Recommendation and Weaving of Reusable Mashup Model Patterns for Assisted Development

SOUDIP ROY CHOWDHURY, INRIA Saclay
FLORIAN DANIEL and FABIO CASATI, University of Trento

With this article, we give an answer to one of the open problems of mashup development that users may face when operating a model-driven mashup tool, namely the *lack of modeling expertise*. Although commonly considered simple applications, mashups can also be complex software artifacts depending on the number and types of Web resources (the components) they integrate. Mashup tools have undoubtedly simplified mashup development, yet the problem is still generally nontrivial and requires intimate knowledge of the components provided by the mashup tool, its underlying mashup paradigm, and of how to apply such to the integration of the components. This knowledge is generally neither intuitive nor standardized across different mashup tools and the consequent lack of modeling expertise affects both skilled programmers and end-user programmers alike.

In this article, we show how to effectively assist the users of mashup tools with contextual, interactive recommendations of composition knowledge in the form of reusable mashup model patterns. We design and study three different recommendation algorithms and describe a pattern weaving approach for the one-click reuse of composition knowledge. We report on the implementation of three pattern recommender plugins for different mashup tools and demonstrate via user studies that recommending and weaving contextual mashup model patterns significantly reduces development times in all three cases.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; H.5.4 [**Information Systems**]: Information Interfaces and Presentation—*Hypertext/Hypermedia*; H.3.5 [**Information Systems**]: Information Storage and Retrieval

General Terms: Design, Algorithms, Languages

Additional Key Words and Phrases: Mashups, mashup patterns, pattern recommendation, pattern weaving

ACM Reference Format:
Soudip Roy Chowdhury, Florian Daniel, and Fabio Casati. 2014. Recommendation and weaving of reusable mashup model patterns for assisted development. ACM Trans. Internet Technol. 14, 2-3, Article 21 (October 2014), 23 pages.
DOI: http://dx.doi.org/10.1145/2663500

1. INTRODUCTION

Mashups are composite Web applications integrating data, application logic, and/or User Interfaces (UIs) sourced from the Web (we collectively call these *components*) [Yu et al. 2008]. A typical example and one of the first mashups available online is `housingmaps.com`, which plots housing offers by `craigslist.org` on a Google map, allowing its users to explore offers by navigating the map and thereby adding *value* to the two applications taken individually. Mashups can also be much more complex

This work was supported by the European Commission (project OMLETTE, contract 257635) and by the project "Evaluation and enhancement of social, economic and emotional wellbeing of older adult" under the agreement no.14.Z50.31.0029.
Authors' addresses: S. Roy Chowdhury, INRIA Saclay, University Paris-Sud 11, 91405 Orsay, France; F. Daniel (corresponding author) and F. Casati, University of Trento, Via Sommarive 5, 38123 Provo (TN), Italy; email: daniel@disi.unitn.it.

DOI: http://dx.doi.org/10.1145/2663500

than this (and typically are), for example, if they integrate a large number of different components, if the components themselves are just complex to use, if some components require authentication and others not, and the like. To cope with this complexity, dedicated *mashup tools* have emerged. These are typically, but not mandatorily, Web applications that provide the user with an integrated development environment comprising a *mashup editor* (for the development of mashups) and a *runtime infrastructure* (for the execution of mashups). Prominent examples of mashup tools are Yahoo! Pipes (`http://pipes.yahoo.com/pipes/`) or JackBe's Presto (`http://www.jackbe.com/products/presto`), but also academic tools like our own platform mashArt [Daniel et al. 2009] or CRUISe by Pietschmann et al. [2009].

All these tools usually come with a set of readily reusable resources (e.g, a Google map, a calendar, an RSS feed adapter, or data transformation operators), that are abstracted as components able to be configured and integrated with each other and for which the tool provides reusable implementations and runtime support. Development starting from such components is commonly supported via *model-driven development paradigms* equipped with suitable *visual development notations*, that represent components and composition activities graphically and aim to ease the development task. Except for a very few attempts of standardizing XML-based mashup languages (with little success so far) such as EMML (`http://www.openmashup.org/omadocs/v1.0/`) and OMDL (`http://omdl.org/`), mashup tools are, however, still very diverse in the features they offer, also their visual notations are proprietary, with their own abstractions and interpretations. When using a new tool, developers therefore easily lack awareness of which components a tool provides, their specific function, how to combine them, how to model the propagation of data, and so on. The problem is not dissimilar in Web service composition (e.g., with BPEL) or business process modeling (e.g., with BPMN).

It is therefore not uncommon that developers have to look for help when trying to solve a given modeling problem. Examples of existing mashup models are one of the main sources of knowledge in these situations. The problem is finding suitable examples, that is, examples that have an analogy with the modeling situation faced by the developer. Also tutorials, expert colleagues or 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., related to and immediately applicable to the given modeling problem.) Also, in a user study we conducted with 10 university accountants in the context of our own research on mashup development [De Angeli et al. 2011], we found that end-users are not reluctant at all to use mashup tools (provided they help them improve their everyday business), but that they require assistance, not only via tutorials or manuals, but continuously throughout the development process.

These observations inspired the research we describe in this article, that is, the interactive, contextual recommendation and weaving of reusable mashup model patterns. The idea is to assist the modeler in each step of his/her development task with recommendations of possible next modeling steps, such as suggesting a candidate next component or a whole chain of related activities, much like how Google's Instant feature delivers search results even as the keywords are being typed into the search field. The goal of the work is twofold: on the one hand, we aim to speed up mashup development with model-driven mashup tools; on the other hand, we want to ease mashup development and make it more accessible to both skilled and less skilled developers.

This article summarizes three years of research and covers the concepts, algorithms, implementations, and evaluations catering for an assisted modeling paradigm for model-based mashup tools. We started this research with a study of requirements for assisted composition [De Angeli et al. 2011], studied a first recommendation algorithm

in Roy Chowdhury et al. [2011b], demonstrated a first recommendation plugin in Roy Chowdhury et al. [2012], and described how to mine patterns from mashup models in Rodríguez et al. [2014b]. The contributions of this article are as follows.

—We provide a conceptual model for mashups (Section 2) and mashup patterns as well as a generic architecture for the interactive and contextual recommendation of mashup model patterns (Section 3). The model and architecture are generic in nature and can easily be adapted to a variety of different model-driven mashup or composition tools.
—We design, implement, and evaluate the precision and recall of three pattern recommendation algorithms (Section 4), respectively, for contextual, approximate pattern retrieval, for pattern retrieval taking into account the personal preferences of modelers, and for pattern retrieval taking into account the preferences of expert modelers.
—We design and implement a generic pattern weaving algorithm (Section 5) able to mimic modeling actions inside an interactive modeling environment.
—We implement three pattern recommendation plugins for different mashup tools, namely Yahoo! Pipes, Apache Rave, and MyCocktail, and demonstrate the effectiveness and viability of the interactive pattern recommendation approach via dedicated user studies carried out partly by ourselves and partly independently by partners in the context of the EU FP7 project OMELETTE (Section 6).

Before closing the article, we discuss the lessons learned and limitations (Section 7) and review the most relevant related works (Section 8).

2. MASHUPS: CONCEPTS AND MODEL

The mashup landscape is very diverse and heterogeneous, ranging from mashups for data integration to those that focus on all the three layers of the application stack (data, logic, presentation). To clearly circumscribe the problem space that we want to focus on, next we introduce a suitable mashup model, first by example, then more formally.

2.1. Example Mashup Scenario

Let us focus on Yahoo! Pipes (`http://pipes.yahoo.com/pipes/`), one of the most well-known and used data mashup platforms today. Let us also assume we want to develop a simple pipe that fetches news items from Google News, filters them according to a predefined condition (in our case, we want to search for news on products and services by a given vendor), and plots them on a Yahoo! Map.

The pipe that implements the required feature is illustrated in Figure 1. It is composed of five components: The URL Builder is needed to set up the remote GeoNames service `http://ws.geonames.org` that takes the news RSS feed, analyzes its content, and inserts geo-coordinates into each news item (where applicable). Doing so requires setting some parameters of the URL Builder component: "Base"="http://ws.geonames.org", "Path elements"="rssToGeoRSS", and "Query parameters"="feedUrl:news.google.com/news?topic=t&output=rss&ned=us." The so-created URL is fed into the Fetch Feed component that fetches the geo-enriched news feed from the Google News site. In order to filter out those news items in which we are interested, we need to use the Filter component, which requires the setting of proper filter conditions via the "Rules" input fields. Feeding the filtered feed into the Location Extractor component eventually instructs the mashup to plot the news on a Yahoo! Map. The Pipe Output component identifies the end of the data processing pipe.

If we analyze Figure 1, we easily acknowledge that developing even such a simple mashup requires good knowledge of Yahoo! Pipes: The URL Builder requires setting configuration parameters. Components need to be connected in order to allow data to flow, that is, the outputs of the components must be mapped to inputs of other

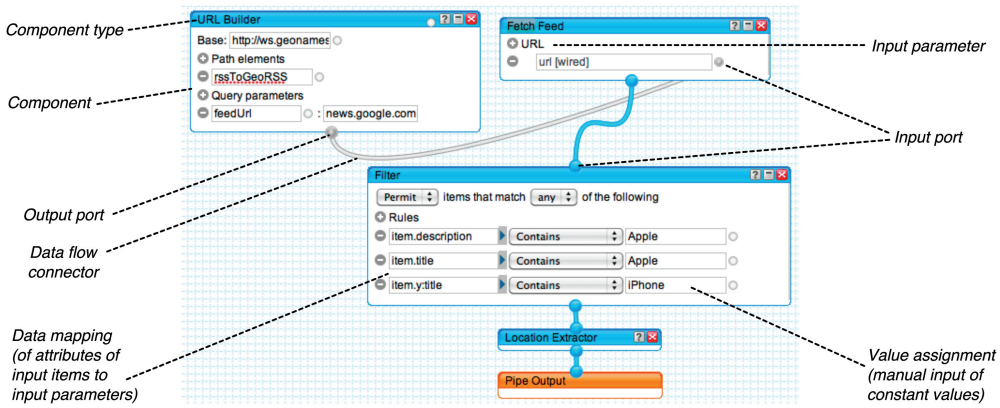


Fig. 1. A simple Yahoo! pipe that plots news on a map.

components. More importantly, plotting news onto a map implies knowing the requirement of enriching an RSS feed with geo-coordinates, fetching the actual feed, and only then plotting the items on a map. This logic is neither trivial nor intuitive without profound prior knowledge—and the example is still simple.

2.2. Reference Mashup Model

Without loss of generality of the approach we present in this article, we specifically focus our attention on data mashups such as those supported by Yahoo! Pipes, JackBe Presto, or MyCocktail. Yahoo! Pipes, in particular, is well known and comes with a large body of readily available mashup models on which we can leverage. Data mashups are relatively simple in terms of modeling constructs and expressive power and the respective mashup patterns don't grow too complex (which would hinder comprehension). However, in Section 6 we also show an example of how the resulting approach can, for instance, be adapted to pure user interface mashups.

We model a *data mashup* as a tuple $m = \langle C, DF, RES \rangle$, where C is a set of components, DF is a set of dataflow connectors, and RES is a set of result parameters of the mashup (refer to Figure 1). Specifically, we have the following.

—$C = \{c_k | c_k = \langle id_k, type_k, IP_k, IN_k, DM_k, VA_k, OP_k, OUT_k \rangle\}$ is the set of *components*, where c_k is a component instance characterized by a unique id id_k and a type $type_k$. Moreover, IP_k, IN_k, OP_k, and OUT_k, are the sets of input ports, input parameters, output ports, and output attributes (the attributes of the items in the output dataflow).
  —$DM_k \subseteq IN_k \times (\bigcup_{c \in C} c.OUT)$ is the set of *data mappings* that map attributes of input data items to the input parameters of c_k.
  —$VA_k \subseteq IN_k \times (STR \cup NUM)$ is the set of *value assignments* to the input parameters IN_k; values are either strings (STR) or numbers (NUM).
—$DF = \{df_j | df_j = \langle srccid_j, srcop_j, tgtcid_j, tgtip_j \rangle\}$ is a set of *dataflow connectors*, each assigning the output port $srcop_j$ of a source component with identifier $srccid_j$ to an input port $tgtip_j$ of a target component $tgtcid_j$, such that $srccid \neq tgtcid$.
—$RES \subseteq \bigcup_{c \in C} c.OUT$ is the set of outputs computed by the mashup.

Starting from this conceptual model, the problem we approach in this article is: (i) identifying meaningful mashup model patterns that can be reused; (ii) representing and storing them for search and retrieval; (iii) recommending contextual patterns while modeling; and (iv) weaving patterns inside visual mashup editors.

3. ASSISTED MASHUP DEVELOPMENT: APPROACH

3.1. Requirements and Assumptions

From the previous problem statement, we derive a set of requirements and principles, that we want to make explicit.

—*Reusable mashup knowledge*. The goal of this work is to assist mashup developers in the context of model-driven mashup tools. Model-driven development requires drawing diagrams, namely models, that express the internal logic of a mashup. This is the knowledge-intensive activity where we identify the need for assistance. Ogrinz [2009] proposes a set of *mashup patterns* as abstract configurations of desired mashup functionalities (the *what*). We propose a more concrete focus on *mashup model patterns*, that carry knowledge of *how* to implement desired functionalities.

—*Identification of patterns*. In De Angeli et al. [2011], we identified a need for continuous assistance throughout the modeling process. We thus propose an interactive recommendation approach, that suggests possible next modeling steps in response to modeling actions by the developer. This aim helps identify pattern types: patterns should resemble modeling actions that can be applied on behalf of the developer.

—*Representation and storage*. Our goal is to aid developers without distracting them. This requires fast recommendation times. Johnson [2010], for instance, identifies 0.1 seconds as the limit below which our brain is able to unconsciously identify correlations (e.g., between a modeling action and a recommendation) without interrupting its main task (e.g., modeling). Complying with this limit asks for client-side solutions and smart pattern storage, indexing, and query logics.

—*Recommendation*. In order for recommended patterns to be understandable and useful, it is of utmost importance that retrieved patterns are *contextual*, that is, directly related to the modeling actions of the developer. In order for patterns to be useful, it is important that they can be used straightaway and are operative. This asks for contextual recommendation algorithms.

—*Weaving*. Finally, we also want to automate the application of patterns to a partial mashup model under development. Automated weaving concretely helps by taking over modeling actions. Doing so requires resolving possible conflicts between the pattern and the partial mashup model and deriving a set of modeling actions to be executed on behalf of the developer.

The key assumptions of our work are therefore as follows. Mashups are expressed via models. Models are designed via a visual editor. It is possible to intercept modeling actions inside the editor and to access the partial mashup model under development. It is possible to extend the editor with a recommendation plugin (e.g., an own panel). It is also possible to mimic modeling actions. Most of the Web-based mashup tools available today satisfy these requirements; we will show tool examples in Section 6.

3.2. Reusable Mashup Model Patterns

Analyzing the typical modeling actions performed by a developer (e.g., filling input fields, connecting components, copying/pasting model fragments) in Pipes-like mashup tools, we specifically identify the following types of patterns (in Roy Chowdhury et al. [2011a], we discuss concrete examples). We express pattern types in terms of the mashup model introduced earlier ($m = \langle C, DF, RES \rangle$; identifiers are system generated):

Parameter value pattern. The parameter value pattern expresses possible value assignments VA for the input fields IN of a component c.

$ptype^{par} = \langle \{c\}, \varnothing\, \varnothing \rangle$; with $c = \langle 0, type, \varnothing, IN, \varnothing, \varnothing, VA, \varnothing \rangle$

This pattern helps fill input fields of a component with explicit user input.

Connector pattern. The connector pattern defines a connector df_{xy} between two components c_x and c_y, along with the respective data mapping DM_y of the output attributes OUT_x to the input parameters IN_y.

$ptype^{con} = \langle \{c_x, c_y\}, \{df_{xy}\}, \varnothing \rangle$; with

$c_x = \langle 0, type_x, \varnothing, \varnothing, \varnothing, \varnothing, \{op_x\}, OUT_x \rangle$; $c_y = \langle 1, type_y, \{ip_y\}, IN_y, DM_y, \varnothing, \varnothing, \varnothing \rangle$.

This pattern helps connect a newly placed component to the partial mashup model in the canvas.

Component co-occurrence pattern. The component co-occurrence pattern captures a pair of components c_x and c_y that occur together, along with their connector, parameter values, and c_y's data mapping logic.

$ptype^{com} = \langle \{c_x, c_y\}, \{df_{xy}\}, \varnothing \rangle$; with

$c_x = \langle 0, type_x, \varnothing, IN_x, \varnothing, VA_x, \{op_x\}, OUT_x \rangle$; $c_y = \langle 1, type_y, \{ip_y\}, IN_y, DM_y, VA_y, \varnothing, OUT_y \rangle$.

This pattern helps develop a mashup model incrementally, producing at each step a connected mashup model.

Component embedding pattern. The component embedding pattern captures which component c_z is typically embedded into a component c_y preceded by a component c_x. The pattern hence contains the three components with their connectors, data mappings, and parameter values.

$ptype^{emb} = \langle \{c_x, c_y, c_z\}, \{df_{xy}, df_{xz}, df_{zy}\}, \varnothing \rangle$; with $c_x = \langle 0, type_x, \varnothing, \varnothing, \varnothing, \{op_x\}, OUT_x \rangle$;

$c_y = \langle 1, type_y, \{ip_y\}, IN_y, DM_y, VA_y, \varnothing, \varnothing \rangle$; $c_z = \langle 2, type_z, \{ip_z\}, IN_z, DM_z, VA_z, \{op_z\}, OUT_z \rangle$.

This pattern helps, for example, model loops, commonly a nontrivial modeling task.

Multi-component pattern. The multi-component pattern represents fragments composed of multiple components. The pattern captures information about the full model fragment, along with its constituent set of components and connectors.

$ptype^{mul} = \langle C, DF, \varnothing \rangle$; with $C = \{c_i | c_i.id = i; i = 0, 1, 2, \ldots\}$.

This pattern helps in understanding, for example, domain knowledge and best practices as well as keeping agreed-upon modeling conventions.

It is important to note that these pattern types are the result of a sensible selection driven by the specific goal to assist developers in model-driven mashup editors. Given a set of mashup models of type m, in principle there is a large number of possible pattern types, but not all are applicable straightaway and sufficiently intuitive if interactively recommended. For instance, we do not propose disconnected patterns (after weaving a pattern, we always want to have connected mashup models) or connector co-occurrence patterns (this latter we actually studied in the past but discarded here because they were not useful enough). Each pattern is conceived to provide as much development assistance as possible. Where meaningful, patterns therefore already come with dataflow connectors DF, data mappings DM, and value assignments VA. This aims to ease the design of data propagation logics and the configuration of components. Focusing on predefined patterns further eases the actual identification of pattern instances (we already know the structure), allows lowering threshold values of automated pattern mining algorithms, and improves the quality of the body of reference patterns, thus the effectiveness of the overall recommender. Of course, pattern types may always be extended and tailored to the needs of individual mashup tools to increase effectiveness.

Instances of composition patterns may come from different sources, such as usage examples or tutorials (expert knowledge), modeling best practices (domain expert knowledge), or recurrent modeling fragments mined from a repository of mashup models (community knowledge). How to source patterns is an opportunistic decision that depends on the purpose of the recommendation approach (e.g., speeding up development requires automating recurrent modeling actions, while teaching how to model may benefit from a selection of more diverse patterns) and on the availability of reusable mashup models (e.g., automatically mining patterns requires large numbers

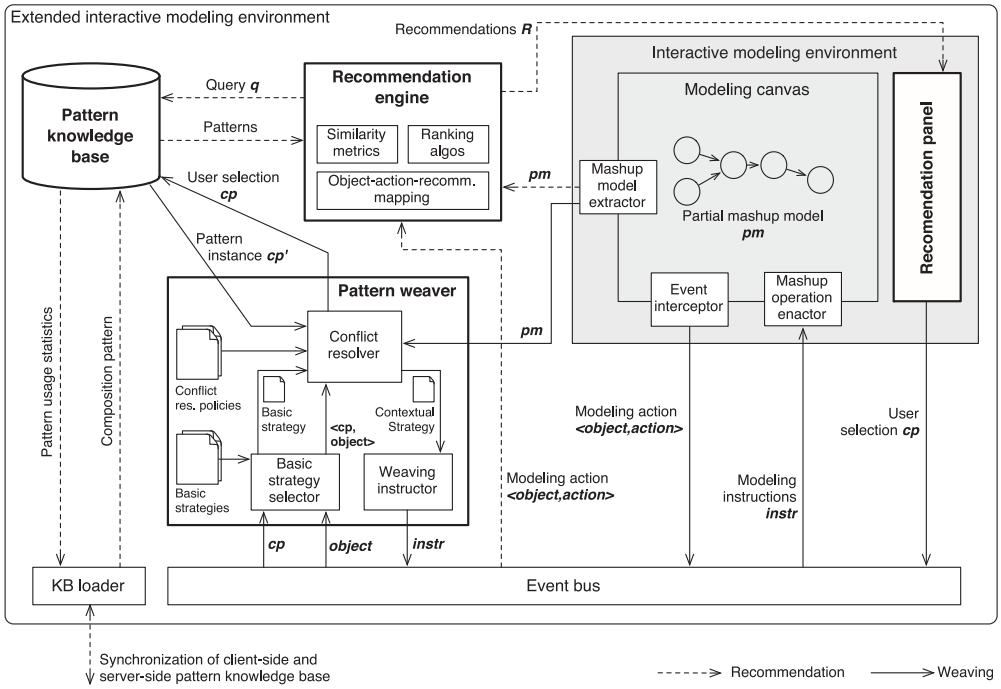


Fig. 2. Functional architecture of the interactive pattern recommendation and weaving approach, based on the extension of existing interactive modeling environments with new capabilities.

of reference examples). For our experiments with Yahoo! Pipes, we make use of automatically mined patterns [Rodríguez et al. 2014b], whereas the other two of our use cases (Section 6) make use of expert-provided patterns.

3.3. Architecture

Figure 2 details the architecture of our pattern recommendation and weaving infrastructure. The pattern Knowledge Base (KB) stores reusable mashup model patterns. The recommendation engine intercepts modeling actions on model constructs ($\langle object, action \rangle$) inside the modeling canvas and queries (q) the knowledge base for contextual patterns, that it renders in the recommendation panel, extending the modeling environment. The pattern weaver intercepts user selections (cp), fetches the necessary low-level details from the knowledge base, and weaves the pattern into the partial mashup model pm by emulating modeling actions/instructions ($instr$).

Everything is executed inside the client hosting the modeling environment and acts as an *extension* (or plugin) of the modeling environment, extending it with pattern recommendation and weaving capabilities. On the server side, we only maintain a knowledge base for the persistent storage of patterns.

We detail the knowledge base, the pattern recommender, and the pattern weaver next. In Section 6, we show how we extended three concrete modeling environments.

4. RECOMMENDING PATTERNS

In order to better understand how to recommend patterns that are indeed suitable to a given user, we develop and compare three different recommendation algorithms.

—*Contextual recommendations*. This algorithm retrieves patterns that are contextual to the latest modeling action made by the developer, that is, candidate patterns must contain the *object* of the latest modeling action. The algorithm performs both an exact and an approximate search for patterns. The ranking of patterns is based on structure only. This is the most basic algorithm we consider.
—*Personalized recommendations*. Shani and Gunawardana [2011] suggest to recommend items the users are already familiar with in order to gain their trust. A close investigation we did on users' development histories in Yahoo! Pipes in fact revealed that users tend to reuse the same set of components and data sources. With this algorithm, we extend the contextual algorithm to take into account personal component usage preferences when ranking retrieved patterns.
—*Expert recommendations*. Our investigation of the development histories also revealed that, in open, online mashup platforms like Yahoo! Pipes, users tend to learn from other users, such as by replicating or cloning existing mashup models. We assume that the more mashups of a given user are cloned, the more expert and valuable this user. In order to capture this value, with this algorithm we extend the contextual algorithm to take user expertise into account when ranking retrieved patterns.

The three algorithms make use of a common Knowledge Base (KB) for pattern retrieval. KB stores patterns, decomposed into their constituent parts, so as to enable a fast, incremental recommendation approach: to recommend patterns, we only retrieve components and connectors; only to weave a pattern, we also fetch parameter values and data mappings. The observation underlying this practice is that conveying a recommendation does not require details; it is enough to communicate the essence of a pattern. Only the parameter value pattern immediately requires detailed values. We refer the reader to Appendix A for the details of the KB.

To enable the evaluation of the recommendation algorithms, we filled the KB with 660 distinct mashup model patterns (259 component co-occurrence, 292 parameter value, and 109 multi-component patterns) automatically mined from a dataset of 970 Yahoo! Pipes models (so-called pipes) retrieved through the YQL Console (`http://developer.yahoo.com/yql/console/`). Pattern clones (patterns identified multiple times) are taken into account by the mining algorithm to compute support values, but eliminated from the KB. We selected pipes from the list of "most popular" pipes, as popular pipes are more likely to be functioning and useful. The average numbers of components, connectors, and input parameters of the retrieved pipes were 11.1, 11.0, and 4.1, respectively, indicating the selection of reasonably complex and realistic pipes. The used mining algorithms are described in detail in Rodríguez et al. [2014b].

4.1. Evaluation Metrics: Precision and Recall

For the evaluation of the three recommendation algorithms, we adopt the widely accepted precision and recall metrics of recommendation systems evaluation. The two metrics are based on the concepts of true positives, false positives, and false negatives.

—*True Positive (TP)*. We are in the presence of a *true positive* if one of the top-k recommendations retrieved is accepted by the user as the next modeling action.
—*False Positive (FP)*. We have a *false positive* if none of the retrieved top-k recommendations is accepted.
—*False Negative (FN)*. We are in the presence of a *false negative* if no recommendations can be retrieved at all.

Given a body of recommendations and the respective user reactions, the precision and recall metrics can be calculated as $P = \frac{|TP|}{|TP|+|FP|}$ and $R = \frac{|TP|}{|TP|+|FN|}$.

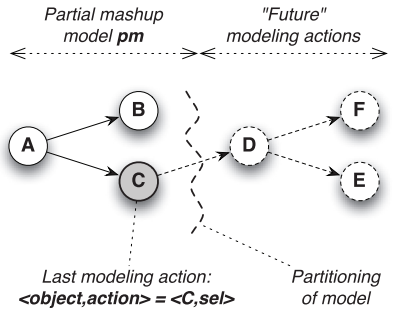


Fig. 3. Partitioning of a pipes model into a partially developed model and "future" modeling actions (action object size 1).

ALGORITHM 1: getPatterns

Data: Query $q = \langle object, action, pm \rangle$, knowledge base KB, object-action-recommendation mapping OAR, component similarity matrix $CompSim$, similarity threshold T_{sim}
Result: $Patterns$ with similarity with $object$ greater than T_{sim}

1 $Patterns$ = set();
2 $recTypes$ = getSuitableRecTypes($object, action, OAR$); `// get types of recommendations to be given`
3 **foreach** $recType \in recTypes$ **do**
4 **if** $recType \in \{ParValue, Connector, DataMapping, CompCooccur\}$ **then**
5 $Patterns$ = $Patterns \cup$ queryPatterns($object, KB, recType$) ; `// exact`
6 **else**
7 $Patterns$ = $Patterns \cup$ getSimilarPatterns($object, KB.MultiComponent, CompSim, T_{sim}$) ; `// similar`
8 **return** $Patterns$;

To create the necessary test data, we again selected 100 pipes, making sure they were not already considered when mining mashup model patterns to fill the KB. The strategy to derive modeling actions (to trigger the recommendation process) as well as to derive reactions to the recommendations is to partition a pipe model into a partial mashup model *pm* and "future modeling actions" (everything after the partition). We then assume the developer selected a part of *pm* as the last modeling action, for instance, a component. We are therefore in the presence of a true positive if any of the top-k recommendations retrieved matches the future modeling action directly following the object. The partitioning procedure is illustrated in Figure 3. We applied it to generate 856 test cases with different object sizes (in parentheses): 356 (1) + 227 (2) + 212 (3) + 61 (4).

4.2. Contextual Recommendations

The contextual recommendation algorithm is the basic one we introduced in Roy Chowdhury et al. [2011b] and that we extend next with personalized and expert recommendations. We split the algorithm into two parts: *getPatterns* (Algorithm 1) retrieves candidate patterns from the KB, while *getContextualRecommendations* (Algorithm 2) ranks retrieved patterns and returns them for rendering.

With the help of an *Object-Action-Recommendation* (*OAR*) rule, getPatterns first determines which pattern types are to be retrieved (line 2). A given type of modeling action on a given type of object always triggers a specific type of recommendation, such as when the selection of an input field triggers parameter value patterns, or when adding a new component may trigger all types of patterns. Given the recommendation

ALGORITHM 2: getContextualRecommendations

Data: Query $q = \langle object, action, pm \rangle$, knowledge base KB, object-action-recommendation map OAR, component similarity $CompSim$, similarity threshold T_{sim}, ranking threshold T_{rank}, k for top-k threshold
Result: Recommendations $R = [\langle cp_i, rank_i \rangle]$ with $rank_i \geq T_{rank}$

```
1 R = array();
2 Patterns = getPatterns(q, KB, OAR, CompSim, T_sim) ;          // retrieve patterns
3 foreach pat ∈ Patterns do
4     if rank(pat.cp, pat.sim, pm) ≥ T_rank then
5         append(R, ⟨pat.cp, rank(pat.cp, pat.sim, pm)⟩) ;      // rank, threshold, remember
6 OrderGroupTruncate(R, k);
7 return R;
```

types, recommendations are retrieved by the algorithm in two ways: (i) it queries the KB for parameter value, connector, data mapping, and component co-occurrence patterns (lines 4 and 5); and (ii) it *matches* the object against complex patterns (line 7). The former approach is based on exact matches with the object, whereas 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 (i.e., for multi-component patterns) and therefore are not able to write efficient exact queries.

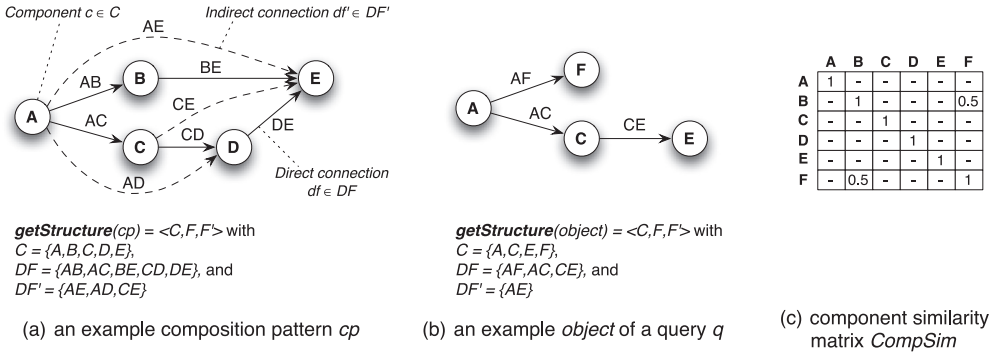
As for the retrieval of similar patterns, our goal is to help rather then to disorient, modelers. This leads us to the following principles for the identification of "similar" patterns: (1) the preference should be given to exact matches of components and connectors in object; (2) the candidate patterns may differ for the insertion, deletion, or substitution of at most one component in a given path in object; and (3) among the nonmatching components, preference should be given to functionally similar ones, for example, it may be reasonable to allow a Yahoo! Map instead of a Google Map.

Figure 4 illustrates the resulting matching logic (for the details, refer to Roy Chowdhury et al. [2011b]). Each multi-component pattern is represented as a tuple $\langle C, DF, DF' \rangle$, where C is the set of components, DF the set of direct connections, and DF' the set of indirect connections, skipping one component for approximate search. This preprocessing logic is represented by the function *getStructure* in Figure 4(a) and can be evaluated offline for each multi-component pattern in the pattern KB; results are stored in the *MultiComponent* entity of the KB as shown in Figure 9. Also the component similarity matrix *CompSim* (Figure 4(c)) can be set up by an expert or automatically derived by mining the pattern KB. Given a query object such as that in Figure 4(b), similarity search now matches the result of getStructure(*object*) against the patterns in KB, comparing components, direct, and indirect connections.

The actual getPersonalizedPatterns algorithm is Algorithm 2. For each pattern retrieved by Algorithm 1, it computes a rank based on the pattern's description (containing usage and date), the computed similarity, and the usefulness of the pattern inside the partial mashup (if the pattern is already contained in *pm*, we do not recommend it again). Then, it orders and groups the recommendations by type and filters out the best k patterns for each recommendation type.

4.3. Personalized Recommendations

The key ingredient to provide personalized recommendations are suitable *user profiles* (in our case, user-item matrices matching users with items) that allow the recommendation algorithm to take into account individual preferences. Some approaches [Sarwar



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

Fig. 4. Pattern preprocessing and example of component similarity matrix *CompSim* [Roy Chowdhury et al. 2011b]. Components are identified with characters, connectors by their endpoints.

et al. 2000] leverage on explicit item ratings provided by users. In our mashup scenario, however, collecting this kind of information is only possible if mashup tools are purposefully extending. We therefore derive implicit ratings of components by analyzing the development behavior of users, in the same spirit as Hu et al. [2008].

For each user and component, we derive the respective frequency of use, normalized in the range [0, 1] (i.e., 0 meaning the user has never used the component and 1 meaning he/she has used the component in all his/her mashups). Concretely, for the purpose of this article, we crawled the catalog of existing pipes as well as the individual user profile pages of Yahoo! Pipes and collected profile information for 441 unique Yahoo! Pipes users and the 56 components of Yahoo! Pipes. Each user profile therefore consists of a matrix capturing the association of users with components/modules and of users with data sources.

An analysis of the identified implicit rating matrices revealed a high sparsity, indicating that most users only use few components and are unaware of others. This makes conventional collaborative filtering techniques [Su and Khoshgoftaar 2009] ineffective, hence we opted for an approach based on singular value decomposition. *Singular Value Decomposition* (SVD), such as the Alternating Least-Square (ALS) algorithm by Koren et al. [2009], is a matrix factorization technique that effectively discovers latent features from a user-item rating matrix while dealing with high sparsity. Berry et al. [1995] also point out that the reduced orthogonal dimensions resulting from SVD are even less noisy than the original data.

SVD characterizes items and users by vectors of features inferred from the rating matrix. A high correlation between item and user features implies common rating patterns. During matrix factorization, the SVD algorithm learns these patterns, allowing it then to predict item ratings for users also for items without direct item-user relationships in the original dataset. Technically, SVD factors an $m \times n$ matrix R into three matrixes: $R = U \cdot S \cdot V^T$, where U and V^T are two orthogonal matrices of size $m \times r$ and $r \times n$, respectively, and S is a diagonal matrix of singular values (ranked from greatest to lowest) of size $r \times r$, where r is called the *rank* of R. In our case, R is the implicit user-item rating matrix derived before, U is the association users $\times$ features, S is the association features $\times$ features, and V^T is the association features $\times$ components.

SVD now aims to find r' independent feature vectors (that represent the whole feature space) by applying a feature reduction technique identifying the best lower-rank approximation R' of the original matrix R. For example, if every user who used component1 also used component2, the features of these two components are linearly dependent and can be approximated by a single combined feature, thereby

reducing the dimension of S. Given a reduced matrix of singular values S', we can derive a prediction of user-item ratings R' and infer missing ones by calculating $R' = U \cdot S' \cdot V^T$.

We specifically implemented the ALS variant of SVD defined by Zhou et al. [2008] using the following parameters (determined via experiments): number of hidden features $n_f = 30$, regularization metric $\lambda = 0.05f$ to prevent overfitting, and number of iterations $i = 20$. To calculate the accuracy of the rating prediction of our recommendation algorithm, we used a standard Root Mean Square Error (RMSE) metric $RMSE = \sqrt{\frac{1}{|S_{test}|} \sum_{(c,u) \in S_{test}} (R_{c,u} - R'_{c,u})^2}$, in which $R_{c,u}$ denotes the rating of a component c by a user u in the input rating matrix for the ALS algorithm and $R'_{c,u}$ denotes the predicted rating for the same component and user as produced by the ALS algorithm. As test dataset S_{test} for the accuracy measure calculation, we considered those entries in the original rating matrix R that have nonzero rating values, that is, for which we already knew the rating. The calculated RMSE for the prediction of components by our implementation is only 0.1206, confirming the quality of the prediction algorithm. Algorithm 4 in Appendix B.1 shows how we use the inferred user-item rating matrix R' to retrieve personalized recommendations.

4.4. Expert Recommendations

The preceding algorithm gives preference to those patterns that have components with which the modeler is already familiar. Here, we want to identify those patterns expert modelers are likely familiar with. As we do not make any assumptions regarding the provenance of patterns, that is, in general we do not know which pattern stems from which modeler (this information is, for instance, lost when automatically mining patterns), the question is how to correlate patterns with experts. The approach we propose is similar to the previous algorithm in that we rank patterns according to component preferences, this time, however, considering the modeling preferences of experts. The primary challenge is therefore to identify experts in a system like Yahoo! Pipes.

Thus we rely on the *clone count*, which tells how many times a given pipe by a given modeler has been cloned, that is, used as starting point for the development of a new pipe. If a modeler clones a pipe by another modeler, this is like an endorsement or citation (of a scientific paper). Leveraging on this latter analogy, we extend the well-known *h-index* [Hirsch 2005] to the domain of mashups and define a *clone h-index*: A modeler has a clone index h if h of his/her pipes have been cloned h times, while the rest of his/her pipes have no more than h clones.

We compute the clone h-index for all users for which we already identified a personal user-item rating matrix R' in the previous section, order them in descending order, and select the top-n modelers. We fix n such that the n modelers collectively have used each of the 56 components at least once in the past. In order to construct an expert-item rating matrix E, we then sum their individual user-item rating matrices and divide the result by n: $E = \frac{1}{n} \sum_{i \in (1..n)} R'_i$. Out of the 441 users of Yahoo! Pipes we identified in the previous section, with this procedure we selected $n = 36$ expert modelers with clone h-index values ranging from 46 to 4 (average 8.44). The resulting getExpertRecommendation algorithm is detailed in Algorithm 5 in Appendix B.2.

4.5. Evaluation

We used the precision and recall metrics defined earlier to assess the three recommendation algorithms. All tests were based on the 856 test cases identified in Section 4.1; the KB was filled with the 660 patterns described in Section 4. Figure 5 illustrates the results obtained in three different test settings.

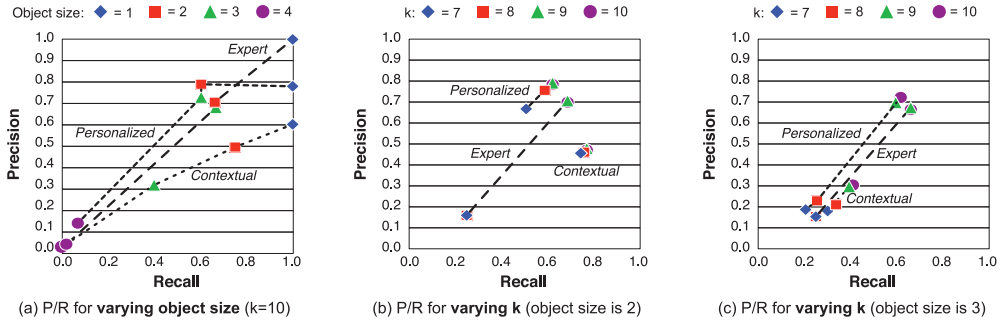


Fig. 5. Precision and recall for the three recommendation algorithms under different parameters settings.

Figure 5(a) studies the behavior of the algorithms by varying the size of object from 1 to 4 components while keeping the number k of recommended patterns constant ($k = 10$). For object size 1, we note that all algorithms have a recall of 1, indicating that if the query considers only one component, (e.g., the latest one, which is the most important modeling situation), all algorithms are able to recommend some patterns. The precision is slightly lower. As soon as the object size increases, both precision and recall drop as expected, since the query becomes more complex. Object sizes 2 and 3 still produce good results; with object size 4 the algorithms are no longer able to retrieve a set of meaningful patterns. This result is less an issue of the recommendation algorithms and more a problem of the patterns in the knowledge base: out of the 109 multi-component patterns, very few are composed of more than 4 components. Except for the case of object size 1 where the expert recommendations excel (the result of precision 1 needs further investigation), the personalized recommendations consistently outperform the other two. While this result is partly expected, is it interesting to see that the expert recommendations also perform well. This can be read as an indication that, given the relatively low complexity of data mashups and the limited modeling expertise required for mashup tools like Yahoo! Pipes, average modelers do not develop mashups that are very different from the ones made by expert modelers.

Figures 5(b) and (c) study more specifically the effect of k for object sizes 2 and 3, respectively; for object size 1, we have seen that precision and recall stay consistently high. The best results are, of course, obtained with $k = 10$. Also $k = 9$ produces very similar precision and recall values, which, however, significantly drop in both charts for $k \leq 8$. This confirms that, in order to effectively assist a modeler in his task, the recommendation panel should be able to render at least 9 patterns at a time.

5. WEAVING PATTERNS

Weaving a given pattern cp into a partial mashup model pm is not straightforward and requires a thorough analysis of both pm and cp, in order to understand how to apply cp to the constructs already present in pm without resulting in modeling conflicts. The problem is not as simple as just copying and pasting the pattern, in that new identifiers of all constructs of cp need to be generated, connectors must be rewritten based on the new identifiers, and connections with existing constructs must be defined by satisfying the modeling constraints. The implementation of the respective weaving functionality relies on two different strategies, namely a basic and a contextual strategy.

5.1. Basic Weaving Strategy

Given an object and the pattern cp of a recommendation, the basic weaving strategy BS provides the sequence of mashup operations necessary to weave cp into the object. In

other words, the BS tells how to expand object into *cp*, without taking into consideration *pm*. The basic weaving strategy is static for each pattern type.

The key ingredients of the *BS* are therefore the basic mashup operations available to express the strategy. Mashup operations resemble the operations a developer can typically perform in the modeling canvas when designing a mashup, such as addComponent(*ctype*), deleteComponent(*cid*), assignValue(*cid*, *VA*), addConnector(df_{xy}) and the like (we provide the details of the full list of mashup operations identified for dataflow-based mashups in Appendix C.1). Mashup operations modify the partial mashup *pm* and produce an updated version *pm*′. All operations assume the *pm* is globally accessible. The internal logic of these operations is highly platform specific in that they need to operate inside the target modeling environment; in our case, our implementation manipulates the JSON representation of Yahoo! Pipes mashup models.

For instance, let's assume we want to weave a component co-occurrence pattern of type $ptype^{comp}$. If we further assume $object = comp_x$ (i.e., one single component) with $comp_x.type = c_x.type$, where c_x is the first component of the selected pattern, then the basic weaving strategy is as follows: To weave the pattern, a new component c_y must be added to the composition. This step is denoted by the instruction "$newcid = addComponent($c_y$.*type*)", where the "$" prefix denotes variables, followed by connecting the existing component $comp_x$ with the new component ("addConnector(⟨$comp_x$.id, $comp_x$.op,$newcid,$c_y$.ip⟩)"). Then, we apply the data mappings of c_y and value assignments for the parameters of both c_x and c_y.

In Appendix C.2 we describe the function getBasicStrategy($cp, object$) → *BS*, that produces the strategies *BS* for all patterns identified in Section 3.2.

5.2. Conflict Resolution and Contextual Weaving Strategy

Applying the mashup operations in the basic strategy may require the resolution of possible conflicts among the constructs of *pm* and those of *cp*. For instance, if we want to add a new component of type *ctype* to *pm*, but *pm* already contains a component *comp* of type *ctype*, we are in the presence of a conflict; either we decide to reuse *comp*, which is already there, or decide to create a new instance of *ctype*. In the former case, we say we apply a *soft* conflict resolution policy, whereas in the latter case a *hard* resolution policy. In Appendix C.3, we describe the two policies.

Given an object, a pattern *cp*, and a partial mashup *pm*, a *contextual weaving strategy* *WS* is now a sequence of mashup operations that weave *cp* into *pm*.

The core logic of our pattern weaver is expressed in Algorithm 3. First, it derives a basic strategy *BS* for a given composition pattern *cp* and the *object* from *pm* (line 2). Then, for each of the mashup operations *instr* in *BS*, it checks for possible conflicts with the current modeling context *pm* (line 4). In case of a conflict, the function resolveConflict(*pm*, *instr*) derives the corresponding contextual weaving instructions *CtxInstr* replacing the conflicting, basic operation *instr*. Subsequently *CtxInstr* is applied to the current *pm* to compute the updated mashup model *pm*′ (line 5), which is then used as the basis for weaving the next *instr* of *BS*. The contextual weaving structure *WS* is constructed as a concatenation of all conflict-free instructions *CtxInstr*. Algorithm 3 returns both the list of contextual weaving instructions *WS* and the final updated mashup model *pm*′. The former can be used to interactively weave *cp* into *pm*, the latter to convert *pm*′ into native formats.

In Appendix C.4 we go through a concrete weaving example and explain each step of the weaving process.

6. IMPLEMENTATION AND USER STUDIES

In order to test the viability of the assisted mashup development approach described in the previous sections, we implemented three different extensions (called *Baya*—the

ALGORITHM 3: getWeavingStrategy

Data: partial mashup model *pm*, composition pattern *cp*, object *object* that triggered the recommendation
Result: contextual weaving instructions *WS*, i.e., a sequence of abstract mashup operations; updated mashup model *pm′*

```
1 WS = array();
2 BS = getBasicStrategy(cp, object);
3 foreach instr ∈ BS do
4 |   CtxInstr = resolveConflict(pm, instr);
5 |   pm = apply(pm, CtxInstr);
6 |   append(WS, CtxInstr);
7 return ⟨WS, pm⟩;
```

name stems from a weaverbird that weaves its nest with long strips of leaves) of existing mashup environments and performed suitable user studies. Specifically, we extended Yahoo! Pipes, Apache Rave, and MyCocktail, the latter two in the context of the EU FP7 project OMELETTE (`http://www.ict-omelette.eu/`).

The purpose of the user studies was to test the following hypotheses (where possible).

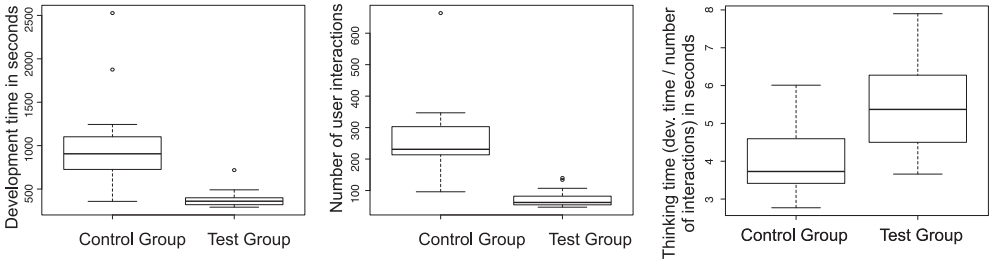
—*H1*. Baya speeds up mashup development; that is, the average development time by users with Baya is lower than that by users without.
—*H2*. Mashup design with Baya requires fewer user interactions (number of clicks) than without.
—*H3*. Mashup design with Baya requires less thinking time (time between two user interactions) to take modeling decisions than without.

The respective user studies were partly carried out by ourselves, partly independently by our partners in the OMELETTE project.

6.1. Baya for Yahoo! Pipes

Baya for Yahoo! Pipes is a Mozilla FireFox add-on that extends Pipes inside the browser with a pattern recommendation panel, implementing the architecture in Figure 2. The implementation is based on JavaScript for the business logic (e.g., the pattern search, retrieval, matching, and weaving algorithms) and XUL (`https://developer.mozilla.org/En/XUL`) for UI development. The use of JavaScript is the basis for the implementation of custom event listeners that intercept modeling events on elements in the DOM tree (e.g., dropping a new component) and of the mashup operations described in Table I in Appendix C.2. Mashup operations are written in JavaScript to manipulate the mashup model's internal JSON representation. Both weaving strategies (basic and contextual) are encoded as JSON arrays of mashup operations, thereby enabling us to use the native `eval()` command for fast and easy parsing of the weaving logic. The implementation of the KB is based on SQLite (`http://www.sqlite.org`). A modeling example and demo of Baya in action can be found at `http://goo.gl/YXuqvR`.

User study. In order to enroll a sufficient number of people familiar with Yahoo! Pipes, we designed a crowdsourced user study [Stolee and Elbaum 2010] using Amazon's Mechanical Turk platform (`https://www.mturk.com`). We elaborated a mashup scenario to be developed in Yahoo! Pipes and designed two different test settings: a control group developed the scenario without Baya, while a test group developed the same scenario with the help of Baya. A Google Form replica of the questionnaire we used for the control group can be found at `http://goo.gl/9UnZW`, while that for the test group can be found at `http://goo.gl/B3ZE0`. Both groups had to install the Baya add-on in order to objectively measure the development time and number of user interactions; for the



(a) mean development time in designing a pipe without and with Baya

(b) mean number of user interactions in designing a pipe without and with Baya

(c) mean thinking time for designing a pipe without and with Baya

Fig. 6. User study of Baya in Yahoo! Pipes with 30 participants split into a control and a test group.

control group, the recommendation panel was disabled. After the development task, participants were asked to fill out an online questionnaire evaluating their satisfaction with their development experience (we used a 5-point Likert scale ranging from *strongly agree* to *strongly disagree* to collect feedback). In three days (from May 16 to May 18, 2012) and with a reward of $1 for developing the pipe and filling the questionnaire and $0.10 for additional free feedback, we could attract 32 participants, out of which 30 provided useful data (15 in each group). Participants were required to pass a qualification test with questions about Yahoo! Pipes in order to prevent junk answers.

The data collected are illustrated in Figure 6. In order to accept or reject our hypotheses, we used Welch's t-test for equal sample sizes and unequal variance.

—*H1*. Figure 6(a) shows the collected development times, with $\mu_{dev,ctrl} = 1027.1s$ and $\mu_{dev,test} = 384.9s$. The null hypothesis is $\mu_{dev,test} - \mu_{dev,ctrl} = 0$, that is, there is no significant difference between the two development times. The p-value for this null hypothesis is 0.00045, which is very small. Hence, we reject the null hypothesis, proving that Baya speeds up mashup development in Yahoo! Pipes.
—*H2*. Figure 6(b) shows the number of interactions; $\mu_{int,ctrl} = 258.9$ and $\mu_{int,test} = 74.3$. The null hypothesis is $\mu_{int,text} - \mu_{int,ctrl} = 0$. The p-value for this hypothesis is 0.00009, again very small, thus we have to reject the null hypothesis. Hence, Baya requires fewer user interactions in Yahoo! Pipes.
—*H3*. Figure 6(c) shows the thinking times; $\mu_{th,ctrl} = 4.0s$ and $\mu_{th,test} = 5.5s$, that is, the control group has lower thinking time (as opposed to H3). The null hypothesis is $\mu_{th,test} - \mu_{th,ctrl} = 0$. The p-value for this hypothesis was 0.00209, once again a very small probability. Hence, Baya increases thinking time in Yahoo! Pipes.

The feedback collected via the questionnaire further reinforced these conclusions: 73% of the control group agreed that understanding which module to use in their pipe took most of their time, whereas 100% agreed (27% strongly) that assigning the right parameter values to a module took most of their design time. Moreover, 73% of the control group agreed that some form of automated assistance would have helped them in their task. Out of the test group subjects, 80% strongly agreed that the interactive recommendations saved time, whereas 73% agreed that the automatic weaving feature did so (27% expressed neutral feedback). As for the scenario, 80% of the control group and 73% of the test group strongly agreed that the scenario was nontrivial, a property we considered necessary for the collection of meaningful data.

6.2. Baya for Apache Rave

Apache Rave (`http://rave.apache.org/`) is an open-source mashup environment for OpenSocial and W3C UI widgets. It follows a relatively simple and live development

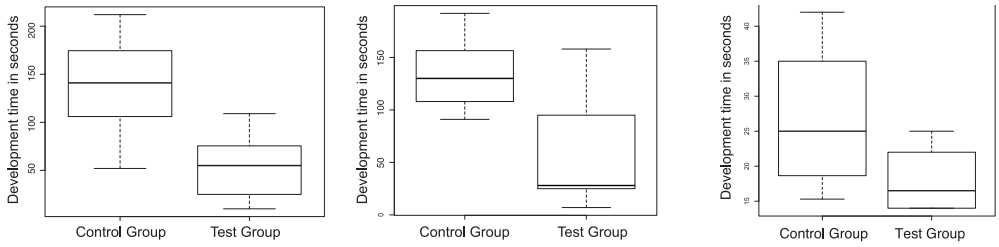


Fig. 7. Results of the user study of Baya for Apache Rave and MyCocktail.

paradigm: graphical widgets can be placed into a so-called workspace and are immediately rendered. In order to enable communication among widgets, the OMELETTE consortium has implemented an eventing extension of the W3C widget model [Wilson et al. 2011] and a suitable event bus for Rave. Baya for Rave comes as: (i) an extension of Rave to intercept events and to enact weaving operations; and (ii) a UI widget for the recommendation panel. Its internal architecture is very similar to the one presented in Figure 2 and supports widget co-occurence and multiwidget patterns. Parameter settings inside widgets are out of the control of Rave; events are broadcast and interpreted by all widgets (hence no need for connector patterns) and it is not possible to embed widgets inside each other. In absence of workspaces to mine from, we manually filled the pattern knowledge base with expert-provided patterns for the chosen domain (communication and collaboration). All algorithms are again implemented in JavaScript. The pattern KB runs in SQLite and stores patterns in JSON. JavaScript event listeners capture the triggering events for pattern retrieval, namely DOM modifications (e.g., adding a widget, deleting a widget) of the workspace model.

User study. This study was conducted independently by our partners Huawei and T-Systems MMS in the OMELETTE project [OMELETTE Consortium 2013]. The study was conducted in China and Germany and involved a total of 44 participants (again equally distributed into test and control groups). Due to the embedding of this study inside the broader one of the OMELETTE products, this study asked the two groups to extend an existing mashup (workspace) according to a given set of requirements, again both with and without the help of recommendations. Only the development time for both groups could be collected in this setting. We reused Welch's t-test for equal sample sizes and unequal variance to verify Hypothesis 1.

—*H1 (China)*. Figure 7(a) shows the collected development times, with $\mu_{dev,ctrl} = 139.8s$ and $\mu_{dev,test} = 54.6s$. The null hypothesis is $\mu_{dev,test} - \mu_{dev,ctrl} = 0$, namely there is no significant difference between the two development times. The p-value for the null hypotheses is 0.0001, very small. Hence, we reject the null hypothesis, confirming that Baya speeds up development in Apache Rave.
—*H1 (Germany)*. Figure 7(b) shows the collected development times, with $\mu_{dev,ctrl} = 132.1s$ and $\mu_{dev,test} = 58.9s$ with a p-value of 0.0007 for the null hypothesis. We thus reject the null hypotheses, reconfirming that Baya speeds up development in Apache Rave also in the German test setting.

Qualitatively, 67% of all users considered the recommendation widget important or even essential for a mashup environment. Some users stated that Baya's assistance helped them learn the usage of new widgets of which they were previously unaware.

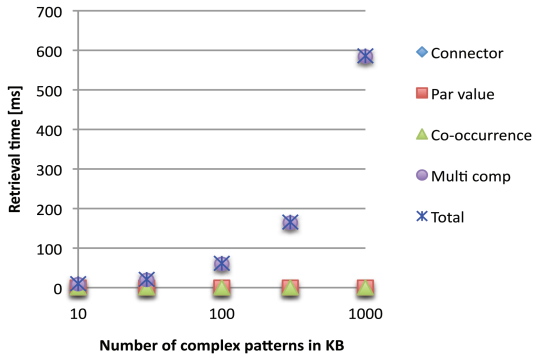


Fig. 8. Performance of pattern recommender.

6.3. Baya for MyCocktail

MyCocktail (`http://www.ict-romulus.eu/web/mycocktail`) is a data mashup tool similar to Yahoo! Pipes that has been extended in the context of the OMELETTE project with the possibility to deploy mashups as W3C widgets. The tool features three different types of components: *service* components (fetching data), *operation* components (manipulating data), and *renderer* components (visualizing data). Data is propagated among components via global, shared variables. Baya for MyCocktail comes as a native extension of MyCocktail developed together with CGI/Logica, who owns the tool. It supports three types of mashup patterns, namely *parameter value*, *component co-occurrence*, and *multi-component* patterns. All algorithms are implemented in JavaScript and data are stored using SQLite.

User study. Also this user study was carried out independently by our partner CGI/Logica in the OMELETTE project [OMELETTE Consortium 2013]. In total, 8 employees of the company took part in this study, equally distributed into test and control groups. Participants were asked to design a tag cloud widget that visualizes RSS feed data on some topic. The study once more logged the total design time for each user, thus allowing us to verify only hypothesis H1 (see Figure 7(c) for the detailed data).

—*H1*. The average time to complete the development of the widget with and without Baya was, respectively, $\mu_{dev,test} = 18s$ and $\mu_{dev,ctrl} = 26.8s$, with a p-value of the null hypothesis of 0.22. This result does not allow us to reject the null hypothesis with high statistical confidence (partly also due to the low number of participants), but it nevertheless indicates a tendency toward the reduction of development time.

Participants consistently provided positive feedback on the interactive, step-by-step development assistance provided by the pattern recommender.

7. PERFORMANCE AND LIMITATIONS

As for the performance, namely the response time, of the pattern recommendation algorithms, in Roy Chowdhury et al. [2011b] we performed a set of tests with Baya for Yahoo! Pipes with synthetically generated patterns. As explained earlier, all algorithms and the knowledge base were implemented as client-side extensions of Yahoo! Pipes. Tests were performed with a Firefox 3.6.17 Web browser and a common MacBook Pro with OS X 10.6.1, a 2.26 GHz Intel Core 2 Duo processor, and 2GB of memory. Response times were measured with the FireBug 1.5.4 plugin for Firefox. Figure 8 illustrates 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

parameter value, connector, component co-occurrence, and multi-component patterns. 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 multi-component patterns that grows almost linearly (consider the logarithmic scale of the x-axis).

The important observation is that, even with 109 multi-component patterns as employed in our user studies, the overall response time remains well below the 0.1s mark, our target chosen so as not to distract users from their modeling task. The result outperforms our expectations and applies to all three algorithms as they share the same pattern retrieval logic and only apply different ranking logics.

For a better assessment of the applicability of our approach to other contexts, it is good to point out also some limitations (and assumptions) of the work.

—*Web-based implementation*. The goal of this work was not to develop yet another standalone mashup tool, therefore we propose an extension of existing mashup tools with a special focus on Web-based tools. It is good to note that the approach could also be applied to desktop-based modeling environments such as Eclipse or the like. In terms of performance, we expect this would further decrease response times.

—*Coverage of the pattern knowledge base*. We are aware that the recommendation of patterns and hence the success of the approach also depend on the availability of suitable patterns. We intentionally do not propose any specific source for patterns, as we consider identifying good patterns a research area on its own. One of our user studies is based on automatically mined patterns (details are in Rodríguez et al. [2014b]), the other two on patterns provided by domain experts. The three user studies achieve consistently good results across all evaluations. Yet, we acknowledge that precision and recall directly depend on the quality and number of available patterns, respectively. Also, we did not focus on the evolution of the pattern knowledge base itself, a problem we would like to approach with the periodic injection of new patterns (if available) and the analysis of pattern usage data and ratings.

—*Size of patterns*. In Section 4.5, we have seen that both precision and recall dramatically drop for query object sizes greater than 2 to 3 components. An analysis of the patterns in the knowledge base revealed this drop must be ascribed to the lack of suitable multi-component patterns in the knowledge base, and not to the recommendation algorithms. This lack, in turn, derives from the use of automated mining algorithms, that leverage on the support (the number of occurrences) of patterns in the source dataset for the identification of patterns. The bigger a pattern, the harder it is to find it repeated multiple times with exactly the same implementation.

—*Computation of user/expert-item rating matrices*. The approach we presented in this article for the computation of the user-/expert-item rating matrices assumes the availability of sufficient data, namely, the availability of at least some mashup models per user. If new users join a mashup platform, they will of course not yet have mashup models from which to infer preferences. The use of incremental SVD algorithms [Sarwar et al. 2002] in this context may help achieve fast suitable matrices also for new users, improve them over time, and reduce the respective computation cost. It is, however, important to note that, in our personalized algorithm, SVD is used only to rank patterns retrieved by the contextual algorithm. As a consequence, in absence of suitable user-item ratings, the recommender is nevertheless able to seamlessly deliver valuable recommendations.

—*Didactic value to novice users*. One of the wishes of this work is to help especially novice modelers or even end-users to develop their own mashups, for instance, through patterns with high didactic value. The user study of Baya for Apache Rave specifically focused on nonprogrammers and the results obtained are in line with the

other two studies involving more expert developers. Yet, the claim that our approach helps novice modelers to learn how to model strongly depends on the mashup environment hosting the recommendation extension and needs further investigation and analysis.

—*Pattern quality*. The effectiveness of the approach also depends on the quality of recommended patterns and components. Our assumption here is that patterns proven successful in the past (e.g., assessed by experts) or that have been used repeatedly (e.g., verified via minimum support threshold values) are also of good quality. Yet, explicit quality models as, for instance, in Chen et al. [2013] and Cappiello et al. [2009, 2011] could be used to analytically ascertain quality and to rerank patterns.

—*Trust in patterns*. It is important to note that—from a technical point of view—the problem of recommending and weaving patterns is independent of where patterns actually come from (e.g., experts versus automated mining). However, we also acknowledge that comprehension of the source of a pattern may help create trust in recommendations and help developers to better interpret patterns and avoid incoherences. It is our intention to add this information in the future and to allow developers to provide feedback on patterns (see the video references earlier).

—*UI design*. Also the way recommendations are communicated to users strongly depends on the hosting mashup environment. How recommendations are rendered may have a significant effect on the performance of modelers and the perceived usefulness of the recommendations. We developed our recommendation extensions following a best-effort and best-knowledge approach. A dedicated study on how to communicate modeling recommendations most effectively would complement this work.

8. RELATED WORK

Traditionally, recommender systems have focused on the retrieval of information likely of interest to a given user, such as newspaper articles or books. The likelihood of interest is typically computed based on a user profile containing the user's areas of interest. In our work, we first of all aim to recommend contextual model patterns, namely patterns that are related to what a user does in the modeling canvas. Only then do we refine by taking preferences into account. Model-driven development is a complex task and our user studies show that pattern usefulness is more important than component preferences (although these may further improve the quality of recommendations).

In the specific context of Web mashups, there are also other works aiming to assist developers in their development task. Syntactic approaches [Wong and Hong 2007] suggest modeling constructs based on syntactic similarity, for instance, comparing output and input data types. Semantic [Ngu et al. 2010] or goal-oriented [Riabov et al. 2008] approaches target at automatically deriving compositions that satisfy user-specified goals, such as leveraging on annotations of modeling constructs with semantic descriptions. In *programming by demonstration* [Cypher et al. 1993], the goal is to auto-complete a process definition starting from a set of user-selected model examples. Hybrid approaches extend semantic and goal-based techniques; for instance, Elmeleegy et al. [2008] propose an interactive goal recommender that suggests a set of composition goals to the user based on his/her current composition context. If the user accepts a goal, the mashup engine accordingly auto-completes the partial mashup composition with the help of an AI planner. A pattern-based approach similar to our work is proposed by Deutch et al. [2010], who aim to recommend so-called *glue patterns* in response to user-selected components (so-called *mashlets*) in order to auto-complete a partial mashup; supported patterns fall only into the category of what we call component co-occurrence patterns.

All these methods typically require advanced modeling skills (which not all users have), or expect the user to specify complex rules for defining goals (thereby breaking

the actual modeling paradigm), or expect domain experts to specify and maintain complex semantic networks describing modeling constructs and their relationships (which they don't do). Cappiello et al. [2012] specifically focus on end-user development and on a widget-based live mashup paradigm that comes without explicit models (making development more intuitive) and complement it with quality-aware recommendations of UI widgets. Quality awareness may indeed improve the quality in our approach also, but so far our focus has been on model-driven mashup paradigms.

The focus in the context of *service-oriented computing* has traditionally been more on service selection and automated composition. Most notably, the area of QoS-based service selection [Lin et al. 2012] targets selecting and/or recommending services based on nonfunctional properties of services. Chen et al. [2013] envision a very similar approach to the one proposed in this article and recommend service sequence patterns based on QoS and service usage frequency. Automated approaches typically propose semantic languages, such as WSMO or OWL-S, to equip service descriptions with metadata that can be machine processed.

It is the business process management community that more specifically focuses on model patterns. For instance, Gschwind et al. [2008] propose an automated pattern application technique for the workflow patterns introduced by van der Aalst et al. [2003], namely control-flow patterns like simple merge, exclusive choice, parallel branch, etc. Patterns are recommended based on structural similarity with a process model. Control-flow patterns, however, are generic and do not capture domain knowledge. The key problem with business process models is nonetheless less the control-flow structure and more their semantics such as typically derived from task labels specified by the modeler individually for each model. Klinkmüller et al. [2013], for example, well clarify the problem and achieve good results with their process model matching approach. Constructs in mashup tools have constant names, thus easing model matching. What is more complicated is that also variables, parameter values, and data mappings must be taken into account and, more importantly, also woven.

9. CONCLUSION AND FUTURE WORK

This article comprehensively treats the problem of recommending patterns in mashup tools. We propose the use of mashup model patterns as a means to foster knowledge reuse and thereby speed up and ease mashup development. We: (i) define a set of typical mashup model patterns; (ii) describe three different recommendation algorithms; (iii) study their performance, precision, and recall; (iv) describe a weaving technique to apply patterns automatically; and (v) demonstrate the viability of the approach in the context of three different extensions of existing mashup tools equipped with suitable user studies (done by ourselves and by partners in the EU FP7 project OMELETTE). All prototypes consistently lowered development times and user satisfaction was consistently high throughout the three user studies.

We specifically focused on data mashups. Yet, the approach and underlying concepts and algorithms are generic in nature and can easily be applied to different modeling contexts. For instance, we used the same algorithms, patterns, and architecture for the development of three different recommendation plugins, one of which specifically for UI mashups. Other modeling notations and environments can be similarly empowered.

In our future work, we focus more specifically on the identification of multi-component patterns with more than 3 components. We already developed a crowd-based pattern mining algorithm that involves human expertise [Rodríguez et al. 2014a]. The first results achieved are very promising in terms of obtaining bigger patterns than those derived with traditional mining algorithms and to attack the cold start problem. We further plan to leverage on developer reactions to recommendations and feedback to improve the precision of the recommendation algorithms. Finally, we also would like

to study the value of the approach to novice users and the influence of different pattern visualizations and pattern quality. This will require the design of controlled user studies specifically tailored to the study of different user groups, pattern visualizations, and pattern quality levels, as well as the adoption of more qualitative assessment techniques (e.g., speak-aloud practices or contextual interviews) able to go beyond statistical significance and to elicit subjective opinions and feelings. Excluding the study of the different visualization techniques, these studies can be carried out with Baya as-is, without the need for new technical implementations.

ACKNOWLEDGMENTS

We are grateful to C. Rodríguez (U. Trento) for providing access to the mashup patterns mined from Yahoo! Pipes. We thank S. Wilson (U. Bolton) for helping with the implementation of Baya in Apache Rave, S. Pietschmann and M. Niederhausen (T-Systems MMS) and C. Heng (Huawei) for the user study of Baya in Apache Rave, and C. Villalonga, R. Illera and J.C. Ruiz Dominguez (Logica/CGI) for the user study of Baya in MyCocktail. We thank S. Thirumuruganathan (U. Texas at Arlington) for his help with the collection of user profiles.

REFERENCES

M. W. Berry, S. Dumais, G. O'brien, M. W. Berry, S. T. Dumais, and Gavin. 1995. Using linear algebra for intelligent information retrieval. *SIAM Rev.* 37, 573–595.

C. Cappiello, F. Daniel, A. Koschmider, M. Matera, and M. Picozzi, 2011. A quality model for mashups. In *Proceedings of the 11th International Conference on Web Engineering (ICWE'11)*. 137–151.

C. Cappiello, F. Daniel, and M. Matera, 2009. A quality model for mashup components. In *Proceedings of the 9th International Conference on Web Engineering (ICWE'09)*. 236–250.

C. Cappiello, M. Matera, M. Picozzi, F. Daniel, and A. Fernandez 2012. Quality-aware mashup composition: Issues, techniques and tools. In *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology (QUATIC'12)*. 10–19.

L. Chen, J. Wu, H. Jian, H. Deng, and Z. Wu 2013. Instant recommendation for web services composition. *IEEE Trans. Services Comput.* 99, 1.

A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, EDS. 1993. *Watch What I Do: Programming by Demonstration*. MIT Press.

F. Daniel, F. Casati, B. Benatallah, and M.-C. Shan 2009. Hosted universal composition: Models, languages and infrastructure in mashart. In *Proceedings of the 28th International Conference on Conceptual Modeling (ER'09)*. Springer, 428–443.

A. De Angeli, A. Battocchi, S. Roy Chowdhury, C. Rodriguez, F. Daniel, and F. Casati 2011. End-user requirements for wisdom-aware eud. In *Proceedings of the 3rd International Conference on End-User Development (IS-EUD'11)*. 245–250.

D. Deutch, O. Greenshpan, and T. Milo 2010. Navigating in complex mashed-up applications. *Proc. VLDB Endow.* 3, 1–2, 320–329.

H. Elmeleegy, A. Ivan, R. Akkiraju, and R. Goodwin 2008. Mashup advisor: A recommendation tool for mashup development. In *Proceedings of the IEEE International Conference on Web Services (ICWS'08)*. IEEE Computer Society, 337–344.

T. Gschwind, J. Koehler, and J. Wong 2008. Applying patterns during business process modeling. In *Proceedings of the 6th International Conference on Business Process Management (BPM'08)*. Springer, 4–19.

J. E. Hirsch 2005. An index to quantify an individual's scientific research output. *Proc. Nat. Acad. Sci.* 102, 46.

Y. Hu, Y. Koren, and C. Volinsky 2008. Collaborative filtering for implicit feedback datasets. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM'08)*. IEEE Computer Society, 263–272.

J. Johnson 2010. *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules*. Morgan Kaufmann, San Fransisco.

C. Klinkmuller, I. Weber, J. Mendling, H. Leopold, and A. Ludwig 2013. Increasing recall of process model matching by improved activity label matching. In *Proceedings of the 11th International Conference on Business Process Management (BPM'13)*. 211–218.

Y. Koren, R. Bell, and C. Volinsky 2009. Matrix factorization techniques for recommender systems. *Comput.* 42, 8, 30–37.

D. Lin, C. Shi, and T. Ishida 2012. Dynamic service selection based on context-aware qos. In *Proceedings of the 9th IEEE International Conference on Services Computing (SCC'12).* IEEE Computer Society, 641–648.

A. Ngu, M. Carlson, Q. Sheng, and AIK, H. Young 2010. Semantic-based mashup of composite applications. *IEEE Trans. Services Comput.* 3, 1, 2–15.

M. Ogrinz 2009. *Mashup Patterns: Designs and Examples for the Modern Enterprise*. Addison-Wesley.

Omelette Consortium. 2013. D7.4: Evaluation of demonstrators report. Project deliverable. http://www.ict-omelette.eu/public_deliverables.

S. Pietschmann, M. Voigt, A. Rumpel, and K. Meissner 2009. CRUISe: Composition of rich user interface services. In *Proceedings of the 9th International Conference on Web Engineering (ICWE'09).* Springer, 473–476.

A. V. Riabov, E. Boillet, M. D. Feblowitz, Z. Liu, and A. Ranganathan 2008. Wishful search: Interactive composition of data mashups. In *Proceedings of the 17th International Conference on World Wide Web (WWW'08).* ACM Press, New York, 775–784.

C. Rodriguez, F. Daniel, and F. Casati 2014a. Crowd-based mining of reusable process model patterns. In *Proceedings of the 12th International Conference on Business Process Management (BPM'14).* 51–66.

C. Rodriguez, S. Roy Chowdhury, F. Daniel, H. R. Motahari Nezhad, and F. Casati 2014b. Assisted mashup development: On the discovery and recommendation of mashup composition knowledge. In *Web Services Foundations*. Springer, 683–708.

S. Roy Chowdhury, A. Birukou, F. Daniel, and F. Casati 2011a. Composition patterns in data flow based mashups. In *Proceedings of the 16th European Conference on Pattern Languages of Programs (EuroPLoP'11).* 27–28.

S. Roy Chowdhury, F. Daniel, and F. Casati 2011b. Efficient, interactive recommendation of mashup composition knowledge. In *Proceedings of the 9th International Conference on Service- Oriented Computing (ICSOC'11).* 374–388.

S. Roy Chowdhury, C. Rodriguez, F. Daniel, and F. Casati 2012. Baya: Assisted mashup development as a service. In *Proceedings of the 21st International Conference on World Wide Web Companion (WWW'12).* ACM Press, New York, 409–412.

B. Sarwar, G. Karypis, J. Konstan, and J. Riedl 2002. Incremental singular value decomposition algorithms for highly scalable recommender systems. In *Proceedings of the 5th International Conference on Computer and Information Science (ICCIT'02).* 27–28.

B. M. Sarwar, G. Karypis, J. A. Konstan, and J. T. Riedl 2000. Application of dimensionality reduction in recommender system – A case study. In *Proceedings of the ACM WEBKDD Workshop.*

G. Shani and A. Gunawardana 2011. Evaluating recommendation systems. In *Recommender Systems Handbook.* Springer, 257–297.

K. T. Stolee and S. Elbaum 2010. Exploring the use of crowdsourcing to support empirical studies in software engineering. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'10).* ACM Press, New York, 35:1–35:4.

X. Su and T. M. Khoshgoftaar 2009. A survey of collaborative filtering techniques. *Adv. Artif. Intell. 2009,* 4:2–4:2.

W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros 2003. Workflow patterns. *Distrib. Parallel Databases* 14, 5–51.

S. Wilson, F. Daniel, U. Jugel, and S. Soi 2011. Orchestrated user interface mashups using w3c widgets. In *Proceedings of the 11th International Conference on Current Trends in Web Engineering (ICWE'11).* Springer, 49–61.

J. Wong and J. I. Hong 2007. Making mashups with marmite: Towards end-user programming for the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'07).* 1435–1444.

J. Yu, B. Benatallah, F. Casati, and F. Daniel 2008. Understanding mashup development. *IEEE Internet Comput.* 12, 5, 44–52.

Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan 2008. Large-scale parallel collaborative filtering for the netflix prize. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management (AAIM'08).* Springer, 337–348.

Received October 2013; revised March 2014; accepted June 2014