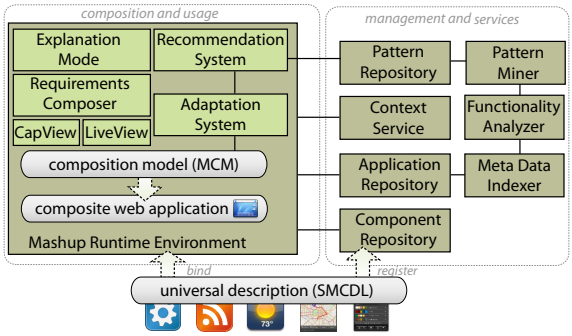

Figure 1: Architectural overview of our platform.

A fundamental characteristic of our approach is that run time and development time of a CWA are strongly interwoven. End users – in our case domain experts which know their problem and possible solutions in terms of domain tasks to perform, but fail to map such solutions on technical mashup compositions – can seemingly switch between editing and using the application. Thereby, they are not bothered with composition model concepts. Instead, communication with users takes place on capability level and necessary mappings of composition steps to composition model changes are handled transparently.

To this end, a mashup runtime environment (MRE) is equipped with a set of tools and mechanisms, see Figure 1. For instance, the recommendation system covers the whole recommendation loop, starting from identifying when recommendations may

be necessary (triggers), querying recommendations from a pattern repository, and displaying candidate patterns to the end user. The latter is done utilizing capabilities of patterns and is contextualized with respect to the CWA at hand. Optionally, the user can define functional and quality requirements in a requirements composer. Furthermore, an MRE provides different views on the current CWA: In the live view, mainly intended for usage, only component UIs are visible to the user, while there are overlay views, like CapView, that display component and composition model details and mainly serve for development purposes. In addition, an MRE offers tools explaining the functional interplay of components in a textual and visual manner, like the explanation mode.

Components are registered at the component repository using SMCDL descriptors and can be queried. Analogously, composition models of CWAs are managed on server-side in a repository separated from a concrete MRE. There are also modules attached to repositories that analyze the persisted items. For instance, composition models are classified regarding the approximate capabilities they provide by the meta data index, which uses the functionality analyzer. The same holds for patterns which are detected by pattern miners using semantic technologies exploiting component interface annotations or using statistical analysis methods. In any case the pattern functionality in terms of capabilities is derived, too. Required models, algorithms and applications of such a functional classification is in scope of this paper.

The following platform features build up on the derivation of capabilities of composition fragments, i. e., mashup applications and patterns:

- Explanation of CWA functionality (single components and especially the interplay of components);
- Awareness for inter-widget communication;
- Entering functional requirements towards composition fragments;
- Calculating composition fragment recommendations and presenting them based on the functionality they provide;
- Composition steps on whole functionality blocks rather than single technical concepts like components and channels;

We argue that utilizing capabilities is beneficial for all those use cases and ease communication with the end user. In order to enable such features, it is obviously necessary to model and estimate capabilities of composition fragments. In the following sections, we introduce our solutions to this end.

4 MODELLING ASPECTS

Based on our previous work (Radeck et al., 2013) and research on task models, we developed a model for capabilities, shown in Figure 2. The main idea and assumption is that components serve to solve tasks, and that a composition of components can fulfill more complex tasks accordingly. The proposed model is more lightweight than traditional task models, uses semantic annotations and is dedicated to CWA since it is possible to establish links to UI elements.

Capabilities describe functional and, tough restricted, behavioral semantics of a composition fragment, i. e. what it 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 concepts like classes, properties and individuals described in Web Ontology Language (OWL) ontologies back the description with formal semantics. There are optional attributes to address activity and entity more precisely: In case the entity is an OWL property e. g. `hasName`, *entity context* can define the domain, e. g. `person`; similarly, an *activity modifier* can clarify the activity without the need to blow up ontologies with individuals or sub concepts, e. g. `sort` with activity modifier `hasName` instead of declaring an individual `sortByName` in the ontology. Optionally, a capability belongs to a *domain* or a certain topic. In order to achieve a capability, it may be necessary for the user to partake and interact with the component UI or not. Thus, UI and system capabilities are distinguished.

Our model allows to build *composite capabilities* i. e. establish hierarchical structures. The relation of children of a composite capability is expressed with the help of a *connective*. Currently, we support parallel and sequential relations. In case of sequences, capabilities are chained to define the order using the relation *next* and *previous*. As an example, it is possible to describe the capability `search route` as a sequence of `select start`, `select destination`, `search route` and `display route`.

Relating capabilities with *requirements* allows to state that the provision of a capability depends on certain parameters and conditions of the user, usage or execution context. For instance, the capability `take picture` requires access to a camera within the runtime environment context.

A concept particular for UI capabilities are *view bindings*. They link the semantic layer and the user interface of the according component. Basically, a

view binding describes interaction steps via atomic, parallel or sequential operations. These point to UI elements using a selector language, e. g. CSS selectors, and define the interaction technique, like click and sweep. In case a capability has multiple view bindings, they are considered alternative, for instance, if it is possible to select a location via typing something in a text field or double clicking a map.

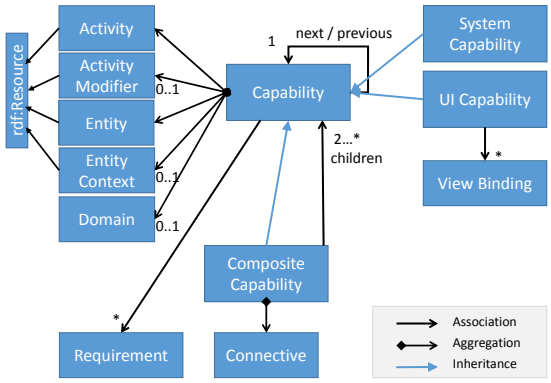

Figure 2: Schematic overview of the capability metamodel

All composition fragments, i. e. components, applications and patterns, can carry capabilities.

Capabilities of components are statically annotated by component developers in the corresponding SMCDL descriptor. Listing 1 shows an excerpt from a map component's descriptor. Most concepts of the capability metamodel are reflected by XML elements and attributes. Thereby, capabilities can be located at two positions: at level of the whole component (see Listing 1 lines 2–15), where especially UI capabilities are annotated, and at interface level (lines 22–24), where only system capabilities occur that are exclusively achieved when invoking an operation or setting a component property. It is not necessary to declare composite capabilities in order to reduce annotation effort for component developers. However, capabilities can and should be linked via `causes` and `causedBy` (see e. g. line 6) with other capabilities. This reflects causality and is a replacement that enables to derive composite capabilities afterwards. Single entries in `causes` and multiple ones connected via `and` map to a sequences, multiple `or`-ed entries are mapped to a parallel composite capability. Details on this step are provided in Section 5.2.

Events and properties reference existing capabilities via *causedBy* (and *causes* in case of properties) rather than declaring new ones, see line 19.

```
1 <component id="..." name="Map">
2  <capability id="capDispLoc" activity="
       act:Display" entity="geo:Location">
3   <viewbinding>
4    <atomicoperation element="div[id$='_map']" />
5   </viewbinding>
6   <causedBy>capInpLoc or cap02 or capInpLocDet</
       causedBy>
7  </capability>
8  <capability id="cap02" activity="act:Select"
       entity="geo:Location">
9   <viewbinding>
10   <atomicoperation element="input[id$='
       mapTextField']" interactionTech="
       i:TypeOperation" />
11  </viewbinding>
12  <viewbinding>
13   <atomicoperation element="div[id$='
       gMapCurrentLocationIcon']"
       interactionTech="i:DragNDrop" />
14  </viewbinding>
15 </capability> ...
16 <interface>
17  <event name="locationSelected">
18   <parameter name="loc" type="geo:Location" />
19   <causedBy>cap02</causedBy>
20  </event>
21  <operation name="showLocation">
22   <capability entity="geo:Location" activity="
       act:Input" id="capInpLoc">
23    <causes>capDispLoc</causes>
24   </capability>
25   <parameter name="loc" type="geo:Location" />
26  </operation> ...
```

Listing 1: Excerpt of a map's SMCDL descriptor.

While the implementation of our capability metamodel in SMCDL has its specificities, CWA and patterns are directly equipped with arbitrarily structured capabilities. Since components are the atomic building blocks, capabilities of patterns and mashups result of the statically declared capabilities of components and especially how these are connected via communication channels. Thus, capabilities of patterns and CWA are not predefined and consequently have to be derived for each composition fragment. Our solution for that is presented next.

5 CLASSIFICATION ALGORITHM

In this section we go into details on our algorithm for estimating the capabilities of an arbitrary, valid composition fragment.

5.1 Foundation

As a prerequisite we briefly describe some basic concepts and foundations of the algorithm in this section.

A *capability graph* is a set of *capability nodes* and directed edges called *capability links*, see Figure 3. It may be cyclic and represents the capabilities of a composition fragment since for each communication channel and *causes* or *causedBy* relation a capability link is created between nodes encapsulating the coupled capabilities. Each capability link comprises a start and a target capability node and stores selected composition model information, e. g. mediation techniques applied on a channel.

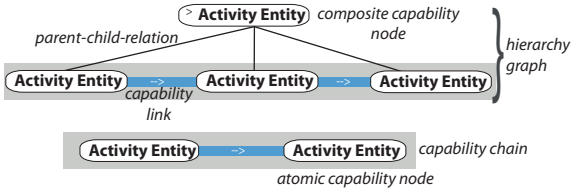

Figure 3: Schematic example of a capability graph.

Besides the dataflow or causality-oriented graph built from atomic capabilities and links between them, there is an overlay structure, the *hierarchy graph*. It is created from deriving composite capabilities with the help of our algorithm.

In case a capability graph consists of multiple isolated subgraphs which are coherent in themselves, these are called *capability chains*.

As mentioned earlier, entities refer to OWL concepts. The latter can be related in different ways using OWL properties, like `subClassOf` and defining range and domain. When deriving a composite capability it is necessary to identify an entity as expressive as possible, which we call *dominant entity*. Due to space limitation we only can give a brief overview on how we determine a dominant entity. It is calculated by analyzing the semantic entity annotations of all direct child capabilities of the composite capability at stake, denoted as set E. Thereby, we utilize inheritance (`subClassOf`, `subPropertyOf`) to identify coarse grained concepts subsuming other entities. Further, we assume that a class C_1 aggregates or subsumes C_2 if there are OWL properties with domain C_1 and range C_2. In this step we skip symmetric and inverse properties. A concept is dominant if it subsumes all $e \in E$. Such a concept does not have to be element of E. Lets consider a simple example: The entities `location` and `route` are given and the ontology states that each `route` has OWL ObjectProperties `hasStart` and `hasDestination`, both with range `location`. Then `route` is the dominant entity.

5.2 Detailed Procedure

Basic ideas and assumptions of our algorithm can be summarized as follows. The core functionality of a CWA is achieved by components and their interplay based on capability links. Through transitive connections more complex functional relations are established within a CWA. Facilitating semantic information of capabilities, heuristics and learned data, composite capabilities can be estimated and describe functionality of whole composition fragments.

Figure 4 shows the essential workflow of our algorithm, which is explained in detail in the following.

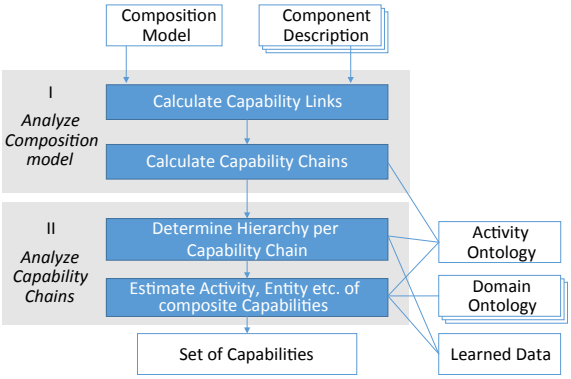

Figure 4: Inputs, main steps and outputs of the algorithm.

Phase I. Given a composition model representing the composition fragment, a main goal of the first phase is to calculate capability links by analyzing MCM as well as SMCDLs of included components.

In a preparation step, information about components and their annotations are gathered, for instance, references in element *causes* are resolved to actual capabilities and for each component property, a capability with activity `set` and an entity according to the property type is created.

Then all communication channels in the composition model are considered. We assume that a channel has exactly one publisher and one subscriber interface element, and more complex communication patterns are build on top of such "atomic" channels. For all combinations of relevant capabilities of publisher and subscriber a capability link is created. Subsequently, those capability links are completed by following the intra-component relations *causes* and *causedBy* and creating additional capability links for each of them.

Optionally, if requested by the client or if there are no capability links so far, the capability graph is extended by intra-component capability nodes and links. Thereby, only capabilities that are not yet part of the capability graph and which can be performed by users, i. e. no capabilities at operation level and none exclusively caused by operation calls, are considered. As described above, capability links are established based on the relations *causes* and *causedBy*.

Capability chains, i. e. functionality blocks of a

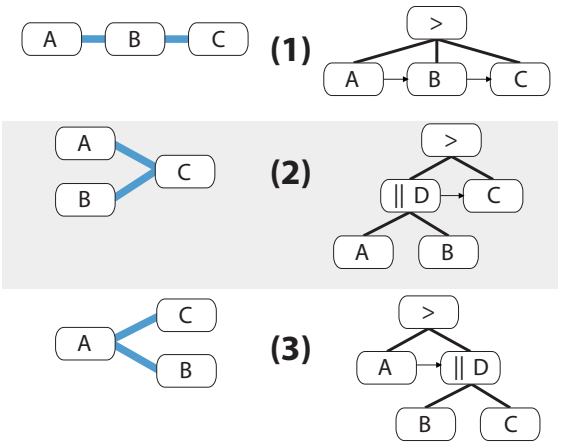

Figure 5: Supported graph patterns for hierarchically structuring composite capabilities.

CWA, are identified then. Beginning at capability nodes with outgoing links only, capability links are followed until either another chain or a capability node without outgoing links is reached. In the first case, both chains are merged.

Phase II. In this phase the algorithm strives for determining a hierarchy graph per capability chain. To this end, certain graph structures, inspired by workflow patterns (van der Aalst et al., 2003), are identified in a capability chain. Each structure has a well defined effect on the resulting hierarchy graph leading to the creation of composite nodes, see Figure 5.

(1) Sequence If there are two or more capability nodes connected in a line pattern and all nodes have max. 1 in and max. 1 outgoing link, they are assigned as children to a composite capability node with sequence connective.

(2) Synchronization Converge several capability links in a capability node, the latter is a synchronization point. In the resulting overlay hierarchy, all sources (A and B in Figure 5) are grouped to a parallel composite node (D), which is source in a sequence with the target node (C).

(3) Parallel split In this case, a capability node has multiple outgoing capability links. The target nodes (B and C in Figure 5) are assigned to a parallel composite node (D), which is in sequence with the source node (A), this time as target node.

These rules are applied to create composite capabilities forming the hierarchy graph whereby (2) and (3) are higher prioritized than (1).

As described in Section 3, the MCM allows to define different views on a CWA, affecting the visibility of UI components and, from an end-user-perspective, consequently the accessibility of corresponding UI capabilities. Thus, rules (2) and (3) are adapted: In case the underlying components of D's child capability nodes do not occur in the same view, the connective of D is set to sequential, otherwise parallel. The order in a sequence corresponds to the view order.

Next, sub sequencing takes place. In this central step, child nodes of sequence nodes are analyzed regarding their activity concept in order to detect potential subdivisions. According to a system-theoretical paradigm, we assume that functionality essentially consists of inputting something, transforming it and outputting a result. Based on this, we define the following rules determining potential borders between sub functionalities in a sequence of capability nodes. Further, we classify activities or actions according to the superclasses `input`, `transform`, `output` in our activity ontology (Tietz et al., 2013). Let act_i denote the superclass of the activity of the i-th capability in a sequence. Then a potential border is after capability i

- if $act_i = \texttt{output}$ and $act_{i+1} \neq \texttt{output}$ or
- if $act_i = \texttt{transform}$ and $act_{i+1} = \texttt{input}$

In case all resulting sequences would have more than one child, the hierarchy graph is adapted accordingly. Please refer to Figure 6 for an example, where the sequence in the upper part is analyzed accordingly, and the resulting structure is shown below. Potential borders are depicted in orange.

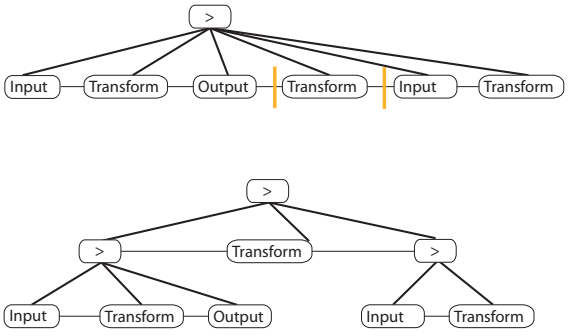

Figure 6: Exemplified sub sequencing approach.

The intermediate result at this point is a hierarchy graph per capability chain whose composite capability nodes are not semantically annotated yet. All hierarchy graphs are assigned as children to the root node. Thus, semantic annotations are estimated next. To this end, composite capability nodes are arranged in layers according to the distance from the root. Then, the procedure begins on the lowest layer and performs for each composite capability node c_{comp} a number of steps. All child nodes $cap_{children}$ are analyzed to try to estimate the most likely capability for c_{comp}. External knowledge for this is provided by ontologies used

to annotate activity and entity concepts, as well as learned data from previous runs in shape of confirmed capability graphs. First, a look up for known solutions in learned data is performed by graph matching. If there exists an identical case, c_{comp} is set accordingly. Otherwise the estimation proceeds and calculates for every entity concept, which is annotated in $cap_{children}$, a rating that is influenced by the following factors.

- Activity rating r_a is defined as the maximum of all weights w_a for activity concepts an entity occurs with. Given the superclass of an activity concept we propose the following order $w_{transform} > w_{output} > w_{input}$. If there are multiple activities with the same w_a, learned knowledge is incorporated, by looking for similar constellations of capabilities and increasing w_a of the activity chosen in such cases.
- Structural rating r_s states the relevance of an entity with respect to its position and role in capability and hierarchy graph. This comprises factors like:
  - Position within a sequence, whereby entities located at the end are rated higher.
  - Entities of composite nodes are rated higher.
  - Entities of capability nodes partaking in capability links derived from communication channels, are considered more important.
- Frequency rating r_f denotes the relative frequency of an entity with the set under investigation.
- Semantic rating r_{sem} expresses, if an entity is dominant regarding the set under investigation.

The overall rating for an entity is defined as

$$rating_{entity} = r_a + r_s + r_f + r_{sem}$$

If there are multiple entities with the same rating, we determine if one of them is the dominant entity with respect to that set, and increase the rating. Furthermore, we incorporate learned knowledge by looking for a capability node where the children are equipped with the same annotations. If a similar case exists, we set that entity's rating to the highest value since we consider the data as validated.

Finally a composite capability cap_{result} is created with the best rated entity, the corresponding activity and its activity modifier. In addition, the *domain* of cap_{result} is derived. We use the ontology defining the entity concept of cap_{result} and expect it to provide `rdfs:label` annotations, which serve as a brief domain descriptor. In some cases we also set cap_{result}'s entity context: We check if the child capability node carrying the best rated entity is connected via a capability link to the previous node cap_p and if this link originates from a communication channel which uses projection for mediating source and target interface, e. g., `Event` → `hasName`. Then, the parent nodes entity context is set to the entity of cap_p. This enables to distinguish slightly different capabilities, like `search article` by name of an event or a location.

Additionally, a *confidence* value is calculated and attached to cap_{result}. It is proportional to the distance of the highest and second highest $rating_{entity}$. In order to increase the plausibility of the overall result, the hierarchy graphs root node is removed if its *confidence* value is below a threshold c_{min}, leading to several capability nodes as a result.

6 EVALUATION

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

6.1 Implementation

We implemented the algorithm and a set of clients as part of the CRUISE platform. In the following, the conceptual architecture and some implementation details are presented utilizing Figure 7. The algorithm is situated on server-side and encapsulated in a dedicated package, to which the `Functionality Analyzer` is the central access point to. Therefore, it provides several interfaces, e. g., as SOAP web service. The `Functionality Analyzer` orchestrates several other modules and performs pre-processing steps like format transformations in order to answer incoming requests (A).

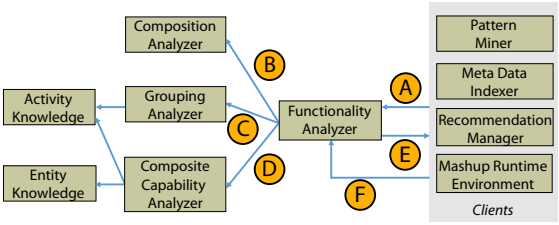

Figure 7: Architectural overview of our prototype.

Then (B), the `Composition Analyzer` is responsible for analyzing the given composition fragment in terms of a composition model and the SMCDL component descriptors. The resulting capability links are handed over to the `Grouping Analyzer` in step (C) which derives capability chains and the capability hierarchy. The latter is enriched with semantic annotations by the `Composite Capability Analyzer` then (D). It utilizes semantic knowledge from the modules `Activity Knowledge` and `Entity Knowledge`, which manage ontologies and provide access to reasoning facilities and to answer queries.

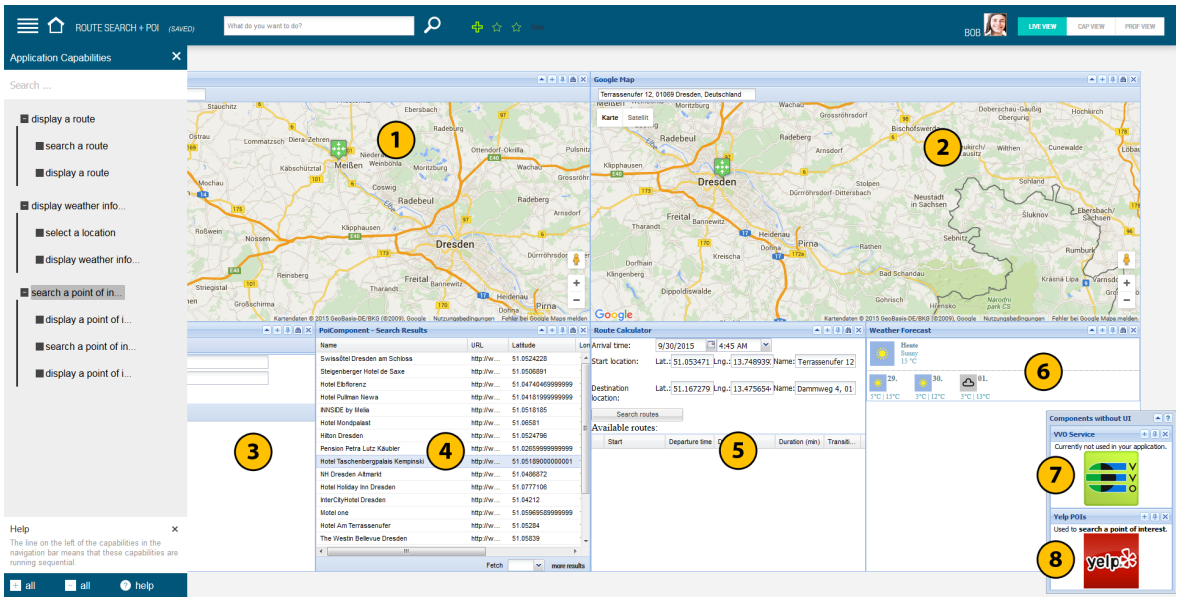

Figure 8: Screenshot of a mashup in our test bed.

For these tasks, our prototype employs the framework Apache Jena in both modules. Finally, results are delivered to the client after some post-processing (E). User feedback on algorithm results is transfered from an MRE to the server side and stored as confirmed solutions, see (F).

There are several clients to be considered. The Recommendation Manager requires the algorithm for calculating recommendations paying attention to capabilities required by the user or already part of a mashup. Further, within our repositories for applications and components, the Meta Data Indexer and Pattern Miners use the `Functionality Analzyer` to derive the capabilities of persisted mashups or newly identified composition patterns. An MRE provides several tools for understanding and developing mashups. On the left-hand side of Figure 8 there is for instance a widget visualizing capabilities of the current application. Results of the algorithm are also used to present capabilities by generating short natural language sentences (Radeck et al., 2013) when giving recommendations, e. g., in the CapView and a recommendation menu, and when composing functional requirements in a wizard. Additionally, we are currently working on mechanisms interactively explaining application capabilities to users.

6.2 Experiments

In order to validate the prototypical implementation of the proposed algorithm, i. e., to test if it works as expected, we defined test cases with increasing complexity in terms of number of components N_{co}, channels N_{ch} and capability links N_l. Our test bed consists of the following types of composition fragments (CF):

- CF comprising a single component.
- CF with two non-connected components.
- CF with two components that are connected via one channel, e. g. map and weather widget. We also varied the connection on composition model level to test if the result is semantically the same, for instance, both components can be coupled via event and operation or via properties.
- CF from different application domains with 2-7 components and 1-7 channels, for instance, travel planning, POI search, news scenario, appointment scheduling, hotel search. Again we tested structural variations if applicable.
- CF consisting of separate capability chains, e. g., the mashup shown in Figure 8 allowing to `search routes` (components (1), (2), (5), (7)), to `display weather information` at the destination ((2) and (6)) and to `search POIs` ((3), (4), (8))

Based on the test bed described above, we were also interested in the performance of our research prototype to show the practicability and applicability. To this end, we measured the average calculation time needed by our algorithm to process increasingly complex composition fragments. For each data set we performed 100 runs in a single thread in order to lower the impact of outliers. The test system features an Intel i7-4900 with 2.8 GHz and 32 GB RAM. Table 1 shows the results in case of local calls.

Table 1: Benchmark results.

Test case	N_{co}	N_{ch}	N_l	$T_\varnothing$
News scenario	2	1	1	181 ms
Appointment app	4	2	2	280 ms
Travel planning	8	7	12	458 ms

The results indicate that calculation time increases proportional to the structural complexity of the inputted composition fragment. Even for rather complex mashups, the calculation time is far below one second, which we consider good performance taking into account the prototypical character of our implementation. Further, none of our use cases poses hard time constraints with particularly low response time.

6.3 Expert Evaluation

Methodology. In order to validate both our capability model and the estimation algorithm, we conducted an expert evaluation. Seven computer scientists or master students, which work in and have contributed to the area of mashups or service-oriented architectures, participated. All participants have profound knowledge about using and building component-based applications. We sketched nine mashups of our test bed with increasing complexity on paper, like the CWA depicted in Figure 8. Thereby, components, their capabilities and capability links were schematically represented. If required, a short introduction to our capability model was given. Further, we showed live mashups in our platform if necessary to avoid misinterpretations. Then, the experts were asked to answer the following questions for one CWA at a time by sketching capability graphs on paper. Explanations and thoughts were noted by the interviewer.

Q1 *How would you describe the overall functionality the CWA provides in terms of capabilities?*

Q2 *Would you decompose those capabilities? If yes, how?*

Our main goal was to show that the proposed capability model is well suited to describe functionality and that our algorithm is able to derive adequate capability model instances for composition fragments covering a broad variety of cases. To this end, we then compared the capability models our experts would assign with the output of our algorithm.

Results and Discussion. Experts were in nearly all cases able to express what they wanted using our capability model. Often they qualified activities or entities, e. g., "search article for location", which is mappable to *activity modifier* and *entity context*. Repeatedly the following suggestions were made. It is possible to use one capability, e. g. `select location`, as source for multiple capability links or to provide several sources in multiple components. Some experts remarked that in the latter case, a distinction of those capabilities should be possible, since there are several instances e. g. of location. We agree, and required information are only implicitly part of our model, given by ID and corresponding components of capabilities. Thus, it is mainly a matter of properly analyzing and presenting the model in a front-end. Additionally, few experts suggested to allow optional capabilities.

Regarding *Q1* the results are promising. We calculated a matching degree for activities and entities. We considered semantically similar concepts as 50% match, e.g. `show` and `display`. In case, experts derived additional hierarchy levels, we matched the layer comparable to the algorithmic result. An entity match of 96.83 % and an activity match of 80.16 % lead to an overall accuracy of 88.49 % in our test.

There was no consensus about if `transform` or `output` activities are more important. However, in all cases at least 5 of 7 experts decided for the first, which confirms our prioritization. In case there are multiple capabilities with the same type of activity in sequence, e. g., `search song` → `search article`, 6 of 7 experts prioritized entity `article` when deriving a parent capability. That is consistent with our heuristics, which pay attention to flow direction.

We did not incorporate learned data in order to validate the base concepts and heuristics of our algorithm. Due to this, in more complex scenarios, our algorithm was not able to derive a meaningful root capability like experts did. For instance, for the CWA in Figure 8 our prototype calculates three composite capabilities. Though this is in line with what experts derived, 6 of 7 experts additionally defined `plan trip` or similar as additional parent capability. In the test case "appointment app", our prototype derives `edit` as activity of the root capability based on annotated concepts, while experts often used similar terms, like `manage` or `plan`, based on assumptions and additional knowledge. Deriving such capabilities is far from trivial, especially in a generic automatic way, in some cases even for experts. Combining community and semantic knowledge seems the most promising solution. However, semantics-based heuristics enable to avoid cold start problems in case there are no feedback, training or learned data available.

Regarding *Q2* results showed, that the capability graphs experts drew were in principle similar to our concept. However, it becomes evident that experts tend to subsume capabilities and leave them out. For instance, some experts stated, that it is clear to them that to `search` something implies to `input` search cri-

teria first. Due to the multitude of use cases our algorithm keeps such capabilities. It is up to the concrete client to apply filters if necessary. In the most complex scenario, experts struggled to structure the hierarchy up to the leaves, while the upper hierarchy layers were without difficulty. This underpins the necessity of an automated approach. We observed, that experts created sub-sequences similarly to our concept, although not in every case our algorithm would do. However, this mainly leads to flatter hierarchies rather than different semantics. Regarding the importance of non-linked component capabilities opinions differed. Some experts ignored them, others subsumed or grouped them in a composite capability, e. g., with activity `display`.

In general, we noticed that experts were influenced by experiences with web applications and consequently assumed functionalities when reading component names, even if there was no adequate capability presented. The same holds for incomplete annotations like missing links, which were assumed by experts. This underpins the crucial role of careful semantic component annotations. Annotating is a potentially cumbersome and error prone task. Thus, component developers should be provided with proper tooling. Also the quality of ontologies used for annotation has a strong impact on the results. Therefore, well accepted ontologies should be utilized. However, we argue that mashup platforms benefit from semantic annotations — we have indicated some use cases throughout this paper. Further, based on our proposal, annotations of composition fragments can be derived without explicit modeling of developers or users.

7 CONCLUSIONS

Mashup development and usage are still cumbersome tasks for non-programmers, for instance, when it comes to understanding the composite nature of the functionality of unfamiliar CWA. Our model-driven mashup platform strives for capability-centered EUD, which basic characteristics are interwoven runtime and development time, capabilities as description of functionality of composition fragments, and a palette of EUD tools building up on capabilities as communication means with end users. This novel approach aims to overcome limitations of current mashup platforms. Therein, knowledge about the capabilities of arbitrary (parts of) composition models is a central aspect. We use it, e. g., to present recommendations and explain application functionality. Even with semantically annotated components it is far from trivial to derive the functionality of a set of connected components. We introduce our capability metamodel which allows to describe functional semantics of composition fragments. Based on this, we propose an algorithm for estimating capabilities of a given composition fragment which analyzes annotations of components and the communication channels between them.

Future work includes backend extensions, e. g. completion of causes relations, and frontend concepts for capability-centered mashup EUD, like implementing and evaluating the explanation mode.

ACKNOWLEDGEMENTS

The work of Carsten Radeck is funded by the European Union and the Free State of Saxony within the EFRE program. Gregor Blichmann is funded by the German Federal Ministry of Economic Affairs and Energy (ref. no. 01MU13001D).

REFERENCES

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

Bai, L., Ye, D., and Wei, J. (2012). A goal decomposition approach for automatic mashup development. In van Sinderen, M., Johnson, P., Xu, X., and Doumeingts, G., editors, *Enterprise Interoperability*, volume 122 of *Lecture Notes in Business Information Processing*, pages 20–33. Springer Berlin Heidelberg.

Bianchini, D., De Antonellis, V., and Melchiori, M. (2010). A recommendation system for semantic mashup design. In *Database and Expert Systems Applications (DEXA), 2010 Workshop on*, pages 159 –163.

Bouillet, E., Feblowitz, M., Liu, Z., Ranganathan, A., and Riabov, A. (2008). A tag-based approach for the design and composition of information processing applications. *SIGPLAN Not.*, 43(10):585–602.

Chudnovskyy, O., Nestler, T., Gaedke, M., Daniel, F., Fernández-Villamor, J. I., Chepegin, V., Fornas, J. A., Wilson, S., Kögler, C., and Chang, H. (2012). End-user-oriented telco mashups: The omelette approach. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, pages 235–238, New York, NY, USA. ACM.

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 Daniel, F., Dolog, P., and Li, Q., editors, *Web Engineering*, volume 7977 of *Lecture Notes in Computer Science*, pages 114–122. Springer Berlin Heidelberg.

Matera, M., Picozzi, M., Pini, M., and Tonazzo, M. (2013). Peudom: A mashup platform for the end user devel-

opment of common information spaces. In Daniel, F., Dolog, P., and Li, Q., editors, *Web Engineering*, volume 7977 of *Lecture Notes in Computer Science*, pages 494–497. Springer Berlin Heidelberg.

Radeck, C., Blichmann, G., and Meißner, K. (2013). Capview – functionality-aware visual mashup development for non-programmers. In Daniel, F., Dolog, P., and Li, Q., editors, *Web Engineering*, volume 7977 of *Lecture Notes in Computer Science*, pages 140–155. Springer Berlin Heidelberg.

Radeck, C., Lorz, A., Blichmann, G., and Meißner, K. (2012). Hybrid Recommendation of Composition Knowledge for End User Development of Mashups. In *ICIW 2012, The Seventh International Conference on Internet and Web Applications and Services*, pages 30–33.

Tietz, V., Mroß, O., Rümpel, A., Radeck, C., and Meißner, K. (2013). A requirements model for composite and distributed web mashups. In *Proc. of the 8th Intl. Conf. on Internet and Web Applications and Services (ICIW 2013). XPS*.

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., and Barros, A. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51.