Efficient, Interactive Recommendation of Mashup Composition Knowledge

Soudip Roy Chowdhury, Florian Daniel, and Fabio Casati

University of Trento
Via Sommarive 5, 38123 Povo (TN), Italy
{rchowdhury,daniel,casati}@disi.unitn.it

Abstract. In this paper, we approach the problem of interactively querying and recommending composition knowledge in the form of reusable composition patterns. The goal is that of aiding developers in their composition task. We specifically focus on mashups and browser-based modeling tools, a domain that increasingly targets also people without profound programming experience. The problem is generally complex, in that we may need to match possibly complex patterns on-the-fly and in an approximate fashion. We describe an architecture and a pattern knowledge base that are distributed over client and server and a set of client-side search algorithms for the retrieval of step-by-step recommendations. The performance evaluation of our prototype implementation demonstrates that - if sensibly structured - even complex recommendations can be efficiently computed inside the client browser.

1 Introduction

Mashing up, i.e., composing, a set of services, for example, into a data processing logic, such as the data-flow based data processing pipes proposed by Yahoo! Pipes (`http://pipes.yahoo.com/pipes/`), is generally a ***complex task*** that can only be managed by skilled developers. People without the necessary programming experience may not be able to profitably use mashup tools like Pipes – to their dissatisfaction. For instance, we think of tech-savvy people, who like exploring software features, author and share own content on the Web, that would like to mash up other contents in new ways, but that don't have programming skills. They might lack appropriate awareness of which composable elements a tool provides, of their specific function, of how to combine them, of how to propagate data, and so on. The problem is analogous in the context of web service composition (e.g., with BPEL) or business process modeling (e.g., with BPMN), where modelers are typically more skilled, but still may not know all the features of their modeling languages.

Examples of ready mashup models are one of the main sources of ***help*** for modelers who don't know how to express or model their ideas – provided that suitable examples can be found (examples that have an analogy with the modeling situation faced by the modeler). But also tutorials, expert colleagues or

G. Kappel, Z. Maamar, H.R. Motahari-Nezhad (Eds.): ICSOC 2011, LNCS 7084, pp. 374–388, 2011.

friends, and, of course, Google are typical means to find help. However, searching for help does not always lead to success, and retrieved information is only seldom immediately usable as is, since the retrieved pieces of information are not contextual, i.e., immediately applicable to the given modeling problem.

Inspired by a study on how end users would like to be assisted in mashup development [1], we are working toward the interactive, contextual recommendation of composition knowledge, in order to assist the modeler in each step of his development task, e.g., by suggesting a candidate next component or a whole chain of tasks. The knowledge we want to recommend is re-usable ***composition patterns***, i.e., model fragments that bear knowledge that may come from a variety of possible sources, such as usage examples or tutorials of the modeling tool (developer knowledge), best modeling practices (domain expert knowledge), or recurrent model fragments in a given repository of mashup models (community knowledge [2]). The vision is that of developing an assisted, web-based mashup environment (an evolution of our former work [3]) that delivers useful composition patterns much like Google's Instant feature provides search results already while still typing keywords into the search field.

In this paper, we approach one of the core ***challenges*** of this vision, i.e., the fast search and retrieval of a ranked list of contextual development recommendations. The problem is non-trivial, in that the size of the respective knowledge base may be large, and the search for composition patterns may be complex; yet, recommendations are to be delivered at high speed, without slowing down the modeler's composition pace. Matching a partial mashup model with a repository of modeling patterns, in order to identify which of the patterns do in fact represent useful information, is similar to the well-known inexact sub-graph isomorphism problem [4], which has been proven to be NP-complete in general. Yet, if we consider that the pattern recommender should work as a plug-in for a web-based modeling tool (such as Pipes or mashArt [3], but also instruments like the Oryx BPMN editor [`http://bpt.hpi.uni-potsdam.de/Oryx/`]), fast response times become crucial.

We provide the following ***contributions***, in order to approach the problem:

- We *model* the problem of interactively recommending composition knowledge as pattern matching and retrieval problem in the context of data mashups and visual modeling tools (Section 2). This focus on one specific mashup/composition model is without loss of generality as for what regards the overall approach, and the model can easily be extended to other contexts.
- We describe an *architecture* for an assisted development environment, along with a client-side, recommendation-specific knowledge base (Section 3).
- We describe a set of *query* and *similarity search algorithms* that enable the efficient querying and ranking of interactive recommendations (Section 4).
- We study the *performance* of the conceived algorithms and show that interactively delivering composition patterns inside the modeling tool is feasible (Section 5).

In Section 6 we have a look at related works, and in the conclusion we recap the lessons we learned and provide hints of our future work.

2 Preliminaries and Problem Statement

Recommending composition knowledge requires, first of all, understanding how such knowledge looks like. We approach this problem next by introducing the mashup model that accompanies us throughout the rest of this paper and that allows us to define the concept of composition patterns as formalization of the knowledge to be recommended. Then, we characterize the typical browser-based mashup development environment and provide a precise problem statement.

2.1 Mashup Model and Composition Patterns

As a first step toward more complex mashups, in this paper we focus on ***data mashups***. Data mashups are simple in terms of modeling constructs and expressive power and, therefore, also the structure and complexity of mashup patterns is limited. The model we define in the following is inspired by Yahoo! Pipes and JackBe's Presto (`http://www.jackbe.com`) platform; in our future work we will focus on more complex models.

A data mashup model can be expressed as a tuple $m = \langle name, C, F, M, P \rangle$, where *name* is the unique name of the mashup, C is the set of components used in the mashup, F is the set of data flow connectors ruling the propagation of data among components, M is the set of data mappings of output attributes[1] to input parameters of connected components, and P is the set of parameter value assignments for component parameters. Specifically:

- $C = \{c_i | c_i = \langle name_i, desc_i, In_i, Out_i, Conf_i \rangle\}$ is the non-empty set of ***components***, with $name_i$ being the unique name of the component c_i, $desc_i$ being a natural language description of the component (for the modeler), and $In_i = \{\langle in_{ij}, req_{ij} \rangle\}, Out_i = \{out_{ik}\}$, and $Conf_i = \{\langle conf_{il}, req_{il} \rangle\}$, respectively, being the sets of input, output, and configuration parameters/attributes, and $req_{ij}, req_{il} \in \{yes, no\}$ specifying whether the parameter is required, i.e., whether it is mandatory, or not. We distinguish three kinds of components:
  - *Source* components fetch data from the web or the local machine. They don't have inputs, i.e., $In_i = \varnothing$. There may be multiple source components in C.
  - *Regular* components consume data in input and produce processed data in output. Therefore, $In_i, Out_i \neq \varnothing$. There may be multiple regular components in C.
  - *Sink* components publish the output of the data mashup, e.g., by printing it onto the screen or providing an API toward it, such as an RSS or RESTful resource. Sinks don't have outputs, i.e., $Out_i = \varnothing$. There must always be exactly one sink in C.

[1] We use the term *attribute* to denote data attributes in the data flow and the term *parameter* to denote input and configuration parameters of components.

- $F = \{f_m | f_m \in C \times C\}$ are the ***data flow connectors*** that assign to each component c_i it's predecessor c_p $(i \neq p)$ in the data flow. Source components don't require any data flow connector in input; sink components don't have data flow connectors in output.
- $M = \{m_n | m_n \in IN \times OUT, IN = \cup_{i,j} in_{ij}, OUT = \cup_{i,k} out_{ik}\}$ is the ***data mapping*** that tells each component which of the attributes of the input stream feed which of the input parameters of the component.
- $P = \{p_o | p_o \in (IN \cup CONF) \times (val \cup null), CONF = \cup_{i,l} conf_{il}\}$ is the ***value assignment*** for the input or configuration parameters of each component, *val* being a number or string value (a constant), and *null* representing an empty assignment.

This definition allows models that may not be executed in practice, e.g., because the data flow is not fully connected. With the following properties we close this gap:

Definition 1. *A mashup model m is* **correct** *if the graph expressed by F is connected and acyclic.*

Definition 2. *A mashup model m is* **executable** *if it is correct and all required input and configuration parameters have a respective data mapping or value assignment.*

These two properties must only hold in the moment we want to execute a mashup m. Of course, during development, e.g., while modeling the mashup logic inside a visual mashup editor, we may be in the presence of a *partial* mashup model $pm = \langle C, F, M, P \rangle$ that may be neither correct nor executable. Step by step, the mashup developer will then complete the model, finally obtaining a correct and executable one, which can typically be run directly from the editor in a hosted fashion.

Given the above characterization of mashups, we can now define composition knowledge that can be recommended as re-usable ***composition patterns*** for mashups of type m, i.e., model fragments that provide insight into how to solve specific modeling problems. Generically – given the mashup model introduced before – we express a composition pattern as a tuple $cp = \langle C, F, M, P, usage, date \rangle$, where C, F, M, P are as defined for m, *usage* counts how many times the pattern has been used (e.g., to compute rankings), and *date* is the creation date of the pattern. In order to be useful, a pattern must be correct, but not necessarily executable. The size of a pattern may vary from a single component with a value assignment for at least one input or configuration parameter to an entire, executable mashup; later on we will see how this is reflected in the structure of individual patterns.

Finally, to effectively deliver recommendations it is crucial to understand *when* to do so. Differently from most works on pattern search in literature (see Section 6), we aim at an ***interactive recommendation*** approach, in which patterns are queried for and delivered in response to individual modeling actions performed by the user in the modeling canvas. In visual modeling environments, we typically

have $action \in \{select, drag, drop, connect, delete, fill, map, ...\}$, where *action* is performed on an $object \subseteq C \cup F \cup IN \cup CONF$, i.e., on the set of modeling constructs affected by the last modeling action. For instance, we can *drop* a component c_i onto the canvas, or we can *select* a parameter $conf_{il}$ to fill it with a value, we can *connect* a data flow connector f_m with an existing target component, or we can *select* a set of components and connectors.

2.2 Problem Statement

In the composition context described above, providing interactive, contextual development recommendations therefore corresponds to the following problem statement: given a query $q = \langle object, action, pm \rangle$, with pm being the partial mashup model under development, how can we obtain a list of ranked composition patterns $R = [\langle cp_i, rank_i \rangle]$ (the recommendations), such that (i) the provided recommendations help the developer to stepwise draw an executable mashup model and (ii) the search, ranking, and delivery of the recommendations can be efficiently embedded into an interactive modeling process?

3 Recommending Composition Knowledge: Approach

The key idea we follow in this work is not trying to crack the whole problem at once. That is, we don't aim to match a query q against a repository of generic composition patterns of type *cp* in order to identify best matches. This is instead the most followed approach in literature on graph matching, in which, given a graph g_1, we search a repository of graphs for a graph g_2, such that g_1 is a sub-graph of g_2 or such that g_1 satisfies some similarity criteria with a sub-graph of g_2. Providing *interactive* recommendations can be seen as a specific instance of this generic problem, which however comes with both a new challenge as well as a new opportunity: the new challenge is to query for and deliver possibly complex recommendations *responsively*; the opportunity stems from the fact that we have an interactive recommendation consumption process, which allows us to *split* the task into optimized sub-steps (e.g., search for data mappings, search for connectors, and similar), which in turn helps improve performance.

Having an interactive process further means having a user performing modeling actions, inspecting recommendations, and accepting or rejecting them, where accepting a recommendation means weaving (i.e., connecting) the respective composition pattern into the partial mashup model under development. Thanks to this process, we can further split recommendations into what is needed to *represent* a pattern (e.g., a component co-occurrence) from what is needed to *use* the pattern in practice (e.g., the exact mapping of output attributes to input parameters of the component co-occurrence). We can therefore further leverage on the separation of pattern representation and usage: representations (the recommendations) don't need to be complete in terms of ingredients that make up a pattern; completeness is required only at usage time.

3.1 Types of Knowledge Patterns

Aiming to help a developer to stepwise refine his mashup model, practically means suggesting the developer which next modeling action (that makes sense) can be performed in a given state of his progress and doing so by providing as much help (in terms of modeling actions) as possible. Looking at the typical modeling steps performed by a developer (filling input fields, connecting components, copying/pasting model fragments) allows us to define the following ***types of patterns*** (for simplicity, we omit the *usage* and *date* attributes):

- *Parameter value* pattern: $cp^{par} = \langle c_x, \varnothing, \varnothing, p_o^x \rangle$. Given a component, the system suggest values for the component's parameters.
- *Connector* pattern: $cp^{conn} = \langle \{c_x, c_y\}, f^{xy}, \varnothing, \varnothing \rangle$. Given two components, the system suggests a connector among the components.
- *Data mapping* pattern: $cp^{map} = \langle \{c_x, c_y\}, f^{xy}, \{m_n^{xy}\}, \varnothing \rangle$. Given two components and a connector among them, the system suggests how to map the output attributes of the first component to the parameters of the second component.
- *Component co-occurrence* pattern: $cp^{co} = \langle \{c_x, c_y\}, f^{xy}, \{m_n^{xy}\}, \{p_o^x\} \cup \{p_o^y\} \rangle$. Given one component, the system suggests a possible next component to be used, along with all the necessary data mappings and value assignments.
- *Complex* pattern: $cp^{com} = \langle C, F, M, P \rangle$. Given a fragment of a mashup model, the system suggests a pattern consisting of multiple components and connectors, along with the respective data mappings and value assignments.

Our definition of *cp* would allow many more possible types of composition patterns, but not all of them make sense if patterns are to be re-usable as is, that is, without requiring further refinement steps like setting parameter values. This is the reason for which we include also connectors, data mappings, and value assignments when recommending a component co-occurrence pattern.

3.2 The Interactive Modeling and Recommender System

Figure 1 illustrates the internals of our prototype modeling environment equipped with an interactive knowledge recommender. We distinguish between client and server side, where the whole application logic is located in the client, and the server basically hosts the *persistent pattern knowledge base* (KB; details in Section 3.3). At startup, the *KB loader* loads the patterns into the client environment, decoupling the knowledge recommender from the server side.

Once the editor is running inside the client browser, the developer can visually compose components (in the *modeling canvas*) taken from the *component tool bar*. Doing so generates modeling events (the *actions*), which are published on a browser-internal *event bus*, which forwards each modeling action to the *recommendation engine*. Given a modeling *action*, the *object* it has been applied to, and the partial mashup model *pm*, the engine queries the *client-side pattern KB* via the *KB access API* for recommendations (pattern representations).

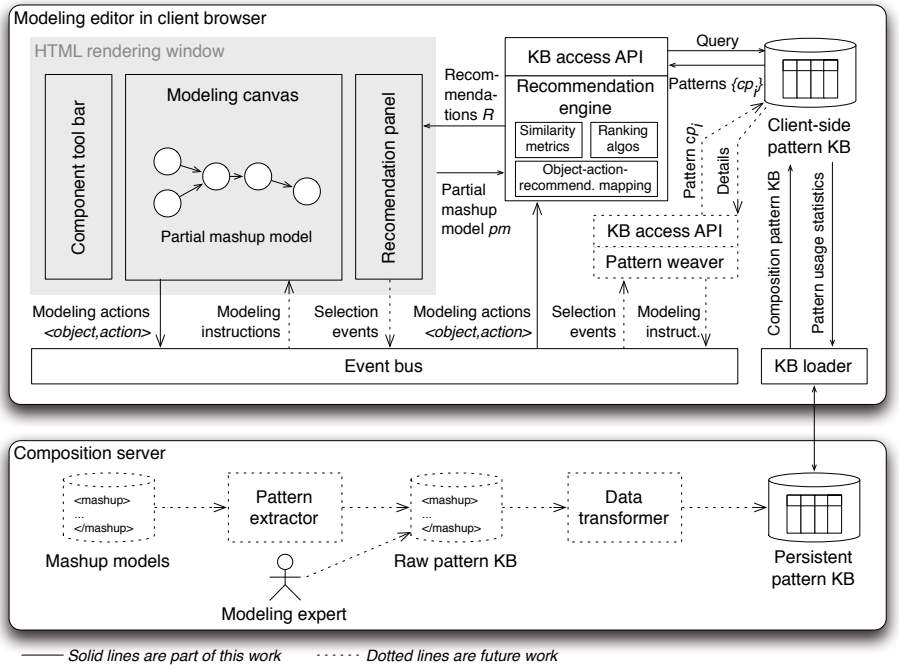


Fig. 1. Simplified architecture of the assisted modeling environment with client-side knowledge base and interactive recommender. We focus on recommendations only and omit elements like the mashup runtime environment, the component library, etc.

An *object-action-recommendation mapping* (OAR) tells the engine which type of recommendation is to be retrieved for each modeling action on a given object.

The list of patterns retrieved from the KB (either via regular queries or by applying dedicated similarity criteria) are then ranked by the engine and rendered in the *recommendation panel*, which renders the recommendations to the developer for inspection. In future, selecting a recommendation will allow the *pattern weaver* to query the KB for the usage details of the pattern (data mappings and value assignments) and to automatically provide the modeling canvas with the necessary modeling instructions to weave the pattern into the partial mashup model.

3.3 Patterns Knowledge Base

The core of the interactive recommender is the KB that stores generic patterns, but decomposed into their constituent parts, so as to enable the incremental recommendation approach. If we recall the generic definition of composition patterns, i.e., $cp = \langle C, F, M, P, usage, date \rangle$, we observe that, in order to convey the structures of a set of complex patterns inside a visual modeling tool, typically C and F (components and connectors) will suffice to allow a developer

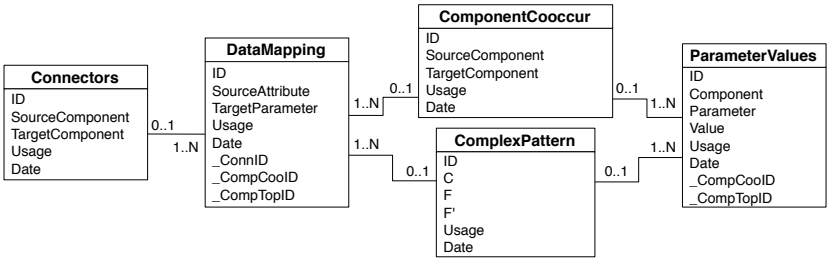


Fig. 2. Model of the pattern knowledge base for client-side knowledge management

to select a candidate pattern. Ready data mappings and value assignments are then delivered together with the components and connectors only upon selection of a pattern by the developer.

This observation leads us to the KB illustrated in Figure 2, whose structure enables the retrieval of the representations of the types of recommendations introduced in Section 3.1 with a one-shot query over a single table. For instance, the entity *Connectors* contains all connector patterns, and the entity *ComplexPattern* contains the structure of the complex patterns (in Section 4 we explain the meaning of the attributes C, F, F'). The KB is partly redundant (e.g., the structure of a complex pattern also contains components and connectors), but this is intentional. It allows us to defer the need for joins to the moment in which we really need to retrieve all details of a pattern, i.e., when we want to use it. In order to retrieve, for example, the representation of a component co-occurrence pattern, it is therefore enough to query the *ComponentCooccur* entity for the *SourceComponent* and the *TargetComponent* attributes; weaving the pattern then into the modeling canvas requires querying $ComponentCooccur \bowtie DataMapping \bowtie ParameterValues$ for the details.

4 Exact and Approximate Search of Recommendations

Given the described types of composition patterns and a query q, we retrieve composition recommendations from the described KB in two ways: (i) we *query* the KB for parameter value, connector, data mapping, and component co-occurrence patterns; and (ii) we *match* the *object* against complex patterns. The former approach is based on *exact* matches with the *object*, the latter leverages on *similarity search*. Conceptually, all recommendations could be retrieved via similarity search, but for performance reasons we apply it only in those cases (the complex patterns) where we don't know the structure of the pattern in advance and, therefore, are not able to write efficient conventional queries.

Algorithm 1 details this ***strategy*** and summarizes the logic implemented by the recommendation engine. In line 3, we retrieve the types of recommendations that can be given (*getSuitableRecTypes* function), given an *object-action* combination. Then, for each recommendation type, we either query for patterns (the

Algorithm 1. getRecommendations

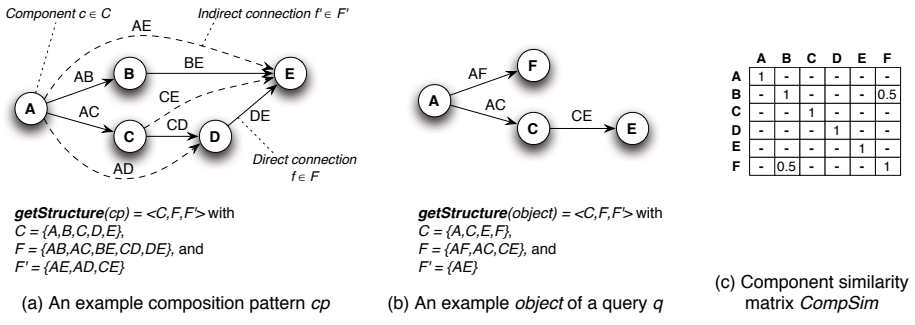
Data: query $q = \langle object, action, pm \rangle$, knowledge base KB, object-action-recommendation mapping OAR, component similarity matrix $CompSim$, similarity threshold T_{sim}, ranking threshold T_{rank}, number n of recommendations per recommendation type
Result: recommendations $R = [\langle cp_i, rank_i \rangle]$ with $rank_i \geq T_{rank}$

```
1  R = array();
2  Patterns = set();
3  recTypeToBeGiven = getSuitableRecTypes(object, action, OAR);
4  foreach recType ∈ recTypeToBeGiven do
5      if recType ∈ {ParValue, Connector, DataMapping, CompCooccur} then
6          Patterns = Patterns∪ queryPatterns(object, KB, recType) ;      // exact query
7      else
8          Patterns = Patterns∪
           getSimilarPatterns(object, KB.ComplexPattern, CompSim, T_sim) ; // similarity
           search
9  foreach pat ∈ Patterns do
10     if rank(pat.cp, pat.sim, pm) ≥ T_rank then
11         append(R, ⟨pat.cp, rank(pat.cp, pat.sim, pm)⟩) ;    // rank, threshold, remember
12 orderByRank(R);
13 groupByType(R);
14 truncateByGroup(R, n);
15 return R;
```

queryPatterns function can be seen like a traditional SQL query) or we do a similarity search (*getSimilarPatterns* function, see Algorithm 2). For each retrieved pattern, we compute a rank, e.g., based on the pattern description (e.g., containing *usage* and *date*), the computed similarity, and the usefulness of the pattern inside the partial mashup, order and group the recommendations by type, and filter out the best n patterns for each recommendation type.

As for the retrieval of ***similar patterns***, our goal was to help modelers, not to disorient them. This led us to the identification of the following principles for the identification of "similar" patterns: preference should be given to exact matches of components and connectors in *object*, candidate patterns may differ for the insertion, deletion, or substitution of at most one component in a given path in *object*, and among the non-matching components preference should be given to functionally similar components (e.g., it may be reasonable to allow a Yahoo! Map instead of a Google Map).

Algorithms 2 and 3 implement these requirements, although in a way that is already optimized for execution, in that they don't operate on the original, graph-like structure of patterns, but instead on a pre-processed representation that prevents us from traversing the graph at runtime. Figure 3(a) illustrates the pre-processing logic: each complex pattern is represented as a tuple $\langle C, F, F' \rangle$, where C is the set of components, F the set of direct connections, and F' the set of indirect connections, skipping one component for approximate search. This pre-processing logic is represented by the function *getStructure*, which can be evaluated offline for each complex pattern in the raw pattern KB; results are stored in the *ComplexPattern* entity introduced in Figure 2. Another input that can be computed offline is the component similarity matrix *CompSim*, which can be set up by an expert or automatically derived by mining the raw pattern KB. For the purpose of recommending knowledge, similarity values should



	A	B	C	D	E	F
A	1	-	-	-	-	-
B	-	1	-	-	-	0.5
C	-	-	1	-	-	-
D	-	-	-	1	-	-
E	-	-	-	-	1	-
F	-	0.5	-	-	-	1

Fig. 3. Pattern pre-processing and example of component similarity matrix *CompSim*. Components are identified with characters, connectors with their endpoints.

reflect semantic similarity among components (e.g., two *flight search* services); syntactic differences are taken into account by the pattern structures. Figure 3(c) illustrates a possible matrix for the components in the sub-figures (a) and (b); similarity values are contained in [0..1], 0 representing no similarity, 1 representing equivalence.

Algorithm 2 now works as follows. First, it derives the optimized structure of *object* (line 2). Then, it compares it with each complex pattern $cp \in CP$ in four steps: (i) it computes a similarity value for all components and connectors of *obj* and *cp* that have an exact match (line 5); (ii) it eliminates all matching components and connectors from the structure of *obj* (lines 6-8); (iii) it computes the best similarity value for the so-derived *obj* by approximating it with other components based on *CompSim* (lines 9-16); and it aggregates to two similarity values (line 17). Specifically, the algorithm substitutes one component at a time in *obj* (using *getApproximatePattern* in line 13), considering all possible substitutes *simc* and their similarity values *simc.sim* obtained from *CompSim*. The actual similarity value between two patterns is computed by Algorithm 3.

Let's consider the pattern, object, and similarity matrix in Figure 3. If in Algorithm 3 we use the weights $w_i \in \{0.5, 0.2, 0.1, 0.1, 0.1\}$ in the stated order, *sim* in line 4 of Algorithm 2 is 0.57 (exact matches for 3 components and 2 connectors). After the elimination of those matches, $obj = \langle\{F\}, \{AF\}, \varnothing\rangle$, and substituting F with B as suggested by *CompSim* allows us to obtain an additional approximate similarity of $approxSim = 0.35$ (two matches and $simc.sim = 0.5$), which yields a total similarity of $sim = 0.57 + 0.35/4 = 0.66$.

5 Implementation and Performance Evaluation

We implemented the *recommendation engine*, the *KB access API*, and the *client-side pattern KB* along with the recommendation and similarity search algorithms, in order to perform a detailed performance analysis. The ***prototype implementation*** is entirely written in JavaScript and has been tested with a Firefox 3.6.17 web browser. The implementation of the *client-side KB* is based

Algorithm 2. getSimilarPatterns

Data: query object *object*, set of complex patterns CP, component similarity matrix $CompSim$, similarity threshold T_{sim}
Result: $Patterns = \{\langle cp_i, sim_i \rangle\}$ with $sim_i \geq T_{sim}$

```
1  Patterns = set();
2  objectStructure = getStructure(object) ;   // computes object's structure for comparison
3  foreach cp ∈ CP do
4      obj = objectStructure;
5      sim = getSimilarity(obj, cp) ;                // compute similarity for exact matches
6      obj.C = obj.C − cp.C ;          // eliminate all exact matches for C, F, F' from obj
7      obj.F = obj.F − cp.F − cp.F';
8      obj.F' = obj.F' − cp.F' − cp.F;
9      approxSim = 0;          // will contain the best similarity for approximate matches
10     foreach c ∈ obj.C do
11         SimC = getSimilarComponents(c, CompSim) ;  // get set of similar components
12         foreach simc ∈ SimC do
13             approxObj = getApproximatePattern(obj, c, simc) ;   // get approx. pattern
14             newApproxSim = simc.sim*getSimilarity(approxObj, cp) ; // get similarity
15             if newApproxSim > approxSim then
16                 approxSim = newApproxSim ;  // keep highest approximate similarity
17     sim = sim + approxSim * |obj.C|/|objectStructure.C| ;   // normalize and aggregate
18     if sim ≥ Tsim then
19         Patterns = Patterns ∪ ⟨cp, sim⟩ ;      // remember patterns with sufficient sim
20 return Patterns;
```

Algorithm 3. getSimilarity

Data: query object *object*, complex pattern *cp*
Result: *similarity*

```
1  initialize w_i for i ∈ 1..5 with Σ_i w_i = 1;
2  sim_1 = |object.C ∩ cp.C|/|object.C| ;                          // matches components
3  sim_2 = |object.F ∩ cp.F|/|object.F| ;                          // matches connectors
4  sim_3 = |object.F ∩ cp.F'|/|object.F| ;          // allows insertion of a component
5  sim_4 = |object.F' ∩ cp.F|/|object.F'| ;          // allows deletion of a component
6  sim_5 = |object.F' ∩ cp.F'|/|object.F'| ;      // allows substitution of a component
7  similarity = Σ_i w_i * sim_i;
8  return similarity;
```

on Google Gears (`http://gears.google.com`), which internally uses SQL Lite (`http://www.sqlite.org`) for storing data on the client's hard drive. Given that SQL Lite does not support set data types, we serialize the representation of complex patterns $\langle C, F, F' \rangle$ in JSON and store them as strings in the respective *ComplexPattern* table in the KB; doing so slightly differs from the KB model in Figure 2, without however altering its spirit. The implementation of the *persistent pattern KB* is based on MySQL, and it is accessed by the *KB loader* through a dedicated RESTful Java API running inside an Apache 2.0 web server. The prototype implementation is installed on a MAC machine with OS X 10.6.1, a 2.26 GHz Intel Core 2 Duo processor, and 2 GB of memory. Response times are measured with the FireBug 1.5.4 plug-in for Firefox.

For the generation of realistic ***test data***, we assumed to be in the presence of a mashup editor with 26 different components ($A-Z$), with a random number of input and configuration parameters (ranging from $1-5$) and a random number of output attributes (between $1-5$). To obtain an *upper bound* for the performance of the *exact queries* for parameter value, connector, data mapping, and

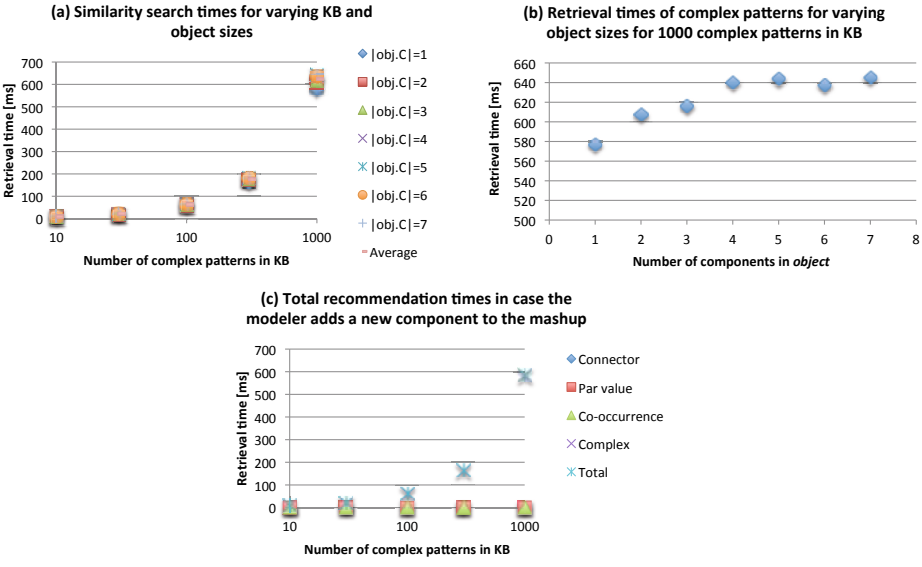


Fig. 4. Performance evaluation of the client-side knowledge recommender

component co-occurrence patterns, we generated, respectively, $26 * 5 = 130$ parameter values for the 26 components, $26 * 25 = 650$ directed connectors, $650 * 5 = 3250$ data mappings, and 650 component co-occurrences. To measure the performance of the *similarity search* algorithms, we generated 5 different KBs with 10, 30, 100, 300, 1000 complex patterns, where the complexity of patterns ranges from $3 - 9$ components. The patterns make random use of all available components and are equally distributed in the generated KBs. Finally, we generated a set of query objects with $|obj.C| \in \{1..7\}$.

In Figure 4, we illustrate the tests we performed and the respective ***results***. The first test in Figure 4(a) studies the performance in terms of pattern retrieval times of Algorithm 2 for different *KB sizes*; the figure plots the retrieval times for different *object sizes*. Considering the logarithmic scale of the x-axis, we note that the retrieval time for complex patterns grows almost linearly. This somehow unexpected behavior is due to the fact that, compared to the number of patterns, the complexity of patterns is similar among each other and limited and, hence, the similarity calculation can almost be considered a constant. We also observe that there are no significant performance differences for varying object sizes. In Figure 4(b) we investigate the effect of the object size on the performance of Algorithm 2 only for the KB with 1000 complex patterns (the only one with notable differences). Apparently, also the size of the query object does not affect much retrieval time. Figure 4(c), finally, studies the performance of Algorithm 1, i.e., the performance perceived by the user, in a typical modeling situation: in response to the user placing a new component into the canvas, the recommendation engine retrieves respective parameter value, connector, co-occurrence, and complex patterns (we do not recommend data mappings for single components);

the overall response time is the sum of the individual retrieval times. As expected, the response times of the simple queries can be neglected compared to the one of the similarity search for complex patterns, which basically dominates the whole recommendation performance.

In summary, the above tests confirm the validity of the proposed pattern recommendation approach and even outperform our own expectations. The number of components in a mashup or composition tool may be higher, yet the number of really meaningful patterns in a given modeling domain only unlikely will grow beyond several dozens or 100. Recommendation retrieval times of fractions of seconds will definitely allow us – and others – to develop more sophisticated, assisted composition environments.

6 Related Work

Traditionally, ***recommender systems*** focus on the retrieval of information of likely interest to a given user, e.g., newspaper articles or books. The likelihood of interest is typically computed based on a *user profile* containing the user's areas of interest, and retrieved results may be further refined with collaborative filtering techniques. In our work, as for now we focus less on the user and more on the partial mashup under development (we will take user preferences into account in a later stage), that is, recommendations must match the partial mashup model and the object the user is focusing on, not his interests. The approach is related to the one followed by research on *automatic service selection*, e.g., in the context of QoS- or reputation-aware service selection, or adaptive or self-healing service compositions. Yet, while these techniques typically approach the problem of selecting at runtime a concrete service for an abstract activity, we aim at interactively assisting developers at design time with more complex information in the form of complete modeling patterns.

In the context of ***web mashups***, Carlson et al. [5], for instance, react to a user's selection of a component with a recommendation for the next component to be used; the approach is based on semantic annotations of component descriptors and makes use of WordNet for disambiguation. Greenshpan et al. [6] propose an auto-completion approach that recommends components and connectors (so-called glue patterns) in response to the user providing a set of desired components; the approach computes top-k recommendations out of a graph-structured knowledge base containing components and glue patterns (the nodes) and their relationships (the arcs). While in this approach the actual structure (the graph) of the knowledge base is hidden to the user, Chen et al. [7] allow the user to mashup components by navigating a graph of components and connectors; the graph is generated in response to the user's query in form of descriptive keywords. Riabov et al. [8] also follow a keyword-based approach to express user goals, which they use to feed an automated planner that derives candidate mashups; according to the authors, obtaining a plan may require several seconds. Elmeleegy et al. [9] propose MashupAdvisor, a system that, starting from a component placed by the user, recommends a set of related components (based on conditional co-occurrence probabilities and semantic matching); upon

selection of a component, MashupAdvisor uses automatic planning to derive how to connect the selected component with the partial mashup, a process that may also take more than one minute. Beauche and Poizat [10] apply automatic planning in the context of ***service composition***. The planner generates a candidate composition starting from a user task and a set of user-specified services.

The ***business process management*** (BPM) community more strongly focuses on patterns as a means of knowledge reuse. For instance, Smirnov et al. [11] provide so-called co-occurrence action patterns in response to action/task specifications by the user; recommendations are provided based on label similarity, and also come with the necessary control flow logic to connect the suggested action. Hornung et al. [12] provide users with a keyword search facility that allows them to retrieve process models whose labels are related to the provided keywords; the algorithm applies the traditional TF-IDF technique from information retrieval to process models, turning the repository of process model into a keyword vector space. Gschwind et al. [13] allow users in their modeling tool to insert control flow patterns, as introduced by Van der Aalst et al. [14], just like other modeling elements. The proposed system does not provide interactive recommendations and rather focuses on the correct insertion of patterns.

In summary, the mashup and service composition approaches either focus on single components or connectors, or they aim to automatically plan complete compositions starting from user goals. The BPM approaches do focus on patterns as reusable elements, but most of the times pattern similarity is based on label/text similarity, not on structural compatibility. We assume components have stable names and, therefore, we do not need to interpret text labels.

7 Conclusion and Future Work

In this paper, we focused on a relevant problem in visual mashup development, i.e., the recommendation of composition knowledge. The approach we followed is similar to the one adopted in data warehousing, in which data is transformed from their operational data structure into a dimensional structure, which optimizes performance for reporting and data analysis. Analogously, instead of querying directly the raw pattern knowledge base, typically containing a set of XML documents encoding graph-like mashup structures, we decompose patterns into their constituent elements and transform them into an optimized structure directly mapped to the recommendations to be provided. We access patterns with fixed structure via simple queries, while we provide an efficient similarity search algorithm for complex patterns, whose structure is not known a-priori.

We specifically concentrated on the case of client-side mashup development environments, obtaining very good results. Yet, the described approach will perform well also in the context of other browser-based modeling tools, e.g., business process or service composition instruments (which are also model-based and of similar complexity), while very likely it will perform even better in desktop-based modeling tools like the various Eclipse-based visual editors. As such, the pattern recommendation approach discussed in this paper represents a valuable, practical input for the development of advanced modeling environments.

Next, we will work on three main aspects: The complete development of the interactive modeling environment for the interactive derivation of *search queries* and the automatic *weaving* of patterns; the *discovery* of composition patterns from a repository of mashup models; the fine-tuning of the similarity and ranking algorithms with the help of suitable *user studies*. This final step will also allow us to assess and tweak the set of proposed composition patterns.

Acknowledgments. This work was partially supported by funds from the European Commission (project OMELETTE, contract no. 257635).

References

1. De Angeli, A., Battocchi, A., Roy Chowdhury, S., Rodríguez, C., Daniel, F., Casati, F.: End-user requirements for wisdom-aware eud. In: IS-EUD 2011. Springer, Heidelberg (2011)
2. Roy Chowdhury, S., Rodríguez, C., Daniel, F., Casati, F.: Wisdom-aware computing: On the interactive recommendation of composition knowledge. In: WESOA 2010, pp. 144–155. Springer, Heidelberg (2010)
3. 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)
4. Hlaoui, A., Wang, S.: A new algorithm for inexact graph matching. In: ICPR 2002, vol. 4, pp. 180–183 (2002)
5. Carlson, M.P., Ngu, A.H., Podorozhny, R., Zeng, L.: Automatic Mash up of Composite Applications. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 317–330. Springer, Heidelberg (2008)
6. Greenshpan, O., Milo, T., Polyzotis, N.: Autocompletion for mashups. In: VLDB 2009, vol. 2, pp. 538–549 (2009)
7. Chen, H., Lu, B., Ni, Y., Xie, G., Zhou, C., Mi, J., Wu, Z.: Mashup by surfing a web of data apis. In: VLDB 2009, vol. 2, pp. 1602–1605 (2009)
8. Riabov, A.V., Boillet, E., Feblowitz, M.D., Liu, Z., Ranganathan, A.: Wishful search: interactive composition of data mashups. In: WWW 2008, pp. 775–784. ACM (2008)
9. Elmeleegy, H., Ivan, A., Akkiraju, R., Goodwin, R.: Mashup advisor: A recommendation tool for mashup development. In: ICWS 2008, pp. 337–344. IEEE Computer Society (2008)
10. Beauche, S., Poizat, P.: Automated Service Composition with Adaptive Planning. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 530–537. Springer, Heidelberg (2008)
11. Smirnov, S., Weidlich, M., Mendling, J., Weske, M.: Action Patterns in Business Process Models. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 115–129. Springer, Heidelberg (2009)
12. Hornung, T., Koschmider, A., Lausen, G.: Recommendation Based Process Modeling Support: Method and User Experience. In: Li, Q., Spaccapietra, S., Yu, E., Olivé, A. (eds.) ER 2008. LNCS, vol. 5231, pp. 265–278. Springer, Heidelberg (2008)
13. Gschwind, T., Koehler, J., Wong, J.: Applying Patterns during Business Process Modeling. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 4–19. Springer, Heidelberg (2008)
14. Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases 14, 5–51 (2003)