Task-Based Recommendation of Mashup Components

Vincent Tietz*, Gregor Blichmann**, Stefan Pietschmann, and Klaus Meißner

Technische Universität Dresden, Faculty of Computer Science
01062 Dresden, Germany
{vincent.tietz,gregor.blichmann,stefan.pietschmann,
klaus.meissner}@tu-dresden.de

Abstract. Presentation-oriented mashup applications are usually developed by manual selection and assembly of pre-existent components. The latter are either described on a very technical, functional level, or using informal descriptors, such as tags, which bear certain ambiguities. With regard to the increasing number and complexity of available components, their discovery and integration has become a challenge for non-programmers. Therefore, we present a novel concept for the task-based recommendation of mashup components, which comprises a more natural, task-driven description of user requirements and a corresponding semantic matching algorithm for universal mashup components. By its realization and integration with an composition platform, we could prove the feasibility and sufficiency of our approach.

Keywords: Requirements specification, task modeling, mashup component recommendation, semantics, methodology.

1 Introduction

Presentation-oriented *mashups* introduce the user interface (SWS) as a new integration layer for service-based applications and have become a prominent approach for the lightweight integration of distributed and decoupled web resources. Originally, mashups have been developed by manual, script-based integration of heterogeneous application programming interfaces (SWSs). Addressing non-programmers, *mashup tools* like *Yahoo! Pipes*, *JackBe Presto* or the *mashArt editor* [7] have emerged to support the visual composition of technology-independent web services, SWSs and SWS components.

Despite the simplicity of composition metaphors, the discovery of components remains difficult. The search is occasionally facilitated by recommendations based on keywords, interface descriptions and community feedback, e. g., in *programmableweb.com* and *IBM Mashup Center*. However, in the light of growing repositories and ambiguous tags, the identification of proper search criteria becomes an increasing challenge for unexperienced users.

* Funded by the European Social Fund (ESF), Free State Saxony (Germany) and Saxonia Systems AG (Germany, Dresden), filed under ESF-080939514.
** Funded by the ESF and Free State Saxony (Germany), filed under ESF-080951805.

A. Harth and N. Koch (Eds.): ICWE 2011 Workshops, LNCS 7059, pp. 25–36, 2011.

Instead of coping with technical details, users – typically domain experts – need to express their requirements in a more natural way. Since task analysis is considered as an intuitive way to gather user requirements for interactive systems [12], we strive for a task-based elicitation of user requirements. Thereby, user activities can be identified at design-time, avoiding low-level implementation details and using intuitive decomposition into smaller parts as well as the identification of used domain and application objects [16].

Fig. 1 shows an exemplary task description for planning a conference participation. In order to receive suggestions for routes of the public transportation services, a participant needs to input start and destination location as well as corresponding temporal constraints. In addition he or she needs information about available hotels and the weather near the conference location. Therefore, the task "Conference Participation" is decomposed into "Specify Criteria", "Calculation" and "Read Travel Information". Mashup components can be considered as self-contained entities solving these tasks. As an example, a map component could be used to specify start and destination location *(interaction task)*. Similarly, list components can display routes and hotels. In contrast, for "Search Hotels" and "Search Routes" components encapsulating web services could be employed, as these tasks are performed by the system *(system task)*.

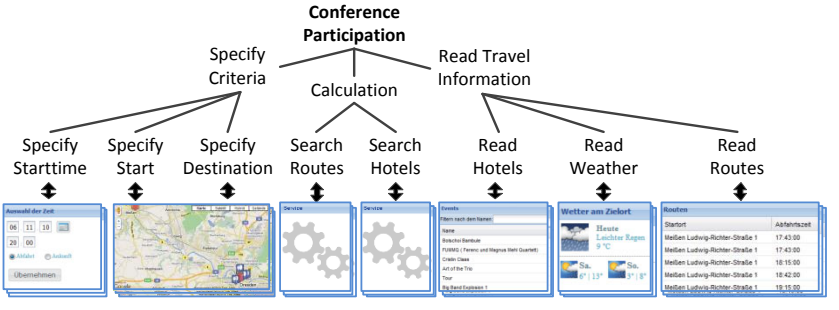

Fig. 1. Travel planning scenario

In this paper, we present a concept for the task-based recommendation of mashup components. It comprises a formal task model, whose instances serve as requirements descriptions, and a corresponding semantic matching algorithm that enables recommendation during design-time. The basis of our approach is the idea of universal mashup composition that we outline in Section 2. In Section 3, we summarize the related work for task modeling and task-based web service discovery. Building on that, we describe our ontology-based task model in Section 4 and our approach for task-based recommendation of mashup components in Section 5. Finally, we discuss the results in Section 6 and outline further work in Section 7.

2 Model-Driven Semantic Mashup Composition

As mashup components are considered as task-solving entities, a component description beyond the exposure of interface signatures is needed, representing both functional and data semantics. Therefore, our concept builds on the component model of the CRUISe project [19], as it provides a universal semantic interface to heterogeneous web resources, ranging from UI widgets to SOAP and RESTful web services. Additionally, automatic and semi-automatic encapsulation of web content and applications is gradually improved by ongoing work.

The central idea of CRUISe is the extension of the service-oriented paradigm to the presentation layer supporting *universal composition* [7]. Therein, mashups are built from uniform constituents residing on all application layers. Back-end services can be seamlessly integrated with UI components using the same principles and abstract interface descriptors. Thus, we denote a *mashup application* as a composition of uniform components encapsulating distributed web resources, i. e., services providing data, business logic, or user interface parts. With respect to this paper, all of those components represent tasks or subtasks – either involving user interaction or application logics.

CRUISe proposes a model-driven development process for building mashup applications from these components. It includes a platform-independent composition model as well as a service-oriented infrastructure for the dynamic, context-aware composition and adaptation at runtime. As our concept employs the same component models and covers the design-time phase of this process, we briefly outline the most relevant conceptual foundations for our work in the following.

2.1 Semantic Component Model

The universal composition of a mashup requires that all constituent parts adhere to a generic component model [20]. In the following, we highlight the semantic annotations, as they form the basis for our task-based recommendation.

In our conceptual space, every component – representing a service, application logic, or UI – is a black-box of independent software with an internal state. All components are described using three abstractions, namely *property*, *event*, and *operation*. The set of properties resembles the visible state and allows the configuration of components. Whenever the internal state changes, events are issued to inform the runtime system and other components. Finally, state changes, calculations and other arbitrary functionality of a component can be triggered by invoking their operations with the help of events. Events and operations may themselves contain *parameters*, realizing the data flow within the mashup.

The *Semantic Mashup Component Description Language (SWS)* allows the description of a component interface – comparable to WSDL for web services – and the semantic annotation of a component descriptor C_c at three different levels by linking certain parts with semantic models: typing of properties and parameters (*data semantics*), the definition of *functional semantics* of components A_c, operations OP_c, and events EV_c, as well as *non-functional semantics*, e. g., for pricing, licensing, and other metadata M_c.

Listing 1.1 shows a partial description of a SWS component "RouteHotelComp" (C_1), which facilitates the search for public transportation service routes (using start/destination location and start/destination time) and hotels in a certain area. As a result, the component displays routes and hotels using sortable lists. The semantic annotation of its interface is realized by linking it to semantic descriptors for "functionality" and data "type". The prefix "to" denotes concepts of the travel domain as part of a domain ontology *TO*, while "ao" is used for concepts of the action ontology *AO*, which currently represents actions through a combination of specializations out of *ao:Input*, *ao:Output* and *ao:Manipulate* via inheritance. Since the functionality of sortable lists is not represented by the component interface (because it is triggered only by user interaction), the corresponding semantic concept *ao:Sort* is annotated at component level (line 1). The data semantics of the parameter *location* in the operation *setStart* is related to the concept *to:Location* (line 3). The functional semantics of *setStart* (line 2), *setDest* (line 5), *setStartTime* (line 8), *setDestTime* (line 11) and *rSearched* (line 14) is equally *ao:SearchRoute*, because all this pieces are necessary to realize the search of routes. The attribute "trigger" (e. g., line 14) indicates the source of the event, which is either *system*, *operation*, or *interaction*.

```
1  <mcdl ... name="RouteHotelComp" functionality="ao:Sort"> ...
2  <operation name="setStart" functionality="ao:SearchRoute">
3    <parameter name="location" type="to:Location"/>
4  </operation>
5  <operation name="setDest" functionality="ao:SearchRoute ao:SearchHotel">
6    <parameter name="location" type="to:Location"/>
7  </operation>
8  <operation name="setStartTime" functionality="ao:SearchRoute">
9    <parameter name="time" type="to:StartTime"/>
10 </operation>
11 <operation name="setDestTime" functionality="ao:SearchRoute">
12   <parameter name="time" type="to:DestinationTime"/>
13 </operation>
14 <event name="rSearched" trigger="operation" functionality="ao:SearchRoute">
15   <parameter name="result" type="to:RouteList"/>
16 </event>
17 <event name="hSearched" trigger="operation" functionality="ao:SearchHotel">
18   <parameter name="result" type="to:HotelList"/>
19 </event>
20 <event name="rSelected" trigger="interaction" functionality="ao:Input">
21   <parameter name="result" type="to:Route"/>
22 </event> ... </mcdl>
```

Listing 1.1. Example mashup component C_1 for searching routes and hotels

2.2 Semantic Mashup Composition

In CRUISe, a platform-independent composition model [20] specifies the components to be integrated, incorporating information from their descriptors and defining aspects like the data and control flow, the visual layout, the adaptive behavior of the overall composition. It is interpreted by a runtime environment, which further integrates all components from a repository and executes the mashup, correspondingly. This infrastructure and integration process as well as the adaptivity concepts have been realized and validated. Yet, it is important to realize the crucial role of the design-process, i. e., the authoring, in this

context. They key challenge in rapid mashup development – especially with regard to *end-user development* – is the discovery and seamless integration of the right components in a certain context. Hence, the remaining paper addresses the question, how non-programmers may be able to find components and build such models. Before we get more into details, we discuss related efforts from the fields of task modeling and task-based service discovery.

3 Related Work

As already stated, our work envisions the recommendation of mashup components from task descriptions. Therefore, the latter must feature a formal representation with semantic references, so that actions and data of the tasks can be semantically matched with functionality and data of mashup components.

In this context, the lack in using semantic technologies and in formalism of action and domain modeling impede the use of traditional task modeling approaches (e. g., HTA [1], GOMS [5], GTA [22] and K-MAD [3]). A prominent task modeling approach is CTT [17] that is used in many model-based user interface development approaches, e. g., MARIA [18] and UsiXML [14]. However, based on the CAMELEON reference framework [4], which includes a four-stage transformation starting with a task model and ending up with the final SWS, only the manual identification of presentation items and sets is utilized.

With regard to the semantic matching of data and functional concepts, semantic web service (SWS) discovery utilizes logic-based, e. g., [6], non-logic-based, e. g., [9], or hybrid matchings [10]. While logic-based approaches use deduction to decide if concepts are equal (exact match), part of each other (subsume) and (plug-in) or distinct (fail), non-logic-based ones rely on syntactic, structural and numerical analysis, and hybrid approaches combine both. Overall, the major drawback of SWS is the use of technical service templates for discovering web services, which impedes non-expert users from expressing and satisfying their business demands [21].

Task-based recommendation usually involves the mapping of an interaction or system task to a SWS or non-SWS component, whereas also sets of tasks and components need to be considered. Corresponding task-based discovery mechanisms are supported by an extension of MARIA [11] and the SeTEF framework [21]. However, the former only supports the discovery of web services for system tasks, while the latter uses an ontology-based task description OWL-T that is transformed to SAWSDL and, therefore, is restricted to non-UI components. Furthermore, the description of tasks highly depends on knowledge about available service operations, and only one-on-one mapping between tasks and service operations are supported, which impedes the search for combinations of operations. In contrast, our approach facilitates the task-based recommendation of SWS and non-SWS mashup components during design-time by using semantic annotations in tasks and components across operations and events.

In the following, we introduce the underlying ontology-based task model and the matching algorithm employed for component recommendation.

4 Ontology-Based Task Model

Based on specific [15,17] and uniforming [8,13] task models, we derived a minimalistic task ontology – illustrated in Fig. 2 – to support user-centered analysis and description of a specific domain problem. Since mashup components are considered as black-boxes, we focus rather on the expression of required data and functional semantics than on conditions and effects in order to recommend components. Therefore, a *task* is mainly characterized by its inputs *(hasInputObject)*, outputs *(hasOutputObject)*, manipulating actions *(hasAction)* and category *(hasCategory)*. A *composite task* consists of at least two subtasks *(hasChildTask)*, whereas, subtasks are always a specialization of a parent task. *Grouping* enables the temporal relations *sequence*, *arbitrary sequence*, *choice* and *parallel* between subtasks of a composite task [2]. Both, task hierarchy and grouping facilitate task analysis and description at different abstraction levels.

In order to express *what* is intended to be done, *actions* can be assigned to composite and atomic tasks. Because, *atomic tasks* comprises only one action, we can specify exactly the data objects involved to realize the functionality represented by the action (e. g., the search of a list of hotels based nearby a certain location). Data objects *(hasInputObject* and *hasOutputObject)* are represented as ontology concepts or individuals from a domain ontology. Actions are formalized by the independent classification *AO* to represent the task's functionality (e. g., *ao:Sort* and *ao:Search* in Fig. 3).

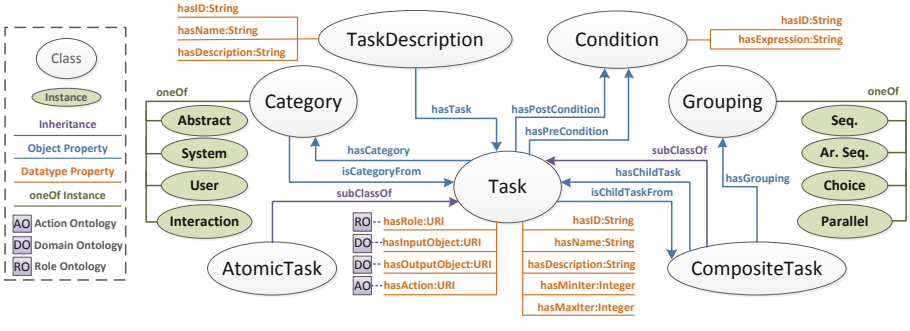

Fig. 2. Task ontology

Since mashups provide SWS and non-SWS components, we follow [17] and distinguish the *task categories*: *system*, *interaction*, *user* and *abstract*. System tasks are exclusively performed by components. For example, "Search Routes" in our scenario could be modeled as a system task. Whereas, *interaction* indicates that an interaction between humans and an SWS is required, e. g., setting a marker on a map in order to specify the start location. User tasks require no interaction with the system, e. g., fetching a folder. An abstract task groups heterogeneous subtasks (e. g., "Conference Participation") and is the default task category providing all kinds of components during recommendation.

Finally, we formalize tasks as a tuple $T_t = (M_t, A_t, c_t, IN_t, OUT_t, R_t, C_t)$ defined by metadata M_t like name and description, a set of actions $A_t \subseteq AO$, a category $c_t \in \{interaction,\ system,\ user,\ abstract\}$, a set of inputs and outputs $IN_t, OUT_t \subseteq DO$, roles R_t and conditions C_t. Considering our scenario, an example task "Search Routes" requires a start and a destination location as well as a start time to search a list of routes. Therefore, the task T_1 is defined as ({"Search Routes"}, {*ao:Search*}, *system*, {*to:StartLocation*, *to:DestinationLocation*, *to:StartTime*}, {*to:RouteList*}, ∅, ∅). Because roles and conditions are required neither in the scenario nor to explain the recommendation algorithm, these sets are empty. However, a role could be *Administrator*, represented by a concept of a role ontology. The restriction that *to:StartLocation* needs to be a European city is a possible pre-condition.

Using semantic technologies this ontology-based task model enables the task-based recommendation of previously introduced mashup components which is presented in the following section.

5 Task-Based Recommendation

Our approach aims to fill the gap between a user-centered requirements specification and semantic mashup component discovery. In order to match different semantic concepts annotated in task and component descriptions, we propose the two subsumption-based functions *CoreMatch* and *SetMatch*. Further, we consider the mapping of inputs and outputs between tasks and components as well as their functional semantics. Building on that, we present our task-based recommendation algorithm.

5.1 Calculation of Subsumption-Based Similarity

In order to compare and rate different semantic annotations used in tasks and components, we propose a subsumption-based matching, referring to logic-based web service matchings [6].

The core matching degrees $CoreMatch(r, a)$ between the request r and the advertisement a are defined by *exact* ($5 \Leftrightarrow r \equiv a$), *plug-in* ($\frac{4}{s} \Leftrightarrow r \sqsubseteq a$), *subsume* ($3 \Leftrightarrow a \sqsubseteq r$) and *fail* ($0 \Leftrightarrow else$), where s is the number of sibling nodes s (with $s \geq 1$) at the same distance. Fig. 3 shows an example for requesting *ao:SearchRoute*. Therein, the advertisement *ao:SearchBusRoute* subsumes *ao:SearchRoute* and, therefore, the result is 3. The distance $dist(r, a)$ is defined by the number of inheritances related from the request r to the advertisement a. In the case of a plug-in, we divide the result by s in order to consider partial concepts, e. g., the advertisement of *ao:Sort* represents only one part of the requested functionality and can be potentially combined with other sub-functionalities. For example, the functionality represented by *ao:Search* is a specialization of *ao:Sort* and *ao:Calculate*. Therefore, *ao:Sort* and *ao:Calculate* are siblings at the distance 2 and $CoreMatch(ao{:}SearchRoute, ao{:}Sort) = \frac{4}{2} = 2$.

Further, we define the function $SetMatch(R, A)$ that calculates the rank of R and A as sets of requested and advertised ontology concepts. First, this function

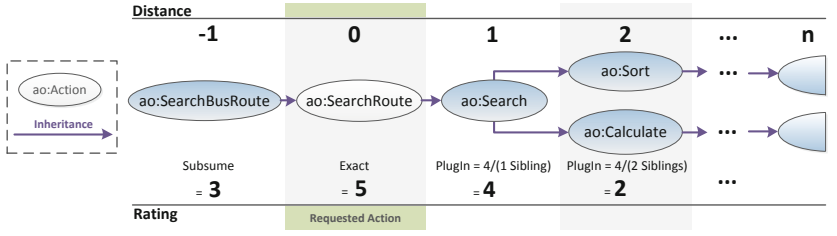

Fig. 3. Rating and distances related to a request of *ao:SearchRoute*

groups all best core matches of each member of R and A and, finally, returns the average of all best matching degrees. For example, $SetMatch(\{ao{:}SearchRoute, ao{:}SearchHotel\},\{to{:}Search\}) = 2.0$, because *ao:SearchRoute* builds the best assignment with *ao:Search* as subsumption (4) and *ao:SearchHotel* could not be further matched because A has no concepts left. The final result is 2.0, the average of 4.0 and 0.0. In the following, we use $SetMatch(R, A)$ for rating different aspects (e. g. data and functional semantics) of tasks and components.

5.2 Mapping Data Semantics of Components with Tasks

In general, component descriptions include events representing output and operations representing input (cf. Section 2). In order to match tasks and components, a task-aligned interpretation of these descriptions is required. From the perspective of user interaction, a SWS component signalizes the input of data by triggering an event, e. g., as result of a user selecting a location on a map component. On the other hand, the output of data, e. g., its visualization on the map, is realized by an operation. Therefore, inputs of an interaction task must be mapped to the data semantics of events $IN_t \rightarrow d(EV_c)$, while its outputs must be mapped to operations of a component $OUT_t \rightarrow d(OP_c)$, accordingly.

While this applies to interaction tasks, it does not for system tasks. In this case, inputs are mapped naturally to operations ($IN_t \rightarrow d(OP_c)$), e. g., to invoke a search for hotels based on a location as parameter, and the outputs are mapped to events ($OUT_t \rightarrow d(EV_c)$). If the task category is *abstract*, it cannot be decided how to map inputs and outputs. Therefore, the mapping and rating needs to be carried out for both, whereas the maximum of both ratings is returned.

5.3 Task-Based Recommendation

In the following, we describe the algorithm $Match(T, C)$ that returns an ordered list of rated recommendations out of a set of components C for a task instance T that is compliant with the proposed task ontology. The algorithm exploits the previously mentioned matching principles in order to rate CRUISe components for each task.

Reducing complexity. In the first step, the complexity of the task model is reduced in order to optimize the matching performance. Therein, user tasks are omitted, because no component is required. Next, parent tasks are removed, because subtasks are an equal or more detailed representation of their parents regarding actions, inputs, and outputs. Finally, the amount of component candidates is reduced for each task based on its category. If the category is *interaction*, service and logic components are excluded, because they offer no SWS.

Interim ratings and data structure. In the second step, we compare each task T_t with each component candidate C_c. The final rating rtc_{fin} of a component, is the maximum of the two interim ratings rtc_1 and rtc_2. rtc_1 reflects the matching of the overall semantic annotation of the component and rtc_2 considers the semantics of operations and events (including data and functionality) of the component. In order to calculate the interim rating rtc_2, a data structure similar to Table 1 is created that represents a task-like interpretation of each component description. With the help of the table, the ratings of the functional semantics m^i_{act} and the data semantics m^i_{in} and m^i_{out} of the component are determined. In the following, we describe in detail, how the table is filled and how both interim ratings are calculated. For this, we use the previously introduced task T_1 (cf. Section 4) and component C_1 (cf. Section 2).

Table 1. Intermediate results of the matching algorithm $Match(T_1, C_1)$

i	d_{ti}	act_i	m^i_{act}	$D_{in}(act_i)$	m^i_{in}	$D_{out}(act_i)$	m^i_{out}	r_i
1	-1	ao:SearchRoute	3.0	to:Location to:Location to:StartTime to:DestinationTime	4.33	to:RouteList	5.0	**3.83**
2	-1	ao:SearchHotel	3.0	to:Location	1.67	to:HotelList	0.0	**1.92**
3	∞	ao:Input	0.0	$\emptyset$	0.0	to:Route	0.0	**0.0**
							rtc_2	**3.83**

Functional semantics of components. The first interim rating rtc_1 reflects the matching of the overall semantic annotation of the component A_c requesting the actions of the task A_t. Using our example, this is $rtc_1 = SetMatch(A_t, A_c) = SetMatch(\{ao{:}Search\}, \{ao{:}Sort\}) = 2.0$ (cf. Fig. 3). Because we define that functional and data semantic are equally weighted and no data semantic is annotated at this level, rtc_1 is divided by 2 which results in 1.0.

Functional semantics of operations and events. In order to calculate the ratings of the functional semantics m^i_{act}, each annotated action act_i of all operations and events is added to a distinct action list. If an interaction task is requested, only events with the trigger "interaction" are considered (e. g., line 20 in Listing 1.1). Then, the rating for the functional semantics m^i_{act} is calculated by $SetMatch(A_t, act_i)$. This means that all actions of the task A_t are requested for each functionality act_i of the component. In our example, $m^1_{act} = SetMatch(\{ao{:}Search\}, \{ao{:}SearchRoute\}) = 3.0$.

Data semantics. The rating of the data semantics m^i_{in} and m^i_{out} is based on both columns $D_{in}(act_i)$ and $D_{out}(act_i)$ as advertisements and the input IN_t and output OUT_t of the task as requests. According to our example, the task category equals *system*, therefore, $D_{in}(act_i)$ gets filled with the data semantics of all operations annotated by the functionality act_i and $D_{out}(act_i)$ gets filled with the data semantics of all events annotated the functionality act_i (cf. Section 5.2). In general, m^i_{in} is calculated by $SetMatch(IN_t, D_{in}(act_i))$ and m^i_{out} by $SetMatch(OUT_t, D_{out}(act_i))$.

For example, $m^1_{in} = SetMatch(\{to{:}StartLocation,\ to{:}DestinationLocation,\ to{:}StartTime\},\ D_{in}(act_1)) = 4.33$. Further, $m^1_{out} = SetMatch\ (\{to{:}RouteList\},\ D_{out}(act_1)) = 5.0$. Because we weight functional and data semantics equally, the rating for each row (r_i) is the average of m^i_{act} and the average of m^i_{in} and m^i_{out}. For example, $r_1 = \frac{1}{2}(3.0 + \frac{1}{2}(4.33 + 5.0)) = 3.83$.

Detecting sub-functionalities. As mentioned in Section 5.1, it is possible to detect and merge associated sub-functionalities like *ao:Calculate* and *ao:Sort*. For this, we use the distance d_{ti} between the requested A_t and advertised act_i and group all *subsumes* (where $0 < d < \infty$) having the same distance.

Then, we sum their functional semantics rating and build the average of their data semantic rating, to end up in one new row including all sub-functionalities. This allows us to handle functionalities across multiple operations and events. In our example, no grouping is necessary, because we get two subsuming concepts (*ao:SearchRoute*, *ao:SearchHotel*) and two fails (*ao:Input*, *ao:Output*).

Final rating result. As previously mentioned, rtc_1 considers the overall functionality of the component and rtc_2 represents the best match for functional and data semantics of all operations and events. Therefore, rtc_2 is the maximum of all r_i. Finally, the highest value out of rtc_1 and rtc_2 is the final result r_{fin} of the matching algorithm for a task and a component. In our example r_{fin} equals 3.83, because this is the maximum of $rtc_1 = 1.0$ and $rtc_2 = 3.83$. The matching is done for all tasks $\in T$ and all components $\in C$. In the end, the result tuple $RT = (T_t, \{(C_c, rtc_{fin})\})$ includes for every task T_t a set of component proposals, represented by their id and rating.

6 Implementation and Discussion

We have successfully implemented the proposed algorithm as a part of a service-oriented and Java-based component repository of CRUISe. The repository registers, manages, matches and ranks components and offers a web service interface. The matching can be based on a SMCDL template or, as used in this case, on an instance of the task ontology. The repository and the matching algorithm use the semantic web framework Jena (`http://jena.sourceforge.net/`) in order to access OWL knowledge bases using plain Java.

We have tested the algorithm with a task model representing our scenario and a set of components such as generic and specific input and output components (e. g., for locations, time and routing) getting expected ranks. However, in order

to get reliable results we plan to evaluate the algorithm within a broad user study utilizing more scenarios and components.

Since we address the design-time, performance is negligible to a certain degree. However, the current response time is about 1s for one task and 50 components and tends to be more than proportional with the increasing number of components and tasks. Therefore, we plan to implement caching and other optimizations.

Regarding the use of ontologies, we assume that component developers and task modelers have a common understanding of how functionalities and data are semantically represented. Currently, we use self-developed travel and action ontologies on the basis of the introduced scenario. In principle, any knowledge base can be used and matching as well as aggregating ontology concepts can be applied in future.

7 Conclusion and Further Work

The contribution of this work is twofold. First, we presented an ontology-based task model that allows formal and lightweight modeling of user's requirements for composite mashup applications on the basis of existing knowledge bases. This addresses our key requirements regarding the formalization and abstraction of any specific service operation or user interface component. Second, we provide a matching algorithm based on semantically annotated mashup components in order to support discovery for task-based requirements. The key feature is the proposal and rating of components realizing specific as well as partly supported functionalities across services and components during the design-time.

Regarding the discussion in Section 6, further work addresses the optimization and evaluation of the recommendation algorithm particularly by utilizing a user study. Further, we explore the opportunity of semi-automatic composition utilizing the proposed task model and recommendation of components. Currently, we work on the design and the implementation of an authoring tool in order to allow task modeling for non-programmers and to determine concepts for ontology-based modeling. Finally, this work is an important step towards a task-based development approach for composite mashup applications.

References

1. Annett, J., Duncan, K.: Task analysis and training design. Hull Univ. (England). Dept. of Psychology (1967)
2. Betermieux, S., Bomsdorf, B.: Finalizing Dialog Models at Runtime. In: Baresi, L., Fraternali, P., Houben, G.-J. (eds.) ICWE 2007. LNCS, vol. 4607, pp. 137–151. Springer, Heidelberg (2007)
3. Caffiau, S., Scapin, D.L., Girard, P., Baron, M., Jambon, F.: Increasing the expressive power of task analysis: Systematic comparison and empirical assessment of tool-supported task models. Interacting with Computers 22(6), 569–593 (2010)
4. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A Unifying Reference Framework for multi-target user interfaces. Interacting with Computers 15, 289–308 (2003)
5. Card, S., Moran, T., Newell, A.: The Psychology of Human-Computer Interaction. Lawrence Erlbaum, Hillsdale (1983)

6. Chabeb, Y., Tata, S., Ozanne, A.: YASA-M: A Semantic Web Service Matchmaker. In: 24th IEEE International Conference on Advanced Information Networking and Applications (AINA 2010), pp. 966–973 (2010)
7. Daniel, F., Casati, F., Benatallah, B., Shan, M.-C.: Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In: Laender, A.H.F., Castano, S., Dayal, U., Casati, F., de Oliveira, J.P.M. (eds.) ER 2009. LNCS, vol. 5829, pp. 428–443. Springer, Heidelberg (2009)
8. Goschnick, S., Sonenberg, L., Balbo, S.: A Composite Task Meta-Model as a Reference Model. In: Forbrig, P., Paternó, F., Mark Pejtersen, A. (eds.) HCIS 2010. IFIP Advances in Information and Communication Technology, vol. 332, pp. 26–38. Springer, Heidelberg (2010)
9. Klein, M., König-Ries, B.: Coupled Signature and Specification Matching for Automatic Service Binding. In: Zhang, L.-J., Jeckle, M. (eds.) ECOWS 2004. LNCS, vol. 3250, pp. 183–197. Springer, Heidelberg (2004)
10. Klusch, M.: Semantic web service coordination. In: CASCOM: Intelligent Service Coordination in the Semantic Web. Whitestein Series in Software Agent Tech. and Autonomic Computing, Birkhäuser, pp. 59–104 (2008)
11. Kritikos, K., Paternò, F.: Service discovery supported by task models. In: 2nd ACM SIGCHI Symp. on Engineering Interactive Computing Systems, EICS 2010 (2010)
12. Limbourg, Q., Vanderdonckt, J.: Comparing task models for user interface design. In: The Handbook of Task Analysis for Human-Computer Interaction, pp. 135–154. Lawrence Erlbaum Associates (2003)
13. Limbourg, Q., Pribeanu, C., Vanderdonckt, J.: Towards Uniformed Task Models in a Model-Based Approach. In: Johnson, C. (ed.) DSV-IS 2001. LNCS, vol. 2220, pp. 164–182. Springer, Heidelberg (2001)
14. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López-Jaquero, V.: USIXML: A Language Supporting Multi-Path Development of User Interfaces. In: Feige, U., Roth, J. (eds.) DSV-IS 2004 and EHCI 2004. LNCS, vol. 3425, pp. 134–135. Springer, Heidelberg (2005)
15. Mahfoudhi, A., Abid, M., Abed, M.: Towards a user interface generation approach based on object oriented design and task model. In: Proc. of the 4th Intl. Worksh. on Task Models and Diagrams, pp. 135–142. ACM (2005)
16. Mori, G., Paternò, F., Santoro, C.: CTTE: Support for developing and analyzing task models for interactive system design. IEEE Trans. Software Eng. 28(8) (2002)
17. Paternò, F., Mancini, C., Meniconi, S.: ConcurTaskTrees: A diagrammatic notation for specifying task models, pp. 362–369. Chapman & Hall (1997)
18. Paternò, F., Santoro, C., Spano, L.D.: MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. ACM Trans. Comput.-Hum. Interact. 16(4), 1–30 (2009)
19. Pietschmann, S.: A model-driven development process and runtime platform for adaptive composite web applications. Intl. Journal On Advances in Internet Technology (IntTech) 4(1), 277–288 (2010)
20. Pietschmann, S., Tietz, V., Reimann, J., Liebing, C., Pohle, M., Meißner, K.: A metamodel for context-aware component-based mashup applications. In: Proc. of the 12th Intl. Conf. on Information Integration and Web-Based Applications & Service (iiWAS 2010), pp. 413–420 (2010)
21. Tran, V.X., Tsuji, H.: A task-oriented framework for automatic service composition. In: Proc. of the 2009 Congress on Services - I (SERVICES 2009), pp. 615–620. IEEE (2009)
22. van Welie, M., van der Veer, G.C., Eliëns, A.: An ontology for task world models. In: 5th Int. Worksh. on Design, Specification, and Verification of Interactive Systems, DSV-IS (1998)