PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Assisted Reuse of Pattern-Based Composition Knowledge for Mashup Development

Soudip Roy Chowdhury

Advisor:

Prof. Fabio Casati and Dr. Florian Daniel

Università degli Studi di Trento

March, 2013

Abstract

First generation of the World Wide Web (WWW) enabled users to have instantaneous access to a large diversity of knowledge. Second generation of the WWW (Web 2.0) brought a fundamental change in the way people interact with and through the World Wide Web. Web 2.0 has made the World Wide Web a platform not only for communication and sharing information but also for software development (e.g., web service composition). Web mashup or mashup development is a Web2.0 development approach in which users are expected to create applications by combining multiple data sources, application logic and UI components from the web to cater for their situational application needs. However, in reality creating an even simple mashup application is a complex task *that can only be managed by skilled developers.*

Examples of ready mashup models are one of the main sources of help for users who don't know how to design a mashup, provided that suitable examples can be found (examples that have an analogy with the modeling situation faced by the user). But 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 a success, and retrieved information is only seldom immediately usable as it is, since the retrieved pieces of information are not contextual, i.e., immediately applicable to the given modeling problem.

Motivated by the development challenges faced by a naive user of existing mashup tools, in this thesis we propose to aid such users by enabling assisted reuse of pattern-based composition knowledge. In this thesis we show how it is possible to effectively assist these users in their development task with contextual, interactive recommendations of composition knowledge in the form of mashup model patterns. We study a set of recommendation algorithms with different levels of performance and describe a flexible pattern weaving approach for the one-click reuse of patterns. We prove the generality of our algorithms and approach by implementing two prototype tools for two different mashup platforms. Finally, we validate the usefulness of our assisted development approach by performing thorough empirical tests and two user studies with our prototype tools.

Keywords[recommendation,weaving,assisted mashup development, personalized recommendation, users studies]

Preface

This thesis is structured as a collection of articles that we published during this work and those, thus, have been reviewed and accepted by peers in the scientific community. Chapter 1 presents an executive summary of our research work, providing an overview of the main problems, research challenges and solutions and the references to our articles describing in details various parts of the work. Chapter 1 is structured as follows: first, we provide motivations (in subsection 1.1) for this thesis. Then, we define problems we identified in the context of the motivating example, research challenges (in section 1.3) associated with the identified problem statement, the research methodology that we follow (in section1.4) during this research exploration. Section 1.5 summarizes the main contributions of this thesis. Finally, Section 1.7 discusses the key lessons we learned during this research work, limitations of this work and future work that we are planning to do next. All the articles composing this work (cited in Figure 1.13) are included as appendix at the end of this dissertation.

Acknowledgements

Quitting an esteemed research lab job for pursuing Ph.D. study was a tough call for me back in 2009, however, the great guidances, which I had received from my mentors and colleagues at IBM Research Lab helped me and motivated me to explore the tough path of Ph.D. Now, when I analyze my so far journey in research before and during Ph.D., I can confidently say that my decision to take up this hard path was justified and I am deeply grateful to my research mentor Dr. C. Mohan and other great managers from IBM Dr. Manish Gupta, Dr. Shivkumar Kalyanraman, Dr. Sambit Sahu who had greatly motivated me to take up the challenge for doing Ph.D.

I wish to convey my sincere gratitude to my advisor Prof. Fabio Casati for giving me the opportunity to carry out my research in the University of Trento. I would also like to express my sincere thanks to my co-advisor Dr. Florian Daniel for his advice during my doctoral research endeavor. As my supervisor, he has constantly motivated me to remain focused on achieving my goals. His observations and comments helped me to strive for perfection and to move forward with research in depth.

Besides my advisors, I would like to thank my thesis committee : Prof. Boualem Benatallah, Prof. Ihab Francis Ilyas and Prof. Cinzia Cappiello, for their encouragement, insightful comments and feedback.

I would also like to thank Dr. Carlos Castillo, Dr. Patrick Meier for hosting me and offering me an internship opportunity in Qatar Computing Research Institute that led me to work on diverse exciting projects in social computing. This work experience has certainly broaden my perspective as a researcher. I also wish to convey my sincere thanks to all my colleagues and friends at Qatar Computing Research for providing a friendly research environment.

I also thank Dr. Sihem Amer Yahia and Prof. Gautam Das for guiding and helping at times during my Ph.D. journey with their insightful guidance and help. I wish to thank OMELETTE EU FP7 project and COMPAS EU FP7 project for funding my research. I also wish to convey my sincere thanks to my research collaborators who have worked with me in these projects and who have co-authored with me for various publications. I also thank my colleague and friends at DISI.

Last but not the least, I am deeply thankful to my family for their love and support. I dedicate this thesis to my mother, who has been the continuous source of motivation in my life. This last word of acknowledgment I have saved for my dear wife, who has been with me in all the ups and downs of life. Her sacrifice, unlimited patience, understanding and continuous encouragements through out my Ph.D journey has made this day possible.

(Soudip Roy Chowdhury)

Contents

Chapter 1

Thesis Summary

All knowledge that the world has ever received comes from the mind; the infinite library of the universe is in our own mind.

Swami Vivekananda

Despite several efforts for simplifying the composition process, the learning process for using existing mashup editors remains rather difficult. Due to this difficult learning process, the development of mashup applications is only achievable by expert developers. In this thesis, we describe how this barrier can be lowered by means of an assisted development approach that enables the reuse of existing composition knowledge. We further demonstrate and verify how with the help of our efficient interactive pattern recommendation technique, less-skilled user can successfully build mashup applications.

1.1 Motivating scenario and problem statement

In order to better understand the problem that we address in this thesis, let's have a look at how a mashup is, for instance, composed in Yahoo! Pipes `http://pipes.yahoo.com/pipes/`, one of the most well-known and used mashup platforms as of today. Let us assume we want to develop a simple pipe that fetches a set of news feeds from Google News website, 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 locates them on a Yahoo! Map based upon the geo-location associated with each news item.

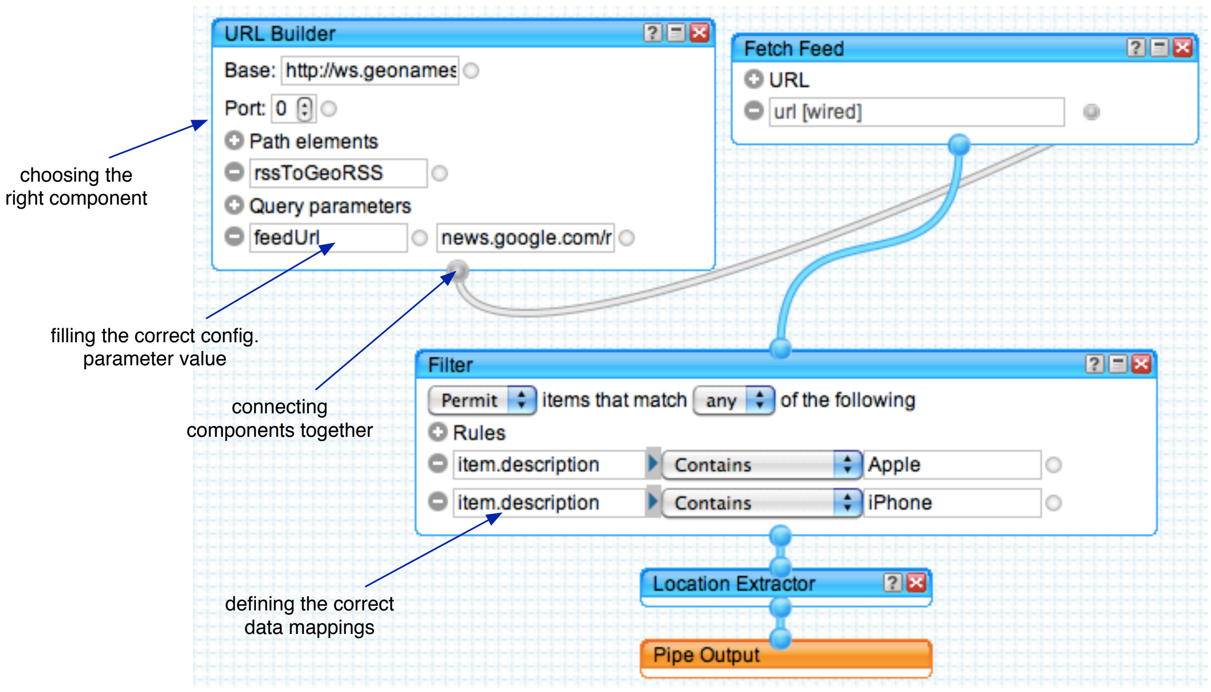
The pipe that implements the required feature is illustrated in Figure 1.1. It is composed of five components: The *URL Builder* is needed to set up the remote GeoNames service `http://www.geonames.org/`, which takes a news RSS feed as an input, analyzes its content, and inserts geo-coordinates, i.e., longitude and latitude, into each news item (where applicable). Doing so requires setting few 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, which fetches the geo-enriched news feed from the Google News site. In order to filter out the news items we are really interested in, we need to use the *Filter* component, which requires the setting of proper filter conditions via the *Rules* input field. Feeding the filtered feed into the *Location Extractor* component causes the plot of the news items on a Yahoo! Map. Finally, the *Pipe Output* component specifies the end of the pipe.

If we analyze the development steps above, we can easily understand that developing even such a simple composition is out of reach for people without sufficient programming knowledge, it is even difficult for a less-skilled developer. As pointed out in the Figure 1.1, the *URL Builder*, for example, requires the correct setting of it's configuration parameter. Then, components need to be correctly connected in order to define a consistent data-flow logic i.e., the output parameter of a component must be mapped correctly to the input parameter/s of other component/s. But more importantly, plotting news onto a map requires knowing that this can be done by first enriching a feed with geo-coordinates, then by fetching the actual feed, and then by plotting the fetched items on a map. Understanding all these programming concepts is *neither trivial nor intuitive* for a less-skilled developer.

To have a more detailed understanding of the problem space i.e., end user development, we performed an initial user study [3] with few end users (10 university accountants), who have little technical expertise. The study revealed that less-skilled developers wished to be assisted with automatic/contextual help through out the development process in case they are developing an application. In one hand the *less-skilled* developers (the users who have limited technical knowledge; e.g., users who know how to use the basic spread sheet functionalities but never used any software programming tools) want to be assisted in their development tasks. On the other hand, *more-skilled* users (people who possess technical expertise for programming) develop useful applications using such platforms and thus generates composition knowledge in terms of best practices. By bridging the gap between these two types of developer communities (less-skilled and more-skilled), i.e., by reusing the knowledge created by the more-skilled users to help the less-skilled ones, we can reduce the overhead of learning time and enhance the usability for any mashup tool like Yahoo! Pipes. Designing a suitable recommendation system that helps to reuse composition knowledge that are either harvested from existing applications developed by more-skilled users, or are provided by the domain experts themselves could be a viable solution. However, designing of such a system for the composition development (mashup) domain, is not the same as designing a recommender for music, books or movie domains.

The non-trivial programming concepts (e.g., data flow, control flow, components, connectors, configuration settings etc.) of mashup development, as explained in Figure 1.1, make the design of a suitable recommendation system more complex and challenging.

Being motivated by this challenge, in this thesis, we address the following problem statement (PS): How can we assist less-skilled developers in their mashup designs by providing them necessary support to reuse the existing composition knowledge?


URL Builder
Fetch Feed
choosing the right component
filling the correct config. parameter value
connecting components together
defining the correct data mappings
Filter
Location Extractor
Pipe Output

Base: http://ws.geonames
Port: 0
Path elements
rssToGeoRSS
Query parameters
feedUrl news.google.com/r
URL
url [wired]
Permit items that match any of the following
Rules
item.description Contains Apple
item.description Contains iPhone

Figure 1.1: Screenshot of the Yahoo! Pipes mashup environment showing typical composition steps

1.2 State of the art

In the context of *web mashups*, several works aim to assist less skilled developers in the design of mashups. Syntactic approaches [23] suggest modeling constructs based on syntactic similarity (comparing output and input data types), while semantic approaches [9] annotate constructs to support suggestions based on the meaning of constructs. In programming by demonstration [1], the system aims to auto complete a process definition, starting from a set of user-selected model examples. Goal-oriented approaches [?] aim to assist the user by automatically deriving compositions that satisfy user-specified goals. Pattern-based development [4] aims at recommending connector patterns (so-called glue patterns) in response to user selected components (so-called mashlets) in order to autocomplete the partial mashup. The limitation of these approaches is that they partly overestimate the skills of less skilled developers, as they either still require advanced modeling skills (which users don't have), or they expect the user to specify complex rules for defining goals (which they are not able to), or they expect domain experts to specify and maintain complex semantic networks describing modeling constructs (which they don't do).

The business process management community more specifically focuses on patterns as a means for knowledge reuse. Among the related works for applying or weaving recommended patterns, the automated pattern application approach [5] uses structural properties of the current composition model to tell the user which pattern (simple merge, exclusive choice, parallel branch, and similar) among the workflow patterns introduced by Van Der Aalst et al. [22] are applicable in the current modeling context. The structural properties of the workflow patterns are verified against the current process model structure to check their applicability. These control flow patterns are not able to capture domain knowledge of the underlying mashup applications, and hence are not contextual.

The syntax-based assistance approach proposed in [8] recommends the user a set of workflow patterns based on his current process model, and, once a user selects one of the recommended patterns, weaves the pattern by considering the structural compatibility among modeling constructs (e.g., a gateway must be followed by an activity in the current composition). However, this approach is limited to only block-structured models, and also the instance level information of a composition model (e.g., an activity of type A must be followed by an activity of type B, and so on) is not captured in the recommended patterns.

To address the limitations as identified in the existing related works, in our research, we aim at designing a more generic knowledge reuse approach, that can also be applicable to design models that are not block-structured. In addition to the structural compatibility, we also consider the underlying mashup language (data flow based mashup) while capturing the knowledge to be reused for the assistance. In summary, in this research we design a more extensive assistance mechanism that not only interactively recommends reusable composition knowledge that are applicable to the current application context, but also enables the *one-click* knowledge reuse approach by applying (weaving) the knowledge in the current composition context on behalf of a user. The detailed analysis of the state-of-the-art, for each of the research aspect as addressed in this research, can be found in corresponding papers as listed in appendices (A-J).

1.3 Research challenges

To find a solution to the problem statement **PS** and to address the gap identified in the existing state-of-the-art approaches, the specific research challenges (RC) that we address in this research are:

- **RC1**: How to design a conceptual model for the interactive development recommendation system that supports the reuse of existing composition knowledge for developing mashups in a step-by-step manner.
- **RC2**: How to represent the composition knowledge that captures the typical modeling steps in mashup designs.
- **RC3**: How to design a set of algorithms that support interactive and contextual retrieval of composition knowledge at runtime.
- **RC4**: How to design a set of algorithms that automate the application (weaving) of the modeling edits (addition or deletion of components, connectors etc.), captured in a composition knowledge, to the current composition context.
- **RC5**: How to implement a system that can help us to integrate the solutions for RC1, RC2, RC3 and RC4 and that also helps us to evaluate the usefulness of our interactive recommendation approach.
- **RC6**: How to assess the usability and the accuracy of our recommendation algorithms and our system via user studies and empirical tests.

1.4 Research steps followed

The summary of how we proceeded in our research exploration and the research methodology that we followed for the execution of this research are described below:

- The foundation of this thesis is based upon a set of hypotheses that emphasize the benefit of the assisted reuse of existing composition knowledge in end user development.
- We confirmed the validity of our initial hypothesis by performing an early user study with a low-fidelity mockup of our perceived assisted mashup platform. This process helped us in eliciting further requirements for the development of new set of recommendation algorithms to aid less-skilled users during the development process.

- By analyzing the existing state of the art related to assistance mechanism across multiple domains, we found a set of novel research challenges that are not yet addressed in the existing body of research and that we addressed in this research work.
- Based upon the elicited requirements, we defined the theoretical foundation (conceptual model of interactive pattern recommendation, pattern representation) of our assisted mashup platform, and then we proposed a set of novel interactive recommendation and an automated weaving algorithm that could satisfy the requirements elicited before and the theoretical foundation defined before.
- To validate the benefit of our proposed algorithms, we developed a prototype system (Baya) that help us in collecting the necessary data for measuring the viability and usability of our approach among the target user groups.
- We further elicited requirements for the improvements (e.g., incorporating user preference aspects for filtering the recommendations) of our algorithms by performing thorough empirical test of our algorithms.
- We performed user studies with the target user groups to validate and fine tune the initial hypotheses underlying our approach.
- Finally by disseminating the results of this thesis across several mashup platforms (Yahoo! Pipes, Apache Rave `http://rave.apache.org/`, MyCocktail `http://www.ict-romulus.eu/web/mycocktail`), we proved the generality as well as the suitability of our approach in end user development paradigm in general, mashup development in particular.

1.5 Contributions

The contributions of this thesis span across theoretical and algorithmic aspects of interactive, contextual development assistance that address all related research challenges listed in Section 1.3.

1.5.1 Theoretical foundation for the assisted re-use of pattern based composition knowledge

The primary goal of the *interactive pattern recommender* is to assist users in designing mashups in a step-by-step manner. The knowledge we want to recommend is reusable 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 [13]). The structure and types of composition patterns that capture the typical design steps in a mashup environment are the core knowledge behind our step-by-step recommendation approach.

Composition pattern types. Composition patterns capture the typical modeling steps performed by a developer (e.g., filling input fields, connecting components etc. as shown in Figure 1.1) in a Pipes-like mashup tool. By analyzing the model of existing mashups [14], we specifically identified following five types of composition patterns.

- *Parameter value pattern.* The parameter value pattern represents a set of value assignments for the input parameters of a component. This pattern helps filling input parameters of a component that require explicit user input.
- *Connector pattern.* The connector pattern represents a connector between a pair of components, along with the data mapping of the target component. This pattern helps connecting a newly placed component to the partial mashup model in the canvas.
- *Component co-occurrence pattern.* The component co-occurrence pattern captures pairs of components that occur together. It comes with two associated components as well as with their connector, parameter values, and data mapping logic. This pattern helps developing mashups incrementally in a connected fashion.
- *Component embedding pattern.* The component embedding pattern captures which component is typically embedded into which other component, both being preceded by another component. This pattern helps, for instance, modeling *loops*, a task that is usually not trivial to non-experts.
- *Multi-component pattern.* The multi-component pattern represents model fragments that are composed of one or more of the patterns as described before.

This list of pattern types is not exhaustive, but it contains the most representative steps in mashup designs. These patterns help in understanding the domain knowledge and the best practices as well as keeping agreed-upon modeling conventions. To read more about the basic intuitions behind coming up with this set of composition patterns, one can refer to our earlier work [14].

Conceptual model for interactive pattern recommender. Figure 1.2 depicts the conceptual model of our interactive pattern recommendation approach. The “white part” in the right hand side of Figure 1.2 represents the condition or context under which

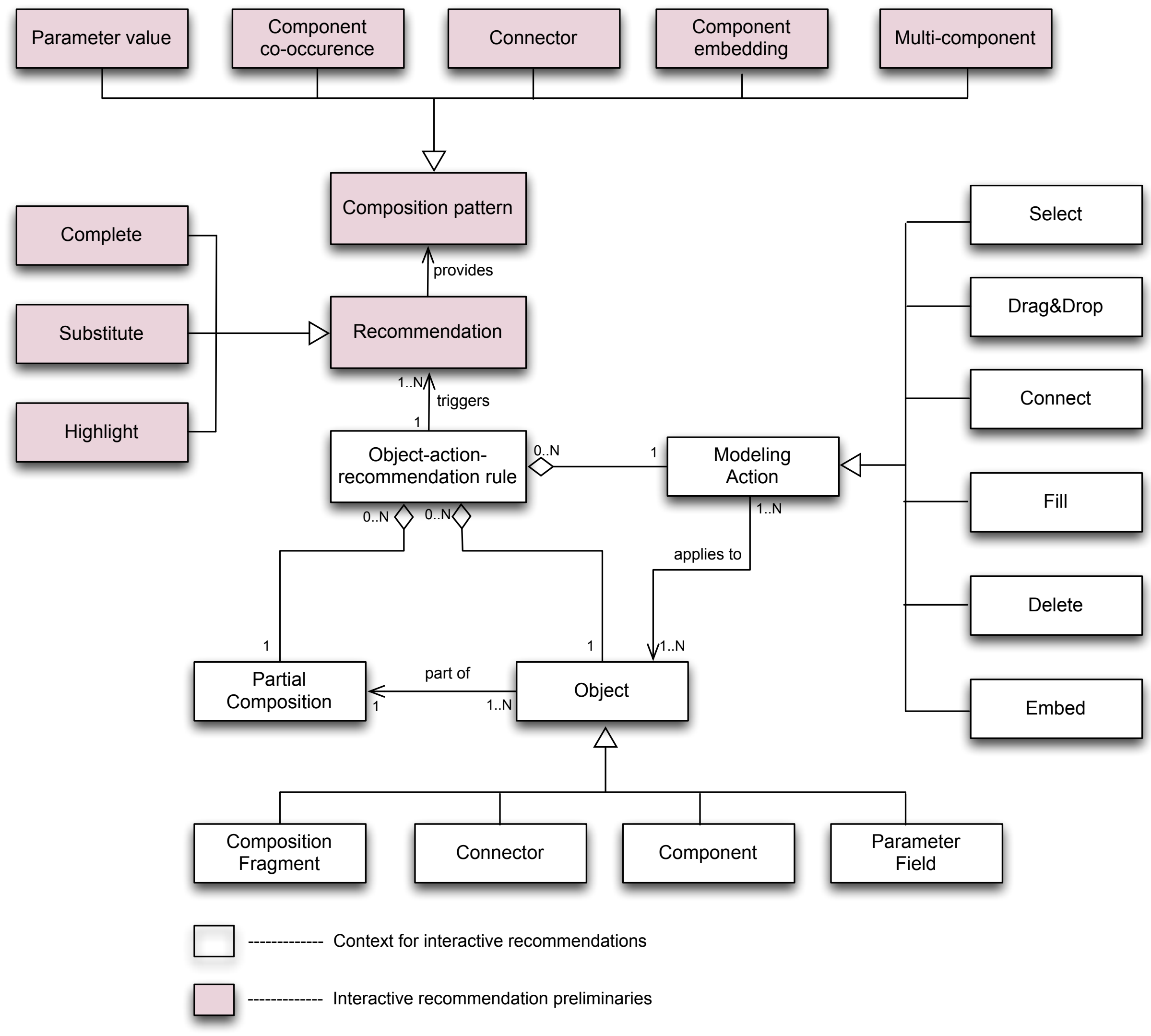


Figure 1.2: Conceptual model of interactive pattern recommendation approach

a recommendation is triggered by the recommendation algorithm. For example, applying a modeling action *fill* to an object of type *parameter field* triggers a recommendation that consists of a parameter value pattern. The *Object-action-recommendation rule* determines the type/s of recommendations to be invoked based on the current partial composition state, modeling action and the object inside the current partial composition. The “grey part” of the Figure 1.2 represents the concepts related to the recommendation. A recommendation can be to complete a partial composition with a composition pattern or to substitute an existing component/s in the current composition with a similar one from the composition pattern, or the recommendation can highlight compatible components in the current modeling canvas. Composition pattern types are recommended to the users

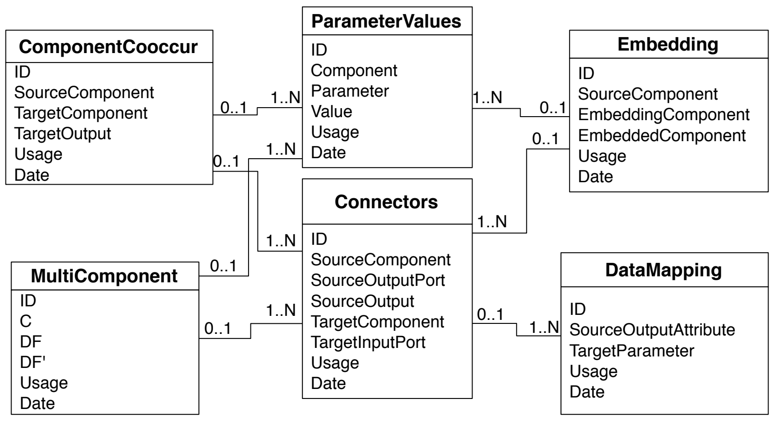


Figure 1.3: KB schema for composition patterns

in order to help them to proceed with the development steps. For more details about the conceptual model and the theoretical foundations of this thesis, we refer the reader to [13], [15].

1.5.2 Efficient, Interactive recommendation of composition patterns

The algorithmic contributions for the pattern recommender consist of a pattern knowledge base (KB) that is structured to support fast retrieval of composition patterns at runtime, an exact and an approximate search algorithm that are designed for the efficient retrieval of graph-like pattern structures and a novel user-preference based filtering and ranking algorithm that improves the accuracy of our pattern retrieval algorithm.

Pattern KB. The core of the interactive recommender is the KB that stores composition patterns, but decomposed into their constituent parts, so as to enable the incremental recommendation approach.

Figure 1.3 illustrates the structure of the pattern KB. This schema enables fast retrieval of the composition patterns with a one-shot query over a single table. 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 avoid expensive database join operations or to defer them to the moment in which we really need to retrieve all details of a pattern. 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 details.

Exact and approximate search of recommendations. Patterns in the KB are retrieved based upon an *object-action-recommendation* rule, which tells which patterns to be retrieved in the current composition context. The *object* of an action is used as a query for searching of suitable patterns from the KB. We develop a similarity based algorithm to retrieve suitable patterns for the given composition context. As for the retrieval of *similar patterns*, our goal was to *help modelers, not to disorient them.* This led us to 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 the 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). For the fast execution of similarity based algorithm we pre-process the graph structure of the original patterns, and the query object to an intermediate representation. This pre-processing step saves us from the costly graph-traversal steps at runtime. For each retrieved pattern, we compute a rank based on the pattern description (e.g., containing *usage* and *date*), the computed similarity, and the current partial mashup context. We then sort the patterns based upon their rank value and group the recommendations by their type, and filter out the top$k$ patterns for each recommendation type. Details of our pattern recommendation algorithm are described in [15].

Performance evaluation of our recommendation algorithm. To prove the efficiency of our recommendation algorithm we wished to verify its performance under different stress conditions. To perform such stress tests, we generate a realistic *test data set*, containing 130 parameter values, 650 component co-occurrence and, 3250 data-mapping, and 1000 multi-component patterns. In the *worst-case scenario* (KB of 1000 multi-component patterns, approximate similarity matching of patterns), the recommendation engine retrieves relevant patterns within 608 milliseconds (cf. Figure 1.4). Details of the performance evaluation tests are described in our previous work [15].

1.5.3 Filtering of recommended patterns by user preferences

A close investigation of users' development histories in Yahoo! Pipes reveals that users tend to use same modules/data sources across all applications he/she develops. [21] suggested to recommend items that the user already knows and likes in order to gain users *trust* on the recommendation system. Our earlier user study [3] also concluded that users like to get personalized recommendations that take into consideration their development preferences. Being motivated by these observations, in this thesis, we hypothesized that filtering of recommendations by considering user's preferences improves the overall accuracy of the system. The user's preferences are derived by analyzing his/her past development history . Figure 1.5) shows how we crawled Yahoo! Pipes existing pipes catalog to search

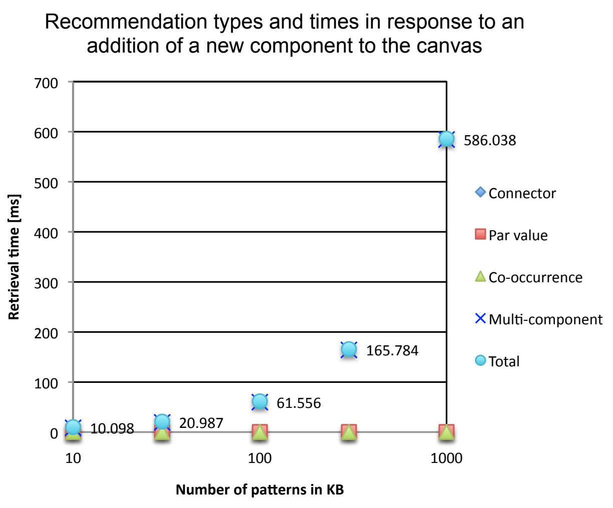
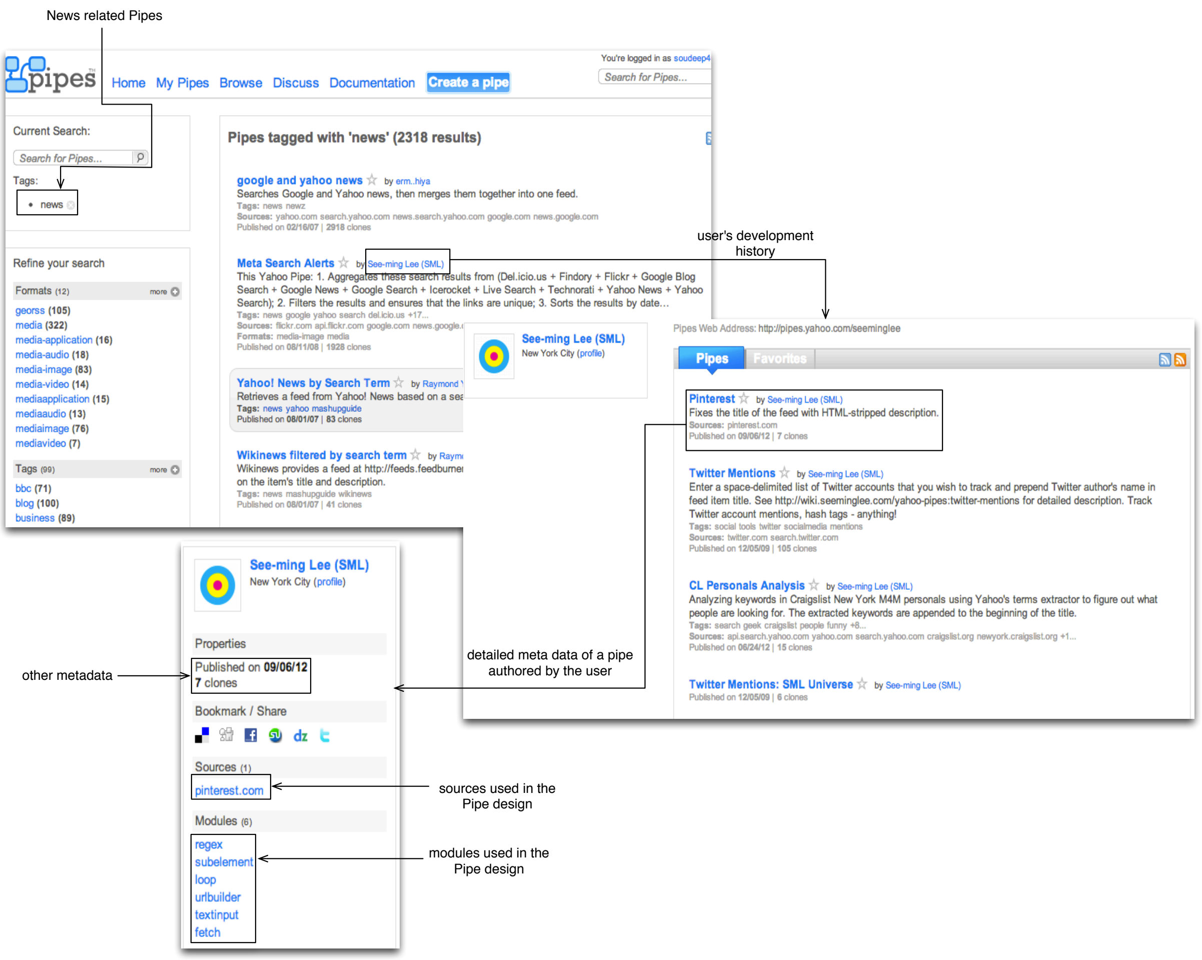


Figure 1.4: Performance evaluation for the client-side pattern recommender

for users' development histories. As shown in the figure, we first crawled all pipes in a given application domain e.g., news related pipes, then for each of the crawled pipes we extracted the author's preference information i.e., number of modules/components and data sources they have used in their pipes by crawling their development profile page containing information about pipes they have developed. Totally, we collected development profiles information for **441** unique Yahoo! Pipes users. The development profile for each user contains two matrices, one of which captures the association between the users (#441) and the modules (#51) and the other matrix contains the association between the users (#441) and the data-sources (#4781). From these association metrics we derived an implicit rating matrix for the users-modules and the users-data-sources. We identified a high sparsity in the rating matrices; a user only uses a few of the modules/data-sources available in the platform in all his/her pipes and that results in high sparsity in the user-module/data-sources rating matrix. This phenomenon motivated us to implement the alternating least square (ALS) algorithm [7], a well known matrix factorization technique that efficiently handles implicit rating dataset with high sparsity. We then incorporated the user preference based ranking of modules and data-sources as a filtering criteria for finding top-k patterns from the retrieved result of our similarity based pattern retrieval algorithm.

Empirical evaluation for calculating the accuracy of pattern recommender algorithms. To evaluate the accuracy of our recommendation algorithms (with and without preference based filtering), we followed the widely accepted recommendation system evaluation metrics such as precision, recall and F_1 metrics in our experimental setting. To calculate these metrics we had to calculate the *true positive*, *false positive* and *false*


News related Pipes
user's development history
other metadata
detailed meta data of a pipe authored by the user
sources used in the Pipe design
modules used in the Pipe design
Home
My Pipes
Browse
Discuss
Documentation
Create a pipe
Search for Pipes...
Pipes
Favorites

You're logged in as soudeep4

Current Search:

Tags:
- news

Refine your search

Formats (12) more
georss (105)
media (322)
media-application (16)
media-audio (18)
media-image (83)
media-video (14)
mediaapplication (15)
mediaaudio (13)
mediaimage (76)
mediavideo (7)

Tags (99) more
bbc (71)
blog (100)
business (89)

Pipes tagged with 'news' (2318 results)

google and yahoo news ☆ by erm..hiya
Searches Google and Yahoo news, then merges them together into one feed.
Tags: news newz
Sources: yahoo.com search.yahoo.com news.search.yahoo.com google.com news.google.com
Published on 02/16/07 | 2918 clones

Meta Search Alerts ☆ by See-ming Lee (SML)
This Yahoo Pipe: 1. Aggregates these search results from (Del.icio.us + Findory + Flickr + Google Blog Search + Google News + Google Search + Icerocket + Live Search + Technorati + Yahoo News + Yahoo Search); 2. Filters the results and ensures that the links are unique; 3. Sorts the results by date...
Tags: news google yahoo search del.icio.us +17...
Sources: flickr.com api.flickr.com google.com news.google.com
Formats: media-image media
Published on 08/11/08 | 1928 clones

Yahoo! News by Search Term ☆ by Raymond
Retrieves a feed from Yahoo! News based on a sea
Tags: news yahoo mashupguide
Published on 08/01/07 | 83 clones

Wikinews filtered by search term ☆ by Raym
Wikinews provides a feed at http://feeds.feedburner
on the item's title and description.
Tags: news mashupguide wikinews
Published on 08/01/07 | 41 clones

See-ming Lee (SML)
New York City (profile)

Pipes Web Address: http://pipes.yahoo.com/seeminglee

Pinterest ☆ by See-ming Lee (SML)
Fixes the title of the feed with HTML-stripped description.
Sources: pinterest.com
Published on 09/06/12 | 7 clones

Twitter Mentions ☆ by See-ming Lee (SML)
Enter a space-delimited list of Twitter accounts that you wish to track and prepend Twitter author's name in feed item title. See http://wiki.seeminglee.com/yahoo-pipes:twitter-mentions for detailed description. Track Twitter account mentions, hash tags - anything!
Tags: social tools twitter socialmedia mentions
Sources: twitter.com search.twitter.com
Published on 12/05/09 | 105 clones

CL Personals Analysis ☆ by See-ming Lee (SML)
Analyzing keywords in Craigslist New York M4M personals using Yahoo's terms extractor to figure out what people are looking for. The extracted keywords are appended to the beginning of the title.
Tags: search geek craigslist people funny +8...
Sources: api.search.yahoo.com yahoo.com search.yahoo.com craigslist.org newyork.craigslist.org +1...
Published on 06/24/12 | 15 clones

Twitter Mentions: SML Universe ☆ by See-ming Lee (SML)
Published on 12/05/09 | 6 clones

See-ming Lee (SML)
New York City (profile)

Properties
Published on 09/06/12
7 clones

Bookmark / Share

Sources (1)
pinterest.com

Modules (6)
regex
subelement
loop
urlbuilder
textinput
fetch

Figure 1.5: Screenshot depicting the development profile information for a user in Yahoo! Pipes

negative metrics for our test results. If the expected modules were found in the top-k list as returned by the recommendation engine then we marked the result as *true positive* (TP), if the expected module was not found in the recommended top-k list then we marked the result as *false positive* (FP), and if the algorithm was not able to return any recommendation for a step then we marked the result as *false negative* (FN). The precision, recall and the F_1 measures were calculated by using the standard formula as described below.

$$\boxed{\begin{aligned} \text{precision} &= \frac{TP}{TP+FP} \\ \text{recall} &= \frac{TP}{TP+FN} \\ \text{F1} &= \frac{2^*\text{precision}^*\text{recall}}{\text{precision}+\text{recall}} \end{aligned}} \tag{1.1}$$

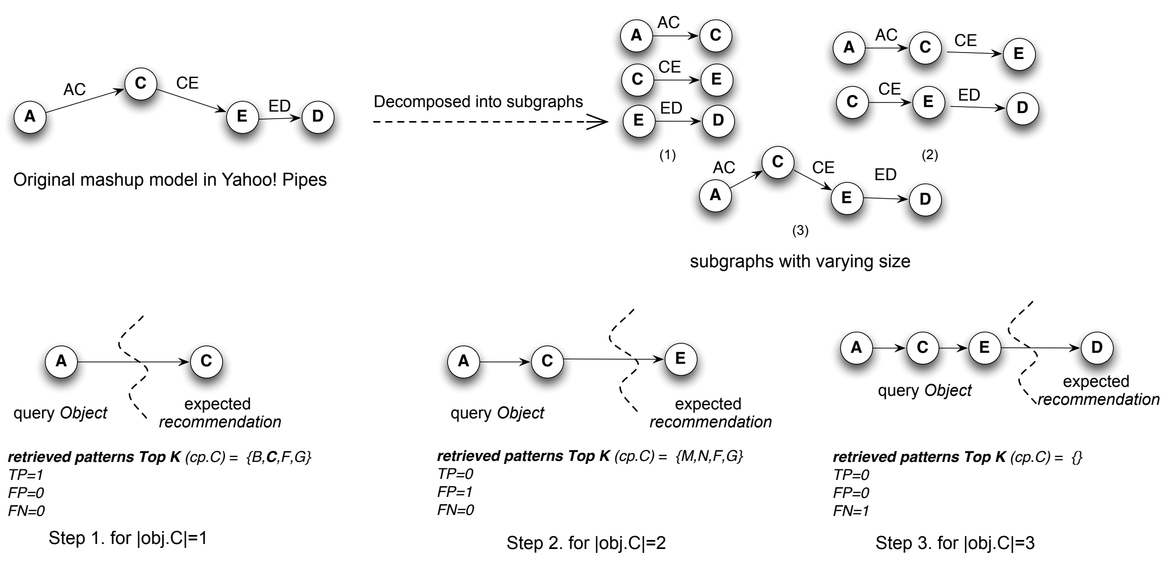

Original mashup model in Yahoo! Pipes
Decomposed into subgraphs
subgraphs with varying size
query Object
expected recommendation
retrieved patterns Top K (cp.C) = {B,C,F,G}
TP=1
FP=0
FN=0
Step 1. for |obj.C|=1
retrieved patterns Top K (cp.C) = {M,N,F,G}
TP=0
FP=1
FN=0
Step 2. for |obj.C|=2
retrieved patterns Top K (cp.C) = {}
TP=0
FP=0
FN=1
Step 3. for |obj.C|=3

Figure 1.6: Evaluation strategy for calculating the accuracy of our recommendation algorithms

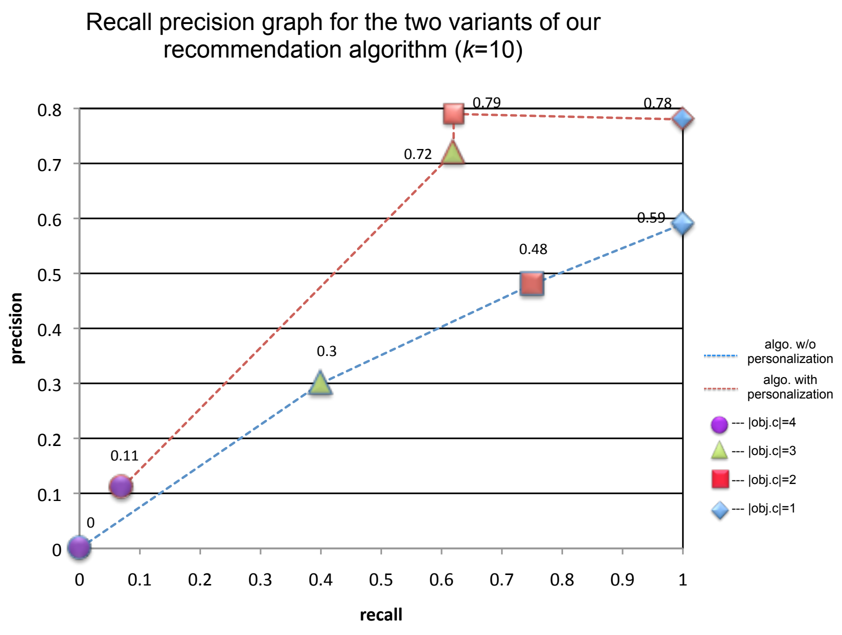


Figure 1.7: Accuracy measures for the top-10 recommendations by our algorithms

The details of the user preference based filtering algorithm and the empirical evaluation tests are explained in the paper [19].

The strategy that we followed to perform the evaluation test is described in Figure 1.6. We decomposed a set of 100 pipes into subgraphs and then used them as query for our recommendation algorithm. In this selection process for 100 test pipes, we also made sure that we didn't consider those pipes that were included for the pattern KB creation. Based upon the retrieved results we calculated the precision, recall and F_1 measures for our algorithms. We used the same evaluation strategy for both of our recommendation algorithms, with and without preference based filtering algorithm and the results of the evaluations are shown in Figure 1.7. In all of our tests, the recommendation algorithm variant that includes user preference based filtering approach *outperformed* the naive version without personalization. This verified our claim that incorporating personalization in the recommendation algorithm can improve accuracy of the overall system. The details of this test setup and detailed evaluation results can be found in the paper [19].

1.5.4 Automated weaving of composition patterns

We don't limit our system functionality to only recommending patterns that can be applied to the current composition context, but we also help users in progressing their development task by applying a selected pattern in the current model on behalf of the user. We call this functionality automated *weaving* of composition patterns. Weaving a given composition pattern into a partial mashup model is not straightforward and requires a thorough analysis of the structure and context for both the pattern and the partial mashup, in order to understand how to connect the pattern to the constructs already present in the current composition model. In essence, weaving a pattern means emulating developer interactions inside the modeling canvas, so as to connect a pattern to the partial mashup.

We approached the problem of pattern weaving by first defining a *basic weaving strategy* that consists of steps (mashup operations) required for weaving a composition pattern type. For example weaving a parameter value pattern requires a mashup operation (assigning a parameter value to a parameter field of a component) to be performed. The Basic weaving strategy is agnostic regarding the current composition context.

While applying a *basic weaving strategy* in the current composition context, we may come across model conflict situations. For example, while applying the value to a parameter field of a component, we may come across a situation in which the selected parameter field for the component already contains a value. In such a case we resolve the modeling conflict with the help of a *conflict resolution policy*, which determines whether to override an existing modeling construct (in this case the value for the parameter of the component) or to keep it. After resolving all possible modeling conflicts, finally, a set of *contextual weaving instructions* are generated which apply a pattern in the current composition context. The first discussion about our automated weaving approach is discussed in the paper [11], and a more detailed version of the related discussion is included in the paper [19].

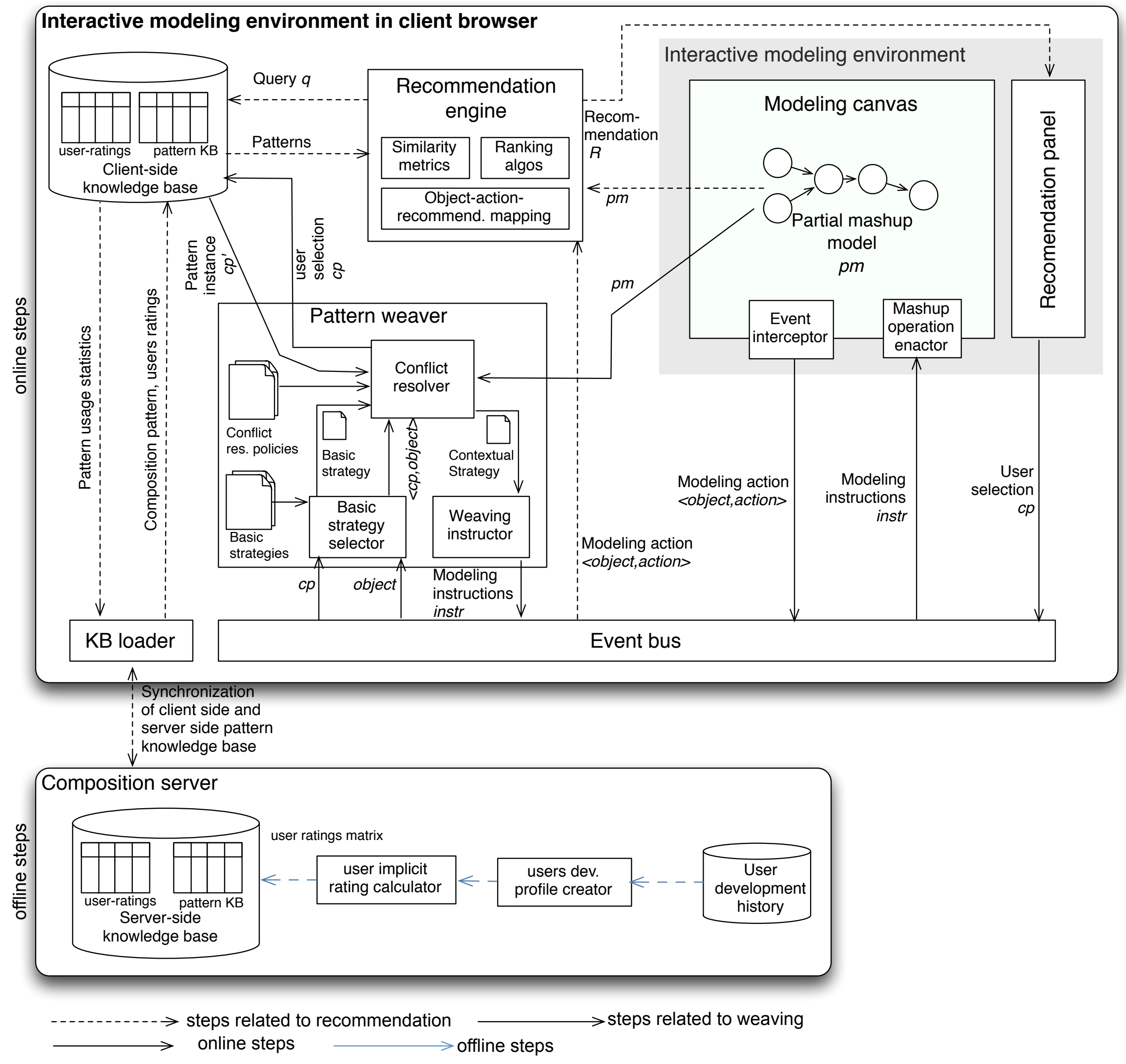


Figure 1.8: Functional architecture of the interactive pattern recommendation and automated weaving approach

1.5.5 System design for our assisted mashup development platform

Figure 1.8 (adapted from the architecture diagram presented in [2]) details the internals of our pattern recommender and weaving approach. In the server side the composition pattern KB is managed. The development profiles for users are also created and stored in the server side knowledge base. The server side knowledge and the client side KB get synchronized at runtime and then on the recommendation algorithm queries the client side knowledge base for the retrieval of the composition patterns and for the user profile information. The recommendation and weaving logic resides in the client side. We also

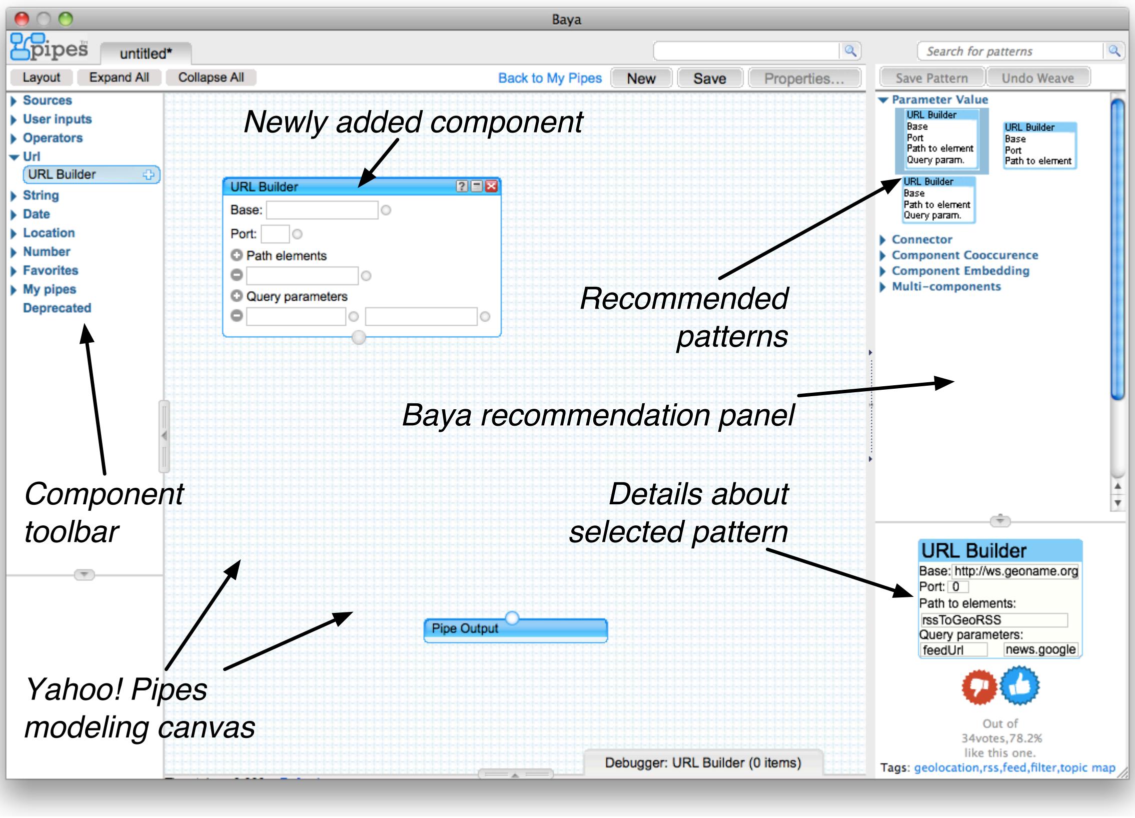


Figure 1.9: Screenshot of Baya: an assisted mashup platform as a service

distinguish between online and offline steps involved in our system.

Baya: Assisted mashup development as a service. To realize our interactive, contextual pattern recommendation and automated weaving algorithms, we developed a prototype tool named Baya [17]. Baya aims to seamlessly extend existing mashup or composition instruments with advanced knowledge reuse capabilities. It targets both expert developers and beginners and aims to speed up the former and to enable the latter. The *design goals* behind Baya can be summarized as follows: We didn't want to develop *yet another* mashup environment; so we opted for an extension of existing and working solutions (e.g., Yahoo! Pipes, Apache Rave etc.). Baya is implemented as Mozilla Firefox (`http://mozilla.com/firefox`) extension for Yahoo! Pipes, adding an interactive recommendation panel to the modeling canvas. Figure 1.9 shows a screenshot of Baya in action, to view a working demonstration of Baya's assisted mashup development approach one can refer to this screencast at `http://www.youtube.com/watch?v=AL0i40NCUmQ`.

Extension of Baya for assisting widget-based mashup. In order to make the Baya approach applicable to any browser-based mashup tool [12], we are working towards

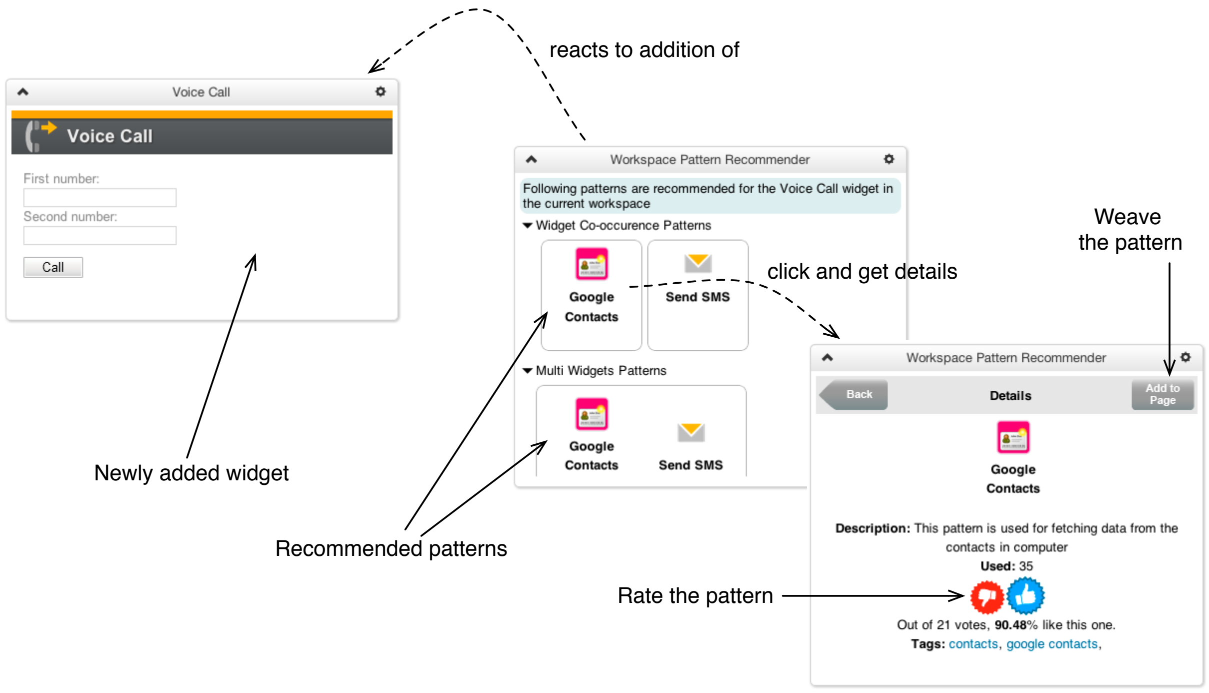
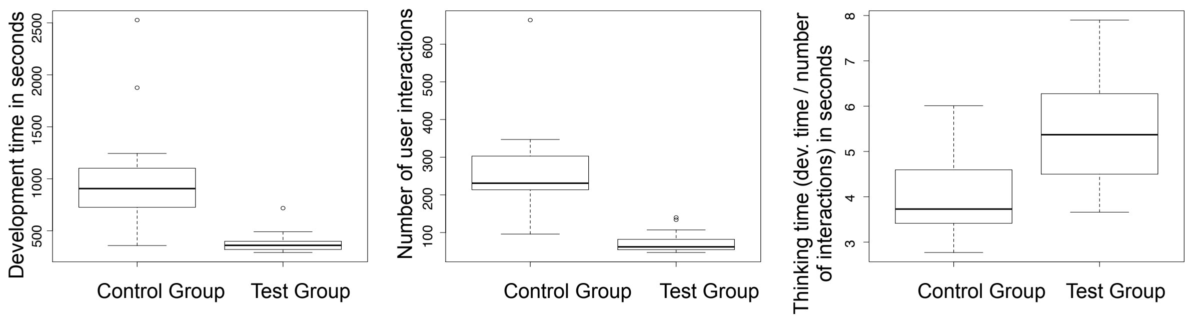

reacts to addition of
Voice Call
Voice Call
First number:
Second number:
Call
Newly added widget
Workspace Pattern Recommender
Following patterns are recommended for the Voice Call widget in the current workspace
Widget Co-occurence Patterns
Google Contacts
Send SMS
Multi Widgets Patterns
Google Contacts
Send SMS
Recommended patterns
click and get details
Weave the pattern
Workspace Pattern Recommender
Back
Details
Add to Page
Google Contacts
Description: This pattern is used for fetching data from the contacts in computer
Used: 35
Rate the pattern
Out of 21 votes, 90.48% like this one.
Tags: contacts, google contacts,

Figure 1.10: Screen shot showing of Baya extension a.k.a workspace pattern recommender widget in action

extending Baya's recommendation and weaving algorithms for other mashup tools. In the context of an European union project OMELETTE (`http://www.ict-omelette.eu/home`), we implemented a pattern recommender widget (an extension of Baya) to support W3C widget based mashup development in an open-source mashup platform Apache Rave (`http://rave.apache.org/`). The pattern recommender (cf. Figure 1.10) aids users of Apache Rave in building their mashups by reusing existing composition knowledge. To understand better how we extended Baya's pattern recommendation and weaving algorithm in the pattern recommender widget implementation, we refer the reader to [10] (page 26-31) and [18].

1.5.6 User studies and validation

Assessing the usability of Baya for end user development. To asses the effectiveness of our assisted development approach, two user studies were performed during the course of this research. The first *user study* was performed by us, with the help of Amazon's Mechanical Turk crowd sourcing platform (`https://www.mturk.com/mturk/welcome`). We specifically tested whether Baya *speeds up* the development process, *reduces* the user interactions, and reduces thinking time during the development. Details of these hypotheses are explained in [18]. We performed Welch's t-test for equal sam-



(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

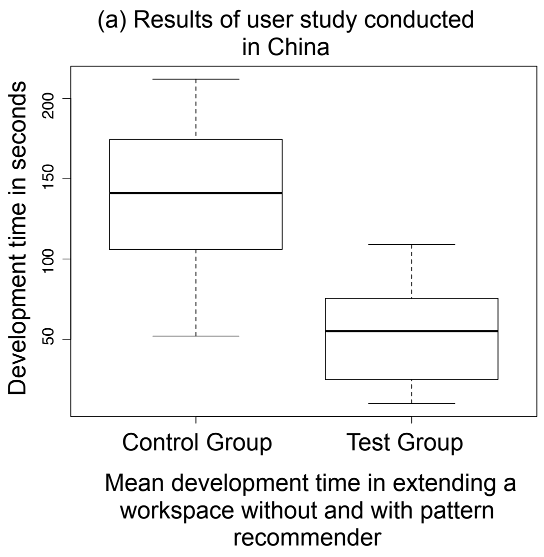
Figure 1.11: Results for Baya's usability validation test executed on the mechanical turk crowdsourcing platform with 30 participants split into a control and a test group

ple sizes and unequal variance on the evaluation results data (cf. Figure1.11) and that could statistically confirm the viability of the first two hypotheses. In response to the questionnaire that we asked users to fill in with feedbacks, the majority of the users (73% of control group and the 80% of test group) confirm the benefits of such an interactive recommendation utility in the end user development.

To cross-validate the result of our first user study another follow-up study was conducted with our protype system by T-Systems `http://www.t-systems-mms.com/` and Huwaei `http://www.huawei.com/en/` in the context of OMELETTE project. The aim of that study was again to evaluate the usefulness of our assisted development approach in an end-user mashup design scenario. The study was conducted in China and in Germany and totally 22 participants took part in that study. The study result as shown in the Figure 1.12 re-confirms the validity of our earlier assumption that Baya *speeds up* the development (mean development time of 57 sec for the test group against 137 sec for the control group). 67% of all users agreed that this feature was important or even essential for a mashup environment. Few users also reported that Baya's assistance helped them to learn the usage of new widgets, which they were unaware of. These study results not only backs our claims about the usefulness of Baya in end-user development paradigm, but they also reveal the didactic value of our interactive step-by-step assistance mechanism. Details of the study and the test results is explained in [18].

1.6 Structure of the thesis

Figure 1.13 describes the organization of chapters and contributions in this thesis. Publications as shown inside each box in the figure are structured by their interdependency.



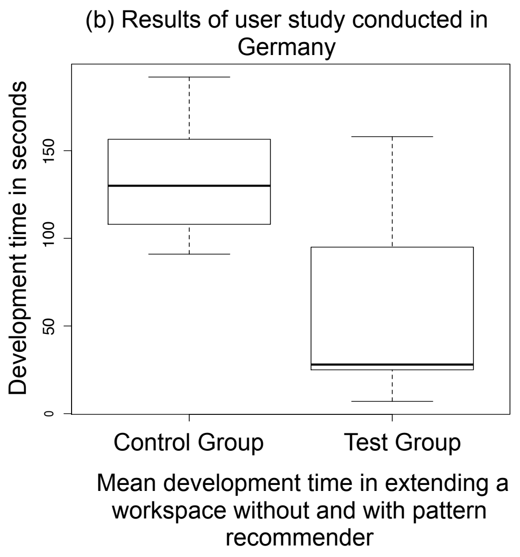


Figure 1.12: Results for usability validation test executed in China and in Germany to evaluate the benefit of pattern recommender tool

Appendix A conveys the generic motivation for this research and introduces the idea of interactive recommendation of composition knowledge. It also describes the theoretical foundation for our assisted development approach. In the paper in Appendix B, we describe how with the help of a user study, we elicited requirements for the assisted development platform. In the paper in Appendix C, we discuss types and structures for each of the composition patterns that represent the reusable knowledge. In Appendix D we discuss the algorithms behind our interactive recommendation approach with the performance evaluation test. In Appendix E we show the architecture of our assisted development platform and in Appendix F we explain our first prototype tool Baya that realizes the interactive recommendation approach as discussed and elaborated in the previous Appendices. In Appendix H we first discuss about the automated weaving algorithms. In Appendix G we proposed how we envision to support different browser-based mashup tools with our pattern recommendation and weaving algorithms. To continue this thread, in Appendix I we explain how we implemented our assisted mechanism for Apache Rave's mashup platform. This paper also discuss about a user study that is performed in China and in Germany with target users to asses the usefulness of our approach in an end-user development scenario. Finally Appendix J elaborates all our contributions (recommendation algorithm and weaving algorithm) in deeper details. It also shows different versions of our recommendation algorithm with detailed empirical results. This also paper also reports another user study that we performed on Mechanical Turk's crowd-sourced platform to assess the viability of three hypotheses behind our interactive pattern recommendation approach.

How to read this thesis

Depending on the reader's interest in details, we suggest three different ways of reading these papers:

- The complete picture of the work done in this thesis and its evolution from the first ideas to the final test of the algorithms, the lesson learned and the conclusion are summarized in Chapter 1 and are described in all details in the papers attached as appendices (A-J).
- The core contributions of the work are best described in publications [11; 15; 19].
- Implementation details for tools, which demonstrate the realization of the our interactive pattern recommendation algorithm and the automated weaving approach are described in [10; 16].

1.7 Conclusion

1.7.1 Dissemination

During the course of this research we disseminated the interim results and the knowledge gathered in several reputed international conferences and journals (cf. appendices A-J). In terms of applications, we developed an open source tool *Baya*, as a Firefox extension for Yahoo! Pipes. The datasets that we have collected for our experimentations (both the pattern KB and the developer profiles in Yahoo! Pipes) is made available for the benefit of larger research communities. In the context of an European Union project OMELETTE (`http://www.ict-omelette.eu/home`), we used the results of our interactive pattern based recommendation and automated weaving algorithms by implementing assistance plugin tools for two actively *open-source* mashup platforms Apache Rave and MyCocktail. This thesis also produces one bachelor thesis and one international recognition (winner for Graduate Category in ACM Student Research Competition 2012 `http://src.acm.org/winners.html`) for the work entitled "Assisting Mashup Development in Browser Based Modeling Tool" [12].

1.7.2 Limitations and future work

We believe that this dissertation has initiated a new line of research in the field of assisted end user development by reusing existing composition knowledge. However, similar to any other work, our proposal has some limitations that we discuss in this section. Some of these limitations can be handled by further analysis of the given methods or extending the experimental study, while others are intrinsic in the design of our models and techniques, and may be addressed by adopting different approaches and/or technical tools.

Some of the limitations that we have identified and are working toward addressing them in our future works are listed below:

- **Coverage of the pattern KB**. As reported in Figure 1.7, the accuracy of our recommendation algorithm falls down sharply once we go beyond the query object size of 3. This is due to the low coverage of our pattern KB for patterns with size more than 4. This is partly because while building the knowledge base for the initial prototype system, we only considered a subset of pipes (tagged as *"most popular"* pipes) as an input for the pattern mining algorithm, that we considered as the source for the composition pattern. The high support value required for identifying multi-component patterns with many components (more than 4) by the pattern mining algorithm also contribute to this low coverage problem.

  To address the pattern KB's coverage issue, in our future work we are planning to crawl the *entire mashup repository* of Yahoo! Pipes, which has several thousands of pipes models available. We also wish to explore other sources for composition pattern knowledge (e.g., knowledge contributed by domain experts) in the future design of our system. As a step towards that, we are working towards defining algorithms for finding expert users in a development platform like Yahoo! Pipes. By analyzing these expert users' mashup designs we can further identify knowledge for our recommendation algorithm. This process is not straight-forward and has it's own research challenges. In our future work we are going to address them one-by-one. We also want to cross-verify the effect of different sources of composition knowledge (e.g., knowledge from the expert users vs knowledge mined from the repository) on the quality of recommendations.

- **Usability issues for the consumption of recommendations**. Accepting/rejecting a recommendation as suggested by any recommendation system requires users to understand the recommended pattern (details of what is recommended and why it is recommended). In our tool we show users the details of a recommended patterns in the form of a preview of a pattern and the associated meta data (e.g., how many users used a pattern or liked a pattern). However, we believe, that to enhance the usability and transparency of our recommendation approach, more work is required in terms of representation of recommendations in the UI.

  As a solution for this limitation, more work is required to understand the most user friendly and intuitive representation for composition patterns inside the UI. However, this is a highly tool-specific issue and, hence, requires different solutions for different mashup environments. In addition to this, we are also working on improving the usability of our recommendation system by providing more *explanations* (why a pattern is suggested, and what would a pattern do in the current context) about the suggested recommendations. We believe that an explanation along with each recommended steps will increase the transparency of our recommendation system to users.

- **Limiting the novelty of recommendations**. Personalization or preference-based filtering of recommendations certainly increases the relevance of recommendations to a user, but it also limits the novelty of recommendations (only preferred modules/data sources always appear in the top-k recommended list). This may limit the usefulness and the didactic aspects of a recommendation system.

  To address this issue, we are working on techniques to introduce *diversity* into recommendations. We hypothesize that a right balance of novelty and personalization in recommendations can make our recommendation system more valuable to users.

- **Scalability of the personalized recommendation algorithm**. New users (for whom the system doesn't have the development preference information) of our assisted development platform poses challenges to the personalized recommendation algorithm. Since the system has no knowledge about the ratings for modules and data-sources for such users, it can't calculate the preference metrics for them. In order to get their development preference information, the system is required to re-execute the implicit-rating matrices calculation for all users. Re-doing this matrix factorization step for a large dataset is an expensive process, and may hinder the overall performance of our recommendation system.

  To address the scalability issue of our personalized recommendation algorithm, in our future work we will explore the applicability of incremental singular value decomposition algorithm [20] and other state-of-the-art collaborative filtering techniques in the context of our recommendation algorithm.

This dissertation presents our work in supporting the reuse of pattern-based composition knowledge for mashup development. Our study provides the theoretical foundation of the assisted mashup development and introduces an efficient mechanism (interactive recommendation of composition patterns and automated weaving of composition patterns) to provide contextual development assistance to less skilled users. We designed and implemented two research prototype tools that realize our interactive development recommendation approach for two different mashup platforms. We performed thorough performance and accuracy evaluation tests to demonstrate the efficiency of our recommendation algorithm. We also reported the results of two evaluation tests which were performed to assess the value of our approach for two different mashup tools and target users. The results of the tests confirm the applicability and the usefulness of our approach in the mashup development domain. During this research, we learned that just like recommendation systems in other domains, the knowledge (composition patterns) is the key behind the usefulness for an assisted platform in the mashup development; when the knowledge is recommended (interactivity, efficiency) and how it is recommended (knowledge representation in the UI), determine the usefulness of an assisted development approach to it's users.

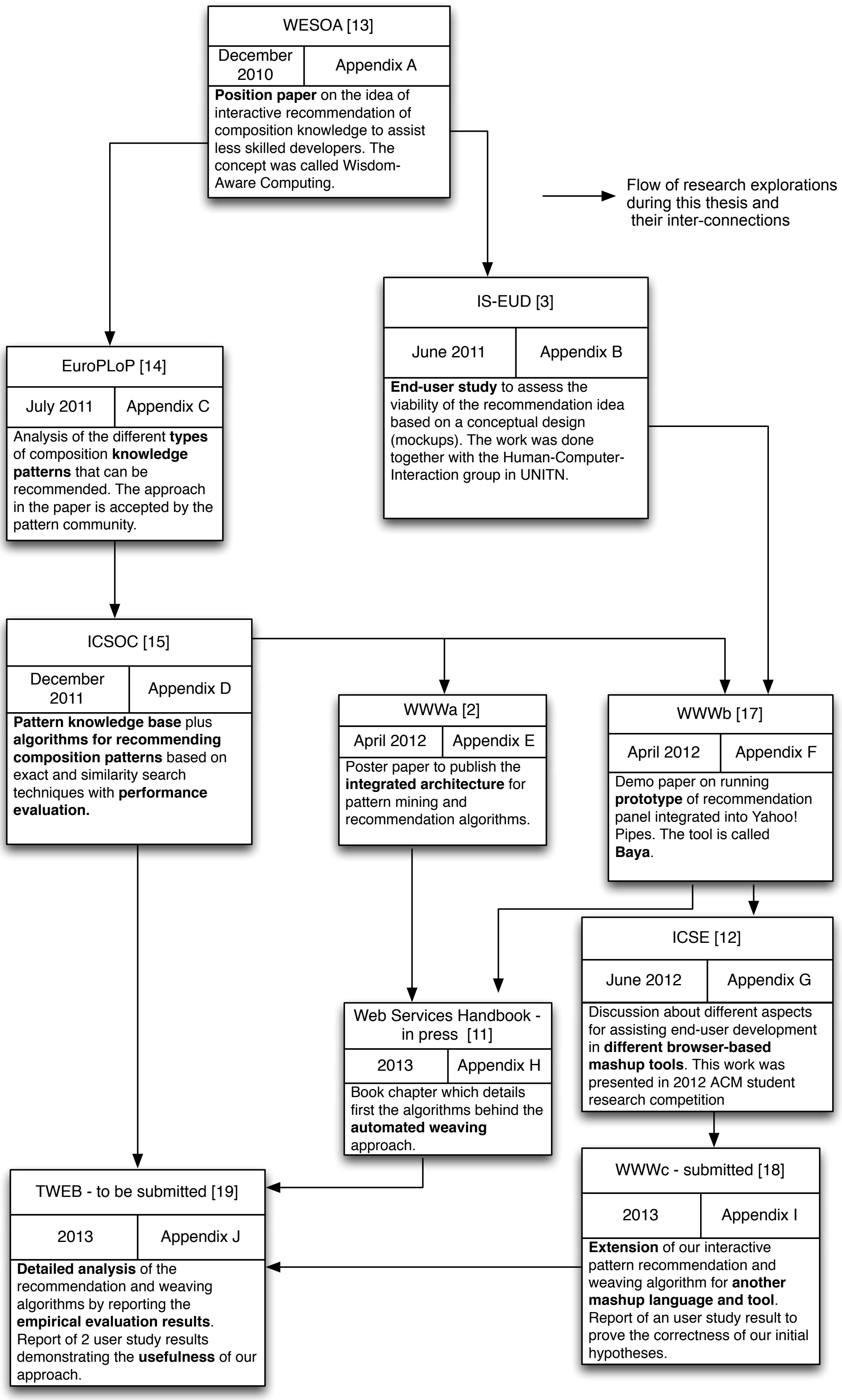

WESOA [13]
December 2010
Appendix A
Position paper on the idea of interactive recommendation of composition knowledge to assist less skilled developers. The concept was called Wisdom-Aware Computing.
Flow of research explorations during this thesis and their inter-connections
IS-EUD [3]
June 2011
Appendix B
End-user study to assess the viability of the recommendation idea based on a conceptual design (mockups). The work was done together with the Human-Computer-Interaction group in UNITN.
EuroPLoP [14]
July 2011
Appendix C
Analysis of the different types of composition knowledge patterns that can be recommended. The approach in the paper is accepted by the pattern community.
ICSOC [15]
December 2011
Appendix D
Pattern knowledge base plus algorithms for recommending composition patterns based on exact and similarity search techniques with performance evaluation.
WWWa [2]
April 2012
Appendix E
Poster paper to publish the integrated architecture for pattern mining and recommendation algorithms.
WWWb [17]
April 2012
Appendix F
Demo paper on running prototype of recommendation panel integrated into Yahoo! Pipes. The tool is called Baya.
ICSE [12]
June 2012
Appendix G
Discussion about different aspects for assisting end-user development in different browser-based mashup tools. This work was presented in 2012 ACM student research competition
Web Services Handbook - in press [11]
2013
Appendix H
Book chapter which details first the algorithms behind the automated weaving approach.
WWWc - submitted [18]
2013
Appendix I
Extension of our interactive pattern recommendation and weaving algorithm for another mashup language and tool. Report of an user study result to prove the correctness of our initial hypotheses.
TWEB - to be submitted [19]
2013
Appendix J
Detailed analysis of the recommendation and weaving algorithms by reporting the empirical evaluation results. Report of 2 user study results demonstrating the usefulness of our approach.

Figure 1.13: Timeline of scientific, peer-reviewed publications describing the major contributions of this thesis and their interdependencies

Bibliography

[1] Cypher, Allen; Halbert, Daniel C.; Kurlander, David; Lieberman, Henry; Maulsby, David; Myers, Brad A., and Turransky, Alan, editors. *Watch what I do: programming by demonstration.* MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-03213-9.

[2] Daniel, Florian; Rodriguez, Carlos; Roy Chowdhury, Soudip; Motahari Nezhad, Hamid R., and Casati, Fabio. Discovery and reuse of composition knowledge for assisted mashup development. In *Proceedings of the 21st international conference companion on World Wide Web*, WWW '12 Companion, pages 493–494, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1230-1. doi: 10.1145/2187980.2188093. URL `http://doi.acm.org/10.1145/2187980.2188093`.

[3] De Angeli, Antonella; Battocchi, Alberto; Roy Chowdhury, Soudip; Rodríguez, Carlos; Daniel, Florian, and Casati, Fabio. End-User Requirements for Wisdom-Aware EUD. In *IS-EUD'11*, pages 245–250, 2011.

[4] Deutch, Daniel; Greenshpan, Ohad, and Milo, Tova. Navigating in complex mashed-up applications. *Proc. VLDB Endow.*, 3(1-2):320–329, September 2010. ISSN 2150-8097. URL `http://dl.acm.org/citation.cfm?id=1920841.1920885`.

[5] Gschwind, Thomas; Koehler, Jana, and Wong, Janette. Applying patterns during business process modeling. In *BPM'08*, pages 4–19. Springer, 2008. ISBN 978-3-540-85757-0. doi: http://dx.doi.org/10.1007/978-3-540-85758-7_4. URL `http://dx.doi.org/10.1007/978-3-540-85758-7_4`.

[6] Henneberger, Matthias; Heinrich, Bernd; Lautenbacher, Florian, and Bauer, Bernhard. Semantic-Based Planning of Process Models. In *Multikonferenz Wirtschaftsinformatik'08*, 2008.

[7] Koren, Yehuda; Bell, Robert, and Volinsky, Chris. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, August 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.263. URL `http://dx.doi.org/10.1109/MC.2009.263`.

[8] Mazanek, Steffen and Minas, Mark. Business Process Models as a Showcase for Syntax-Based Assistance in Diagram Editors. In *MODELS '09*, pages 322–336, 2009.

[9] Ngu, A.H.H.; Carlson, M.P.; Sheng, Q.Z., and young Paik, Hye. Semantic-based mashup of composite applications. *IEEE TSC*, 3(1):2 –15, 2010. ISSN 1939-1374. doi: 10.1109/TSC.2010.8.

[10] OMELETTE Consortium, . D3.4 Interim MDP Prototype Implementation. Technical report, `http://www.ict-omelette.eu/c/document_library/get_file?p_l_id=48742&folderId=139741&name=DLFE-10845.pdf`, 2012.

[11] Rodríguez, Carlos; Roy Chowdhury, Soudip; Daniel, Florian; R. Motahari Nezhad, Hamid, and Casati, Fabio. *Assisted Mashup Development: On the Discovery and Recommendation of Mashup Composition Knowledge – In Press*, pages 683–708. Springer, 2013.

[12] Roy Chowdhury, Soudip. Assisting end-user development in browser-based mashup tools. In *ICSE*, pages 1625–1627, 2012.

[13] Roy Chowdhury, Soudip; Rodríguez, Carlos; Daniel, Florian, and Casati, Fabio. Wisdom-aware computing: On the interactive recommendation of composition knowledge. In *WESOA'10*, pages 144–155. Springer, 2010.

[14] Roy Chowdhury, Soudip; Birukou, Aliaksandr; Daniel, Florian, and Casati, Fabio. Composition patterns in data flow based mashups. In *Proceedings of EuroPLoP 2011*, pages 27–28, 2011.

[15] Roy Chowdhury, Soudip; Daniel, Florian, and Casati, Fabio. Efficient, Interactive Recommendation of Mashup Composition Knowledge. In *ICSOC'11*, pages 374–388, 2011.

[16] Roy Chowdhury, Soudip; Rodríguez, Carlos; Daniel, Florian, and Casati, Fabio. Baya: Assisted Mashup Development as a Service. In *WWW'12*, 2012.

[17] Roy Chowdhury, Soudip; Rodríguez, Carlos; Daniel, Florian, and Casati, Fabio. Baya: assisted mashup development as a service. In *Proceedings of the 21st international conference companion on World Wide Web*, WWW '12 Companion, pages 409–412, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1230-1. doi: 10.1145/2187980.2188061. URL `http://doi.acm.org/10.1145/2187980.2188061`.

[18] Roy Chowdhury, Soudip; Chudnovskyy, Olexiy; Niederhausen, Matthias; Pietschmann, Stefan; Sharples, Paul; Daniel, Florian, and Gaedke, Martin. Complementary assistance mechanisms for end user mashup composition– submitted. In *WWW ' 13*, 2013.

[19] Roy Chowdhury, Soudip; Daniel, Florian, and Casati, Fabio. Interactive recommendation and weaving of mashup model patterns for assisted mashup development– to be submitted. In *ACM Transactions on the Web*, 2013.

[20] Sarwar, Badrul; Karypis, George; Konstan, Joseph, and Riedl, John. Incremental singular value decomposition algorithms for highly scalable recommender systems. In *Fifth International Conference on Computer and Information Science*, pages 27–28, 2002.

[21] Shani, Guy and Gunawardana, Asela. Evaluating recommendation systems. In *Recommender Systems Handbook*, pages 257–297. 2011.

[22] Van Der Aalst, W. M. P.; Ter Hofstede, A. H. M.; Kiepuszewski, B., and Barros, A. P. Workflow patterns. *Distrib. Parallel Databases*, 14:5–51, July 2003. ISSN 0926-8782.

[23] Wong, Jeffrey and Hong, Jason I. Making mashups with marmite: towards end-user programming for the web. In *CHI'07*, pages 1435–1444. ISBN 978-1-59593-593-9. doi: http://doi.acm.org/10.1145/1240624.1240842. URL `http://doi.acm.org/10.1145/1240624.1240842`.

Appendix A

Wisdom-Aware Computing: On the Interactive Recommendation of Composition Knowledge

Soudip Roy Chowdhury, Carlos Rodrguez, Florian Daniel and Fabio Casati. Wisdom-Aware Computing: On the Interactive Recommendation of Composition Knowledge. Proceedings of WESOA 2010, December 2010, Springer.

Wisdom-Aware Computing: On the Interactive Recommendation of Composition Knowledge

Soudip Roy Chowdhury, Carlos Rodríguez, Florian Daniel and Fabio Casati
University of Trento, Via Sommarive 14, 38123 Povo (TN), Italy
{rchowdhury,crodriguez,daniel,casati}@disi.unitn.it

Abstract. We propose to enable and facilitate the development of service-based development by exploiting *community composition knowledge*, i.e., knowledge that can be harvested from existing, successful mashups or service compositions defined by other and possibly more skilled developers (the *community* or *crowd*) in a same domain. Such knowledge can be used to assist less skilled developers in defining a composition they need, allowing them to go beyond their individual capabilities. The assistance comes in the form of *interactive advice*, as we aim at supporting developers while they are defining their composition logic, and it adjusts to the skill level of the developer. In this paper we specifically focus on the case of *process-oriented, mashup-like applications*, yet the proposed concepts and approach can be generalized and also applied to generic algorithms and procedures.

1 Introduction

Although each of us develops and executes various procedures in our daily life (examples range from cooking recipes to low-level programming code), today very little is done to support others, possibly *less skilled developers* (or, in the extreme case, even end users) in developing their own. Basically, there are two main approaches to **enable less skilled people** to "develop": either development is eased by *simplifying* it (e.g., by limiting the expressive power of a development language) or it is facilitated by *reusing knowledge* (e.g., by copying and pasting from existing algorithms).

Among the **simplification** approaches, the workflow and BPM community was one of the first to claim that the abstraction of business processes into tasks and control flows would allow also the less skilled users to define own processes, however with little success. Then, with the advent of web services and the service-oriented architecture (SOA), the web service community substituted tasks with services, yet it also didn't succeed in enabling less skilled developers to compose services. Recently, web mashups added user interfaces to the composition problem and again claimed to target also end users, but mashup development is still a challenge for skilled developers. While these attempts were aimed at simplifying technologies, the human computer interaction community has researched on end user development approaching the problem from the user interface perspective. The result is simple applications that are specific to a very limited domain, e.g., an interactive game for children, with typically little support for more complex applications.

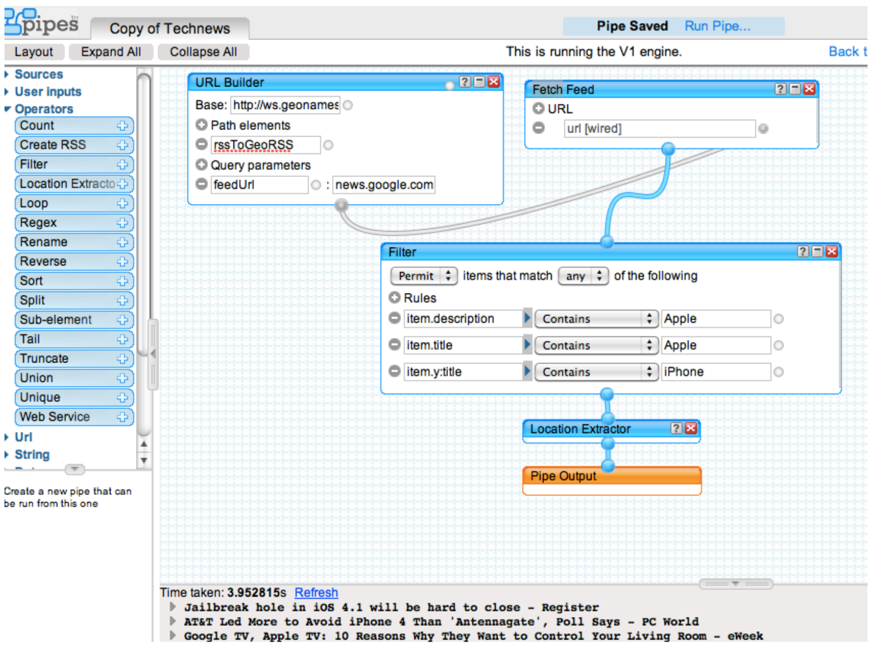
As for what regards **capturing and reusing knowledge**, in IT reuse typically comes in the form of program libraries, services, or program templates (such as generics in Java or process templates in workflows). In essence, what is done today is either providing building blocks that can be composed to achieve a goal, or providing the entire composition (the algorithm – possibly made generic if templates are used), which may or may not suit a developer's needs. In the nineties and early 2000s, AI planning [1] and automated, goal-oriented compositions (e.g., as in [2]) became popular in research. A typical goal there is to derive a service composition from a given goal and a set of components and composition rules. Despite the large body of interesting research, this thread failed to produce widely applicable results, likely because the goal is very ambitious and because assumptions on the semantic richness and consistency of component descriptions are rarely met in practice. Other attempts to extract knowledge are, for example, oriented at identifying social networks of people [3] or at providing rankings and recommendations of objects, from web pages (Google's Pagerank) to goods (Amazon's recommendations). An alternative approach is followed by expert recommender systems [4], which, instead of identifying knowledge, aim at identifying knowledge holders (the experts), based on their code production and social involvement.

In this paper, we describe **WIRE**, a *WIsdom-awaRE development environment* we are currently developing in order to enable less skilled developers to perform also complex development tasks. We particularly target process-oriented, mashup-like applications, whose development and execution can be provided as a service via the Web and whose internals are characterized by relatively simple composition logic and relatively complex tasks or components. This class of programs seems to provide both the benefit of (relative) simplicity and a sufficient information base (thanks to the reuse of components) to learn and reuse programming/service composition knowledge. The idea is to *learn from existing compositions* (or, in general, *computations*) and to provide the learned knowledge in form of *interactive advice* to developers while they are composing their own application in a visual editor. The aim is both to allow developers to go beyond their own development capabilities and to speed up the overall development process, joining the benefits of both simplification *and* reuse.

Next, we discuss a state of the art composition scenario and we show that it is everything but trivial. In Section 3, we discuss the state of the art in assisted composition. In Section 4 and 5, we investigate the idea of composition advices and provide our first implementation ideas, respectively. Then we conclude the paper and outline our future work.

2 Example Scenario and Research Challenges

In order to better understand the problem we want to address, let's have a look at how a mashup is, for instance, composed in Yahoo! Pipes (*http://pipes.yahoo.com/pipes/*), one of the most well-known mashup platforms as of today. Let's assume we want to develop a simple pipe that sources a set of news from *Google News*, filters them according to a predefined filter condition (in our case, we want to search for news on products and services by a given vendor), and locates them on a *Yahoo! Map*.



Base: http://ws.geonames
Path elements: rssToGeoRSS
Query parameters: feedUrl : news.google.com

URL: url [wired]

Permit items that match any of the following
Rules
item.description Contains Apple
item.title Contains Apple
item.y:title Contains iPhone

Time taken: 3.952815s

```
Jailbreak hole in iOS 4.1 will be hard to close - Register
AT&T Led More to Avoid iPhone 4 Than 'Antennagate', Poll Says - PC World
Google TV, Apple TV: 10 Reasons Why They Want to Control Your Living Room - eWeek
```

Figure 1 Implementation of the example scenario in Yahoo! Pipes

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 *Geo Names* service, which takes a news RSS feed as an input, analyzes its content, and inserts geo-coordinates, i.e., longitude and latitude, into each news item (where possible). Doing so requires setting some parameters: *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, which loads the geo-enriched news feed. In order to filter out the news items we are really interested in, we need to use the Filter component, which requires the setting of proper filter conditions via the *Rules* input field. Feeding the filtered feed into the *Location Extractor* component causes Pipes to plot the news items on a *Yahoo! Map*. Finally, the *Pipe Output* component specifies the end of the pipe.

If we analyze the development steps above, we can easily understand that developing even such a simple composition is out of the reach of people without programming knowledge. Understanding which components are needed and how they are used is neither trivial nor intuitive. The URL Builder, for example, requires the setting of some complex parameters. Then, components need to be suitably connected, in order to support the data flow from one component to another, and output parameters must be mapped to input parameters. But more importantly, plotting news onto a map requires knowing that this can be done by first enriching a feed with geo-coordinates, then fetching the actual feed, and only then the map is ready to plot the items.

Enabling non-expert developers to compose a pipe like the above requires telling (or teaching) them the necessary knowledge. In **WIRE**, we aim to do so by providing non-expert developers with interactive development advices for composition, inside an assisted development environment. We want to obtain the knowledge to provide advices by extracting, abstracting, and reusing compositional knowledge from existing compositions (in the scenario above, pipes) that contain community knowledge, best practices, and proven patterns. That is, in *WIRE* we aim at bringing the *wisdom of the crowd* (possibly even a small crowd if we are reusing knowledge within a company) in defining compositions when they are both defined by an individual (where the crowd supports an individual) or by a community (where the crowd supports social computing, i.e., itself in defining its own algorithms). The final goal is to move towards a new frontier of knowledge reuse, i.e., *reuse of computational knowledge*.

Doing so requires approaching a set of **challenges** that are non-trivial:

1. First of all, *identifying the types of advices that can be given and the right times when they can be given*: depending on the complexity and expressive power of the composition language, there can be a huge variety of possible advices. Understanding which of them are useful is crucial to limit complexity.
2. *Discovering computational knowledge*: how do we harvest development knowledge from the crowd, that is, from a set of existing compositions? Knowledge may come in a variety of different forms: component or service compatibilities, data mappings, co-occurrence of components, design patterns, evolution operations, and so on.
3. *Representing and storing knowledge*: once identified, how do we represent and store knowledge in a way that allows easy querying and retrieval for reuse?
4. *Searching and retrieving knowledge*: given a partial program specification under development, how do we enable the querying of the knowledge space and the identification of the most suitable and useful advice to provide to the developer, in order to really assist him?
5. *Reusing knowledge*: given an advice for development, how do we (re)use the identified knowledge in the program under development? We need to be able to "weave" it into the partial specification in a way that is correct and executable, so as to provide concrete benefits to the developer.

In this paper, we specifically focus on the *first challenge* and we provide our first ideas on the second challenge and on the assisted development environment.

3 State of the Art

In **literature**, there are approaches that aim at similar goals as WIRE, yet they mainly focus on the *retrieval* and *reuse* of composition knowledge. In [6], for instance, mashlets (the elements to be composed) are represented via their inputs and outputs, and glue patterns are represented as graphs of connections among them; reuse comes in the form of auto-completion of missing components and connections, selected by the user from a ranked list of top-k recommendations obtained starting from the mashlets used in the mashup. In [8], light-weight semantic annotations for services, feeds, and data flows are used to support a text-based search for data mashups, which are actually generated in an automated, goal-oriented fashion using AI planning (the search tags are the goals); generated data processing pipes can be used as is or further edited. The approach in [9] semantically annotates portlets, web apps, widgets, or Java beans and supports the search for functionally equivalent or matching components; reuse is supported by a semantics-aided, automated connection of components. Also the approach in [10] is based on a simple, semantic description of information sources (name, formal inputs [allowed ones], actual inputs [outputs consumed from other sources], outputs) and mashups (compositions of information sources), which can be queried with a partial mashup specification in order identify goals based on their likelihood to appear in the final mashup; goals are fed to a semantic matcher and an AI planner, which complete the partial mashup. This last approach is the only one that also automatically *discovers* some form of knowledge in terms of popularity of outputs in existing mashup specifications (used to compute the likelihoods of goals).

In the context of business process modeling, there are also some works with similar goals as ours. For instance, in [7], the authors more specifically focus on business processes represented as Petri nets with textual descriptions, which are processed (also leveraging WordNet) to derive a set of descriptive tags that can be used for search of processes or parts thereof; reuse is supported via copy and paste of results into the modeling canvas. The work presented in [13] proposes an approach for supporting process modeling through object-sensitive action patterns, where these patterns are derived from a repository of process models using techniques from association rule learning, taking into consideration not only actions (tasks), but also the business objects to which these actions are related. Finally, [14] presents a model for the reuse data mining processes by extending the CRISP-DM process [15]. The proposed model aims at including data mining process patterns into CRISP-DM and to guide the specialization and application of such patterns to concrete processes, rather than actually exploiting the community knowledge.

In general, the **discovery of community composition knowledge** is not approached by the works above (or they do it in a limited way, e.g., by deriving only behavioral patterns from process definitions). Typically, they start from an annotated representation of mashups and components and query them for functional compatibilities and data mappings, improving the quality of search results via semantics, which are explicit and predefined. WIRE, instead, specifically focuses on the elicitation and collection of *crowd wisdom*, i.e., composition knowledge that derives from the ways other people have solved similar composition problems in the past and that has a significant support in terms of number of times it has been adopted. This means that in order to create knowledge for WIRE, we do not need any expert developer or domain specialist that writes and maintains explicit composition rules or logics; knowledge is instead harvested from how people compose their very own applications, without requiring them to provide additional meta-data or descriptions (which typically doesn't work in practice).

4 Wisdom-Aware Development: Concepts and Principles

Identifying which advices can be provided and which advices do indeed have the potential to help less skilled developers to perform complex development tasks requires, first of all, understanding the *expressive power* of the composition language at hand. We approach this task next. Then we focus on the advices.

4.1 Expressiveness of the Composition Language

Let us consider again Yahoo! Pipes. The platform has a very advanced and pleasant user interface for drag-and-drop development of data mashups and supports the composition of also relatively complex processing logics. Yet, the strong point of Pipes is its *data flow based composition paradigm*, which is very effective and requires only a limited set of modeling constructs. As already explained in the introduction, constraining the expressive power of composition languages is one of the techniques to simplify development, and Pipes shares this characteristic with most of today's mashup platforms.

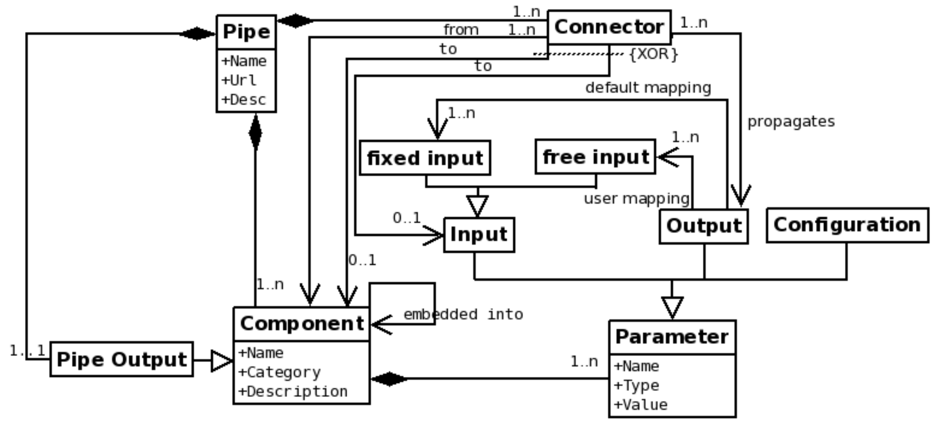


Figure 2 A meta-model for Yahoo! Pipes' composition language

In order to better understand the expressiveness of Yahoo! Pipes, in Figure 2 we derived a **meta-model** for its composition language. A *Pipe* is composed of components and connectors. *Components* have a name and a description and may be grouped into categories (e.g., source components, user input components, etc.). Each pipe contains always one *Pipe Output* component, i.e., a special component that denotes the end of data flow logic or the end of the application. A component may be *embedded* into another component; for example components (except user inputs and operators) can be embedded inside a *Loop Operator* component. Components may also have a set of parameters. A *Parameter* has a name, a type, and may have a value assigned to it. There are basically three types of parameters: *input* parameters (accept data flows attributes), *output* parameters (produce data flow attributes), and *configuration* parameters (are manually set by the developer). For instance, in our example in Section 2, the *URL* parameter of the *Fetch Feed* component is an input parameter; the *longitude* and *latitude* attributes of the RSS feed fetched by the *Fetch Feed* component are output parameters; and the *Base* parameter of the *URL Builder* component is an example of configuration parameter.

Data flows in Pipes are modeled via dedicated connectors. A *Connector* propagates output parameters of one component (indicated in Figure 2 by the *from* relationship) to either another component or to an individual input field of another component. If a connector is connected to a whole component (e.g., in the case of the connector from the *Fetch Feed* component to the *Filter* component in Figure 1), all attributes of the RSS item flowing through the connector can be used to set the values of the target component's input parameters. If a connector is connected only to a single input parameter, the data flow's attributes are available only to set the value of the target input parameter. Input parameters are of two types: either they are *fixed inputs*, for which there are predefined *default mappings*, or they are *free inputs*, for which the user can provide a value or choose which flow attribute to use. That is, for free inputs it is possible to specify a simple attribute-parameter data mapping logic.

Figure 2 shows that Yahoo! Pipes' meta-model is indeed very **simple**: only 10 concepts suffice to model its composition features. Of course, the focus of Pipes is on data mashups, and there is no need for complex web services or user interfaces, two features that are instead present in our own mashup platform, i.e., mashArt [5]. Yet, despite these two additions, mashArt's meta-model only requires 13 concepts. If instead we look at the BPMN modeling notation for business processes [11], we already need more than 20 concepts to characterize its expressive power, and the meta-model of BPEL [12] has almost 60 concepts! Of course, the higher the complexity of the language, the more difficult it is to identify and reuse composition knowledge.

4.2 Advising Composition Knowledge

Given the meta-model of the composition language for which we want to provide composition advices, it is possible to identify which concrete **compositional knowledge** can be extracted from existing compositions (e.g., pipes). The gray boxes in the conceptual model in Figure 3 illustrate the result of our analysis. The figure identifies the key entities and relationships needed to provide composition advices.

An *Advice* provides composition knowledge in form of composition patterns. An advice can be to *complete* a given pattern (given it's partial implementation in the modeling canvas) or to *substitute* a pattern with a similar one, or the advice can *highlight* compatible elements in the modeling canvas or *filter and rank* advices.

Patterns represent the actual recommendation that we deliver to the user. They can be of five types (all these patterns can be identified in the model in Figure 3):

- *Parameter Value Patterns*: Possible values for a given parameter. For instance, in the *URL Builder* component the *Base* parameter value in a pattern can be set to "http://ws.geonames.org", while the *Path elements* parameter value can be "rssToGeoRSS", and feedUrl can be "news.google.com/news?topic=t&output =rss&ned=us", as shown in our example scenario. Alternatively, we can have the *URL Builder* component with the *Base* parameter set to "news.google. com/news" and the *Query parameters* set with different values.
- *Component Association Patterns*: Co-occurrence patterns for pairs of components. For instance, in our scenario, whenever a user drags and drops the *URL Builder* on the design canvas, a possible advice derived from a component association pattern can be to include in the composition the *Fetch Feed* component and connect it to the *URL Builder.*
- *Connector Patterns*: Component-component or component-input parameter patterns. This pattern captures the dataflow logic, i.e., how components are

connected via connector elements. For example, *URL Builder – connector-Fetch Feed* is a connector pattern in our example scenario.

- *Data Mapping Patterns*: Associations of outputs to inputs. In Figure 1, for instance, we map the *description*, *title*, and *y:title* attributes of the fetched feed to the first input field of the first, second, and third rule, respectively, telling the *Filter* component how we map the individual attributes in input to the individual, free input parameters of the component.
- *Complex Patterns:* Partial compositions consisting of multiple components, connectors, and parameter settings. In our example scenario, different combinations of components and connectors, having their parameter values set and with proper data mappings, as a part and as a whole represent complex patterns. For example, the configuration *URL Builder – Fetch Feed – Filter – Location Extractor*, along with their settings, represents a complex pattern.

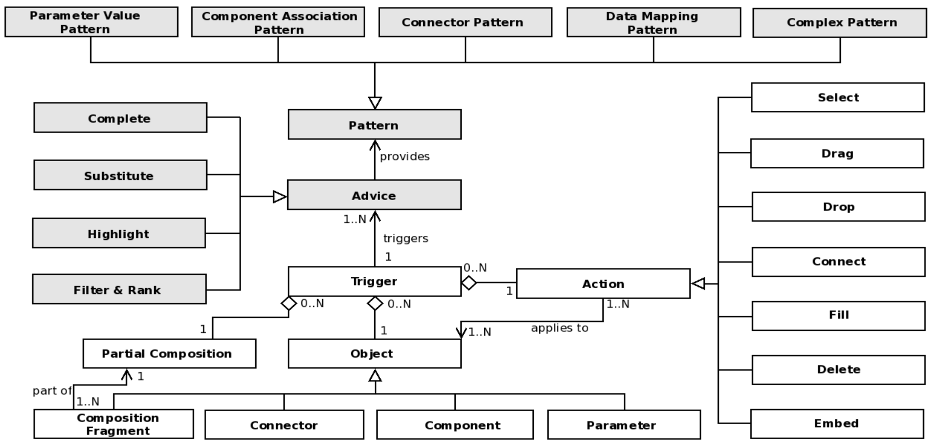


Figure 3 Conceptual model of WIRE's advice approach. Gray entities model the ingredients for advices; white boxes model the advice triggering logic inside the design environment.

An *Advice* provides composition knowledge in form of composition patterns. An advice can be to *complete* a given pattern (given it's partial implementation in the modeling canvas) or to *substitute* a pattern with a similar one, or the advice can *highlight* compatible elements in the modeling canvas or *filter and rank* advices.

Patterns represent the actual recommendation that we would like to deliver to the user. They can be of five different types: *Complex Patterns* (partial compositions possibly consisting of multiple components, connectors, and parameter settings), *Parameter Value Patterns* (possible values for a given parameter), *Component Association Patterns* (co-occurrence patterns for pairs of components), *Connector Patterns* (component-component or component-input parameter patterns), and *Data Mapping Patterns* (associations of outputs to inputs).

Now, let us discuss the "white part" of the model. This part represents the entities that jointly define the conditions under which advices can be triggered. A *Trigger* for an advice is defined by an object, an action of the user in the modeling canvas, and the state of the current composition, i.e., the partial composition in the modeling canvas. This association can be thought of as a triplet that defines the triggering condition. The *Objects* a user may operate are *Composition Fragments* (e.g., a selection of a subset of the pipe in the canvas), individual *Components*, *Connectors*, or *Parameters* (by interacting with the respective graphical input fields). The *Action* represents the action that the user may perform on an object during composition. We identify seven actions: *Select* (e.g., a composition fragment or a connector), *Drag* (e.g., a component or a connector endpoint), *Drop*, *Connect*, *Fill* (a parameter value), *Delete*, and *Embed* (one component into another). Finally, the *Partial Composition* represents the status of the current overall composition.

While the object therefore identifies *which* advice may be of interest to the user, the action decides *when* the advice can be given, and the state *filters* out advices that are not compatible with the current partial composition (e.g., if the *Location Extractor* component has already been used, recommending its use becomes useless).

Regarding the model in Figure 3, not all associations may be needed in practice. For instance, not all components are compatible with the *embed* action. Yet, the model identifies precisely which advices can be given and when.

5 The WIRE Platform

Figure 4 illustrates the high-level architecture of the assisted development environment with which we aim at supporting wisdom-aware development according to the model described in the previous section: developers can design their applications in a *wisdom-aware development environment*, which is composed of an *interactive recommender* (for development advice) and an *offline recommender* as well as the *wisdom-aware editor* implementing the interactive development paradigm. Compositions or mashups are stored in a *compositions repository* and can be executed in a dedicated *runtime environment*, which generates *execution data*. Compositions and execution data are the input for *the knowledge/advice extractor*, which finds the repeated and useful patterns in them and stores them as development and evolution advice in the *advice repository*. Then, the *recommenders* provide them as interactive advices through its query interface upon the current context and triggers of the user's development environment. Here, we specifically focused on development advices related to composition; we will approach evolution advices in our future work (evolution advices will, for instance, take into account performance criteria or evolutions applied by developers over time on their own mashups).

We realize that each domain will have suitable languages and execution engines, such as a mashup engine or a scientific workflow engine. Our goal is not to compete with these, but to define mechanism to "WIRE" these languages and tools with the ability to extract knowledge and provide advice. For this reason, in this paper we started with studying the case of Yahoo! Pipes, which is well known and allows us to easily explain our ideas. We however intend to apply the wisdom-aware development paradigm to our own mashup editor, mashArt [5], which features a *universal composition* paradigm user interface components, application logic, and data web services, a development paradigm that is similar in complexity to that of Pipes.

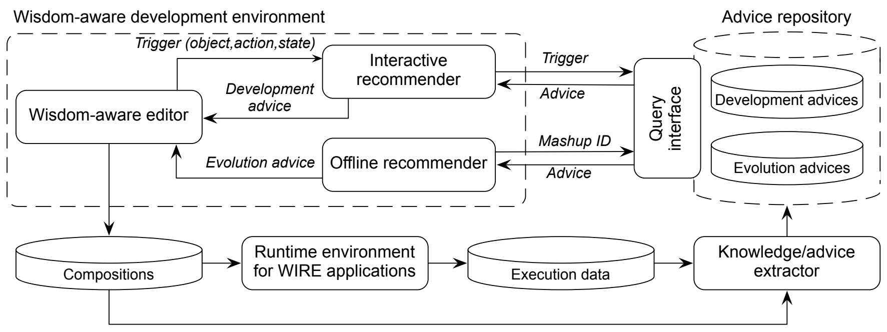


Figure 4 High-level architecture of the envisioned system for wisdom-aware development

As for the reuse of knowledge, the **WIRE approach** is not based on semantic annotations, matching, or AI planning techniques, nor do we aim at automated or goal-driven composition or at identifying semantic similarity among services. We also do not aim at having developers tag components or add metadata to let others better reuse services, processes, or fragments. In other words, we aim at collecting knowledge *implicitly*, as we believe that otherwise we would face an easier wisdom extraction problem but end up with a solution that in practice does not work because people do not bother to add the necessary metadata. WIRE will rather leverage on statistical data analysis techniques and data mining as means to extract knowledge from the available information space. To do so, we propose the following core steps:

1. *Cleaning, integration, and transformation:* We take as input previous compositions and execution data and prepare them for the analysis.
2. *Statistical data analysis and data mining:* On the resulting data, we apply statistical data analysis and data mining techniques, which may include mining of frequent patterns, association rules, correlations, classification and cluster analysis. The results of this step are used to create the composition patterns.
3. *Evaluation and ranking of advices (knowledge):* Once we have discovered the potential advices, we evaluate and rank them using standard interestingness measures (e.g., support and confidence) and ranking algorithms.
4. *Presentation of advices:* The advices are presented to the user through intuitive visual metaphors that are suitable to the context and purpose of the advice.
5. *Gathering of user feedback:* The popularity of advices is gathered and measured in order to better rank them.

Among the techniques we are applying for the discovery tasks, we are specifically leveraging on data mining approaches, such as *frequent itemset mining*, *association rules learning*, *sequential pattern mining*, *graph mining,* and *link mining*. Each of these techniques can be used to discover a different type of advice:

- *Frequent itemset mining:* The objective of this technique is to find the co-occurrence of items in a dataset of transactions. The co-occurrence is considered "frequent" whenever its support equals or exceeds a given threshold. This technique can be used as a support for discovering any of the advices introduced before. For instance, in the case of discovering *Component Association Patterns* we can this technique.

- *Association rules:* This technique aims at finding rules of the form *A→B*, where *A* and *B* are disjoint sets of items. This technique can be applied to help in the discovery of any of the proposed advices. For instance, in the case of the *Parameter Value Pattern*, given the value of two parameters of a component, we can find an association rule that suggests us the value for a third parameter.
- *Sequential pattern mining:* Given a dataset of sequences, the objective of sequential pattern mining is to find all sequences that have a support equal or greater than a given threshold. This technique can be applied to discover *Complex Patterns*, *Component Association Patterns,* and *Connector Patterns*. For instance, in the case of the *Connector Pattern*, we can use this technique to extract patterns that can be then used for suggesting connectors among components placed on the design canvas.
- *Graph mining:* given a set of graphs, the goal of graph mining is to find all subgraphs such that their support is equal or greater than a given threshold. For our purpose, we can use graph mining for discovering *Complex Patterns* and *Connector Patterns*. For instance, for *Complex Patterns* we can suggest a list of existing ready compositions based on the partial composition the user has in the canvas, whenever this partial composition is deemed as frequent.
- *Link mining:* rather than a technique, link mining refers to a set of techniques for mining data sets where objects are linked with rich structures. Link mining can be applied to support the discovery of any of the proposed advices. For example, in the case of *Data Mapping Patterns*, we can discover patterns for mapping the parameters of two components, based on the types these parameters.

Once community composition knowledge has been identified, we store the extracted knowledge in the advice repository in the form of directed graphs. In our advice repository, elements in the patterns, e.g., a component or a connector, are represented as nodes of the graph, and relationships among them, e.g., a component "has" a parameter, are represented as edges between those nodes. We also store a set of rules in our advice repository, which represent the trigger conditions under which a specific knowledge can be provided as an advice. Based upon this information, through our query interface we can match knowledge with the current composition context and retrieves relevant advices from the advice repository. Retrieved advices are filtered, ranked, and delivered based on user profile data (e.g., the programming expertise of the user or his/her preferences over advice types).

6 Conclusion

In this paper we propose the idea of *wisdom-aware computing*, a computing paradigm that aims at *reusing community composition knowledge* (the wisdom) to provide interactive development advice to less skilled developers. If successful, WIRE can e*xtend the "developer base"* in each domain where reuse of algorithmic knowledge is possible and it can *facilitate progressive learning and knowledge transfer*.

Unlike other approaches in literature, which typically focus on *structural and semantic similarities*, we specifically focus on the elicitation of composition knowledge that derives from the expertise of people and that is expressed in the compositions they develop. If, for instance, two components have been used together successfully multiple times, very likely their joint use is both syntactically and semantically meaningful. There is no need to further model complex ontologies or composition rules.

In order to provide identified patterns with the necessary semantics, we advocate the application of the WIRE paradigm to composition environments that focus on *specific domains*. Inside a given domain, component names are self-explaining and patterns can easily be understood. In the Omelette (*http://www.ict-omelette.eu/*) and the LiquidPub (*http://liquidpub.org/)* projects, we are, for instance, working on two domain-specific mashup platforms for telco and research evaluation, respectively.

For illustration purposes, in this paper we used Yahoo! Pipes as reference mashup platform, as Pipes is very similar in complexity to our own *mashArt* platform [5] but better known. In order to have access to the compositions that actually hold the knowledge we want to harvest, we will of course apply WIRE to mashArt.

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

References

1. H. Geffner. Perspectives on artificial intelligence planning. *AAAI'02*, pp.1013-1023.
2. D. Roman, J. de Bruijn, A. Mocan, H. Lausen, J. Domingue, C. Bussler, D. Fensel. WWW: WSMO, WSML, and WSMX in a Nutshell, *ASWC'06*, pp. 516-522.
3. A. Koschmider, M. Song, H.A. Reijers. Social Software for Modeling Business Processes. *BPM'08 Workshops*, pp. 642-653.
4. T. Reichling, M. Veith, V. Wulf. Expert Recommender: Designing for a Network Organization. *Computer Supported Cooperative Work*, vol. 16, no. 4-5, pp. 431-465, Oct. 2007.
5. F. Daniel, F. Casati, B. Benatallah, M.-C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. *ER'09*, pp. 428-443.
6. O. Greenshpan, T. Milo, N. Polyzotis. Autocompletion for mashups. *VLDB'09*, pp.538-549.
7. T. Hornung, A. Koschmider, G. Lausen. Rommendation Based Process Modeling Support: Method and User Experience. *ER'08*, pp. 265-278.
8. A.V. Riabov, E. Bouillet, M.D. Feblowitz, Z. Liu, A. Ranganathan. Wishful Search: Interactive Composition of Data Mashups. *WWW'08*, pp. 775-784.
9. A.H.H. Ngu, M. P. Carlson, Q.Z. Sheng. Semantic-Based Mashup of Composite Applications. *IEEE Transactions on Services Computing*, vol. 3, no. 1, Jan-Mar 2010.
10. H. Elmeleegy, A. Ivan, R. Akkiraju, R. Goodwin. MashupAdvisor: A Recommendation Tool for Mashup Development. *ICWS'08*, pp. 337-344.
11. OMG. Business Process Model and Notation (BPMN) - Version 1.2, January 2009. [Online] *http://www.omg.org/spec/BPMN/1.2*
12. OASIS. Web Services Business Process Execution Language Version 2.0, April 2007. [Online]. *http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html*
13. S. Smirnov, M. Weidlich, J. Mendling, M. Weske. Object-Sensitive Action Patterns in Process Model Repositories. *BPM'10 Workshops*, NJ, USA, September 2010.
14. D. Wegener, S. Rueping. On Reusing Data Mining in Business Processes – A Pattern-based Approach. *BPM'10 Workshops*, NJ, USA, September 2010.
15. C. Shearer. The CRISP-DM model: the new blueprint for data mining. *Journal of Data Warehousing*, Vol. 5, Nr. 4, pp. 13–22, 2000.

Appendix B

End-user requirements for wisdom-aware EUD

Long version of the paper: Antonella De Angeli, Alberto Battocchi, Soudip Roy Chowdhury, Carlos Rodriguez, Florian Daniel, Fabio Casati. End-User Requirements for Wisdom-Aware EUD. Proceedings of IS-EUD 2011, June 2011, Pages 245-250.

Conceptual Design and Evaluation of WIRE: A Wisdom-Aware EUD Tool

Antonella De Angeli, Alberto Battocchi, Soudip Roy Chowdhury, Carlos Rodriguez, Florian Daniel, Fabio Casati

Department of Information Engineering and Computer Science
University of Trento
Via Sommarive, 14 – 38123 Povo, Trento (Italy)
{antonella.deangeli, alberto.battocchi, soudip.roychowdhury, carlos.rodriguez, florian.daniel, fabio.casati}@unitn.it

Abstract. This paper presents the evaluation of the conceptual design of WIRE, a EUD tool for service-based applications. WIRE exploits community composition knowledge harvested from existing programs defined by other developers in the same domain. Such knowledge can assist less skilled developers in defining the composition they need, allowing them to go beyond their individual capabilities. The assistance comes in the form of interactive contextual advices proposed during the definition of composition logic. This idea was evaluated with 10 semi-structured interviews with University accountants. A rich set of information was elicited by means of several probes, including examples of contextual helps, commercial EUD tools, and scenarios in the form of positive and negative user stories. Results informed the definition of a set of requirements for WIRE, and fostered a critical reflection on possibilities and limitations of the general framework of EUD.

Keywords: Contextual help; EUD requirements; Wisdom-aware development; Interactive Advice; Mashups.

1 Introduction

Although the requirement for 'support and help' clearly emerges from user research on EUD for web services [1, 2], little is available to satisfy this need. There are currently two main approaches to enable less skilled users to develop programs: development can be eased by *simplifying* it (e.g., limiting the expressive power of a programming language) or *reusing knowledge* (e.g., copying and pasting from existing algorithms). Among the simplification approaches, the workflow and Business Process Management (BPM) community was one of the first to propose that the abstraction of business processes into tasks and control flows would allow also less skilled users to define their own processes. Yet, according to our opinion, this approach achieved little success and modeling still requires training and knowledge. The advent of the service-oriented architecture (SOA) substituted tasks with services, yet composition is still a challenging task even for expert developers [1, 2]. The reuse approach is implemented by program libraries, services, or templates (such as generics in Java or process templates in workflows). It provides building blocks that can be composed to achieve a goal, or the entire composition (the algorithm – possibly made generic if templates are used), which may or may not suit a developer's needs.

This paper presents the conceptual evaluation of WIRE, a *WIsdom-awaRE development environment* for exploiting the benefits of simplification *and* reuse. WIRE targets process-oriented, mashup-like applications, whose development and execution can be provided as a service via the Web and that are characterized by relatively simple composition logic and complex tasks or components. This class of programs provides the benefit of (relative) simplicity and a sufficient information base (the components) to learn and reuse composition knowledge. The idea is to *learn from existing compositions* (or, in general, *computations*) and provide this knowledge in the form of interactive advices to developers while they compose applications. The paper is organized as follows: Section 2 summarizes current approaches to assist users in development tasks; Section 3 introduces WIRE; Section 4 describes the evaluation and Section 5 concludes presenting implications for contextual help for EUD.

2 State of the Art

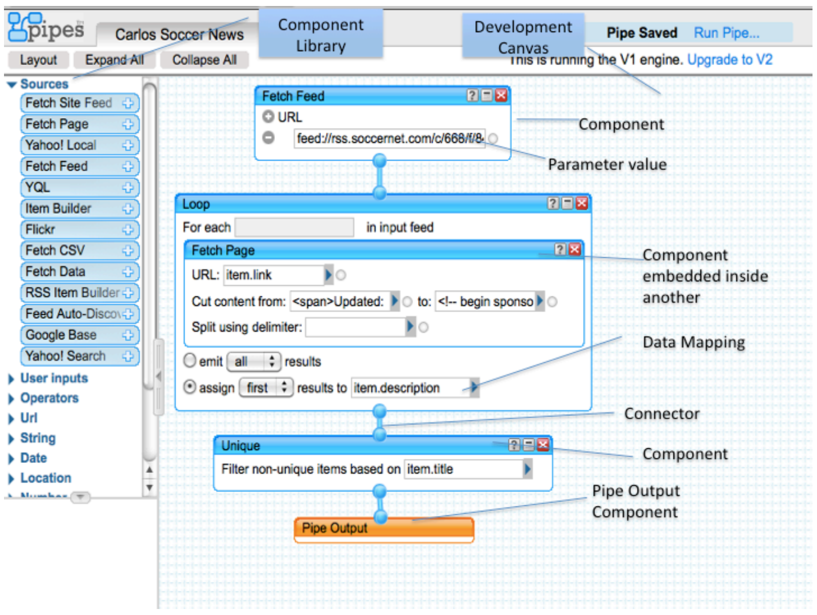
Several approaches are available to assist users in development tasks. In Programming by Demonstration [3], users are given examples of the process to be automated and based upon these examples the system auto-completes the remaining part of the process. Pattern-based Development [4] relies on a library of predefined patterns that represents good development practices. Goal-oriented approaches [5] assist developers by recommending plans that stem from user-specified goals. In semantic-based approaches [6], development ingredients are annotated to generate recommendations based on semantic matching and similarity techniques. Knowledge-discovery approaches [7] exploit a repository of previously developed applications in order to derive patterns that capture the steps performed solving similar problems in the past. All these approaches provide recommendations based upon explicit user inputs in the form of text queries, goals or partial application specification that convey the user intention. However, interactive recommendation systems can be improved by taking into consideration the composition status and the actions performed by users during development, as well as their preferences and skills, which is something that is not fully addressed by current approaches. Moreover, most of these approaches provide recommendations in the form of auto-completion, which has the limitation of masking the intermediate steps from the user, which in turn, results in loosing the control over the development.

In WIRE, we aim at discovering compositional knowledge by observing what other users have done in previous composite applications (i.e., applications that combine data, services and user-interfaces from multiple and possibly heterogeneous sources). We aim at delivering this knowledge in the form of interactive assistance, retrieved, filtered and ranked based upon context, user preferences and skills.

3 WIRE

To illustrate the challenges users face when using mashup environments, let us consider a typical scenario with Yahoo! Pipes[1], a composition environment that is currently presented on-line with words like 'intuitive' and statements like 'learn to build your own pipe in a few minutes'.

John is a soccer fan and an active blogger. He uses his blog to post latest news, articles, videos and updates from media sources and to discuss them with his friends. Keeping his blog updated requires a lot of manual work, such as content aggregation, filtering, and publishing. To automate it, John decides to use Yahoo! Pipes to build a simple pipe that: (1) sources a set of news feeds; (2) includes only the content related to soccer; (3) lists the news with their titles, and; (4) aggregates similar news from different sources under the same title. The pipe that implements the required feature is illustrated in Fig. 1. It is composed of five components. *Fetch Feed* gets the news from the publishing website (e.g., *feed://rss.soccernet.com/c/668/f/8493/index. rss*). *Fetch Page*, embedded inside the *Loop* component, fetches the page content from the feed output and extracts the content specified by 'Cut Content From and to' field. The *Unique* component merges the content of similar news, based upon their title (*item.title*) and groups them. Finally, *Pipe Output* represents the end of the pipe.



URL: feed://rss.soccernet.com/c/668/f/8

For each in input feed

URL: item.link

Cut content from: <span>Updated: to: <!-- begin sponso

Split using delimiter:

emit all results

assign first results to item.description

Filter non-unique items based on item.title

Fig. 1. Implementation of the example scenario in Yahoo! Pipes.

[1] http://pipes.yahoo.com/pipes/

The example shows that this apparently simple application involves development knowledge that goes beyond what typical end users have [8]. It requires knowledge about *programming models* (e.g., how to use components, connectors and data flows), *conventions or standards* (e.g., the structure of URLs), or simply about the right *terminology*. The intuition in WIRE is that this knowledge can be captured by *patterns* and reused as advices, to assist end users in their task. WIRE aims to leverage on the wisdom of the crowd for discovering and reusing *computational knowledge*. In [9] we introduced a detailed conceptual model of WIRE, which we summarize here for the better understanding of the ideas it encompasses. The conceptual model of WIRE can be divided into two parts: (1) the composition advices (the types of composition knowledge patterns) and (2) the triggering logic (when to deliver a given advice). *Advices* represent development recommendations, they can be of four types: *Complete* (e.g., a pipe with a missing connector), *Substitute* (e.g., a component in a pipe), *Highlight* (e.g., a connector between two components) or *Filter* out (e.g., a component from a pipe). Advices provide recommendations in the form of patterns, such as Parameter Value patterns (i.e., possible values for the parameter of a component), Component Association patterns (i.e., which components frequently appear together in a composition), Connector patterns (i.e., possible connections between components), Data Mapping patterns (i.e., mapping between input and output), or Complex patterns (i.e., a combination of the previous patterns). As for the triggering logic, a *Trigger* can be regarded as a triplet that consists of an Action, an Object and a Partial Composition. An *Action* represents what the user is doing, an *Object* represents the element on which the action is performed, and the *Partial Composition* represents the context for giving the advice. An Action can be further specialized as *Select* (a connector), *Drag* and *Drop* (a component from a component library), *Connect* (two components), *Fill* in (a parameter), *Delete* (a component), and *Embed* (a component into another). Finally, an Object can be a *Connector*, *Component*, *Parameter*, or *Composition Fragment*.

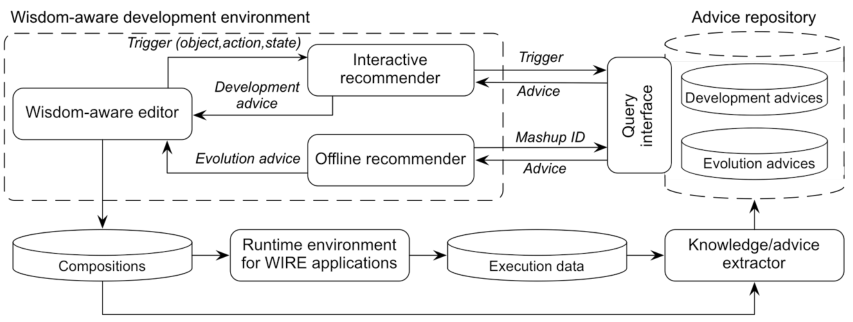


Fig. 2. High-level architecture of the wisdom-aware development (reported in [9]).

Fig. 2 presents the high level architecture that we envision for WIRE [9]. Development takes place in the *wisdom-aware environment*, where users can interact with a *wisdom-aware editor* for developing their applications. While a user builds his composition, a trigger may be fired based upon the actions that the user is performing in the editor. The triggering condition (object, action, partial composition) is sent to the *Interactive recommender*, which queries the *Advice repository* for a development advice that matches the triggering condition. The advice is then sent back to the Wisdom-aware editor. The Advice repository is filled with composition patterns by the *Knowledge/Advice extractor*. It takes as input a repository of existing *Compositions* and leverages on data mining and statistical data analysis techniques to discover and harvest the patterns discussed previously. The retrieved advices are filtered, ranked, and delivered based on user profile data (e.g., the programming expertise of the user or his/her preferences over advice types). We omit the explanation of the *Offline recommender*, as such is less related to the object of this paper.

In summary, the aim of WIRE is to develop an environment that allows end users to build their own composite applications. While we have used Yahoo! Pipes to draw our examples, our goal is to apply the WIRE approach to our own mashup platform [10]. In what regards the reuse of knowledge, we do not base our approach on semantic content and technologies, artificial intelligence planning, or automated goal-driven composition. Instead, we aim at leveraging on the expertise of users expressed through what they do in practice (i.e., composing applications) and that is expressed in their compositions or mashups.

4 Evaluation

An evaluation of the conceptual design of WIRE was run in order to address benefits and limitations of the proposal at a very early phase of the design. The evaluation was conducted by means of semi-structured interviews.

4.1 Method

Participants. The interview was administered to 10 accountants (7 F, 3 M; mean age = 37,2 years of age), all of which were employees of the administration of a local university. No specific inclusion criteria were applied to the sample. None of the participants had a background in computer science. The choice of involving administrative accountants in the evaluation was motivated by the fact that they are non-technical users and that they represent the potential *real targets* of the system.

Procedure. Interviews were administered in a quiet room, in which only the participant and the experimenter were present, and were audio recorded for successive analysis. No time restrictions were applied and participants were informed that they had the right to withdraw at any time or avoid answering any of the questions. Participation to the interview, which lasted approximately one hour, was rewarded with a voucher of 15 Euro that could be spent on an online bookstore. The interview was divided in four sections.

Section A: IT knowledge with specific reference to SOA. This section aimed to understand the level of expertise that our sample had with computer systems, their use of computers during day-to-day work activities, the kind of software they mainly work with, and the level of acquaintance with web-services and mash-up application.

Moreover, we wanted to understand if and how participants were used to describe their work processes through graphical representations.

Section B: Automatization of repetitive and complex tasks. This section aimed at understanding which daily activities participants considered as repetitive or complex and which were the strategies or instruments, if any, that they used to reduce the workload deriving from repetitiveness or complexity. Then, participants were shown an example of automation of workflow created using Automator, an application developed by Apple for Mac OSX that allows implementation of workflows through the creation of batches of tasks. Tasks can be grouped into batches through drag-and-drop of action elements to a workflow space. After seeing the example of automation of the workflow, participants were asked two more questions, which are reported in Table 1.

Table 1. Questions included in Section B/2 of the Interview.

B/2	General questions regarding automatization of repetitive and complex tasks.
B.7	Can you please provide your comments about the application and the procedure of automatization you have just seen?
B.8	Imagine you have a strict deadline. In this case would you prefer to invest some of your time in the process of automatization of the tasks you are dealing with or would you follow a manual procedure, even if facing the risk of missing the deadline and making errors?

Section C: Computer-provided Help and advice. Section C targeted the difficulties that may emerge while using computers during day-to-day work activities and the strategies that people use for overcoming them. This section also aimed at understanding the attitude of participant towards computer-provided help and advice, with particular focus on automatic/contextual modalities that are commonly used to provide them. Section C was divided into two parts: part one (Tab. 2) contains three questions about difficulties and strategies to overcome them, and preference towards either automatic/contextual or on demand modalities for providing help and advice.

Table 2. Questions included in Section C/1 of the Interview.

C/1	Questions about computer-provided help and advice
C.1	(a) What kind of difficulties do you encounter while using computers during your habitual work activity? (b) How did you overcome these difficulties? (c) Where/how did you find help?
C.2	Which is the most effective help strategy among the one you reported?
C.3	Currently there are two main ways through which software provide help or advice to users: 1) following a request by the user (e.g. the user opens the "Help" menu); 2) Automatically (by interpreting the actions performed by users and guessing their objectives). Which of the two ways do you prefer? Why?

After answering the first set of questions, participants were shown a slideshow containing examples of automatic computer-provided help or advice. The presentation included: 1) *Automatic word completion* in the Google search box; 2) *Automatic friend suggestion* in Facebook; 3) *Automatic book suggestions* in Amazon (i.e. "frequently bought together", or "customers who bought this item also bought"); 4) The function offered by web browsers that allows to *automatically save passwords; 5) Pop-up reminder windows* on calendars; 6) The *related videos* sidebar on YouTube. Participants were invited to comment each examples and were probed by three questions that related to advantages and drawbacks of each example, technical understanding on how the advice was created and perceived utility.

Section D: WIRE. Section D aimed at collecting opinions and suggestions about WIRE. We provided interviewees with a plus and a minus scenario reporting of an accountant who is using WIRE for automatizing the process of management of travel reimbursement. Following the methodology described in [11], both scenarios described the effects on work practices, which would be brought forward by WIRE on a new user. These effects were taken to the positive or negative extreme, to help users to think by themselves what consequences the new approach could have in their work practice. In the *Positive Scenario*, the accountant had a successful experience using WIRE, which helped him to save time and speed up repetitive work. So he decided to use it in the future and to share his work with other colleagues. In the *Negative Scenario*, the accountant encountered serious difficulties and eventually decided to go back to his traditional procedures for carrying out his work. Scenarios were presented with a counterbalanced order, meaning that five participants read the negative scenario first and the other five participants read the positive one first. The two scenarios are reported in the Appendix. After reading the two scenarios, interviewees were asked some specific questions about WIRE (Table 3).

Table 3. Questions included in Section D of the Interview.

D	Questions about the WIRE system
D.1	Do you think you would be interested in using WIRE?
D.2	Which advantages do you see in using this approach?
D.3	And what disadvantage do you see?
D.4	Within WIRE, advice and helps are created on the basis of behaviour and hints provided by people who previously used the system. Do you think that this kind of help is efficient in meeting your needs?
D.5	Would you prefer to have the advice automatically displayed or to receive them on request?
D.6	Would you like the system to inform you before doing tasks or during the task?

4.2 Results

Results are reported in different sections according to the thematic area addressed in the interview.

Sample Description. Almost all the participants learned to use a computer by themselves, through use at home and on the work place. Only 2 out of the 10 participants attended specific courses that were aimed at providing the required knowledge on computer use for obtaining the European Computer Driving License (ECDL) certificate. Yet, participants in general reported considering their computer skills as good enough for dealing with the activities required by their job responsibilities. They mainly used programs for the productivity, in particular Microsoft® Office™, platforms for accountancy, e-mail clients and various web browsers. One of the participants had an education in electrical engineering.

IT knowledge with specific reference to SOA. The most part of the interviewees did not know what web services are, but all of them tried to provide an explanation about what they could be. Their definitions tended to encompass practical functionalities related to the meaning of the word 'service' in real life showing almost no understanding of the software engineering definition of the concept. Definitions provided by users varied from *websites for online banking, shopping, or music download*, to *web-based shared documents* (e.g. Google Docs), to *instruments for supporting collaboration* (e.g. Doodle), or software that help people to *improve their navigation skills*. None of the participant could provide a definition of mashup applications.

Only three participants reported to draw flowcharts to represent their work processes, while other two usually draw Gantt diagrams. These diagrams were usually initially sketched using pen and paper and then converted into electronic files using specific software or web-based applications for project management, or Microsoft® Excel™. One participant reported that he usually draws diagrams for describing the results of a process, rather than for planning it. Two participants declared that they have never used diagrams or any type of cognitive artifacts to describe their work.

Automatization of repetitive and complex tasks. All but one of the interviewees could readily report few examples of activities they deal with on a daily basis that they defined as repetitive. Three of the participants explicitly reported that the job of accountants is repetitive in general. The activities that were more frequently described as repetitive consisted in entering data in pre-formatted forms or tables or dealing with paperwork where only few details had to be changed from case to case. Some of the reported activities consisted of long sequences of tasks that were connected by temporal (e.g. you can start task B only after completing task A) or logical (e.g. if task A gives as result X, then continue with task B, otherwise do task C) dependencies. Participants reported that, although some activities were repetitive, they still required a certain level of expertise and that the experience of accountants was a central factor for having things done correctly:

> [P1: "*In general, the most part of administrative work is repetitive, but we know where we can find the information we need...*"].
>
> [P3: *"In administrative jobs all the procedures are quite repetitive [...] anyway some kind of attitudes and experience is required... "*].

Five of the interviewees reported that, in the course of their work experience, they developed a set of *strategies* for speeding up repetitive tasks, making them less boring, and reducing the amount of errors that can derive from the loss of attention that sometimes is connected to repetitiveness.

> [P3: *"Everyone finds his/her ways to organize these processes by creating sets of "personal routines."].*

> [P6: *"I created my personal strategies, procedures for speeding up the process and for avoiding errors that can occur when you are tired or in a hurry."*].

These strategies usually consisted in the way a particular activity is subdivided or managed and differed from person to person. None of the participants reported any strategies that emerged from collaboration with colleagues or from instruction provided by supervisors/heads. Participants also talked about a number of software *tools* that were developed for facilitating administrative work. Some of these tools (e.g. Word™ or Excel™ templates, checklists) were developed directly by users and appeared to be shaped on the individual need and personal preferences of the users and their practices. Other tools appeared to respond more to needs at a "office level" – as opposed to "individual level" - (e.g. databases, shared documents or calendars, web-based applications) and were developed either by accountants that had a better knowledge of computer systems or by software engineers. These strategies and tools were perceived as facilitating work because they were effective in reducing or optimizing the amount of time that activities require, maintaining organization and priorities among tasks, and reducing the number of errors.

When asked if they ever tried to develop some simple programs for automatizing activities that are made by sequences of simple tasks, only two participants responded positively. In one case (P9), the participant wrote a meta-code for an application for the management of exams, which was then implemented by a technical developer; in the second case (P4), the participant reported about a database she developed using Microsoft® Access™ and that was then re-implemented by the system operators of the Department of Computer Science with the purpose of augmenting and customizing its functionalities. In both cases results were reported as being extremely satisfactory. The most part of the interviewees, though, never tried to design software beyond the ready-to-use functionalities of the software they use, and explained this as part of their lack of software engineering skills.

Before seeing the example of automatization of the workflow implemented using Apple Automator, participants were asked how they would have performed the same sequence of tasks in the case this activity had to be performed every day. None of the participants suggested a solution that included automatization of any of the steps. However, after seeing the Automator example, all the participants reported that such software would be of great help in their everyday work:

> [P2: *"A software like this seems useful; I would spend some time to learn how to use it; having quick positive results would be a good motivation for its future use; if first results are not good I would ask help to a system operator"*.].

> [P10: *"This program would be very useful to solve problems that usually take a lot of time"*.].

When asked if they would prefer to automatize an activity using software similar to Automator or to maintain their usual manual procedure in the case of an urgent upcoming deadline, seven out of ten of the interviewees declared that they would prefer to try to automatize the process. However, they put forward some conditions that would be crucial for motivating them to choose automatization: 1) the automated procedure should guarantee to be effective, ready-to-use and safe; and 2) the support of technicians should be available during the process of automatization.

Help and Advice. Understanding how people usually look for and receive help, advice and solutions to problems that may occur while using computers at work is an important question if we want to develop new forms of computer-provided help and suggestions to users. Asking help to colleagues and system operators represented the first option for half of the interviewees. The person to whom they asked for help was usually chosen on the basis of his/her level of expertise in the specific domain in which the problem fell into. Also the level of friendship or acquaintanceship played an important role in the choice of the person. Google represented the first choice of help for four of the participants and the second choice for those participants who could not find a solution to their problems by asking colleagues or system operators. Participants reported using online help and Help menus rarely, and this was the first choice only for 1 interviewee. Advantages and Drawbacks of each of the methods reported by participants are described in Table 4.

Table 4. Advantages and drawbacks of methods for asking help reported by interviewees.

Strategy	Advantages	Drawbacks
Asking colleagues / system operators	Quick and correct answer without wasting time looking elsewhere	Colleagues and system operators are not always present. If expert people always solve problems, you never learn how to solve them by yourself.
Look up solution in Google	Offers quick solutions without bothering other people (especially for simple problems).	-
Online help / help menu	Offer more complete reference.	It is difficult to understand. It makes you waste time. Require precise queries.

When asked which one was the most effective way for receiving help, among the options they reported, eight participants indicated colleagues and technicians. Their choice was motivated by the fact that technicians are very professional and helpful, and that providing support is part of their job. One participant indicated Google as the best source of information *"because you can use it at any time, also when you are at home"* (P10). One interviewee indicated paper manuals (when not in a hurry) and the Help menu (when time is short) as the most effective methods, because *"they provide very complete information"* (P7).

When asked which method for providing help and advice they preferred between *automatic/contextual help* and *help on demand*, seven participants reported that they prefer automatic/contextual help, but two of them also specified that this method works better for new or simple applications, while for more complex or well established procedures they would prefer help on demand. One participant provided an interesting observation about the function of automatic/contextual help:

> [P10: *"Automatic/contextual help has a double function: it appears when you need help and reminds you of potential errors; help on demand covers only the first function"*.].

Only 2 of the participants declared preferring help on demand for all their activities because automatic/contextual help can be a source of distraction and provide unnecessary hints. Participants also suggested that the automatic/help function should be customizable in order to be really useful:

> [P9: *"I prefer the automatic help because it makes me waste less time and spots out errors, but I should be able to deactivate it if I don't need it"*.].

Participants where shown some examples of contextual help and asked to comment on their effectiveness and usefulness. The function of *automatic word completion* by Google was considered very useful by all the participants. They reported that this function helped them in typing queries more quickly and without spelling errors, and that it also provided useful suggestions for other topics that are related to the desired keywords. The *Automatic friend suggestion* in Facebook was considered helpful by only five of the interviewees. However, this function was perceived as too intrusive in people's personal life and some participants declared that they preferred to manually look up for and add their new friends. Five of the participants declared to use the functions that *suggest book titles* in Amazon (i.e. "frequently bought together", or "customers who bought this item also bought") and to find it very useful for finding new books they did not heard of before. Two participants reported that they did not use this function but that they found it potentially useful; three participants reported to be annoyed by this function: they do not like the idea of using a system that interprets their taste for commercial purposes and that "*restricts, instead of expanding, the field of search*" (P4). The possibility of *automatically saving passwords* was considered useful by six of the ten interviewees but two of them also reported the concern that it can be dangerous for security of sensitive data, which is the main reason for four participants not to use it. Willingness to use such function heavily depends on the level of trust that people have on technology, especially when dealing with sensitive data. Five of the participants reported to use on a daily basis the function of Google Calendar that reminds of approaching meetings or commitments. The *related videos* sidebar featured in YouTube was considered very useful by nine of the interviewees; these suggestions were considered relevant, inspirational, and helpful for discovering new things.

When asked to formulate their "theories" about how the suggestions were generated in services like Google, Amazon, Facebook and YouTube, all the participants reported that they are created on the basis of the inserted keywords; one participant also made a distinction between general, or simple, and particular, or complex, suggestions:

> [P8: *"For simple queries, the system works on simple analogies with the inserted keywords; for more complex issues, the system*

does a matching with your personal characteristics (provided while registering to a service)".].

When asked if these suggestions are effective in meeting personal taste and needs, five participants responded positively but two of them also specified that these systems work well only in the case of "objective" suggestions, where the system does not make hypotheses about the user's personality (e.g. in YouTube). Suggestions are considered to be less accurate when they try to enter users' private space (e.g. in Facebook):

> [P4: *"The suggestions are good when referring to "objects"; when they try to catch my personality, they are not effective"*.].

> [P10: *"Suggestions work well for simple things (Amazon, YouTube), but when they try to get more personal, then they are less accurate"*.].

WIRE. Participants provided very useful information about their attitude toward WIRE. When reading the positive scenario, participants recognized several similarities with their work practices and perceived the system as potentially very useful:

> [P2: *"The accountant described in the scenario is very enthusiastic, I also would be like that if I had the right instruments and a good training; WIRE helps in keeping a structure of the work"*.].

> [P10: *"Awesome! This is exactly what a system should do to help people"*.].

Two participants expressed a common concern about the introduction of such system into their common work practices and suggested that, in order to benefit of its potentialities, the use of WIRE should totally replace previous practices, without leaving space for overlapping of the two methodologies:

> [P3: *"I am not sure about the way the accountant shares his experience, few colleagues do that, sharing should be forced from above"*.].

> [P10: *"One weakness could be the difficulty in recreating a process ex-post (after it actually occurred): software are less flexible than people; a software like this should force people to use it: if people are given the chance to use alternative ways in parallel, then things can become complicated"*.].

The Negative Scenario was also perceived as very plausible as it described well fears and frustrations that may emerge when something goes wrong while using computers, especially when dealing with new systems or procedures. Some interesting observations were provided by interviewees about the importance of a system that is well designed and thoroughly tested before being introduced into the work practice:

> [P7: *"I gave for granted that this technology was previously tested and approved by the central administration office. If I don't entirely trust it, I would prefer to perform some manual cross-checks. In the*

> *case of dealing with sensitive or financial matters, I would trust the system only if I am 100% sure that it is effective and functional"*.].
>
> [P9: *"This scenario shows a case in which something went wrong on the technical side; things go wrong when technology is not well designed"*.].

Participants were asked if they would be interested in using WIRE. Nine of the interviewees responded very positively and one was openly sceptical about the new system, adding that *"using WIRE would take the same time it takes doing the procedure manually"* (P1). Anyway, a period of formal training was indicated by two participants as a fundamental prerequisite for the adoption of WIRE:

> [P3: *"I would use it only after receiving a very good training and if it is not too complicated"*.].
>
> [P10: *"It is very important to invest some time to learn new instruments"*.].

Participants indicated *better organization of work*, *optimization of time*, *reduction of errors*, and *sharing of procedures and methodologies with colleagues* as the most relevant benefits connected with the use of WIRE. The risk of a *loss of control over work processes*, in the case that these were entirely completed in an automatic way, was indicated as the major potential drawback of the system:

> [P4: *"I would like to keep track of each step of the process; if everything is made automatically, the users misses the logic that stays behind the process"*.].
>
> [P6: "*If the procedure is too automatic, if I do some mistake in inserting values, then I will not be able to correct my mistakes. Users must be always aware of what they are doing, with a system that is too automatic, some people could get distracted"*.].

All the participants reported that the kind of help and suggestion that WIRE provided in the scenarios would be effective in meeting their personal needs, as the contents of the help message was created on the basis of past experience of colleagues that share the same work procedures and possibly the same difficulties:

> [P6: *"I think that suggestions are useful because they are based on well established and shared experiences"*.].
>
> [P9: *"If suggestions come from people of the same area, who share the same problems I have, then I think that these suggestions are valid"*.].

All the interviewees reported that the help that they would be effective in supporting their work; two of them added that the possibility of personalizing the way suggestions are provided would be a very important feature in order to make help messages really effective. Eight out of the ten participants reported preferring automatic/contextual help to help on demand in the case of an application like WIRE. Two participants motivated this choice by adding that automatic help can remind users about steps that they maybe would forget.

Help messages provided *during the task* were preferred to messages provided before the task by nine of the interviewees. One participant suggested that the two modalities could be combined:

> [P8: *"I can see the two modalities as complementary. At the beginning of the activity the system asks what your needs are in general; during the activity, pop-up windows provide you solutions when the system feels that you are stuck".*].

5 Conclusion

Consistently with [1] and [2], our study showed that people who do not have a specific background in computer science or software engineering know very little about SOA and web-services. They tend to define web-services following the mental model which they have consolidated in real life, as that of a third party which does something on their behalf. In this view, web services are perceived as something that provides assistance to perform actions on-line: they can be any type of software, from online banking to search engines, or Google documents. Participants were not aware of any technical detail about the meaning that the word has assumed in software engineering, nor of the possibility of using services to build or personalize their applications. These results challenge the emerging idea of SOA as a simple panacea to open software engineering to end-users, and call for more research into innovative metaphors to make SOA concepts and tools available to a larger population of non-technical users.

The interviews highlighted the fact that software are still perceived as tools to be used "as they are", and that customization of their functionalities is an option that usually accountants do not consider. The main barriers to EUD uptake were identified as 'concerns over reliability and security of the resulting artifact', 'need for control', 'need for help', and 'organizational barriers'. A clear tension between the requirements for simplicity and user control emerged. If on the one hand, the fact that building blocks were available to be easily combined by the users was considered an important advance over their previous experience with programming tasks, people declared to be reluctant to set-up fully automated systems to support their work, because of their apprehension of loosing control over the intermediate steps. Component-based programming was perceived as risky, because the user did not have direct control over the procedures implemented in each component. These results suggest the importance of developing a transparent system in which the users can keep a high level of control and monitoring over the processes they are dealing with.

End-users acknowledged that the idea of WIRE of providing assistance derived from the experience of colleagues working in a similar context had potential. However, issues of trust, timing and usefulness of the advice still remained important. During the design of WIRE we will need to find new strategies to make transparent how the advice was generated in the form of trust seal, particularly important when people deals with sensitive and financial issues. *Personalization* of procedures is also a desired feature: the system should be able to take into account the strategies that people used to optimize their work practices during the past. Giving users the func-

tionality that their expertise is taken into consideration while developing new practice is likely to facilitate the adoption by end users of a new system.
Finally, our data provide support to the proposal of collaborative tailoring, discussed in [12], as often participants mentioned that their willingness to engage in EUD was mediated by having support from other people and technical help easily available to them. This help was meant not only to alleviate some of the technical difficulties they had to face during development but also to take the responsibility out of their hands, making them less accountable in case of software failures. Issues related to organizational regulations and corporate processes also emerged as barriers to EUD uptake, as people often mentioned the need to have unambiguous approval from their manager as a fundamental step towards making them willing to explore new techniques and tools to automatise their work practices.

Acknowledgments. This work was partially supported by funds from the European Commission (project OMELETTE, contract no. 257635; project SERVFACE, contract no. 216699).

References

1. Namoun, A., Nestler, T., De Angeli, A.: Conceptual and Usability Issues in the Composable Web of Software Services. In: Daniel, F. and Facca, F.M. (eds.) Current Trends in Web Engineering - 10th International Conference on Web Engineering ICWE 2010 Workshops, Revised Selected Papers. LNCS, vol. 6385, pp. 396-407, Springer, Heidelberg (2010)
2. Namoun A., Nestler T., De Angeli, A.: Service Composition for Non Programmers: Prospects, Problems, and Design Recommendations. In: 8th IEEE European Conference on Web Services ECOWS (2010)
3. Cypher, I., Halbert, D.C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B.A., Turransky, A. (eds.): Watch what I do: Programming by Demonstration. MIT Press, Cambridge, MA, USA (1993)
2. Van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases. 14(3), 5--51 (July2003)
3. Henneberger, M., Heinrich, B., Lautenbacher, F., Bauer, B.: Semantic-based Planning Process Models. In: MKWI'08, Munich, Germany (2008)
4. Ngu, A., Carlson, M., Sheng, Q., Paik, H.: Semantic-Based Mashup of Composite Applications. IEEE Transactions on Services Computing. 3(1), 2--15 (2010)
5. Greenshpan, O., Milo, T., Polyzotis, N.: Autocompletion for Mashups. In: Proc. VLDB Endow. 2, no. 1, pp. 538-549 (2009)
6. Yue, K.: Experience on mashup development with end user programming environment. Journal of Information Systems Education. 21(1), 111--119 (2010)
7. Roy Chowdhury, S., Rodríguez, C., Daniel, F., Casati, F.: Wisdom-Aware Computing: On the Interactive Recommendation of Composition Knowledge. In: Proceedings of WESOA 2010, Springer (2010)
8. Daniel, F., Casati, F., Benatallah, B., Shan, M.-C.: Hosted Universal Composition: Models, Languages and Infrastructure. In mashArt. ER'09, pp. 428-443, (2009)
11. Bødker, S.: Scenarios in user-centred design - setting the stage for reflection and action. Interacting with Computers. 13, 61--75 (2000)

12. Wulf, V., Pipek, V., and Won, M.: Component-based tailorability: enabling highly flexible software applications, International Journal of Human-Computer Studies. 66(1), 1--22 (2008)

Appendix: The Scenarios

Introduction (common to both scenarios). Alberto is an accountant at the University of Trento: one of his duties is the reimbursement of travel expenses. This is a sensitive and time-consuming work which requires several repetitive tasks (e.g., checking staff profile, projects details, financial reports and regulations, obtaining proper authorization), which Alberto performs manually. He was told that the university recently bought a web-tool called *WIRE,* which can help him to make repetitive tasks automatic. Although he has never done any type of development before, he is curious to try. A short on-line video shows him that *WIRE* works by connecting *activities*. WIRE provides a set of activities similar to the paperwork Alberto is familiar with; a composition area where these activities can be assembled; and a set of contextual advices generated according to the actions of the user.

Positive Scenario. Alberto starts designing an application for the reimbursement process. He knows from his daily experience that the process starts with a request, so he searches for this activity. He finds an activity called *Reimbursement request* and, after reading its description, he places it in the composition area. Then, he selects the *Get trip document* activity and at that point an automatic advice suggests him to take also the *Translator and* the *Currency Converter* activities. Alberto knows that the advice is right because he deals with many travels in foreigner countries. He thus accepts the advice and WIRE automatically places the two activities on the composition area adding the necessary connections to the other activities. Alberto progresses the design choosing all activities needed and WIRE help him suggesting how to connect them. Once Alberto thinks he has finished, WIRE proposes to add the activity C*heck Authorization* in a specific point of the workflow. He remembers that this task is mandatory and needs to be performed early, so he gladly accepts the advice and the system automatically update the configuration. Now Alberto decides to share the application with all his colleagues: he clicks on the link Publish and the application is automatically posted on the Intranet.

Negative Scenario. The first impression that Alberto has is mixed. WIRE provides a large number of activities that certainly will include those he needs but it is difficult to understand what they do and how to use them. Alberto fears that learning will take quite some time. However, he decides to give it a try and starts designing an application. He knows from his daily experience that the process starts with a request, so he searches for this activity. He spends some time to find an activity called *Reimbursement request* and he places it in the composition area. Then, he selects the *Get trip document* activity and at that point an automatic advice suggests him to take also *Translator and Currency Converter*. Alberto does not need these activities as the application he wants to build is only for national trip. He finds the advice annoying, as he knows exactly which activities are involved in his job and was distracted by the unnecessary suggestion, which took time to be understood. Alberto progresses the design choosing all activities he needs but then he feels unsure on how to connect them. The system however does not have any advice on the problem. Alberto is not sure about the correctness of the application he created. He is worried about the fact that it will handle money and sensitive information of which he is responsible for. He misses the level of control that he feels while processing his task manually one after the other so he goes back to the traditional work.

Appendix C

Composition Patterns in Data Flow Based Mashups

Soudip Roy Chowdhury, Aliaksandr Birukou, Florian Daniel, Fabio Casati. Composition Patterns in Data Flow Based Mashups. Proceedings of EuroPLoP 2011, 2011.

Composition Patterns in Data Flow Based Mashups

Soudip Roy Chowdhury, Aliaksandr Birukou, Florian Daniel, and Fabio Casati
University of Trento, Via Sommarive 5, 38123 Povo (TN), Italy
{rchowdhury,birukou,daniel,casati}@disi.unitn.it

ABSTRACT

Recently, mashup tools have emerged as popular end-user development platform. Composition languages used in mashup tools provide ways (drag-and-drop based visual metaphor for programming) to integrate data from multiple data sources in order to develop *situational applications*. However this integration task often requires substantial technical expertise from the developers in order to use basic composition blocks properly in their composition logic. Reusing of existing composition knowledge is one of the possible solutions to ease mashup development process. This reusable composition knowledge can be harvested from *composition patterns* that have occurred frequently in previously developed mashup. In order to understand composition patterns in mashups, particularly in data flow based mashups, in this paper, we have analyzed the composition language used by one of the most popular data-flow based mashup tools, Yahoo! Pipes. Based upon our analysis we have identified six composition patterns, which represent most commonly used composition steps during mashup application development. To prove the generality of the identified patterns in data-flow based mashup composition languages, we have further shown the applicability of our composition patterns in several other popular data-flow based mashup tools.

Keywords

composition pattern, mashup, data mining, end-user development

1. INTRODUCTION

Recent efforts in end-user development (EUD) focus on enabling domain experts, i.e. business experts who are not typically IT experts to participate in the application development process. Mashup development [6] is particularly in-line with this EUD methodology. Mashup development is conceptualized with a view that domain experts could develop "*situational application*" to cater their immediate business needs without having IT experts in the development loop. Development supports (e.g. visual metaphors like dragging, dropping and connecting visual components instead of writing programs etc.) are provided by the development environment to ease the development process. However these supports are still not sufficient to ease EUD. Developing an application using these development environments requires end-users either to tailor the existing solutions or to create a new solution as per the new requirements. This task involves understanding and defining the complex data flow logic between the components in an application [5]; although this is not a typical skill that an end-user possesses.

The use of the patterns to capture the frequently occurring development styles and insights in computer/software systems design [7,8] is not a new idea. In our approach, we explore the mashup development scenario to identify the potential *mashup development patterns*, which can be useful to the developers (novice or expert) while defining their composition correctly. We also think that mashup platform providers will benefit from our analysis. This analysis of patterns will help them to understand the mashup composition paradigms in a better way. This will also help them to identify what are the functionalities they could provide in the composition language in order to support end-user development. In this paper we have restricted our analysis only to data-flow based mashup composition logic. The patterns, which are discussed in this paper, may not be readily applicable to other composition languages (e.g. control flow based) and may require further refactoring.

Michael Ogrinz et al. [2] have identified 34 different types of mashup patterns classified mainly into 5 main categories for data-flow based applications. The patterns, as presented in this paper, are derived by analyzing the functional and structural aspects of enterprise mashup applications. In our approach, as described in [1], we, however, want to explore the composition patterns in mashups, which are derived by analyzing the frequently occurring development steps (mashup composition models) in existing mashup applications.

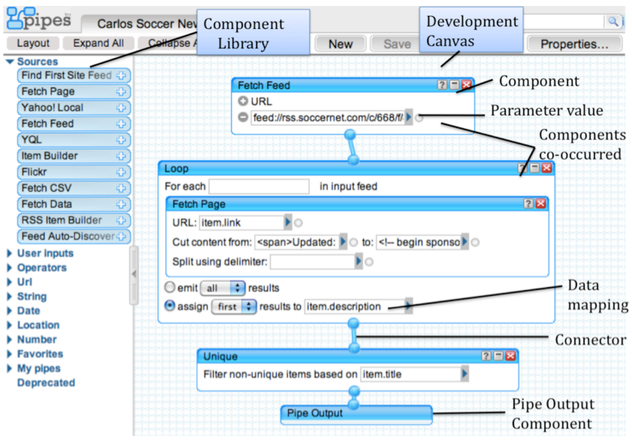
The pattern descriptions in this paper are targeted at both novice and experienced mashup application developers. Novices may choose to treat these patterns as suggestions to be tried and to be applied in their applications. Whereas, experts can use these patterns definitions as a form of checklist, in order to identify them in their application definitions. Experts can further store the definition of the identified composition-pattern in a repository (composition knowledge base) in order to make them reusable by the end-users (domain experts, non-technical users) during their development tasks.

The structure of this paper is as follows; in the next section we explain the development steps that a developer has to follow in order to develop a simple application in a data flow based mashup tool like Yahoo! Pipes. Based upon the scenario, we analyze the mashup composition paradigm and introduce the composition patterns in section 3. In section 4, we show our effort to apply the identified patterns of section 3 in other data mashup platforms. In Section 5, we finally conclude our discussion with possible future work directions.

2. EXAMPLE SCENARIO

In this section, with the help of a use-case implementation scenario in Yahoo! Pipes, we have tried to explain the composition steps that a developer has to follow while developing a mashup application in a tool like Yahoo! Pipes. The example scenario is described as follow:

Carlos is a sports lover and an active blogger. He uses his personal blog to post sports related latest news, articles, videos and updates from different media sources like ESPN sports. Keeping his blog updated with the latest news, requires him to do lot of manual works like content aggregation, filtering and publishing etc. To automate this repetitive and time-consuming job, Carlos intends to use Yahoo! Pipes mashup environment and composition language to create an application, that fetches news feed from ESPN sports, extracts only the content related to soccer news, lists the news with their corresponding headlines and aggregates similar news under the same headline for better readability purpose.



Fetch Feed
URL: feed://rss.soccernet.com/c/668/f/

Loop
For each in input feed

Fetch Page
URL: item.link
Cut content from: <span>Updated: to: <!-- begin sponso
Split using delimiter:

emit all results
assign first results to item.description

Unique
Filter non-unique items based on item.title

Figure 1 Implementation of the example scenario in Yahoo! Pipes.

The pipe that implements the required feature is illustrated in **Figure 1**. It is composed of five components: The *Fetch Feed* is required to get the news article from the publishing website as mentioned by its *URL* parameter. The *URL* address for ESPN news is *feed://rss.soccernet.com/c/668/f/8493/index.rss*. The next component is a container *Loop*, which embeds another component *Fetch Page* inside it. *Fetch Page* Component retrieves the selective page content (*Cut content from* parameter is used as a content selection criteria over the HTML content of the page) from the links as mentioned in *item.link* field of the output coming out of *Fetch Feed* component. *Loop* component runs over every feed item and invokes the *Fetch Page* component. It also assigns the output of the *Fetch Page* component to the *item.description* field. *Unique* component is used for merging the content of the similar news, based upon their title description (*item.title*). Finally, the *Pipe Output* component specifies the end of the pipe.

3. COMPOSTION PATTERNS

Before we discuss about the development patterns in detail, let us first define the preliminaries of a data-flow based mashup application.

A mashup M is a tuple, M = <N, C, T, O>

Where

N- denotes the name of a mashup application.

C - $\{c_1, c_2 \ldots c_n\}$ denotes set of components in an application.

T is a tuple, defined as T =<V, E> denotes the data mapping function between connected components.

Where

V – $\{L_1, L_2, \ldots L_K\}$ denotes set of pair of components which are connected via connectors between them.

Such that L_1 – (c_1, c_2), L_2 - (c_1, c_3), … L_K -(c_{K-1}, c_K),

E – $\{e_{L1} \ldots e_{LK}\}$ denotes set of connectors that can be used for connecting pair of components in V.

O – denotes the output of a mashup application.

Further, a component C can be defined as a tuple.

C = < I, R, Q >

Where

I - $\{P_1, P_2, \ldots P_N\}$ denotes the set of *configuration parameters* (Input) that a component can have.

R - $\{A_i\}$ denotes set of attribute values for the parameters of a component. Given a component c_i, and the set of configuration parameters I, the attribute values that elements of I hold in a mashup, is denoted by the elements of the set R:$\{A_i\}$, where i = 1..N , denotes the index of the parameters. An attribute value can be provided by the developer explicitly or can be assigned with the value of the output of another component in the development canvas.

Q – denotes the output value for the component c_i.

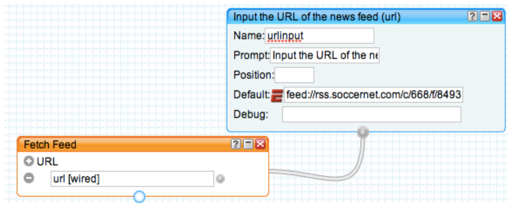
In the light of the above formalization, let us now define composition patterns that we can identify and extract from a mashup application as explained in section 2.

3.1 Frequent Parameter Value

- *Description*: Frequent parameter value captures a set, consisting of possible value assignments for a parameter of a component that have been used frequently in the past compositions. The parameter value can be assigned with an explicit user-specified string value (as shown in Figure 1, *URL* parameter of *Fetch Feed* component) or can be assigned with the output value of another component in the current composition (as shown in Figure 2) via a connector pipe. By analyzing the past successful compositions we can identify the frequent itemsets, which capture the value assignments for a given parameter of a component. Frequent value-assignment itemsets along with associated component, and composition context information are captured and stored as data-pattern.

- *Example:*

  The *Fetch Feed* component as shown in figure 2, has an *URL* parameter. *URL* is assigned with the output value of another component. In this example the output of an *URL Input* component, as shown in the right-top end of Figure2, provides the value to the *URL* parameter of *Fetch Feed.* This value assignment can be captured in frequent parameter value.


Input the URL of the news feed (url)
Name: urlinput
Prompt: Input the URL of the ne
Position:
Default: feed://rss.soccernet.com/c/668/f/8493
Debug:

Fetch Feed
URL
url [wired]

Figure 2 Example of Frequent Parameter Value

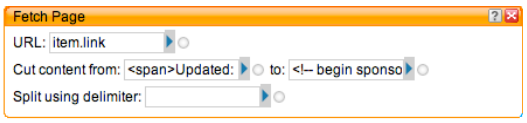
- *Problem:* There may be many possible value options, for the parameter value assignments for a component, remembering all of them are difficult for the developer. Human errors (type mismatch, wrong value assignment etc) in specifying the parameter value lead to erroneous result of the data-flow. Also these types of errors are harder to detect at later stage of the composition design. Learning from the examples of past applications and use it in the current composition takes time and expertise.
- *Forces:*
    - The value can be provided manually as a string value or can be provided by assigning output of another component to the given parameter value field.
    - In case of assigning the input parameter value of a component by the output of another component, it is essential to know which of the components can provide the required input value to be assigned to the parameter. In case of explicit type casting is required for the input parameter value, the developer also needs to know how to do the provisioning (e.g., using filter components to filter out few attributes from the output parameter) in order to make the composition work.
    - Type mismatch is typical problem that arises during parameter value assignment. Due to this when the output of one component is assigned to configuration parameters of another, more care is required to avoid the problem of type mismatch.
- *Solution:* To solve the problems as mentioned above, we capture and store the frequent value assignments information for the parameters of a component along with the associated composition context information. Given a component c_i and its parameter $P_{i,}$ we can identify the possible value assignments for P_i i.e. $\{<P_i,A_1>,<P_i,A_2>..<P_i,A_n>\}$, which have occurred frequently in the past successful compositions. We identify these patterns from the existing composition models stored in the mashup repository and by applying data-mining algorithm (association rule mining). We store the extracted pattern information in our pattern repository to analyze the best practices and common usage of patterns.
- *Consequences:*
    - One component may have multiple parameters and multiple parameters can have many possible values, capturing and storing all the possible values is memory intensive tasks.
    - Frequently used value set doesn't always represent the possible set of values. Hence at time when the user wants to know information about the whole set of possible values this approach may not be useful.

3.2 Associated Parameters Value

- *Description*: Associated Parameters value pattern captures the information related to the value assignments for all the associated parameters for a component. Given a parameter of a component is assigned with a specific value, this pattern captures how the remaining parameters of the selected component are assigned with values. Association rules capturing the relationship and assignments of parameters with their corresponding values for a component along with their support and confidence metric are stored in associated parameter value pattern.
- *Example:*

  As shown in the Figure 3, the value of the parameter *Cut content from* and the subsequent *Split using delimiter* are determined by the value of the parameter *URL* of *Fetch Page*. Hence we can say that there exists an association relation between the other parameter values of *Fetch Page*, given the value of *URL* parameter.
- *Description*: Associated Parameters value pattern captures the information related to the value assignments for all the associated parameters for a component. Given a parameter of a component is assigned with a specific value, it captures how the remaining parameters of the selected component are assigned with values from a set of possible values for them. Association rules capturing the relationship and assignments of parameters with their corresponding values for a component along with their support and confidence metric are stored in parameter value association.
- *Example:*

  As shown in the Figure 3, the value of the parameter *Cut content from* and the subsequent *Split using delimiter* are determined by the value of the parameter *URL* of *Fetch Page*. Hence we can say that there exists an association relation between the other parameter values of *Fetch Page*, given the value of *URL* parameter.



Fetch Page
URL: item.link
Cut content from: <span>Updated: to: <!-- begin sponso
Split using delimiter:

Figure 3 Example of Associated Parameters Value

- *Context:* A selected component has more than one parameter (e.g. input parameter, configuration parameter, and output parameters). User has filled a few of the parameters with their corresponding value assignment and further he wants to assign values for the rest of the parameters of the selected component.
- *Problem:* The problems that a user may face in order to fill up the values for the rest of the parameters, given a few of the parameter values are filled, are due to the fact that there could be many valid options for the parameter value assignments. The assignments of parameters with their corresponding values require the users to know the internal data-flow logic of the composition. This task is not a trivial one, especially the users who do not have enough exposures on service composition and mashup tools, may find it difficult to set these values.
- *Forces:*
    - As the values of the parameters are associated, the values of subsequent parameters are dependent on the values of the preceding parameters. For example selection of *URL* parameter value in Figure 3, determines the possible value options for the subsequent parameters (*cut content from, Split using delimiter etc)*.

- Type mismatch during the value assignment is another typical problem that arises during the value assignment for the parameters of a component. Knowing the proper type information is not very trivial for the developer who does not have enough exposure on mashup tools and also do not have the prior knowledge about the application's data-flow logic.

- *Solution:* Therefore, to solve the problem as described above we need to identify and store the association relation information between the value assignments for the parameters of a component. Given a component c_i has N parameters $(P_1, P_{2,\ldots,} P_N)$,if the parameter value assignments $\{(P_{1,} A_1),(P_2,A_2)\ldots(P_K,A_K)\ldots(P_{N,}A_N)\}$ are found to be the most frequent from the past successful compositions. Then we can infer the association rule $\{(P_{1,}A_1),(P_2,A_2)\ldots(P_k,A_K)\} \rightarrow \{(P_{K+1,} A_{K+1})\ldots(P_N,A_N)\}$ i.e. given P_1 is assigned with $A_{1,}$ P_2 with A_2 and P_K is assigned with A_K etc implies P_{k+1} will be assigned with A_{K+1} and similarly P_N will be assigned with A_N. Association rules, containing the parameter value assignments along with the information of the corresponding component, and composition context reference, are stored as parameter value association in the composition knowledge base. This association information is significant in helping the users to fill the parameters with proper values for a given component in a given composition context mitigating the risk of type mismatch and selecting from multiple options without having enough technical insight about the composition.

- *Consequences:*

  The consequences are similar to the consequences as mentioned for parameter value pattern.

3.3 Components Co-occurrence

- ***Description:*** Components co-occurrence, captures the information in terms of given a component selected what are the other components that can co-exist in a given composition context.

- *Example:*

  Components co-occurrence captures the information about what are the components that may occur together in a given composition context. In the example as shown in Figure 4, component co-occurrence captures the set of components *{Fetch Feed, Loop, Fetch Page, Unique},* given the fact that these components occurred together frequently in the previous successful compositions.

- *Context:* User wants to proceed or complete his current composition design by adding a new component/s in his composition model in the development canvas.

- *Problem:* In the presence of a large database of mashup components, selecting proper component/s that can be used together with the components already existing in the user specified composition design model, is not an easy task for the developer who do not posses sufficient IT knowledge. Learning from the examples of past applications and use it in the current composition requires time and expertise.

- *Forces:*

  - From a database of n different mashup components, the number of possible way that k number of components can be chosen for the mashup design is n^k , in the worst-case scenario. For a less IT skilled developer choosing the best possible option of component out of n^k is not an easy task.

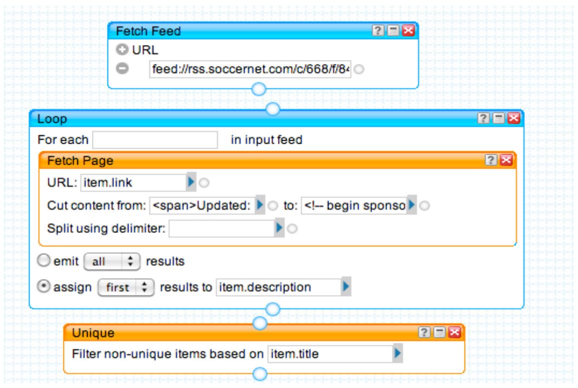

Fetch Feed
URL
feed://rss.soccernet.com/c/668/f/84
Loop
For each in input feed
Fetch Page
URL: item.link
Cut content from: <span>Updated: to: <!-- begin sponso
Split using delimiter:
emit all results
assign first results to item.description
Unique
Filter non-unique items based on item.title

Figure 4 Example of Components Co-occurrence

  - For defining a consistent mashup, not only the co-existence of components in a given composition context but also their inter-dependencies (proper mapping of parameter value from one component to another in order to make the data-flow consistent etc.) have to be defined properly. Developers using such mashup platforms must have the background knowledge about how to satisfy these criteria while defining the data-flow logic for a mashup application.
  - Making a simple mistake in the intermediate steps during the mashup design may lead the whole application to become erroneous. At the later stage of the development, identifying such mistakes, which have occurred in the earlier steps, become difficult.

- *Solution:* Therefore, the components co-occurrence captures the association information of a component or a set of components with the associated set of components with their corresponding support and confidence value to be appeared together in an application. Given a set of components $S=\{c_{i,,}c_{i+1}\ _{,\ldots,}c_N\}$ are present in the current development canvas, we can find the set of other components $Y=\{c_{j,}\ c_{j+1}\ _{..}c_M\}$, such that (S,Y) occurred together in the previous successful compositions and the following conditions satisfy; $S,Y \in C$ also $S_, \cap Y = \varnothing$. The elements in this association rule captures the set of components, which have frequently co-occurred together in the past successful compositions and also the components $(\{c_{current}\})$ in the current development canvas is a subset of either S or Y, i.e., $c_{current} \subset S$ or $c_{current} \subset Y$. While the support value captures the statistical measures of how many times in the past compositions (S,Y) occurred together, the confidence value signifies the probability of occurring Y given any elements of S is present in the composition context. Components that satisfy a certain threshold value of support and confidence are captured and stored in a list that stores the co-occurrence (S,Y) information of the components in a mashup composition knowledge repository. This knowledge may be significant in understanding the possible options for components, which the user may use in his composition.

- *Consequences:*
    - This pattern captures the information about the number of components co-occurred in a composition.
    - But this pattern doesn't capture the information about how those components are connected with each other. In other words how the data flows between the components.

3.4 Data-mapping

- *Description*: *Data-mapping* captures the most frequent dataflow logic definition which consists of components in the current composition, i.e., how in the past compositions the output attribute of one of the existing component is connected via connector to the input parameter of another component/s in the given composition context. Data can be mapped between one component's output to another component's default input or it can be mapped between one component's output to another components' configuration parameter.
- *Example*:

  In a data-flow based composition scenario, as we have described in this paper, the data-mapping can happen between one component's output to another component's default input as shown in Figure 5b or between one component's output to another's configuration parameter as shown in Figure 5a.
- *Context:* A user wants to connect one component with another component in the composition by defining proper data mapping between the output attribute/s of one component to the input parameter on another.

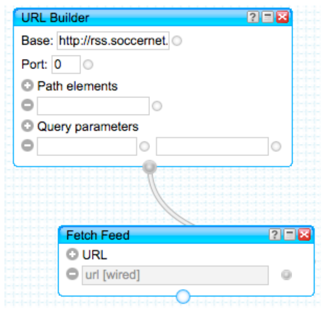

URL Builder
Base: http://rss.soccernet.
Port: 0
Path elements
Query parameters
Fetch Feed
URL
url [wired]

Figure 5a

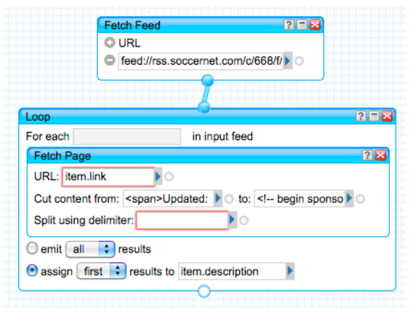

Fetch Feed
URL
feed://rss.soccernet.com/c/668/f/
Loop
For each in input feed
Fetch Page
URL: item.link
Cut content from: <span>Updated: to: <!-- begin sponso
Split using delimiter:
emit all results
assign first results to item.description

Figure 5b

Figure 5 Examples of Data-mapping patterns

- *Problem:* Defining the proper data-mapping logic requires developer to know the technical details about the data-flow logic. If the user makes a mistake in defining the data-mapping logic between the components then the whole composition logic becomes erroneous.
- *Forces:*
    - A user needs to know the type information of the input parameters for the target component as well as the type information for the output parameter of the source component. The type of these two parameters must match for the data mapping between the components.
    - When the output of one component is used as an input parameter value by more than one component, then the type of the output of the source component must match with the input of all the target components.
    - Mapping a specific value from the output set of a component to the input parameter value of another component, for example in Figure 5b, the mapping of *item.link* from the output list *Item* of *Fetch Feed* component to the *URL* parameter of *Fetch Page* is another data-mapping example. However we can observe in this example that knowing this kind of finer mapping details involves technical knowledge as well as knowledge on the data-flow logic.
    - Learning the possible relevant options from the previous application examples is not very easy for the end-users. Also this learning process requires time and expertise.
- *Solution:* To solve the problem as explained above, in data-mapping, we capture the association rule capturing the information about how the output of one component (source component) are mapped to the input parameter (configuration parameter) value of another component (Target component). The data-mapping information is captured in terms of association rules between the parameter values of the components. Let us assume, for a given pair of components (c_i ,c_{i+1}), output object q_i of c_i (Source Component) is mapped to configuration parameter P_j of c_{i+1} (Target Component). Furthermore let us assume that q_i contains set of N values as $\{\beta_1,\beta_2 \ldots\beta_N\}$ and out of that a subset $\{ \beta_{j,\ldots}\beta_K \}$, where 1<=i and K<=N, can be mapped to P_i , in a given composition context. The association relation that captures the relation of $\{\{c_{i,}q_i[\beta_{j,\ldots}\beta_K]\} \rightarrow \{c_{i+1,} P_j\}\}$ with corresponding support and confidence value is stored as *data-mapping*.
- *Consequences:*
    - Given two components this pattern will help users to know in how many ways they can be connected with each other via data-mapping.
    - If the number of components increases, the possible options for their data mapping with each other increase. The viable options for the possible data mappings also become exponentially high.

3.5 Associated Composition Fragments

- *Description: Associated Composition Fragment* captures the association information between two composition fragments.

In other words, given a partial composition definition in the current development canvas, *associated composition fragment* captures the association information between the current partial compositions with the associated components/ composition fragment, which can be used to auto-complete or to extend the current composition definition. *Associated Composition Fragment* consists of set of connected components that have been frequently used together in previous successful applications. This pattern contains partial compositions definition consisting of multiple components, connectors with proper parameter value and data mapping setting.

- *Example:* For instance in Figure 6, the combination of *Filter-Fetch Page* embedded inside *Loop – Unique* component together is an example of *Associated Composition Fragment*. Given *Fetch Feed* component is selected and its *URL* parameter is filled with a specific value as shown in the Figure 6, *Associated Composition Fragment* captures the knowledge that the combination of *Filter- Fetch Page* embedded inside *Loop – Unique* component together is the most frequently used fragment which can be connected to *Fetch Feed* component.
- *Context:* when a user selects a component in the development canvas, and he wants to complete his partial composition definition with fragment consisting of several components connected via connectors with proper data-mapping set among the components etc.
- *Problem:* Completing a mashup composition definition with components, connector and data mapping, requires users to know the internal data flow logic of the application, input and output parameters and their type information for all the constituent components. If the mashup platform contains many components and if the components can have many possible ways to be connected with each other, then the complexity of defining a proper mashup composition becomes exponentially huge. Even a small mistake while selecting a component or filling the parameter value or defining the data mapping logic during the intermediate steps can lead to an erroneous mashup application definition.
- *Forces:*
    - The number of possible ways that a mashup composition can be defined is many. Knowing all of these possible options for defining a proper mashup application is difficult for the end-users. Especially when the mashup application is considerably large, for each of the components and connections user needs know the information regarding the parameter values, data mapping logic etc. Knowing all of them is not a trivial task for a less skilled developer or end-users for instance.
    - Learning the possible options of the intermediate steps from the previous application examples requires time and expertise.
    - As for the large mashup application designing making mistake in defining any of the intermediate steps may become difficult to debug at the later stage.

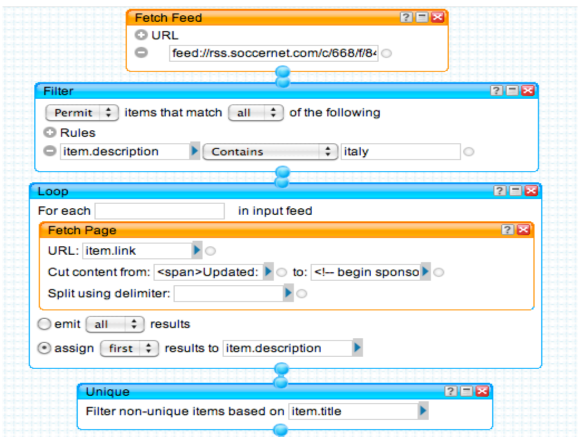

Fetch Feed
URL
feed://rss.soccernet.com/c/668/f/8
Filter
Permit items that match all of the following
Rules
item.description Contains italy
Loop
For each in input feed
Fetch Page
URL: item.link
Cut content from: <span>Updated: to: <!-- begin sponso
Split using delimiter:
emit all results
assign first results to item.description
Unique
Filter non-unique items based on item.title

Figure 6 Example of Associated Composition Fragments (containing component/s, connector/s as a part of meaningful compositions)

- *Solution:* To solve the problem as explained above, associated composition fragment could be used for auto-completing the partial composition definition. Past successful application fragments, which were frequently used and well-tested in similar composition, context, can be used for auto-completing the partial mashup definition fast. Associated composition fragment can be used for this purpose. Associated composition fragment can capture the information such as, given the existing composition definition in the development canvas, the set of frequently occurring partial composition instances, which can be used to auto-complete or extend the definition of the existing composition in the development panel. Let us assume $M_{existing} = < C_{existing}, T_{existing}, O >$ denotes the existing partial composition in the development canvas, where $O= \varnothing$ and $C_{existing}=\{c_i\}$ is the set of components present in the current partial composition such that $i=1..N$. $T_{existing}$ is the set of data mapping function which connects the components in C. Also assume $M_{fragment}= < C_{fragment}, T_{fragment}, O >$ is a partial composition where O may or may not be $\varnothing$. $M_{fragment}$ can be associated with the existing composition in the development canvas to complete the composition definition such that $(M_{existing} \cup M_{fragment}) = M <N, C, T, O>$; where M is consistent. If $O = \varnothing$ then we say that the associated fragment is used for extending the definition of $M_{existing}$, otherwise associated fragment auto-completes $M_{existing}$. Given $M_{existing}$, , associated fragment pattern captures the association rules $(M_{existing} \rightarrow \{M_{fragment}\})$ such that $\{M_{fragment}\}$ contains set of fragments with their corresponding support and confidence values. For each of the elements of the set $\{M_{fragment}\}$ there exists a at least one connection between q_i and P_j, where q_i is the output of a component c_i and $c_j \in M_{existing}$ and P_i is the input parameter of a component c_j and $c_j \in M_{fragment}$. Using standard data mining technique e.g., association rule mining etc we can identify and mine frequently occurring composition fragments sets from the past successful compositions and can store these knowledge in our knowledge-base. This knowledge if provided as development recommendation, can be helpful in automating the composition task and can be used to leverage faster development and maximum reuse of existing composition knowledge in order to help end-users in their composition tasks.

- *Consequences:*
    - Associated composition fragment pattern is basically a union of one or more patterns as described before.

However if the developer wishes to know the final composition model instead of knowing the constituent blocks individually, this pattern can be useful in that scenario.

4. DISCUSSION

For the sake of the simplicity of our analysis, in this paper we have considered Yahoo! Pipes, as a reference data-flow based mashup development environment. Yahoo! Pipes provides a simple visual drag-and-drop metaphor for application development instead of writing code. Based upon the meta-model of Yahoo Pipes as introduced in [1] and composition language provided by the platform, we have identified six types of composition patterns as shown in the previous section. However to verify the applicability of these composition patterns in all the data-mashup domains, we have further explored other popular data mashup platforms e.g. Presto Wires[1] and MyCocktail[2]. To anticipate our analysis on these platforms, we have developed applications in these platforms which implement similar/same scenario as described in Section 2 (as shown in Figure 8 and Figure 9 [Appendix A]). During our analysis of the development steps in these platforms, we could successfully map all the identified composition patterns to the corresponding composition languages as provided by these tools. Based upon our observations, we can hence infer that the 6 composition patterns, as described in this paper can well represent different composition aspects supported by the composition languages that are used for data-flow based mashup development.

In our research approach in WIsdom AwaRE (WIRE) computing [1], we aim at developing an assisted mashup development platform. In WIRE we provide development recommendations during development about the next possible composition steps based upon user actions and partial composition information, with a view that by following the recommendations the users can successfully define their mashup applications. The patterns as discussed in this paper can be a good base for providing development recommendations at different levels of abstraction. We claim that development recommendations on next component, connector, or the possible value set for a given parameter etc which are derived from the composition patterns are more useful to the users during their development tasks. To verify this claim recently we have performed a user study [9] with 10 non-IT administrators of a university. The result of the study reveals the fact that the end-user indeed would like to receive development assistance at different levels of granularity during development. The end-users also expressed their concerns about the existing assisted development platforms, which by auto-completing the partial composition provide little or no room for the end-users to have control over the intermediate steps. However the assistance, which is harvested from the patterns, as discussed in this paper will provide them more control over the intermediate steps. We claim that development recommendations on next component, connector, or the possible value set for a given parameter etc are more useful to an end-user than auto-completion. We also claim that these sets of recommendations will help users to learn about how to define the composition logic in their application. In our approach in WIRE we aim at deriving development recommendations from the *community composition knowledge*, which is again captured from the composition patterns that occurred frequently in past successful compositions. The composition patterns, as discussed in this paper, can be discovered by applying data-mining techniques on the existing composition models. In WIRE in particular, we want to explore and extend the standard data mining techniques like frequent itemsets, *association rule mining* etc for discovering the patterns from the existing composition logs. However we also realize that in case of incomplete or uncertain data these pattern-mining techniques may not work properly. In future work we will direct our research efforts in order to tackle the challenges related to data mining in the presence of incomplete/uncertain data.

The composition patterns in this paper will be helpful in understanding and knowing which composition knowledge are important and are required to be captured as patterns in order to provide them as useful development recommendations. In our future work we will further explore to analyze the contexts under which certain composition patterns can be recommended during the development process.

5. LITERATURE REVIEW

The idea of developing large-scale applications by composing coarse grained, reusable component modules has been well established by [12]. A similar, approach has been proposed in the parallel computing domain [13]. In this case, sequential procedures are composed into a parallel structure using a control flow based graphical notation, where the data flow is derived implicitly by matching parameter names [14], later these parallel structures are reused as knowledge. In the past, there have also been many approaches, which had tried to tackle the problem of extending visual data flow languages with iteration constructs [10]. An example of iteration through vector operators and conditional switches is described in [11]. The main drawback of these approaches is, the patterns only capture the structural behavior of the composition, that too only the variation points (join, split etc), the association between the data sources, relationship of data sources with data flow logic are not captured in these approaches. In our approach as described in this paper, instead of only capturing the iterative structure in a composition, we capture the composition steps, which have occurred frequently over the past successful compositions. The composition patterns as described in this paper capture the iterative structural patterns implicitly along with other related information about the data-flow logic. Hence we can say that the patterns as discussed in this paper are more complete and useful in capturing the composition knowledge in visual programming like mashup development paradigm.

6. Conclusion

In this paper we discussed about the mashup composition patterns, which can be identified during mashup application development. By analyzing the contexts, problems and the factors related to different composition steps, we have identified and formalized *five* mashup composition patterns. To validate the generality of these patterns, we have further explored the mashup composition languages of other data mashup platforms. The result of this experiment shows the applicability and generality of the identified composition patterns in data-flow based mashup platforms. In this

[1] http://www.jackbe.com/products/wires.php

[2] http://www.ict-romulus.eu/MyCocktail/

paper, however, we have restricted our analysis to only data mashup platforms. However this set of composition patterns may not be exhaustive. In "*Process mashup"* we may have different set of representative patterns, which require further research efforts and analysis. In our future work we will analyze the meta-model of such process flow based mashup composition languages and will try to map or extend these composition patterns to support both data-flow based and process-flow based mashup developments.

7. ACKNOWLEDGMENTS

We convey our sincere thanks to our shepherd Christian Kohls for his constructive and supportive help during the shepherding process in order to improve the quality of the patterns and the paper. We would also like to thank Kristian Sorensen, from EuroPLoP 2011 program committee, for his extended help and support during the shepherding process. This work is supported by the funds from European Commission (project OMELETTE, contract no. 257635).

8. REFERENCES

[1] Soudip Roy Chowdhury, Carlos Rodríguez, Florian Daniel and Fabio Casati. Wisdom-Aware Computing: On the Interactive Recommendation of Composition Knowledge. Proceedings of WESOA 2010, December 2010, Springer.

[2] Michael Ogrinz. 2009. Mashup Patterns: Designs and Examples for the Modern Enterprise (1 ed.). Addison-Wesley Professional.

[3] Florian Daniel, Agnes Koschmider, Tobias Nestler, Marcus Roy, Abdallah Namoun. Toward Process Mashups: Key Ingredients and Open Research Challenges. Proceedings of Mashups 2010, December 2010, ACM

[4] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, and L. Yan. From People to Services to UI: Distributed Orchestration of User Interfaces. In Proceedings of BPM'10, pages 310–326., 2010.

[5] David E. Simmen, Mehmet Altinel, Volker Markl, Sriram Padmanabhan, and Ashutosh Singh. 2008. Damia: data mashups for intranet applications. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08). ACM, New York, NY, USA, 1171-1182.

[6] Jeffrey Wong and Jason 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). ACM, New York, NY, USA, 1435-1444

[7] Martin Fowler. 1996. Analysis Patterns: Reusable Objects Models. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

[8] Colette Rolland and Naveen Prakash. 1993. Reusable Process Chunks. In Proceedings of the 4th International Conference on Database and Expert Systems Applications (DDEXA '93), Springer-Verlag, London, UK, 655-666.

[9] De Angeli, Antonella and Battocchi, Alberto and Roy Chowdhury, Soudip and Rodriguez, Carlos and Daniel, Florian and Casati, Fabio (2011) Conceptual Design and Evaluation of WIRE: A Wisdom-Aware EUD Tool. Technical Report DISI-11-353, Ingegneria e Scienza dell'Informazione, University of Trento.

[10] Mosconi, M. and Porta, M. Iteration constructs in data-flow visual programming languages. In Proceedings of Comput. Lang. 2000, 67-104.

[11] M. Auguston and A. Delgado. Iterative constructs in the visual data flow lan- guage. In G. Tortora, editor, Proceedings of the 1997 IEEE Symposium on Visual Languages (VL97), pages 152–159, Capri, Italy, September 1997

[12] G. Wiederhold, P. Wegner, and S. Ceri. Towards mega programming: A paradigm for component-based programming. Communications of the ACM, 35(11):89–99, 1992

[13] J. C. Browne, S. I. Hyder, J. Dongarra, K. Moore, and P. Newton. Visual programming and debugging for parallel computing. IEEE parallel and distributed technology: systems and applications, 3(1):75–83, Spring 1995 .

[14] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. 1994. The PVM concurrent computing system: evolution, experiences, and trends. *Parallel Comput.* 20, 4 (April 1994), 531-545

Appendix A

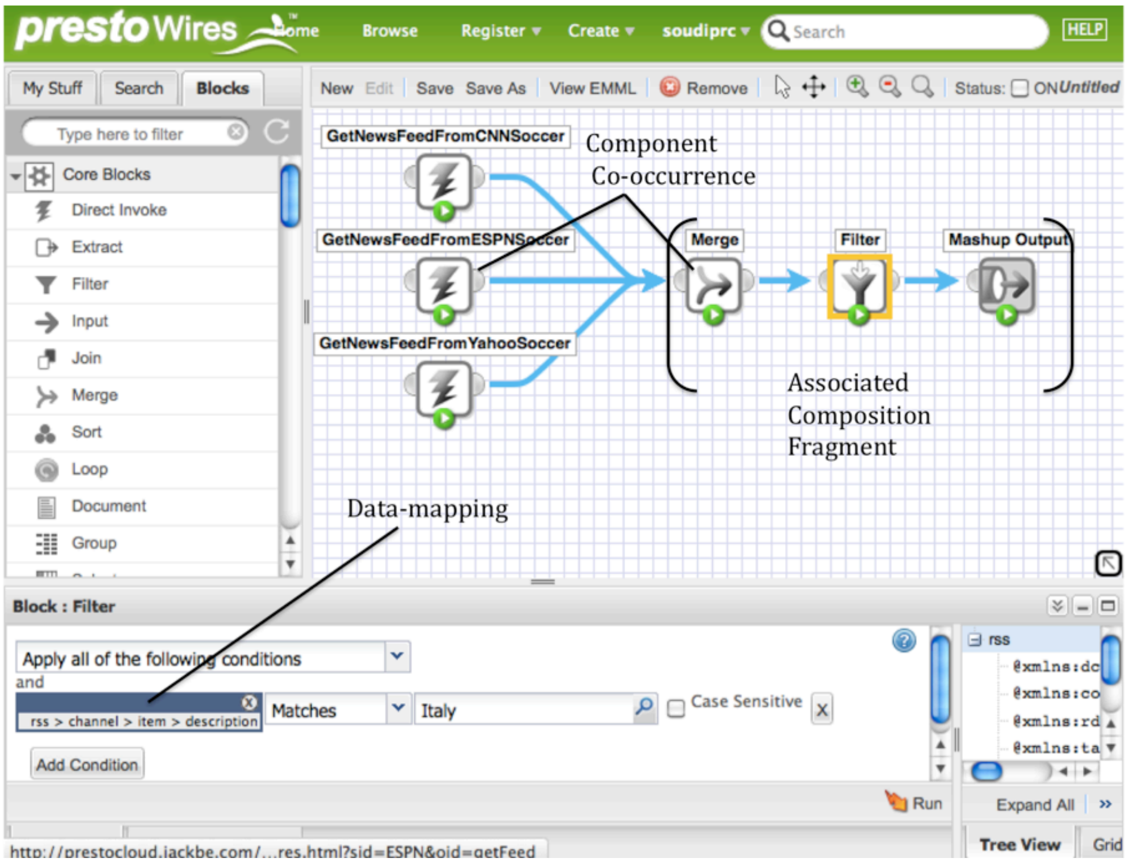
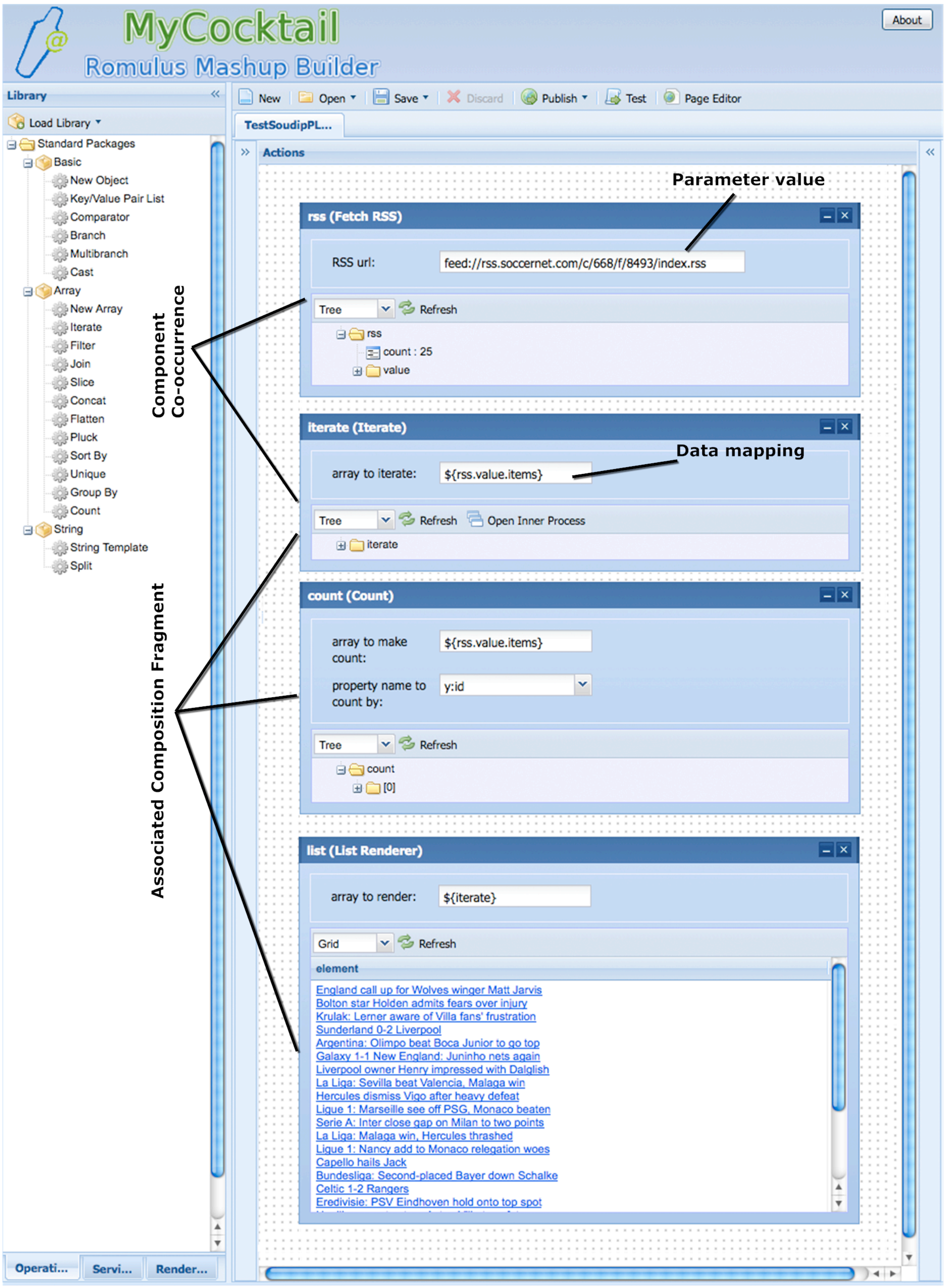


Figure 7 Composition Patterns in Presto Wires

Figure 7 shows an implementation of a simple mashup application in Presto Wires platform. This mashup consists of 6 components. In this example *Direct Invoke* components (*GetNewsFeedFromCNNSoccer*,*GetNewsFeedFromESPNSoccer, GetNewsFeedFromYahooSoccer)* fetch rss-feeds from *URLs* of the websites as mentioned in the *Resource Link* parameter value. *Merge* component then merges the feeds based upon the condition specified in its configuration. Finally *Filter* component filters the merged data based upon the conditions specified by the developer (shown as *Block:Filter Configuration setting* in Figure 7) and provides the filtered data to the *Mashup Output* component. Mapping of composition patterns, as discussed in this paper, to the composition language of Presto Wires validates the applicability of *five composition patterns* in other data flow based mashup composition language as well. To further support this claim we tried to map these five composition patterns to MyCocktail, another data flow based mashup platform. Figure 8 shows an implementation of the mashup scenario as described in section 2 by using MyCocktail mashup builder. This application can also be viewed at this link (http://www.ict-romulus.eu/MyCocktail/#107). This mashup consists of 4 components. The first component in this composition is Fetch RSS service, which fetches the soccer news from the URL as specified in RSS url parameter. The next component *Iterate*, iterates through all the items in the input list and stores them in a temporary array *iterate*. Count component counts the elements of an array based upon some property value of array elements. In this example the elements are counted by the property *id*. Finally UI component *List Renderer* is used for rendering the news in the temporary array. In this example scenario as shown in Figure 8, we can see how the composition patterns, as defined in this paper, can be mapped to the composition language of MyCocktail.



RSS url: feed://rss.soccernet.com/c/668/f/8493/index.rss

count : 25

array to iterate: ${rss.value.items}

array to make count: ${rss.value.items}

property name to count by: y:id

array to render: ${iterate}

element
England call up for Wolves winger Matt Jarvis
Bolton star Holden admits fears over injury
Krulak: Lerner aware of Villa fans' frustration
Sunderland 0-2 Liverpool
Argentina: Olimpo beat Boca Junior to go top
Galaxy 1-1 New England: Juninho nets again
Liverpool owner Henry impressed with Dalglish
La Liga: Sevilla beat Valencia, Malaga win
Hercules dismiss Vigo after heavy defeat
Ligue 1: Marseille see off PSG, Monaco beaten
Serie A: Inter close gap on Milan to two points
La Liga: Malaga win, Hercules thrashed
Ligue 1: Nancy add to Monaco relegation woes
Capello hails Jack
Bundesliga: Second-placed Bayer down Schalke
Celtic 1-2 Rangers
Eredivisie: PSV Eindhoven hold onto top spot

Figure 8 Composition Patterns in MyCocktail Mashup Builder

Appendix D

Efficient, Interactive Recommendation of Mashup Composition Knowledge

Soudip Roy Chowdhury, Florian Daniel, Fabio Casati. Efficient, Interactive Recommendation of Mashup Composition Knowledge. Proceedings of ICSOC 2011, LNCS, December 2011, Springer, Pages 374388.

Efficient, Interactive Recommendation of Mashup Composition Knowledge

Soudip Roy Chowdhury, Florian Daniel, 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 re-usable 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 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). An

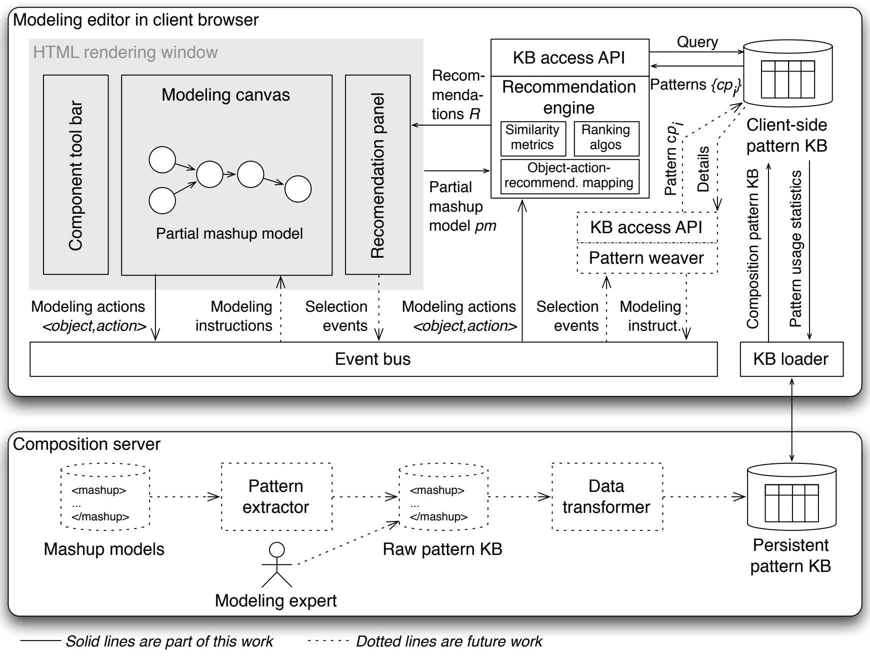


Figure 1. Simplified architecture of the assisted modeling environment with client-side knowledge base and interactive recommender. We focus on recommendations only and omit elements like the mashup runtime environment, the component library, etc.

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

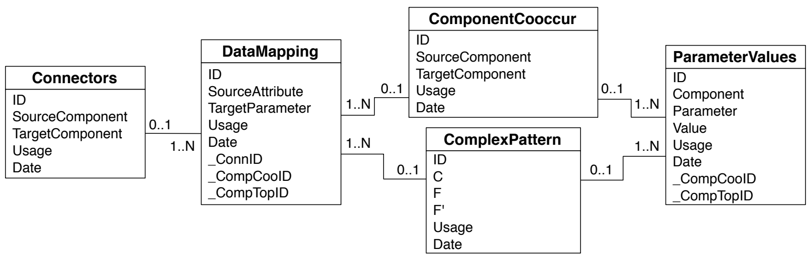


Figure 2. Model of the pattern knowledge base for client-side knowledge management.

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

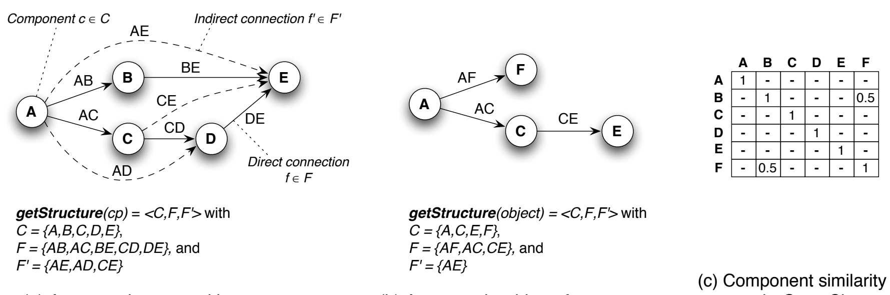
Algorithm 1: getRecommendations

```
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 $recType \in recTypeToBeGiven$ do
5      if $recType \in \{ParValue, Connector, DataMapping, CompCooccur\}$ then
6          $Patterns = Patterns \cup$ queryPatterns($object, KB, recType$) ;      // exact query
7      else
8          $Patterns = Patterns \cup$
           getSimilarPatterns($object, KB.ComplexPattern, CompSim, T_{sim}$) ; // similarity
           search
9  foreach $pat \in Patterns$ do
10     if $rank(pat.cp, pat.sim, pm) \geq T_{rank}$ then
11         append($R, \langle pat.cp, rank(pat.cp, pat.sim, pm) \rangle$) ;      // 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 pat-



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

(a) An example composition pattern *cp* (b) An example *object* of a query *q* (c) Component similarity matrix *CompSim*

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

tern KB. For the purpose of recommending knowledge, similarity values should 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

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 $sim_i \geq T_{sim}$
$Patterns$ = set();
$objectStructure$ = getStructure($object$) ;   // computes object's structure for comparison
foreach $cp \in CP$ do
    $obj = objectStructure$;
    $sim$ = getSimilarity($obj, cp$) ;   // compute similarity for exact matches
    $obj.C = obj.C - cp.C$ ;   // eliminate all exact matches for C, F, F' from obj
    $obj.F = obj.F - cp.F - cp.F'$;
    $obj.F' = obj.F' - cp.F' - cp.F$;
    $approxSim = 0$;   // will contain the best similarity for approximate matches
    foreach $c \in obj.C$ do
        $SimC$ = getSimilarComponents($c, CompSim$) ;   // get set of similar components
        foreach $simc \in SimC$ do
            $approxObj$ = getApproximatePattern($obj, c, simc$) ;   // get approx. pattern
            $newApproxSim = simc.sim *$getSimilarity($approxObj, cp$) ;   // get similarity
            if $newApproxSim > approxSim$ then
                $approxSim = newApproxSim$ ;   // keep highest approximate similarity
    $sim = sim + approxSim * |obj.C|/|objectStructure.C|$ ;   // normalize and aggregate
    if $sim \geq T_{sim}$ then
        $Patterns = Patterns \cup \langle cp, sim \rangle$ ;   // remember patterns with sufficient sim
return $Patterns$;
```

Algorithm 3: getSimilarity

```
Data: query object $object$, complex pattern $cp$
Result: $similarity$
initialize $w_i$ for $i \in 1..5$ with $\sum_i w_i = 1$;
$sim_1 = |object.C \cap cp.C|/|object.C|$ ;       // matches components
$sim_2 = |object.F \cap cp.F|/|object.F|$ ;       // matches connectors
$sim_3 = |object.F \cap cp.F'|/|object.F|$ ;      // allows insertion of a component
$sim_4 = |object.F' \cap cp.F|/|object.F'|$ ;     // allows deletion of a component
$sim_5 = |object.F' \cap cp.F'|/|object.F'|$ ;    // allows substitution of a component
$similarity = \sum_i w_i * sim_i$;
return $similarity$;
```

Firefox 3.6.17 web browser. The implementation of the *client-side KB* is based 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

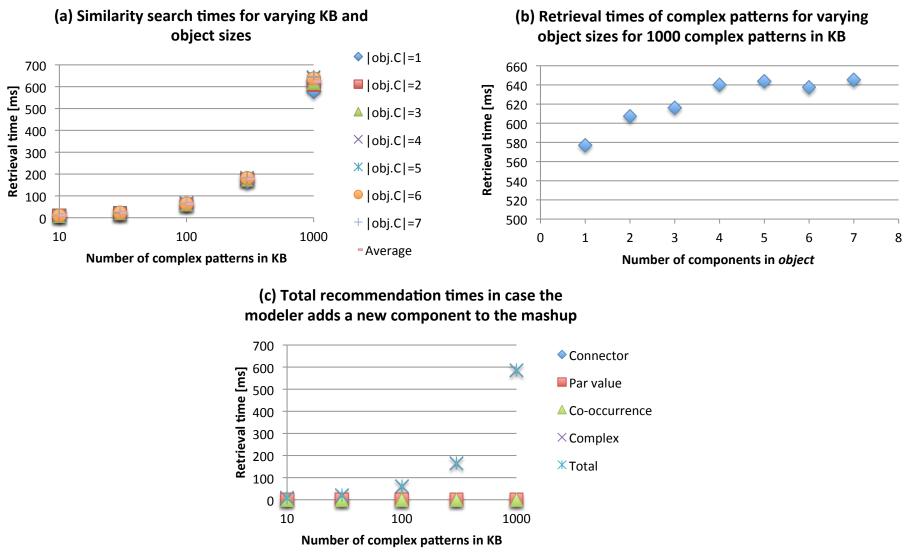


Figure 4. Performance evaluation of the client-side knowledge recommender.

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 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'11, Springer (2011)
2. Roy Chowdhury, S., Rodríguez, C., Daniel, F., Casati, F.: Wisdom-aware computing: On the interactive recommendation of composition knowledge. In: WESOA'10, Springer (2010) 144–155
3. Daniel, F., Casati, F., Benatallah, B., Shan, M.C.: Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In: ER'09. ER '09, Berlin, Heidelberg, Springer-Verlag (2009) 428–443
4. Hlaoui, A., Wang, S.: A new algorithm for inexact graph matching. In: ICPR'02. Volume 4. (2002) 180 – 183
5. Carlson, M.P., Ngu, A.H., Podorozhny, R., Zeng, L.: Automatic mash up of composite applications. In: ICSOC'08, Springer (2008) 317–330
6. Greenshpan, O., Milo, T., Polyzotis, N.: Autocompletion for mashups. VLDB'09 **2** (2009) 538–549
7. Chen, H., Lu, B., Ni, Y., Xie, G., Zhou, C., Mi, J., Wu, Z.: Mashup by surfing a web of data apis. VLDB'09 **2** (2009) 1602–1605
8. Riabov, A.V., Boillet, E., Feblowitz, M.D., Liu, Z., Ranganathan, A.: Wishful search: interactive composition of data mashups. In: WWW'08, ACM (2008) 775–784
9. Elmeleegy, H., Ivan, A., Akkiraju, R., Goodwin, R.: Mashup advisor: A recommendation tool for mashup development. In: ICWS'08, IEEE Computer Society (2008) 337–344
10. Beauche, S., Poizat, P.: Automated service composition with adaptive planning. In: ICSOC'08, Springer-Verlag (2008) 530–537
11. Smirnov, S., Weidlich, M., Mendling, J., Weske, M.: Action patterns in business process models. In: ICSOC-ServiceWave'09, Springer-Verlag (2009) 115–129
12. Hornung, T., Koschmider, A., Lausen, G.: Recommendation based process modeling support: Method and user experience. In: ER'08, Springer (2008) 265–278
13. Gschwind, T., Koehler, J., Wong, J.: Applying patterns during business process modeling. In: BPM'08, Springer (2008) 4–19
14. Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases **14** (2003) 5–51

Appendix E

Discovery and Reuse of Composition Knowledge for Assisted Mashup Development

Florian Daniel, Carlos Rodrguez, Soudip Roy Chowdhury, Hamid R. Motahari Nezhad, Fabio Casati. Discovery and Reuse of Composition Knowledge for Assisted Mashup Development. WWW 2012 Companion, April 2012, Pages 493-494.

Discovery and Reuse of Composition Knowledge for Assisted Mashup Development

Florian Daniel[1], Carlos Rodríguez[1], Soudip Roy Chowdhury[1],
Hamid R. Motahari Nezhad[2], and Fabio Casati[1]
[1]University of Trento, Italy, [2] HP Labs Palo Alto, USA
[1]{daniel,crodriguez,rchowdhury,casati}@disi.unitn.it, [2]hamid.motahari@hp.com

ABSTRACT

Despite the emergence of mashup tools like Yahoo! Pipes or JackBe Presto Wires, developing mashups is still *non-trivial* and requires intimate knowledge about the functionality of web APIs and services, their interfaces, parameter settings, data mappings, and so on. We aim to *assist the mashup process* and to turn it into an interactive co-creation process, in which one part of the solution comes from the developer and the other part from *reusable composition knowledge* that has proven successful in the past. We harvest composition knowledge from a repository of existing mashup models by mining a set of reusable *composition patterns*, which we then use to interactively provide *composition recommendations* to developers while they model their own mashup. Upon acceptance of a recommendation, the purposeful design of the respective pattern types allows us to *automatically weave* the chosen pattern into a partial mashup model, in practice performing a set of modeling actions on behalf of the developer. The experimental evaluation of our prototype implementation demonstrates that it is indeed possible to harvest meaningful, reusable knowledge from existing mashups, and that even complex recommendations can be efficiently queried and weaved also inside the client browser.

Categories and Subject Descriptors

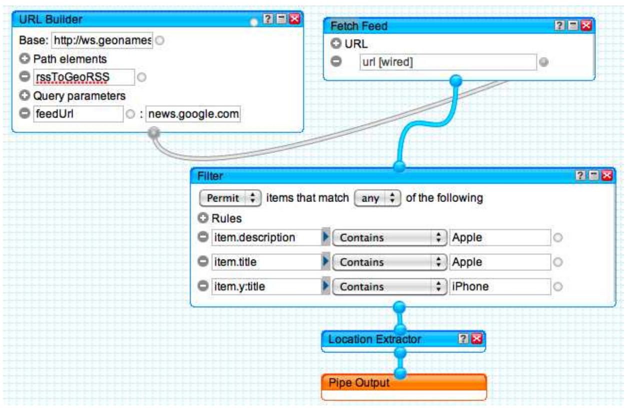
D.2.6 [**Software**]: Software Engineering—*Programming Environments*

Keywords

Assisted mashup development, End user development, Composition patterns, Pattern recommendation, Weaving

1. INTRODUCTION

Mashup tools, such as Yahoo! Pipes (`http://pipes.yahoo.com/pipes/`) or JackBe Presto Wires (`http://www.jackbe.com`), generally promise easy development tools and lightweight runtime environments, both typically running inside the client browser. By now, mashup tools undoubtedly simplified some complex composition tasks, such as the integration of web services or user interfaces. Yet, despite these advances in simplifying technology, mashup development is still a *complex task* that can only be managed by skilled developers.



Base: http://ws.geonames
Path elements
rssToGeoRSS
Query parameters
feedUrl : news.google.com
URL
url [wired]
Permit items that match any of the following
Rules
item.description Contains Apple
item.title Contains Apple
item.y:title Contains iPhone

Figure 1: A typical pattern in Yahoo! Pipes

Figure 1 illustrates a Yahoo! Pipes model that encodes how to plot news items on a map. The lesson that can be learned from it is that plotting news onto a map requires enriching the news feed with geo-coordinates, fetching the actual news items, and handing the items over to the map. Understanding this logic is neither trivial nor intuitive.

In order to aid less skilled developers in the design of mashups like the one above, Carlson et al. [1], for instance, leverage on semantic annotations of components to recommend compatible components, given a component in the canvas. Greenshpan et al. [3] recommend components and connectors (so-called glue patterns) in response to the user providing a set of desired components. Elmeleegy et al. [2] recommend a set of components related to a component in the canvas, leveraging on conditional co-occurrence and semantic matching, and automatically plan how to connect selected components to the partial mashup. Riabov et al. [4] allow users to express goals as keywords, in order to feed an automated planner that derives candidate mashups.

We assist the modeler in each step of his development task by means of ***interactive, contextual recommendations of composition knowledge***. The knowledge is reusable *composition patterns*, i.e., fragments of mashup models. Such knowledge may come from a variety of possible sources; we specifically focus on community composition knowledge (recurrent model fragments in a mashup model repository). In this poster, we describe (i) how we mine *mashup composition patterns*, (ii) the *architecture* of our knowledge recommender, (iii) its *recommendation algorithms*, and (iv) its *pattern weaving algorithms* (automatically applying patterns to mashup models).

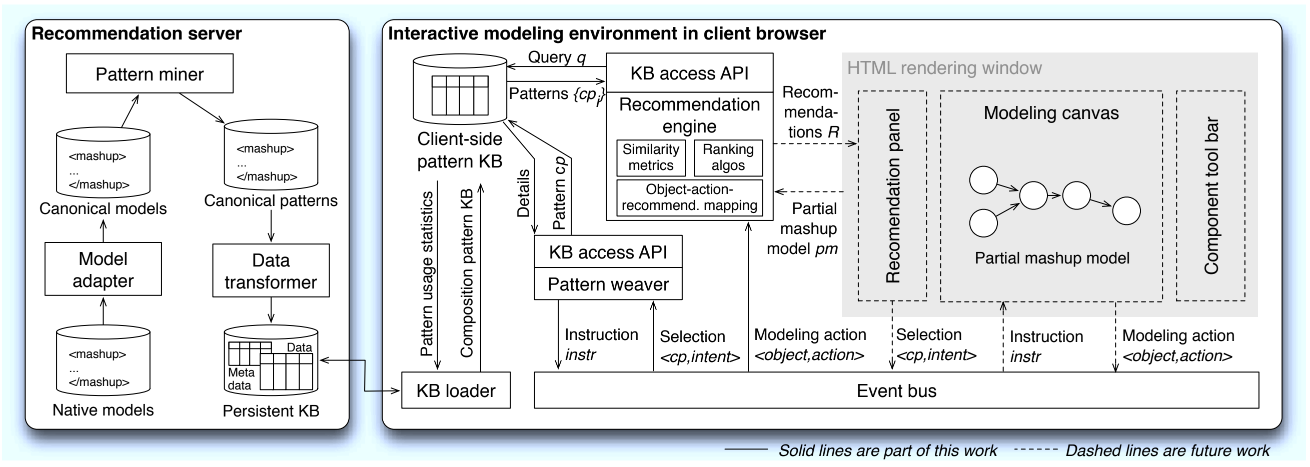


Figure 2: Functional architecture of the composition knowledge discovery and recommendation approach

2. THE RECOMMENDATION PLATFORM

Figure 2 details our knowledge discovery and recommendation prototype. The ***pattern discovery*** logic is located in the server. After converting mashup models into a canonical format, the *pattern miner* extracts patterns, which we store into a knowledge base (KB) that is structured to minimize pattern retrieval at runtime. We support six composition pattern types: *parameter value*, *connector*, *connector co-occurrence*, *component co-occurrence* , *component embedding*, and *multi-component* patterns (cf. Figure 1).

The *interactive modeling environment* runs in the client. It is here where the ***pattern recommendation*** logic reacts to modeling *actions* performed by the modeler on a construct (the *object* of the action) in the canvas. For instance, we can *drop* a component onto the canvas, or we can *select* a parameter. Upon each interaction, the *action* and its *object* are published on a browser-internal *event bus*, which forwards them to the *recommendation engine*. With this information and the partial mashup model *pm* the engine queries the *client-side KB* for recommendations, where an *object-action-recommendation mapping* tells the engine which types of recommendations are to be retrieved. The list of patterns retrieved from the KB are then ranked and rendered in the *recommendation panel*.

Upon the selection of a pattern from the recommendation panel, the ***pattern weaver*** weaves it into the partial mashup model. The *pattern weaver* first retrieves a *basic weaving strategy* (a set of model-agnostic mashup instructions) and then derives a *contextual weaving strategy* (a set of model-specific instructions), which is used to weave the pattern. Deriving the contextual strategy from the basic one may require the resolution of possible *conflicts* among the constructs of the partial model and those of the pattern to be weaved. The pattern weaver resolves them according to a configurable conflict resolution policy.

Our ***prototype*** is a Mozilla Firefox extension for Yahoo! Pipes [6], with the recommendation and weaving algorithms implemented in JavaScript. Event listeners listen for DOM modifications, in order to identify mashup modeling actions inside the modeling canvas. The instructions in the weaving strategies refers to modeling actions, which are implemented as JavaScript manipulations of the mashup model's DOM elements. The server-side part is implemented in Java.

3. EVALUATION

For our experiments we extracted 303 pipes definitions from the repository of Pipes. The average numbers of components, connectors and input parameters were 12.7, 13.2 and 3.1, respectively, indicating fairly complex mashups. We were able to identify patterns of all the types described above. For example, the minimum/maximum support for the *connector patterns* was 0.0759/0.3234, while the one for the *component co-occurrence patterns* was 0.0769/0.2308. We used these patterns to populate our KB and generated additional synthetic patterns to test the performance of the recommendation engine (the sizes of the KBs ranged from 10, 30, 100, 300, 1000 multi-component patterns) [5]. The complexity of the patterns ranged from $3-9$ components per pattern, and we used queries with $1-7$ components. In the worst case scenario (KB of 1000 patterns, approximate similarity matching of patterns), the recommendation engine could retrieve relevant patterns within 608 millisecond – everything entirely inside the client browser. The next step is going online and performing users studies.

Acknowledgment. This work was supported by the European Commission (project OMELETTE, contract 257635).

4. REFERENCES

[1] M. P. Carlson, A. H. Ngu, R. Podorozhny, and L. Zeng. Automatic mash up of composite applications. In *ICSOC'08*, pages 317–330, 2008.

[2] H. Elmeleegy, A. Ivan, R. Akkiraju, and R. Goodwin. Mashup advisor: A recommendation tool for mashup development. In *ICWS'08*, pages 337–344, 2008.

[3] O. Greenshpan, T. Milo, and N. Polyzotis. Autocompletion for mashups. *VLDB'09*, 2:538–549, 2009.

[4] A. V. Riabov, E. Boillet, M. D. Feblowitz, Z. Liu, and A. Ranganathan. Wishful search: interactive composition of data mashups. In *WWW'08*, pages 775–784, 2008.

[5] S. Roy Chowdhury, F. Daniel, and F. Casati. Efficient, Interactive Recommendation of Mashup Composition Knowledge. In *ICSOC'11*, pages 374–388, 2011.

[6] S. Roy Chowdhury, C. Rodríguez, F. Daniel, and F. Casati. Baya: Assisted Mashup Development as a Service. In *WWW'12*, 2012.

Appendix F

Baya: Assisted Mashup Development as a Service

Soudip Roy Chowdhury, Carlos Rodrguez, Florian Daniel, Fabio Casati. Baya: Assisted Mashup Development as a Service. WWW 2012 Companion, April 2012, Pages 409-412.

Baya: Assisted Mashup Development as a Service

Soudip Roy Chowdhury, Carlos Rodríguez, Florian Daniel and Fabio Casati
University of Trento
Via Sommarive 5, 38123 Povo (TN), Italy
{rchowdhury,crodriguez,daniel,casati}@disi.unitn.it

ABSTRACT

In this demonstration, we describe *Baya*, an extension of Yahoo! Pipes that *guides* and *speeds up* development by interactively recommending composition knowledge harvested from a repository of existing pipes. Composition knowledge is delivered in the form of *reusable mashup patterns*, which are retrieved and ranked on the fly while the developer models his own pipe (the mashup) and that are automatically weaved into his pipe model upon selection. Baya mines candidate patterns from pipe models available online and thereby leverages on the *knowledge of the crowd*, i.e., of other developers. Baya is an extension for the Firefox browser that seamlessly integrates with Pipes. It enhances Pipes with a powerful new feature for both *expert developers* and *beginners*, speeding up the former and enabling the latter. The discovery of composition knowledge is provided *as a service* and can easily be extended toward other modeling environments.

Categories and Subject Descriptors

H.m [**Information Systems**]: Miscellaneous; D.1 [**Software**]: Programming Techniques; D.2.6 [**Software**]: Software Engineering—*Programming Environments*

Keywords

Baya, Assisted mashup development, Composition patterns, Pattern mining, Pattern recommendation, Weaving

1. INTRODUCTION

Mashup tools, such as Yahoo! Pipes (`http://pipes.yahoo.com/pipes/`) or JackBe Presto Wires (`http://www.jackbe.com`), simplify the development of composite applications by means of easy development paradigms (e.g., using visual programming metaphors) and hosted runtime environments that do not require the installation of any client-side software. Yet, despite the initial goal of enabling end users to develop own applications and the advances in simplifying technology, mashup development is still a *complex task* that can only be managed by skilled developers.

For instance, Figure 1 illustrates a Yahoo! Pipes model that encodes how to plot news items on a map. The example shows that understanding and modeling the logic for building such a mashup is ***neither trivial nor intuitive***. Firstly,

WWW 2012 Companion, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1230-1/12/04.

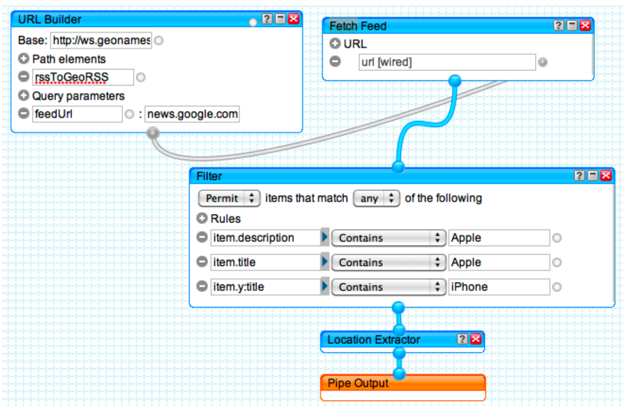


Figure 1: A typical pattern in Yahoo! Pipes

we need to enrich the news feed with geo-coordinates, then, we must fetch the actual news items, and only then we can plot the items on a map. If modeling difficulties arise, it is common practice to manually search the Web for *examples* or *help* on which components to use, on how to fill the respective parameter fields, or on how to propagate data.

In order to aid less skilled developers in the design of mashups like the one above, in programming by demonstration [1], for instance, the system aims to auto-complete a process definition, starting from a set of user-selected model examples. Goal-oriented approaches [4] aim to assist the user by automatically deriving compositions that satisfy user-specified goals. Pattern-based development [3] aims at recommending connector patterns (so-called glue patterns) in response to user selected components (so-called mashlets) in order to autocomplete the partial mashup. Syntactic approaches [7] suggest modeling constructs based on syntactic similarity (comparing output and input data types), while semantic approaches [5] annotate constructs to support suggestions based on the meaning of constructs. The limitations in these approaches lie in the fact that they overlooked the perspectives for end user development, as they either still require advanced modeling skills (which users don't have), or they expect the user to specify complex rules for defining goals (which they are not able to), or they expect domain experts to specify and maintain the semantics of the modeling constructs (which they don't do).

Driven by a user study on how end users would like to be assisted during mashup development [2], we have developed ***Baya***, a plug-in for Yahoo! Pipes that provides *interactive, contextual recommendations of reusable composition knowl-*

edge. The knowledge Baya recommends is re-usable *composition patterns*, i.e., model fragments that bear knowledge about how to compose mashups, such as the one in Figure 1. For instance, Baya may suggest a candidate next component or a whole chain of constructs. Upon selection of a recommendation, Baya weaves the respective pattern automatically into the current model in the modeling canvas[1]. Baya mines community composition knowledge from existing mashup models publicly available in the online Yahoo! Pipes repository and provides the respective patterns as a service to client-side modeling environments.

In this demo paper, we describe Baya, outline the concepts and architecture behind its simple user interface, and provide insight into its implementation and future evolution.

2. THE BAYA APPROACH

Baya aims to seamlessly extend existing mashup or composition instruments with advanced knowledge reuse capabilities. It targets both expert developers and beginners and aims to speed up the former and to enable the latter.

The ***design goals*** behind Baya can be summarized as follows: We didn't want to develop *yet another* mashup environment; so we opted for an extension of existing and working solutions (in this demo, we focus on Yahoo! Pipes; other tools will follow). We wanted to *reuse* composition knowledge that has proven successful in the past; mining modeling patterns from existing mashups allows us to identify exactly this, i.e., recurrent modeling practice. We wanted to support a variety of *different* mashup tools, not just one; as we will see, the sensible design of a so-called canonical mashup model serves exactly this purpose. Modelers should not be required to *ask* for help; we therefore pro-actively and interactively recommend contextual composition patterns. We did not want the *reuse* to be limited to simple copy/paste of patterns, but knowledge should be *actionable*, and therefore, Baya features the automated weaving of patterns.

2.1 Composition Knowledge

Considering the typical actions performed by a developer in a graphical modeling environment (e.g., filling input fields, connecting components, copying/pasting model fragments), Baya specifically supports the following set of ***pattern types***:

- ***Parameter value pattern.*** The parameter value pattern represents a set of recurrent value assignments for the input parameters of a component. This pattern helps filling input parameters of a component that require explicit user input.
- ***Connector pattern.*** The connector pattern represents a recurrent connector between a pair of components, along with the data mapping of the target component. The pattern helps connecting a newly placed component to the partial mashup model in the canvas.
- ***Connector co-occurrence pattern.*** The connector co-occurrence pattern captures which connectors occur together. The pattern also includes the associated data mappings. This pattern is particularly valuable in those cases where people, rather than developing their mashup model in an incremental but connected fashion, first select the desired functionalities (the components) and only then connect them.
- ***Component co-occurrence pattern.*** Similarly, the component co-occurrence pattern captures couples of components that occur together. It comes with the two associated components as well as with their connector, parameter values, and data mapping logic. The pattern helps developing mashups incrementally in a connected fashion.
- ***Component embedding pattern.*** The component embedding pattern captures which component is typically embedded into which other component, both being preceded by another component. The pattern helps, for instance, modeling *loops*, a task that is usually not trivial to non-experts.
- ***Multi-component pattern.*** The multi-component pattern represents recurrent model fragments that are composed of multiple components. It represents more complex patterns, such as the one in Figure 1, that are not yet captured by the other pattern types.

This list of pattern types is extensible and will evolve over time. However, this set of pattern types at the same time leverages on the interactive modeling paradigm of the mashup tools (the patterns represent modeling actions that could also be performed by the developer) and provides as much information as possible.

2.2 Discovery, Recommendation and Weaving

Figure 2 details the internals of the Baya architecture. The overall architecture is devided into two blocks, namely, the *recommendation server* and the *client-side extension* of the chosen mashup tool, i.e., Yahoo! Pipes.

The ***Baya recommendation server*** (at the left in Figure 2) is in charge of discovering and harvesting composition knowledge patterns from existing mashup compositions. The first step for discovering composition patterns consists in taking the *native models* of the target mashup tools from a repository of existing compositions and translating them into a *canonical mashup model*, a step that is performed by a dedicated *model adapter*. The canonical model is able to represent a variety of similar mashup languages and allows the development of more generic mining algorithms. The *pattern miner* runs a set of *pattern mining algorithms* on the data in the canonical model and discovers the above introduced patterns. Discovered patterns are stored back into a database of *canonical patterns*, transformed by the *data transformer*, and loaded into the *persistent knowledge base* (KB). The persistent KB consists in a database that is structured in such a way that patterns can be efficiently queried and retrieved by the *client-side browser extension* for interactive recommendation.

The ***Baya Firefox extension*** consists of two main components: a recommendation engine and a pattern weaver. In the client, we have the actual interactive modeling environment (Pipes), in which the developer can visually compose components by dragging and dropping them from a component tool bar and connecting them together in the canvas. The developer therefore performs composition *actions* (e.g., select, drag, drop, connect, delete, fill, map,...), where the

[1]This is also the capability that inspired the name of the tool: the *Baya weaver* is a so-called weaverbird that weaves its nest with long strips of leaves.

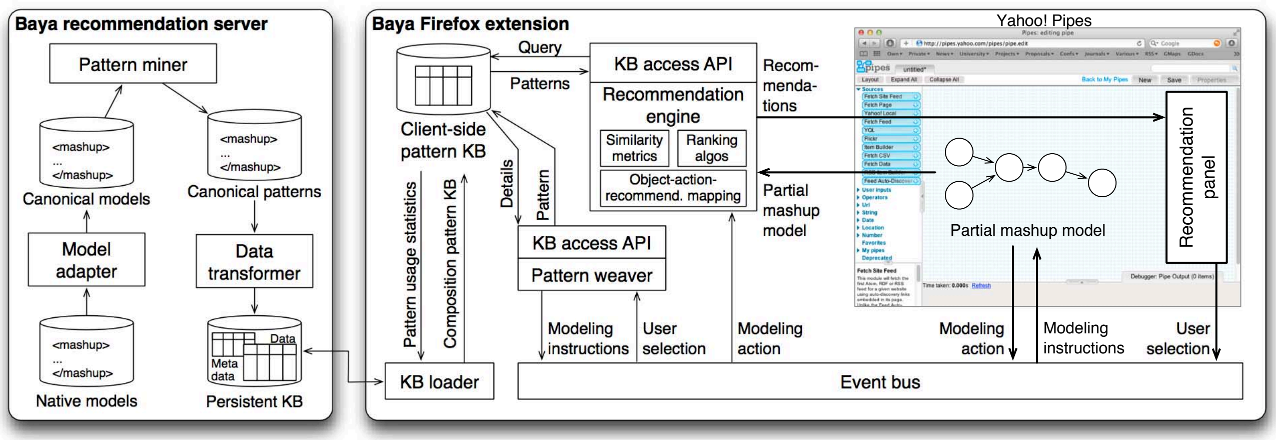


Figure 2: The internals of Baya: functional architecture for pattern discovery, recommendation and weaving

action is performed on a modeling construct in the modeling canvas; we call this construct the *object* of the action. For instance, we can *drop* a component onto the canvas, or we can *select* a parameter to fill it with a value, and so on. Upon each interaction, the *action* and its *object* are published on a browser-internal *event bus*, which forwards them to the ***recommendation engine***. Given a modeling *action*, the *object* it has been applied to, and the partial mashup model, the engine queries the *client-side pattern KB* via the *KB access API* for recommendations (pattern representations) and gets a list of candidate patterns. Baya uses both *exact* and *approximate* pattern matching algorithms [6] to determine the final candidate set of recommendations that also match the current composition context, ranks them in order of their similarity and popularity, and finally renders them in the *recommendation panel*.

Upon the selection of a pattern from the recommendation panel, the ***pattern weaver*** weaves it into the partial mashup model in the modeling canvas. For each supported pattern type, Baya retrieves a *basic weaving strategy* (a static set of modeling instructions; see `http://goo.gl/Xk7VF`), which is independent of the partial mashup model, and derives a *contextual weaving strategy*, which applies the basic strategy to the partial model at runtime. Applying the mashup operations in the basic strategy may require the resolution of possible *conflicts* among the constructs of the partial model and those of the pattern to be weaved. For instance, if we want to add a new component of type *ctype* but the mashup already contains an instance of type *ctype*, say *comp*, we are in the presence of a conflict: either we decide that we reuse *comp*, which is already there, or we decide to create a new instance of *ctype*. In order to choose how to proceed, Baya allows one to choose among different policies (see `http://goo.gl/9jJtK`). Given a final, contextual strategy, the pattern weaver applies the respective modeling actions to the partial mashup model.

Upon successful weaving of a recommended pattern into the partial composition, the *usage statistics* of the selected pattern in the client-side KB get updated, and simultaneously this information is sent to the server-side *persistent KB* via the *KB loader*. This updated metadata is used for future recommendation filtering and ranking. In the Baya client side, we also consider the option for saving patterns, in which users can select and store to the pattern KB new user-defined patterns from their current composition. This feature is part of our on-going development and will be available in future versions of Baya.

3. IMPLEMENTATION

Baya is implemented as Mozilla Firefox (`http://mozilla.com/firefox`) extension for Yahoo! Pipes, adding an interactive recommendation panel at the right of its modeling canvas. Baya implementation is based on JavaScript for the business logic (e.g., the algorithms) and XUL (XML User Interface Language, `https://developer.mozilla.org/En/XUL`) for UI development. The use of JavaScript in Firefox Extension development framework eases the interaction with the HTML DOM elements in the browser window and the implementation of dedicated listeners to intercept modeling events on elements in the DOM tree (e.g., model constructs in the Pipes modeling canvas). A screenshot of Baya in action is shown in Figure 3.

The server side is implemented in Java. This comprises the model adapter (cf. Figure 2), which is able to convert Yahoo! Pipes' internal JSON representation of mashups into our canonical mashup model as well as the necessary mining algorithms for the discovery of the patters (a description of the algorithms can be found at `http://goo.gl/Dis5V`). Parts of our mining algorithms make use of frequent itemset mining, for which we used the tool ARMiner (`http://www.cs.umb.edu/~laur/ARMiner/`).

Discovered patterns are transformed and stored in a knowledge base that is optimized for fast pattern retrieval at runtime. The implementation of the persistent pattern KB at server side, is based on MySQL (`http://www.mysql.com/`). Via a dedicated Java RESTful API, at startup of the recommendation panel the KB loader synchronizes the server-side KB with the client-side KB, which instead is based on SQLite (`http://www.sqlite.org`). The pattern matching and retrieval algorithms are implemented in JavaScript and triggered by events generated by the event listeners monitoring the DOM modifications related to the mashup model.

The weaving algorithms are also implemented in JavaScript. Upon the selection of a recommendation from the panel, they derive the contextual weaving strategy that is necessary to weave the respective pattern into the partial

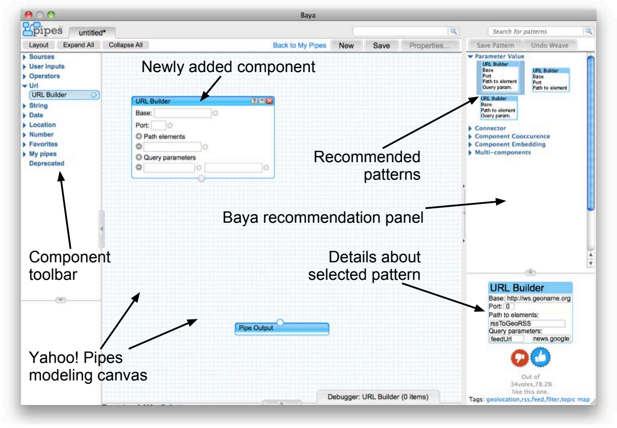

Newly added component
Recommended patterns
Baya recommendation panel
Component toolbar
Details about selected pattern
Yahoo! Pipes modeling canvas

Figure 3: Screenshot of Baya in action.

mashup model. Each of the instructions in the weaving strategy refers to a modeling action, where modeling actions are implemented as JavaScript manipulations of the mashup model's DOM elements. Both the weaving strategies (basic and contextual) are encoded as JSON arrays, which enable us to use the native `eval()` command for fast and easy parsing of the weaving logic.

For our experiments we extracted 303 pipes definitions from the repository of Pipes. The average numbers of components, connectors and input parameters were 12.7, 13.2 and 3.1, respectively, indicating fairly complex mashups. We were able to identify patterns of all the types described above. For example, the minimum/maximum support for the *connector patterns* was 0.0759/0.3234, while the one for the *component co-occurrence patterns* was 0.0769/0.2308. We used these patterns to populate our KB and generated additional synthetic patterns to test the performance of the recommendation engine (the sizes of the KBs ranged from 10, 30, 100, 300, 1000 multi-component patterns) [6]. The complexity of the patterns ranged from $3 - 9$ components per pattern, and we used queries with $1 - 7$ components. In the worst case scenario (KB of 1000 patterns, approximate similarity matching of patterns), the recommendation engine could retrieve relevant patterns within 608 millisecond – everything entirely inside the client browser.

4. DEMONSTRATION STORYBOARD

During the live demonstration, we will showcase Baya at work and take our audience through the theoretical as well as the usage aspects of the tool, using a mix of slides and hands-on examples. In particular, we intend to organize the demonstration as follows:

1. ***Introduction***: A short intro to the goals and key concepts of Baya.

2. ***Example***: A simple example developed by us with the use of the interactive recommendations.

3. ***Non-assisted development by audience***: A similar modeling exercise for a member of the audience, however without the help of the interactive recommender.

4. ***Assisted development by audience***: The same modeling scenario as in 3, this time however with the help of the interactive recommender.

5. ***Patterns and discovery***: An explanation of the pattern types supported by Baya, along with the mining approach underlying the pattern knowledge base.

6. ***Architecture and internals***: Explanation of the internal architecture of Baya and of the recommendation and weaving algorithms working behind the scenes.

7. ***Conclusion***: Lessons learned and outline of future works and the evolution of Baya.

This process will allow us to introduce the audience to Baya and help us evaluate the efficacy and usability of the tool. We hope we will get valuable feedback from the audience, in order to further fine-tune Baya's UI and algorithms.

An introduction to and a screencast of Baya is available at `http://www.youtube.com/watch?v=RNRAsX1CXtE`.

5. STATUS AND LESSONS LEARNED

Baya was born in the context of the EU research project OMELETTE, in order to assist mashup development inside the project's own mashup editors. Soon, however, we recognized that the kind of knowledge discovery algorithms we were working on and the conceptual approach to pattern recommendation and weaving are generic enough to be applied in the context of many other modeling or mashup tools. As a proof of concept, we therefore developed Baya, an apparently simple, yet effective tool. The idea of composition knowledge as a service makes it unique among other assisted development approaches, and a-priori definition of pattern structures allows us to extract meaningful knowledge also from single mashup models.

Next, we will extend the mining algorithms to other composition paradigms and develop dedicated clients for different composition tools. The idea is to make Baya publicly available and to study how effectively pattern recommendation and weaving can help users to develop own mashups.

Acknowledgment. This work was supported by the European Commission (project OMELETTE, contract 257635).

6. REFERENCES

[1] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[2] A. De Angeli, A. Battocchi, S. Roy Chowdhury, C. Rodríguez, F. Daniel, and F. Casati. End-User Requirements for Wisdom-Aware EUD. In *IS-EUD'11*, pages 245–250.

[3] O. Greenshpan, T. Milo, and N. Polyzotis. Autocompletion for mashups. *VLDB'09*, 2:538–549.

[4] M. Henneberger, B. Heinrich, F. Lautenbacher, and B. Bauer. Semantic-Based Planning of Process Models. In *Multikonferenz Wirtschaftsinformatik'08*, 2008.

[5] A. Ngu, M. Carlson, Q. Sheng, and H. young Paik. Semantic-based mashup of composite applications. *IEEE TSC*, 3(1):2 –15, 2010.

[6] S. Roy Chowdhury, F. Daniel, and F. Casati. Efficient, Interactive Recommendation of Mashup Composition Knowledge. In *ICSOC'11*, pages 374–388, 2011.

[7] J. Wong and J. I. Hong. Making mashups with marmite: towards end-user programming for the web. In *CHI'07*, pages 1435–1444.

Appendix G

Assisting End-User Development in Browser-Based Mashup Tools

Soudip Roy Chowdhury. Assisting end-user development in browser-based mashup tools. In Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012). IEEE Press, Piscataway, NJ, USA, 1625-1627.

Assisting End-User Development in Browser-Based Mashup Tools

Soudip Roy Chowdhury
DISI, University of Trento, Italy
rchowdhury@disi.unitn.it

***Abstract*—Despite the recent progresses in end-user development and particularly in mashup application development, developing even simple mashups is still *non-trivial* and requires intimate knowledge about the functionality of web APIs and services, their interfaces, parameter settings, data mappings, and so on. We aim to *assist* less skilled developers in composing own mashups by interactively recommending composition knowledge in the form of modeling patterns and fostering knowledge reuse. Our prototype system demonstrates our idea of *interactive recommendation* and *automated pattern weaving*, which involves recommending relevant composition patterns to the users during development, and once selected, applying automatically the changes as suggested in the selected pattern to the mashup model under development. The experimental evaluation of our prototype implementation demonstrates that even complex composition patterns can be efficiently stored, queried and weaved into the model under development in browser-based mashup tools.**

***Keywords*-assisted development; end-user development; composition pattern; pattern recommendation; weaving**

I. Problem and Motivation

By now, mashup tools, such as Yahoo! Pipes (http://pipes.yahoo.com/pipes/), undoubtedly simplified some complex composition tasks by providing easy modeling constructs and lightweight runtime environments inside the client browser. Yet, despite these advances in simplifying technology, mashup development is still a *complex task* that can only be managed by skilled developers ([1], [2]).

Figure 1 illustrates a Yahoo! Pipes model that encodes how to plot news items on a map. The lesson that can be learned from it is that plotting news onto a map requires enriching the news feed with geo-coordinates, fetching the actual news items, and plotting them on a map. For developing even such a simple application, at every step the user needs to know which component/s to use, how to set the parameter values for the components, how to define the data mapping among the components etc. Understanding this logic is neither trivial nor intuitive. Motivated by a user study on how end users imagine assistance during mashup development [3], we assist the modeler in each step of his development task by means of ***interactive, contextual recommendations of composition knowledge***. The knowledge is re-usable *composition patterns*, i.e., frequently occurred composition fragments of previous successful mashup models. Having such re-usable composition patterns stored in

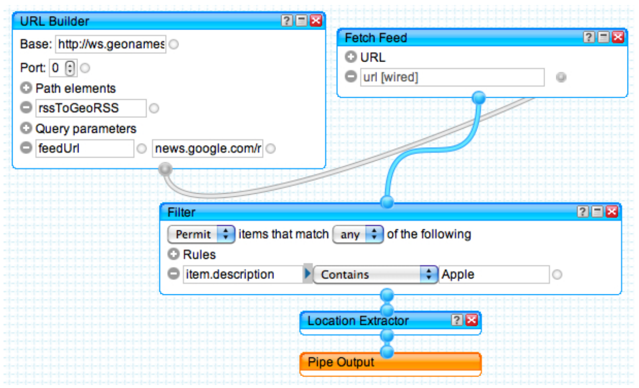


Figure 1. A typical pattern in Yahoo! Pipes

our pattern knowledge base, in this research, we particularly focus on (i) how to efficiently query and retrieve them *interactively* in response to the user's modeling actions and the current composition context inside modeling tool. For instance, once user drags *URL Builder* into the modeling canvas and fills it's parameters with values like in Figure 1, we recommend a pattern which suggests to use *Fetch Feed* as a next component. We further focus on (ii) how to automatically weave a pattern, upon selection, into the current mashup model by performing a set of modeling steps (adding a component, adding a connector between two components etc.) on behalf of the user.

II. Background and Related Work

A. Challenges

To identify the composition patterns for mashup development, in [4] we analyzed data flow based mashup composition models (similar to Yahoo! Pipes) and discovered five types of composition patterns: *parameter value*, *connector*, *data mapping*, *component co-occurence*, and *complex* patterns. However, recommending this patterns proactively and interectively inside the client-side modeling environment requires to design a pattern knowledge base (KB) that supports fast retrieval of the patterns. It also requires to have a set of efficient and interactive pattern querying and retrieval algorithms. In [4], we explained our approach for tackling these challenges. For the weaving, we further require to define a set of algorithms, which automate the contextual

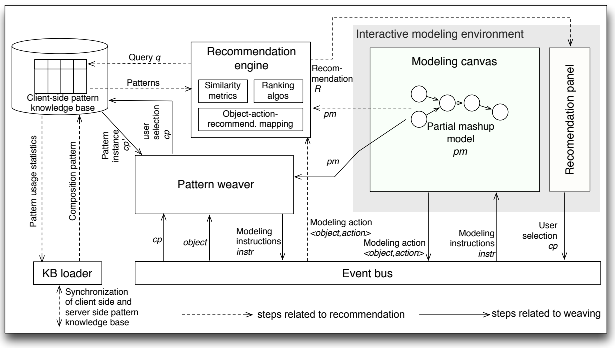

Interactive modeling environment
Recommendation engine
Query q
Patterns
Similarity metrics
Ranking algos
Object-action-recommend. mapping
Recom-mendation R
Modeling canvas
Partial mashup model pm
Recomendation panel
Client-side pattern knowledge base
pm
pm
Pattern instance cp'
user selection cp
Pattern usage statistics
Composition pattern
Pattern weaver
cp
object
Modeling instructions instr
Modeling action <object,action>
Modeling action <object,action>
Modeling instructions instr
User selection cp
KB loader
Event bus
Synchronization of client side and server side pattern knowledge base
steps related to recommendation
steps related to weaving

Figure 2. Functional architecture of our interactive recommendation and weaving approach

weaving steps of a pattern into a model by tackling issues related to model conflicts at runtime.

B. Related Work

Several works aim to assist less skilled developers in the design of mashups. Web macro approaches ([5], [6]) capture and reuse navigation patterns in end user programming. However, the assistance in these approaches are provided only for navigating and configuring data in webpages. In our research, we support compostion pattern reuse, by providing development assistance to end-users, in a more complex web mashup compositions scenario. Among other assisted development approaches, syntactic approaches [7] suggest modeling constructs based on syntactic similarity (comparing output and input data types), while semantic approaches [8] annotate constructs to support suggestions based on the meaning of constructs. In programming by demonstration [9], the system aims to auto-complete a process definition, starting from a set of user-selected model examples. Goal-oriented approaches [10] aim to assist the user by automatically deriving compositions that satisfy user-specified goals. Pattern-based development [11] aims at recommending connector patterns (so-called glue patterns) in response to user selected components (so-called mashlets) in order to autocomplete the partial mashup. The limitation of these approaches is that they partly overestimate the skills of less skilled developers (e.g., end users), as they either still require advanced modeling skills (which users don't have), or they expect the user to specify complex rules for defining goals (which they are not able to), or they expect domain experts to specify and maintain complex semantic networks describing modeling constructs (which they don't do).

III. Approach

In Figure 2, we show the ***architecture*** of our client-side, interactive pattern recommendation and weaving infrastructure. The *interactive modeling environment* runs in the client browser. It is here where the ***pattern recommendation*** logic reacts to modeling *actions* performed by the modeler on a construct (the $object$ of the action) in the canvas. For instance, we can *drop* a component onto the canvas, or we can $select$ a parameter. Upon each interaction, the $action$ and its $object$ are published on a browser-internal *event bus*, which forwards them to the *recommendation engine*. With this information and the partial mashup model pm the engine queries the *client-side KB* for recommendations, where an *object-action-recommendation mapping* tells the engine which types of recommendations are to be retrieved. The list of patterns retrieved from the KB are then ranked and rendered in the *recommendation panel*. Upon the selection of a pattern from the recommendation panel, the ***pattern weaver*** weaves it into the partial mashup model. The *pattern weaver* first retrieves a *basic weaving strategy* (a set of model-agnostic mashup instructions) and then derives a *contextual weaving strategy* (a set of model-specific instructions), which is used to weave the pattern. Deriving the contextual strategy from the basic one may require the resolution of possible *conflicts* among the constructs of the partial model and those of the pattern to be weaved. The pattern weaver resolves them according to a configurable conflict resolution policy.

Our ***prototype*** is a Mozilla Firefox extension for Yahoo! Pipes [12], with the recommendation and weaving algorithms implemented in JavaScript. Event listeners listen for DOM modifications, in order to identify mashup modeling actions inside the modeling canvas, and accordingly recommendation engine queries and retrieves patterns from the KB, which is implemented in SQLite (http://www.sqlite.org/). The instructions in the weaving strategies refers to modeling instructions, which are implemented as JavaScript manipulations of the mashup model's JSON representation.

IV. Evaluation and Discussion

For our experiments we extracted 303 pipes definitions from the repository of Pipes. We were able to identify patterns of all the types described above. We used these patterns to populate our KB and generated additional synthetic patterns to test the performance of the recommendation engine (the sizes of the KBs ranged from 10, 30, 100, 300, 1000 complex patterns) [4]. The complexity of the patterns ranged from $3-9$ components per pattern, and we used queries with varying component set of size $1-7$. In the worst case scenario (KB of 1000 patterns, approximate similarity matching of patterns), the recommendation is provided within 608 millisecond – everything entirely inside the client browser. As a next step, we will work towards evaluating the *precision* and *recall* of our recommendation and weaving algorithms. Subsequently we will expand our algorithms toward other modeling environments (e.g., BPM tools), we will make our prototype tool publicly available online and perform in-depth user studies to fine-tune the usability of our approach.

References

[1] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck, "The state of the art in end-user software engineering," *ACM Comput. Surv.*, pp. 21:1–21:44, 2011.

[2] F. Casati, "How end-user development will save composition technologies from their continuing failures," in *IS-EUD'11*, 2011, pp. 4–6.

[3] A. De Angeli, A. Battocchi, S. Roy Chowdhury, C. Rodríguez, F. Daniel, and F. Casati, "End-User Requirements for Wisdom-Aware EUD," in *IS-EUD'11*, pp. 245–250.

[4] S. Roy Chowdhury, F. Daniel, and F. Casati, "Efficient, Interactive Recommendation of Mashup Composition Knowledge," in *ICSOC'11*, 2011, pp. 374–388.

[5] C. Bogart, M. Burnett, A. Cypher, and C. Scaffidi, "End-user programming in the wild: A field study of coscripter scripts," in *VL/HCC 2008*, 2008, pp. 39 –46.

[6] A. Koesnandar, S. Elbaum, G. Rothermel, L. Hochstein, C. Scaffidi, and K. T. Stolee, "Using assertions to help end-user programmers create dependable web macros," in *SIGSOFT 2008*, ser. SIGSOFT '08/FSE-16, 2008, pp. 124–134.

[7] J. Wong and J. I. Hong, "Making mashups with marmite: towards end-user programming for the web," in *CHI'07*, pp. 1435–1444. [Online]. Available: http://doi.acm.org/10.1145/1240624.1240842

[8] A. Ngu, M. Carlson, Q. Sheng, and H. young Paik, "Semantic-based mashup of composite applications," *IEEE TSC*, vol. 3, no. 1, pp. 2 –15, 2010.

[9] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, Eds., *Watch what I do: programming by demonstration*. Cambridge, MA, USA: MIT Press, 1993.

[10] M. Henneberger, B. Heinrich, F. Lautenbacher, and B. Bauer, "Semantic-Based Planning of Process Models," in *Multikonferenz Wirtschaftsinformatik'08*, 2008.

[11] O. Greenshpan, T. Milo, and N. Polyzotis, "Autocompletion for mashups," *VLDB'09*, vol. 2, pp. 538–549. [Online]. Available: http://dl.acm.org/citation.cfm?id=1687627.1687689

[12] S. Roy Chowdhury, C. Rodríguez, F. Daniel, and F. Casati, "Baya: Assisted Mashup Development as a Service," in *WWW'12*, 2012.

Appendix H

Assisted Mashup Development: On the Discovery and Recommendation of Mashup Composition Knowledge

Carlos Rodrguez, Soudip Roy Chowdhury, Florian Daniel, Hamid R. Motahari Nezhad, Fabio Casati. Assisted Mashup Development: On the Discovery and Recommendation of Mashup Composition Knowledge. To be published in the Handbook on Web Services, Springer, 2012.

Chapter 1

Assisted Mashup Development: On the Discovery and Recommendation of Mashup Composition Knowledge

Carlos Rodríguez, Soudip Roy Chowdhury, Florian Daniel, Hamid R. Motahari Nezhad and Fabio Casati

Abstract Over the past few years, mashup development has been made more accessible with tools such as Yahoo! Pipes that help in making the development task simpler through simplifying technologies. However, mashup development is still a difficult task that requires knowledge about the functionality of web APIs, parameter settings, data mappings, among other development efforts. In this work, we aim at assisting users in the mashup process by recommending development knowledge that comes in the form of *reusable composition knowledge*. This composition knowledge is harvested from a repository of existing mashup models by mining a set of *composition patterns*, which are then used for interactively providing composition recommendations while developing the mashup. When the user accepts a recommendation, it is automatically woven into the partial mashup model by applying modeling actions as if they were performed by the user. In order to demonstrate our approach we have implemented *Baya*, a Firefox plugin for Yahoo! Pipes that shows that it is indeed possible to harvest useful composition patterns from existing mashups, and that we are able to provide complex recommendations that can be automatically woven inside Yahoo! Pipes' web-based mashup editor.

1.1 Introduction

Mashup tools, such as Yahoo! Pipes (`http://pipes.yahoo.com/pipes/`) or JackBe Presto Wires (`http://www.jackbe.com`), generally promise easy development tools and lightweight runtime environments, both typically running inside the client browser. By now, mashup tools undoubtedly simplified some com-

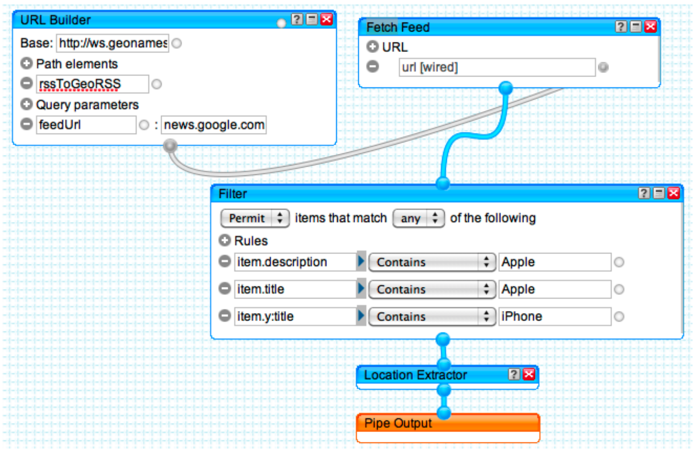
Carlos Rodríguez, Soudip Roy Chowdhury, Florian Daniel and Fabio Casati
University of Trento, Via Sommarive 5, 38123, Povo (TN), Italy, e-mail: `{crodriguez, rchowdhury, daniel, casati}@disi.unitn.it`

Hamid R. Motahari Nezhad
Hewlett Packard Labs, Palo Alto (CA), USA, e-mail: `hamid.motahari@hp.com`

plex composition tasks, such as the integration of web services or user interfaces. Yet, despite these advances in simplifying technology, mashup development is still 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, authoring and sharing 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 functionality, of how to combine them, of how to propagate data, and so on. In short, these are people that do not have software development knowledge. 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 or typical modeling patterns of their tools.

What people (also programmers) typically do when they don't know how to solve a tricky modeling problem is searching for ***help***, e.g., by asking more skilled friends or by querying the Web for solutions to analogous problems. In this latter case, examples of ready mashup models are one of the most effective pieces of information – provided that suitable examples can be found, i.e., examples that have an analogy with the modeling situation faced by the modeler. Yet, 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.


URL Builder
Base: http://ws.geonames
Path elements: rssToGeoRSS
Query parameters: feedUrl : news.google.com

Fetch Feed
URL: url [wired]

Filter
Permit items that match any of the following
Rules
item.description Contains Apple
item.title Contains Apple
item.y:title Contains iPhone

Location Extractor

Pipe Output

Fig. 1.1 A typical pattern in Yahoo! Pipes

For instance, Figure 1.1 illustrates a Yahoo! Pipes model that encodes how to plot news items on a map. Besides showing how to connect components and fill parameters, the key lesson that can be learned from this pipe is that plotting news onto a map requires first enriching the news feed with geo-coordinates, then fetching the actual news items, and only then handing the items over to the map. Understanding this logic is neither trivial nor intuitive.

Driven by a user study on how end users imagine assistance during mashup development [4], we aim to automatically offer them help pro-actively and interactively. Specifically, we are working toward the ***interactive, contextual recommendation of reusable 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 about how to compose mashups, such as the pattern in Figure 1.1. Such knowledge may come from a variety of possible sources. In this work, we specifically focus on community composition knowledge and mine recurrent model fragments from a repository of given mashup models.

The ***vision*** is that of enabling the development of assisted, web-based mashup environments that deliver composition knowledge much like Google's Instant feature delivers search results already while still typing keywords into the search field.

In this chapter, we approach two core ***challenges*** of this vision, i.e., the *discovery* of reusable composition knowledge from a repository of ready mashup models and the *reuse* of such knowledge inside mashup tools, a feature that we call *weaving*. Together with the ability to search and retrieve composition patterns contextually when modeling a new mashup, a problem we approached in [10] and that we summarize in this chapter, these two features represent the key enablers of the vision of assisted development. We specifically provide the following ***contributions***:

- We describe a *canonical mashup model* that is able to represent in a single modeling formalism a variety of data flow mashup languages. The goal is to mine composition knowledge from multiple source languages by implementing the necessary algorithms only once.
- Based on our canonical mashup model, we define a set of *mashup pattern types* that resemble the modeling actions of typical mashup environments.
- We describe an *architecture* of our knowledge recommender that can be used to equip any mashup environment with interactive assistance for its developers.
- We develop a set of *data mining algorithms* that discover composition knowledge in the form of reusable mashup patterns from a repository of mashup models.
- We present our *pattern recommendation* and *pattern weaving* algorithms. The former aims at recommending composition patterns based on the user actions on the design canvas. The later aims at automatically appying patterns to mashup models, allowing the developer to progress in his development task.

In the next section, we start by introducing the canonical mashup model, which will help us to formulate our problem statement, define mashup pattern types and describe our pattern mining algorithms. Section 1.3 is where we describe the types of mashup patterns we are interested in and the architecture of our recommendation platform. In Sections 1.4, 1.5 and 1.6 we, respectively, describe in details the mining, recommendation, and weaving algorithms. Section 1.7 presents the details of the implementation of our approach. In Section 1.8 we overview related work. Then, with Section 1.9, we conclude the chapter.

1.2 Preliminaries and Problem

The development of a data mining algorithm strongly depends on the data to be mined. The data in our case are the mashup models. Since in our work we do not only aim at the reuse of knowledge but also at the reuse of our algorithms across different platforms, we strive for the development of algorithms that are able to accommodate different mashup models in input. Next, we therefore describe a ***canonical mashup model*** that allows us to concisely express multiple data mashup models and to implement mining algorithms that intrinsically support multiple mashup platforms. The canonical model is not meant to be executed; it rather serves as description format.

As a first step toward generic modeling environments, in this chapter we focus on data flow based mashup models. Although relatively simple, they are the basis of a significant number of mashup environments, and the approach can easily be extended toward other mashup environments.

1.2.1 A Canonical Mashup Model

Let CT be a set of ***component types*** of the form $ctype = \langle type, IP, IN, OP, OUT, is\_embedding \rangle$, where $type$ identifies the type of component (e.g., RSS feed, filter, or similar), IP is the set of input ports of the component type (for the specification of data flows), IN is the set of input parameters of the component type, OP is the set of output ports, OUT is the set of output attributes[1], and $is\_embedding \in \{yes, no\}$ tells whether the component type allows the embedding of components or not (e.g., to model a loop). We distinguish three types of components:

- *Source* components fetch data from the web (e.g., from an RSS feed) or the local machine (e.g., from a spreadsheet), or they collect user inputs at runtime. They don't have input ports, i.e., $IP = \emptyset$.
- *Data processing* components consume data in input and produce processed data in output. Therefore: $IP, OP \neq \emptyset$. Filter components, operators, and data transformers are examples of data processing components.
- *Sink* components publish the output of a mashup, e.g., by printing it onto the screen (e.g., a pie chart) or providing an API toward it, such as an RSS or RESTful resource. Sinks don't have outputs, i.e., $OP = \emptyset$.

Given a set of component types, we are able to instantiate components in a modeling canvas and to compose mashups. We express the respective ***canonical mashup model*** as a tuple $m = \langle name, id, src, C, GP, DF, RES \rangle$, where $name$ is the name of the mashup in the canonical representation, id a unique identifier, $src \in \{\text{“}Pipes\text{”}, \text{“}Wires\text{”}, \text{“}myCocktail\text{”}, ...\}$ keeps track of the source platform of

[1] We use the term *attribute* to denote data attributes produced as output by a component or flowing through a data flow connector and the term *parameter* to denote input parameters of a component.

the mashup, C is the set of components, GP is a set of global parameters, DF is a set of data flow connectors propagating data among components, and RES is a set of result parameters of the mashup. Specifically:

- $GP = \{gp_i | gp_i = \langle name_i, value_i \rangle\}$ is a set of ***global parameters*** that can be consumed by components, $name_i$ is the name of a given parameter, $value_i \in (STR \cup NUM \cup \{null\})$ is its value, with STR and NUM representing the sets of possible string or numeric values, respectively. The use of global parameters inside data flow languages is not very common, yet tools like Presto Wires or myCocktail (`http://www.ict-romulus.eu/web/mycocktail`) support the design-time definition of globally reusable variables.
- $DF = \{df_j | df_j = \langle srccid_j, srcop_j, tgtcid_j, tgtip_j \rangle\}$ is a set of ***data flow connectors*** that, each, assign the output port $srcop_j$ of a source component with identifier $srccid_j$ to an input port $tgtip_j$ of a target component identified by $tgtcid_j$, such that $srccid \neq tgtcid$. Source components don't have connectors in input; sink components don't have connectors in output.
- $C = \{c_k | c_k = \langle name_k, id_k, type_k, IP_k, IN_k, DM_k, VA_k, OP_k, OUT_k, E_k \rangle\}$ is the set of ***components***, such that $c_k = instanceOf(ctype)$[2], $ctype \in CT$ and $name_k$ is the name of the component in the mashup (e.g., its label), id_k uniquely identifies the component, $type_k = ctype.type$[3], $IP_k = ctype.IP$, $IN_k = ctype.IN$, $OP_k = ctype.OP$, $OUT_k = ctype.OUT$, and:
  - $DM_k \subseteq IN_k \times (\bigcup_{ip \in IP_k} ip.source.OUT)$ is the set of ***data mappings*** that map attributes of the input data flows of c_k to input parameters of c_k.
  - $VA_k \subseteq IN_k \times (STR \cup NUM \cup GP)$ is the set of ***value assignments*** for the input parameters of c_k; values are either filled manually or taken from global parameters.
  - $E_k = \{cid_{kl}\}$ is the set of identifiers of the ***embedded components***. If the component does not support embedded components, $E_k = \emptyset$.
- $RES \subseteq \bigcup_{c \in C} c.OUT$ is the set of ***mashup outputs*** computed by the mashup.

Without loss of generality, throughout this chapter we exemplify our ideas and solutions in the context of Yahoo! Pipes, which is well known and comes with a large body of readily available mashup models that we can analyze. Pipes is very similar to our canonical mashup model, with two key differences: it does not have global parameters, and the outputs of the mashup are specified by using a dedicated *Pipe Output* component (see Figure 1.1). Hence, $GP, RES = \emptyset$ and a pipe corresponds to a restricted canonical mashup of the form $m = \langle name, id, \text{“}Pipes\text{”}, C, \emptyset, DF, \emptyset \rangle$ with the attributes as specified above. In general, we refer to the generic canonical model; we explicitly state where instead we use the restricted Pipes model.

[2] To keep models and algorithms simple, we opt for a *self-describing* instance model for components, which presents both type and instance properties.

[3] We use a *dot notation* to refer to sub-elements of structured elements; *ctype.type* therefore refers to the *type* attribute of the component type *ctype*.

1.2.2 Problem Statement

Given the above canonical mashup model, the problem we want to address in this chapter is understanding (i) which *kind of knowledge* can be extracted from the canonical mashup model so as to automatically assist users in developing their mashups, (ii) what *algorithms* we need to develop in order to be able to discover such knowledge from existing mashup models, (iii) how to *interactively recommend* discovered patterns inside mashup tools in order to guide users with the next modeling step/s and (iv) how to automatically apply (*weave*) the selected recommendation inside the current mashup design.

1.3 Approach

The current trend in modeling environments in general, and in mashup tools in particular, is toward intuitive, web-based solutions. The key principles of our work are therefore to conceive solutions that *resemble the modeling paradigm* of graphical modeling tools, to develop them so that they can *run inside the client browser*, and to specifically *tune their performance* so that they do not annoy the developer while modeling. These principles affect the nature of the knowledge we are interested in and the architecture and implementation of the respective recommendation infrastructure.

1.3.1 Composition Knowledge Patterns

Starting from the canonical mashup model, we define composition knowledge as reusable ***composition patterns*** for mashups of type m, i.e., model fragments that provide insight into how to solve specific modeling problems, such as the one illustrated in Figure 1.1. In general, we are in the presence of a set of composition pattern types PT, where each pattern type is of the form $ptype = \langle C, GP, DF, RES \rangle$, where C, GP, DF, RES are as defined for m.

The size of a pattern may vary from a single component with a value assignment for one input parameter to an entire, executable mashup. The most ***basic patterns*** are those that represent a co-occurrence of two elements out of C, GP, DF or RES. For instance, two components that recur often together form a basic pattern; given one of the components, we are able to recommend the other component. Similarly, an input parameter plus its value form a basic pattern, given the parameter, we can recommend a possible value for it. As such, the most basic patterns are similar to *association rules*, which, given one piece of information, are able to suggest another piece of information.

Aiming, however, to help a developer refine his mashup model step by step with as less own effort as possible, we are able to identify a set of pattern types that allow the developer to obtain more practical and meaningful composition knowledge. Such knowledge is represented by sensible combinations of basic patterns, i.e., by ***composite patterns***.

Considering the typical modeling steps performed by a developer (e.g., filling input fields, connecting components, copying/pasting model fragments), we specifically identify the following set PT of ***pattern types***:

Parameter value pattern. The parameter value pattern represents a set of recurrent value assignments VA for the input fields IN of a component c:

$ptype^{par} = \langle \{c\}, GP, \emptyset, \emptyset \rangle$;
$c = \langle name, 0, type, \emptyset, IN, \emptyset, \emptyset, VA, \emptyset, \emptyset \rangle$[4];
$GP \neq \emptyset$ if VA also assigns global parameters to IN;
$GP = \emptyset$ if VA assigns only strings or numeric constants.

This pattern helps filling input fields of a component that require explicit user input.

Connector pattern. The connector pattern represents a recurrent connector df_{xy}, given 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:

$ptype^{con} = \langle \{c_x, c_y\}, \emptyset, \{df_{xy}\}, \emptyset \rangle$;
$c_x = \langle name_x, 0, type_x, \emptyset, \emptyset, \emptyset, \emptyset, \{op_x\}, OUT_x, \emptyset \rangle$;
$c_y = \langle name_y, 1, type_y, \{ip_y\}, IN_y, DM_y, \emptyset, \emptyset, \emptyset, \emptyset \rangle$.

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

Connector co-occurrence pattern. The connector co-occurrence pattern captures which connectors df_{xy} and df_{yz} occur together, also including their data mappings:

$ptype^{coo} = \langle \{c_x, c_y, c_z\}, \emptyset, \{df_{xy}, df_{yz}\}, \emptyset \rangle$;
$c_x = \langle name_x, 0, type_x, \emptyset, \emptyset, \emptyset, \emptyset, \{op_x\}, OUT_x, \emptyset \rangle$;
$c_y = \langle name_y, 1, type_y, \{ip_y\}, IN_y, DM_y, \emptyset, \{op_y\},$
$OUT_y, \emptyset \rangle$
$c_z = \langle name_z, 2, type_z, \{ip_z\}, IN_z, DM_z, \emptyset, \emptyset, \emptyset, \emptyset \rangle$.

This pattern helps connecting components. It is particularly valuable in those cases where people, rather than developing their mashup model in an incremental but connected fashion, proceed by first selecting the desired functionalities (the components) and only then by connecting them.

Component co-occurrence pattern. Similarly, the component co-occurrence pattern captures couples of components that occur together. It comes with two components c_x and c_y as well as with their connector, global parameters, parameter values, and c_y's data mapping logic:

[4] The identifier $c.id = 0$ does not represent recurrent information. Identifiers in patterns rather represent internal, system-generated information that is necessary to correctly maintain the structure of patterns. When mining patterns, the actual identifiers are lost; when weaving patterns, they need to be re-generated in the target mashup model.

$ptype^{com} = \langle\{c_x, c_y\}, GP, \{df_{xy}\}, \emptyset\rangle$;
$c_x = \langle name_x, 0, type_x, \emptyset, IN_x, \{op_x\}, OUT_x, VA_x, \emptyset, \emptyset\rangle$;
$c_y = \langle name_y, 1, type_y, \{ip_y\}, IN_y, DM_y, VA_y, \emptyset, \emptyset, \emptyset\rangle$.

This pattern helps developing 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 has three components, in that both the embedded and the embedding component have access to the outputs of the preceding component. How these outputs are jointly used is valuable information. The pattern, hence, contains the three components with their connectors, data mappings, global parameters, and parameter values:

$ptype^{emb} = \langle\{c_x, c_y, c_z\}, GP, \{df_{xy}, df_{xz}, df_{zy}\}, \emptyset\rangle$;
$c_x = \langle name_x, 0, type_x, \emptyset, \emptyset, \{op_x\}, OUT_x, \emptyset, \emptyset, \emptyset\rangle$;
$c_y = \langle name_y, 1, type_y, \{ip_y\}, IN_y, DM_y, VA_y, \emptyset, \emptyset, \emptyset\rangle$;
$c_z = \langle name_z, 2, type_z, \{ip_z\}, IN_z, DM_z, VA_z, \{op_z\},$
$OUT_z, \emptyset\rangle$.

This pattern helps, for instance, modeling cycles, a task that is usually not trivial to non-experts.

Multi-component pattern. The multi-component pattern represents recurrent model fragments that are generically composed of multiple components. It represents more complex patterns, such as the one in Figure 1.1, that are not yet captured by the other pattern types alone. It allows us to obtain a full model fragment, given any of its sub-elements, typically, a set of components or connectors:

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

Besides providing significant modeling support, this pattern helps understanding domain knowledge and best practices as well as keeping agreed-upon modeling conventions.

This list of pattern types is extensible, and what actually matters is the way we specify and process them. However, this set of pattern types, at the same time, leverages on the interactive modeling paradigm of the mashup tools (the patterns represent modeling actions that could also be performed by the developer) and provides as much information as possible (we do not only tell simple associations of constructs, but also show how these are used together in terms of connectors, parameter values, and data mappings).

Given a set of pattern types, an actual pattern can therefore be seen as an ***instance*** of any of these types. We model a composition pattern as $cp = instanceOf(ptype)$, $ptype \in PT$, where $cp = \langle type, src, C, GP, DF, RES, usage, date\rangle$, $type \in \{$"*Par*", "*Con*","*Coo*","*Com*","*Emb*","*Mul*"$\}$, $src \in \{$"*Pipes*","*Wires*","*myCockail*",...$\}$ specifies the target platform of the pattern, C, GP, DF, RES, src are as defined for the pattern's *ptype*, *usage* counts how many times the pattern has been used (e.g., to compute rankings), and *date* is the creation date of the pattern.

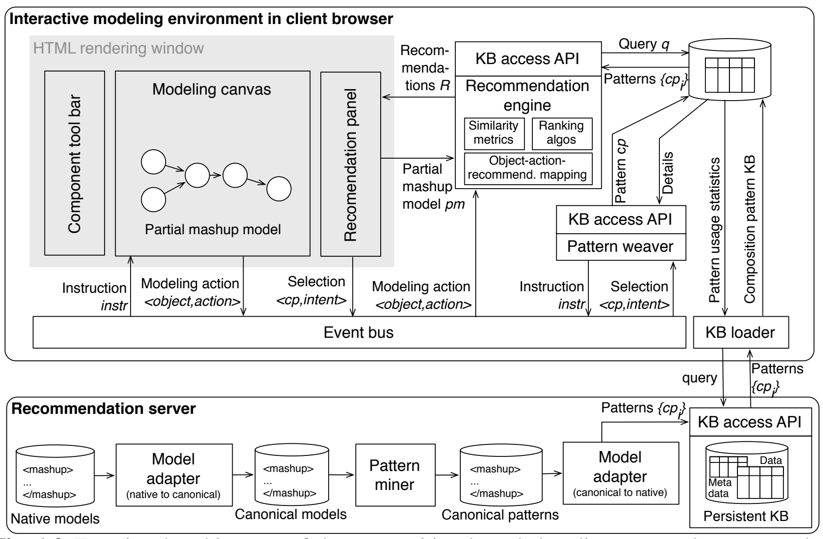


Fig. 1.2 Functional architecture of the composition knowledge discovery and recommendation approach

1.3.2 Architecture

Figure 1.2 details the internals of our knowledge discovery and recommendation prototype. We distinguish between client and server side, where the discovery logic is located in the server and the recommendation and weaving logic resides in the client. In the *recommendation server*, a *model adapter* imports the native mashup models into the canonical format. The *pattern miner* then extracts reusable composition knowledge in the form of composition patterns, which is then handed to a second *model adapter* to convert the canonical patterns into native patterns and load them into a knowledge base (KB). This KB is structured to maximize the performance of pattern retrieval at runtime.

In the client, we have the *interactive modeling environment*, in which the developer can visually compose components (in the *modeling canvas*) taken from the *component tool bar*. It is here where patterns are queried for and delivered in response to modeling actions performed by the modeler in the modeling canvas. In visual modeling environments, we typically have $action \in \{$"*select*","*drag*","*drop*", "*connect*", "*delete*","*fill*","*map*",...$\}$, where the *action* is performed on a modeling construct in the canvas; we call this construct the *object* of the action. For instance, we can *drop* a component onto the canvas, or we can *select* a parameter to fill it with a value, we can *connect* a data flow with a target component, or we can *select* a set of components and connectors. Upon each interaction, the *action* and its *object* are published on a browser-internal *event bus*, which forwards them 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). An *object-*

action-recommendation mapping (*OAR*) tells the engine which types of recommendations are to be retrieved for each modeling action on a given object (for example, when selecting an input field, only recommending possible values makes sense). The client-side KB is filled at startup by the *KB loader*, which loads the available patterns into the client environment, decoupling the knowledge recommender from the server side.

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. Selecting a recommendation enacts the *pattern weaver*, which queries the KB for the usage details of the pattern (data mappings and value assignments) and generates a set of modeling instructions that emulate user interactions inside the modeling canvas and thereby weave the pattern into the partial mashup model.

1.4 Discovering Patterns

The first step in the information flow described in the above architecture is the discovery of mashup patterns from canonical mashup models. To this end, we look into the details of each individual pattern and implement dedicated mining algorithms for each of them, which allow us to fine-tune each mashup-specific characteristic (e.g., to treat threshold values for parameter value assignments and data mappings differently). The pattern mining algorithms make use of standard statistics as well as frequent itemset and subgraph mining algorithms [13].

1.4.1 Mining algorithms

For each of the pattern types identified in Section 1.3.1, we have implemented a respective pattern mining algorithm, the details of which we provide in the following.

Parameter value pattern. In the case of the parameter value pattern, we are interested in finding suitable values for the input fields in a given component. Most of the components in mashup compositions contain more than one parameter and more often than not the values of these parameters are related to one another and therefore we need take into account the co-occurrence of parameter values. In order to discover such co-occurrences, we map this problem to the well-known problem of itemset mining [13]. Algorithm 1 outlines the approach for finding parameter value patterns. Here, we first get all component instances from the mashups in the mashup repository (line 2) and group them together by their type (line 5-6) and then perform the parameter value pattern mining by component type (line 7). Finally, we construct the actual set of patterns that consists in tuples $\langle ct, VA\rangle$, where *ct* represents a component type and *VA* represents the value assignment for its parameters.

Algorithm 1: mineParameterValues

```
Data: repository of mashup compositions $M$ and minimun support ($minsupp_{par}$) for the frequent itemset mining
Result: set of parameter value patterns $\langle ct, VA \rangle$.
1  $Patterns$ = set();
2  $C$ = set of component instances in $M$;
3  $CT$ = array();
4  $Patterns$ = set();
5  foreach type of component ct in C do
6      $CT[ct]$ = $c_x.VA$ with $c_x \in C$ such that $c_x.type = ct$ ;   // get all the parameter value
       assignments of component instances of type ct
7      $FI$ = mineFrequentItemsets($CT[ct], minsupp_{par}$);
8      foreach $VA \in FI$ do
9          $Patterns = Patterns \cup \{\langle ct, VA \rangle\}$;
10 return $Patterns$;
```

Connector pattern. A connector pattern is composed of two components, the source component c_x and the target component c_y, their data flow connector df_{xy}, and the data mapping DM_y of the target component. Given a repository of mashup models $M = \{m_i\}$ and the minimum support levels for the data flow connectors and data mappings, the pseudo-code in Algorithm 2 shows how we mine connector patterns.

We start the mining task by getting the list of all recurrent connectors in M (line 1). The respective function *getRecurrentConnectors* is explained in Algorithm 3; in essence, it computes a recurrence distribution for all connectors and returns only those that exceed the threshold $minsupp_{df}$. The function returns a set of connector types without repetitions and without information about the instances that generated them. Given this set, we construct a database of concrete instances of each connector type (using the *getConnectorInstances* function in line 5 and described in Algorithm 4) and, for each connector type, derive a database of the data mappings for the connectors' target component c_y (lines 7-9). We feed the so constructed database into a standard *mineFrequentItemsets* function [13], in order to obtain a set of recurrent data mappings for each connector type. Finally, for each identified data mapping DM_y, we construct a tuple $\langle df_{xy}, DM_y \rangle$ (lines 11-12), which concisely represents the connector pattern structure introduced in Section 1.3.1; the rest of the pattern comes from the component definitions.

Connector co-occurrence pattern. The *connector pattern* introduced previously is about how pairs of components are connected together. *The connector co-occurrence pattern* goes a step further: it tells how connectors between different pairs of components co-occur together in compositions and how data mappings are defined for them. Algorithm 5 presents the logic for computing *connector co-occurrence patterns*. The main difference with respect to Algorithm 2 is that, instead of computing the frequency of individual dataflow connectors between pairs of components, we compute frequent itemsets of dataflow connectors (lines 2-4).

Component co-occurrence pattern. The component co-occurrence pattern is an extension of the connector pattern; in addition to the connectors and data mappings, it also contains the parameter value assignments of the two components involved in the connector. As shown in Algorithm 6, the respective mining logic is similar to

Algorithm 2: mineConnectors

```
Data: repository of mashup models $M$, minimum support of data flow connectors ($minsupp_{df}$) and data
      mappings ($minsupp_{dm}$)
Result: set of connectors with their corresponding data mappings $\{\langle df_{xy,i}, DM_{y,i}\rangle\}$
1  $F_{df}$ = getRecurrentConnectors($M, minsupp_{df}$);
2  $DB$ = array();                          // database of recurrent connector instances
3  $Patterns$ = set();                                  // set of connector patterns
4  foreach $df_{xy} \in F_{df}$ do
5      $DB[df_{xy}]$ = getConnectorInstances($M, df_{xy}$);
       // create database for frequent itemset mining
6      $DBDM_y$ = array():
7      foreach $dfi_{xy} \in DB[df_{xy}]$ do
8          $c_y$ = target component of $dfi_{xy}$;
9          append($DBDM_y$, $c_y.DM$);
10     $FI_{dy}$ = mineFrequentItemsets($DBDM_y$, $minsupp_{dm}$);
       // construct the connector patterns
11     foreach $DM_y \in FI_{dy}$ do
12         $Patterns = Patterns \cup \{\langle df_{xy}, DM_y\rangle\}$;
13 return $Patterns$;
```

Algorithm 3: getRecurrentConnectors

```
Data: repository of mashup models $M$, minimum support of data flow connectors ($minsupp_{df}$)
Result: set of recurrent connectors $F_{df}$
1  $DB_{df}$ = array();                     // database of data flow connector instances
2  foreach $m_i \in M$ do
3      append($DB_{df}, m_i.DF$) ;                              // fill with instances
4  $F_{df}$ = set();                          // set of recurrent data flow connectors
5  foreach $df_{xy} \in DB_{df}$ do
6      if $computeSupport(df_{xy}, DB_{df}) \geq minsupp_{df}$ then
7          $F_{df} = F_{df} \cup \{df_{xy}\}$;
8  return $F_{df}$;
```

Algorithm 4: getConnectorInstances

```
Data: repository of mashup models $M$, reference connector $df_{xy}$
Result: array of connector instances $DB_{xy}$
1  $DB_{xy}$ = array();                     // database of data flow connector instances
2  foreach $m_i \in M$ do
3      append($DB_{xy}], m_i.DF \cap \{df_{xy}\}$) ;          // fill with instances of the reference
       connector type
4  return $DB_{xy}$;
```

the one of the connector pattern, with two major differences: in lines 6-17 we also mine the recurrent parameter value assignments of c_x and c_y, and in lines 18-21 we consider only those combinations of VA_x, VA_y and DM_y that co-occur in mashup instances for the given connector. Notice that, for the purpose of explaining this algorithm, we perform a cartesian product of VA_x,, VA_y and DM_y in line 22. Doing this can be computational expensive if implemented as-is. In practice, the implementation of this algorithm is performed in such a way that we do not have to explore

Algorithm 5: mineConnectorCooccurrences

```
Data: repository of mashup compositions $M$, minimun support for dataflow connectors ($minsupp_{df}$) and data
        mappings ($minsupp_{dm}$)
Result: list of connector patterns with their corresponding data mappings $\langle DF_{xy}, DM_y \rangle$
// find the co-occurrence of dataflow connectors
$DB_{df}$ = array();
foreach $m_i \in M$ do
    append($DB_{df}$, $m_i.DF$);
$F_{df}$ = mineFrequentItemsets($DB_{df}, minsupp_{df}$);
$DB_{ci}$ = array();
foreach $m_i \in M$ do
    foreach $DF_{xy} \in F_{df}$ do
        if $DF_{xy} \cap m_i.DF = DF_{xy}$ then
            foreach $dfi_{xy} \in DF_{xy}$ do
                append($DB_{ci}[DF_{xy}]$, getConnectorInstances($\{m_i\}, dfi_{xy}$);
// find data mappings for the frequent dataflow connectors obtained above
$DBDM_y$ = array();
foreach $DF_{xy} \in DB_{ci}$ do
    foreach $dfi_{xy} \in DF_{xy}$ do
        $c_y$ = target component of $dfi_{xy}$;
        append($DBDM_y$, $c_y.DM$);
$FI_{dy}$ = mineFrequentItemsets($DBDM_y$, $minsupp_{dm}$);
// construct the connector patterns
$Patterns$ = set();
foreach $DM_y \in FI_{dy}$ do
    $Patterns = Patterns \cup \{\langle DF_{xy}, DM_y \rangle\}$;
return $Patterns$;
```

the whole search space. This comment also applies to the rest of the algorithms presented in this section.

Component embedding pattern. Mashup composition tools typically allow for the embedding of components inside other components. However, not all components present this capability. A common example is the *loop* component: it takes as input a set of data items and then loops over them executing the operations provided by the embedded component (e.g., a *filter* component). Embedding one component into another is not a trivial task, as there may be complex dataflow connectors and data mappings between the outer and inner component as well as between the last two and the component that proceeds the outer component in the composition flow. Algorithm 7 shows the logic for mining component embedding patterns. First, we get the instances of component embeddings from the mashup repository and then we keep only those that have a support greater or equal to $minsupp_{em}$ (lines 2-10). Using these *frequent embeddings*, we look for *frequent dataflows* that involve these embeddings (lines 11 to 17). For these patterns, we are also interested in finding data mapping and parameter value patterns and thus we proceed as in the previous algorithms to mine them (lines 18-31). In the last part of the algorithm (lines 32-37), we proceed with building the actual patterns with tuples $\langle \{c_x, c_y, c_z\}, DF, DM, VA \rangle$ that include information about the components involved in the pattern as well as the dataflow connectors, data mappings and parameter value assignments.

Algorithm 6: mineComponentCooccurrences

```
Data: repository of mashup models $M$, minimum support of data flow connectors ($minsupp_{df}$), data mappings
      ($minsupp_{dm}$), parameter value assignments ($minsupp_{va}$) and pattern co-occurrence ($minsupp_{pc}$).
Result: set of component co-occurrence patterns with their corresponding dataflow connectors, data mappings
        and parameter values $\{\langle df_{xy,i}, VA_{x,i}, VA_{y,i}, DM_{y,i}\rangle\}$
$F_{df}$ = getRecurrentConnectors($M, minsupp_{df}$);
$DB$ = array();                              // database of recurrent connector instances
$Patterns$ = set();                            // set of component co-occurrence patterns
foreach $df_{xy} \in F_{df}$ do
    $DB[df_{xy}]$ = getConnectorInstances($M, df_{xy}$);
    // create databases for frequent itemset mining
    $DBVA_x$ = array();
    $DBVA_y$ = array();
    $DBDM_y$ = array();
    foreach $dfi_{xy}$ in $DB[df_{xy}]$ do
        $c_x$ = source component of $dfi_{xy}$;
        $c_y$ = target component of $dfi_{xy}$;
        append($DBVA_x$, $c_x.VA$);
        append($DBVA_y$, $c_y.VA$);
        append($DBDM_y$, $c_y.DM$);
    $FI_{vx}$ = mineFrequentItemsets($DBVA_x$, $minsupp_{par}$);
    $FI_{vy}$ = mineFrequentItemsets($DBVA_y$, $minsupp_{par}$);
    $FI_{dy}$ = mineFrequentItemsets($DBDM_y$, $minsupp_{dm}$);
    // keep only those combinations of value assignments and data mappings
       that occur together in mashup instances
    $Coo$ = set();
    foreach $\langle VA_x, VA_y, DM_y\rangle \in FI_{vx} \times FI_{vy} \times FI_{dy}$ do
        if $computeSupport(\langle VA_x, VA_y, DM_y\rangle, DB[df_{xy}]) \geq minsupp_{pc}$ then
            $Coo = Coo \cup \{\langle VA_x, VA_y, DM_y\rangle\}$;
    // construct the component co-occurrence patterns
    foreach $\langle VA_x, VA_y, DM_y\rangle \in Coo$ do
        $Patterns = Patterns \cup \{\langle df_{xy}, VA_x, VA_y, DM_y\rangle\}$;
return $Patterns$;
```

Multi-component pattern. The multi-component pattern represents recurrent model fragments that are composed of multiple components. It represents more complex patterns, which are not yet captured by the other pattern types alone. This pattern helps understanding domain knowledge and best practices as well as keeping modeling conventions. Multi-component patterns consists in a combination of the patterns we have introduced before. Algorithm 8 provides the details of the mining algorithm. We start by obtaining the graph representation of the mashups in the repository and mining frequent sub-graphs out of them (lines 2-5). For the sub-graph mining we can choose among the state of the art sub-graph mining algorithms [13]. Then, we get from the mashup repository the list of mashup fragments that match the frequent sub-graphs mined in the previous step (lines 6-11). We do this, so that next we can mine both the parameter value and data mapping patterns using again standard itemset mining algorithms (lines 13-21). Finally, we build the actual multi-component patterns by going through the mashup repository and keeping only those combinations of patterns that co-occur in the mashup instances (lines 22-25).

Algorithm 7: mineComponentEmbeddings

```
Data: repository of mashup compositions $M$, minimum supports for component embeddings ($minsupp_{em}$), data
      flows ($minsupp_{df}$), data mappings ($minsupp_{dm}$), parameter value ($minsupp_{par}$) and pattern co-occurrence
      ($minsupp_{pc}$)
Result: list of component embedding patterns with their corresponding components, dataflow connectors, data
        mappings and parameter value assignments $\langle\{c_x, c_y, c_z\}, DF, DM, VA\rangle$
// get the list of component embeddings
1  $DB_{em}$ = array();
2  foreach $m_i \in M$ do
3      foreach $\langle c_x, c_y, c_z\rangle \in m_i.C \times m_i.C$ do
4          if ($c_x$ preceeds $c_y$) and ($c_y$ embeds $c_z$) then
5              $em_{xyz} = \langle c_x, c_y, c_z\rangle$;
6              append($DB_{em}, em_{xyz}$);
// find the frequent component embeddings
7  $F_{em}$ = set();
8  foreach $em_{xyz} \in DB_{em}$ do
9      if computeSupport($em_{xyz}, DB_{em}$) $\geq minsupp_{em}$ then
10         append($F_{em}, em_{xyz}$);
// get dataflows involving the frequent component embeddings
11 $DB_{df}$ = array();
12 $F_{df}$ = array();
13 foreach $m_i \in M$ do
14     foreach $em_{xyz} \in F_{em}$ do
15         if $em_{xyz} \in m_i$ then
16             append($DB_{df}[em_{xyz}], \langle m_i.df_{xy}, m_i.df_{xz}, m_i.df_{yz}\rangle$);
17     $F_{df}$ = mineFrequentItemsets($DB_{df}, minsupp_{df}$);
// get parameter value and data mapping instances and compute the
   corresponding frequent itemsets
18 $DB_{va}$ = array(); $DB_{dm}$ = array();
19 foreach $m_i \in M$ do
20     foreach $\langle df_{xy}, df_{xz}, df_{yz}\rangle \in F_{df}$ do
21         if $\langle df_{xy}, df_{xz}, df_{yz}\rangle \in m_i$ then
22             $c_x$ = component instance $c_x \in m_i$ corresponding to $df_{xy}$;
23             $c_y$ = component instance $c_y \in m_i$ corresponding to $df_{xy}$;
24             $c_z$ = component instance $c_z \in m_i$ corresponding to $df_{yz}$;
25             $VA_x = c_x.VA$; $DM_x = c_x.DM$;
26             $VA_y = c_y.VA$; $DM_y = c_y.DM$;
27             $VA_z = c_z.VA$; $DM_z = c_z.DM$;
28             append($DB_{va}, VA_x \cup VA_y \cup VA_z$);
29             append($DB_{dm}, DM_x \cup DM_y \cup DM_z$);
30 $F_{va}$ = mineFrequentItemsets($DB_{va}, minsupp_{par}$);
31 $F_{dm}$ = mineFrequentItemsets($DB_{dm}, minsupp_{dm}$);
// construct the component embedding pattern
32 $Patterns$ = set();
33 foreach $\langle EM, DF, DM, VA\rangle \in F_{em} \times F_{df} \times F_{dm} \times F_{va}$ do
34     if $computeSupport(\langle EM, DF, DM, VA\rangle, M) \geq minsupp_{pc}$ then
35         $c_x, c_y, c_z$ = components corresponding to the dataflows $df \in DF$;
36         $Patterns = Patterns \cup \{\langle\{c_x, c_y, c_z\}, DF, DM, VA\rangle\}$;
37 return $Patterns$;
```

1.5 Recommending Patterns

Recommending patterns is non-trivial, in that the size of the knowledge base may be large, and the search for composition patterns may be complex; yet, recommen-

Algorithm 8: mineMulticomponentPatterns

```
Data: repository of mashup compositions $M$ and minimun support for multi-components ($minsupp_{mc}$),
      parameter value ($minsupp_{par}$) and data mapping ($minsupp_{dm}$) patterns.
Result: set of multi-component patterns $\langle mf.C, mf.DF, VA, DM \rangle$.
1  $DB_g$ = array() ;                    // database of graph representations of mashups
2  foreach $m_i \in M$ do
       // get a graph representation of mashup $m_i$ where the nodes represent
          components and arcs represent dataflows; here, the arcs are labeled
          with the output and input ports involved in the dataflow
3      $g_i$ = getGraphRepresentation($m_i$);
4      append($DB_g, g_i$);
5  $FG$ = mineFrequentSubraphs($DB_g, minsupp_{mc}$);
6  $DB_{mc}$ = array();
7  foreach $m_i \in M$ do
8      foreach $fg_i \in FG$ do
9          if getGraphRepresentation($m_i$) contains $fg_i$ then
               // get the fragment $mf$ from mashup instance $m_i$ that matches $fg_i$;
                  notice that $mf$ is represented as a canonical mashup model
10             $mf$ = getSubgraphInstance($m_i, fg_i$);
11             append($DB_{mc}[fg_i], mf$)
12 $Patterns$ = set();
13 foreach $MC \in DB_{mc}$ do
       // get parameter values and data mappings and compute the corresponding
          frequent itemsets
14     $DBVA$ = array();
15     $DBDM$ = array();
16     foreach $mf \in MC$ do
17         foreach $c_x \in mf.C$ do
18             append($DBVA, c_x.VA$);
19             append($DBDM, c_x.DM$);
20     $FI_{va}$ = mineFrequentItemsets($DBVA, minsupp_{par}$);
21     $FI_{dm}$ = mineFrequentItemsets($DBDM, minsupp_{dm}$);
       // construct the multi-component pattern
22     foreach $\langle VA, DM \rangle \in FI_{va} \times FI_{dm}$ do
23         foreach $mf \in MC$ do
24             if $\langle VA, DM \rangle \in mf$ then
25                 $Patterns = Patterns \cup \{\langle mf.C, mf.DF, VA, DM \rangle\}$ ;    // using $mf$, build the
                   patterns with its components ($mf.C$), dataflows ($mf.DF$),
                   value assignments ($mf.VA$) and data mappings ($mf.DM$)
26 return $Patterns$;
```

dations are to be delivered at high speed, without slowing down the modeler's composition pace. Recommending patterns is platform-specific. The following explanations therefore refer to the specific case of Pipes-like mashup models. In [10], we show all the details of our approach; in the following we summarize its key aspects.

1.5.1 Pattern Knowledge Base

The core of the interactive recommender is the pattern KB. In order to enable the incremental and fast recommendation of patterns, we *decompose* them into their constituent parts and focus only on those aspects that are necessary to convey the meaning of a pattern. That is, we leverage on the observation that, in order to convey the structure of a pattern, already its components and connectors enable the developer to choose in an informed fashion. Data mappings and value assignments, unless explicitly requested by the developer, are then delivered only during the weaving phase upon the selection of a specific pattern by the developer.

This strategy leads us to the ***KB*** illustrated in Figure 1.3, whose structure enables the retrieval of each of the patterns introduced in Section 1.3.1 with a one-shot query over a single table. For instance, let's focus on the component co-occurrence pattern: to retrieve its representation, it is enough to query the *ComponentCooccur* entity for the *SourceComponent* and the *TargetComponent* attributes. The query is assembled automatically upon interactions in the modeling canvas and is of the form $q = \langle object, action, pm \rangle$. Only weaving the pattern into the mashup model requires querying $ComponentCooccur \bowtie Connectors \bowtie DataMapping$ and $ComponentCooccur \bowtie ParameterValues$.

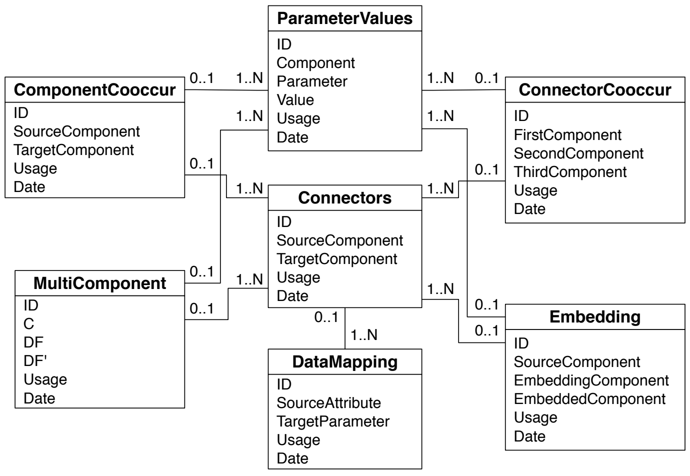


Fig. 1.3 KB structure optimized for Pipes

1.5.2 Exact and Approximate Pattern Matching

The described KB supports both *exact queries* for the patterns with pre-defined structure and *approximate matching* for multi-component patterns whose structure is not known a priori. Patterns are queried for or matched against the $object$ of the query, i.e., the last modeling construct manipulated by the developer. Conceptually, all recommendations could be retrieved via similarity search, but for performance reasons we apply it only when strictly necessary.

Algorithm 9 details this ***strategy*** and summarizes the logic implemented by the recommendation engine. In line 3, we retrieve the types of recommendations that

Algorithm 9: 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 $R = [\langle cp_i, rank_i \rangle]$
1  $R$ = array();
2  $Patterns$ = set();
3  $recTypeToBeGiven$ = getRecTypes($object, action, OAR$);
4  foreach $recType \in recTypeToBeGiven$ do
5      if $recType \neq$ "$Mul$" then
6          $Patterns = Patterns \cup$ queryPatterns($object, KB, recType$) ;          // exact query
7      else
8          $Patterns = Patterns \cup$ getSimilarPatterns($object$,
           $KB, CompSim, T_{sim}$) ;                                               // similarity search
9  foreach $pat \in Patterns$ do
10     if $rank(pat.cp, pat.sim, pm) \geq T_{rank}$ then
11         append($R, \langle pat.cp, rank(pat.cp, pat.sim, pm) \rangle$) ;   // rank, threshold, remember
12 orderByRank($R$);
13 groupByType($R$);
14 truncateByGroup($R, n$);
15 return $R$;
```

can be given (*getSuitableRecTypes* function), given an *object-action* combination. Then, for each recommendation type, we either query for patterns (the *queryPatterns* function can be seen like a traditional SQL query) or we do a similarity search (*getSimilarPatterns* function). 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***, we give preference to exact matches of components and connectors in *object* and allow candidate patterns to differ for the insertion, deletion, or substitution of at most one component in a given path in *object*. Among the non-matching components, we give preference to functionally similar components (e.g., it may be reasonable to allow a Yahoo! Map instead of a Google Map); we track this similarity in a dedicated *CompSim* matrix. For the detailed explanation of the approximate matching logic we refer the reader to [10].

1.6 Weaving Patterns

Weaving a given composition pattern *cp* into a partial mashup model *pm* is not straightforward and requires a thorough analysis of both *cp* and *pm*, in order to understand how to connect the pattern to the constructs already present in *pm*. In essence, weaving a pattern means emulating developer interactions inside the modeling canvas, so as to connect a pattern to the partial mashup. 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 may be required.

We approach the problem of pattern weaving by first defining a *basic weaving strategy* that is independent of *pm* and then deriving a *contextual weaving strategy* that instead takes into account the structure of *pm*.

1.6.1 Basic Weaving Strategy

Given an $object$ and a pattern cp of a recommendation, the ***basic weaving strategy BS*** provides the sequence of mashup operations that are necessary to weave cp into the $object$. The basic weaving strategy does not use *pm*; it tells how to expand $object$ into cp ($object$ being a part of cp). This basic strategy is *static* for each pattern type and it consists a set of ***mashup operations*** that resemble the operations a developer can typically perform manually in the modeling canvas. Typical examples of mashup operations are *addComponent* that corresponds to adding a new component to *pm*, *addConnector* that corresponds to adding a connector between two selected components in *pm*, *assignValues* that corresponds to assigning values to configuration parameters of a component, and similar. Mashup operations are applied on the partial mashup *pm* and result in an updated pm'. All operations assume that the *pm* is globally accessible. The internal logic of these operations are highly platform-specific, in that they need to operate inside the target modeling environment.

For instance, the basic weaving strategy for a component co-occurrence pattern of type $ptype^{comp}$ is as follows (we assume $object = comp$ with $comp.type = c_x.type$, c_x being one of the components of the pattern):

1 \$*newcid*[5]=addComponent($c_y.type$);
2 addConnector($\langle comp.id, c_x.op, \$newcid, c_y.ip \rangle$);
3 assignDataMapping($\$newcid, c_y.DM$);
4 assignValues($comp.id, c_x.VA$);
5 assignValues($\$newcid, c_y.VA$);

That is, given a component c_x, we add the other component c_y (line 1) as mentioned in the selected pattern to the *pm*, connect c_x and c_y together (line 2) and then apply the respective data mappings (line 3) and value assignments (line 4 and line 5). Note that, the basic strategy is not yet applied to *pm*; it represents an array of basic modeling operations to be further processed before being able to weave the pattern.

[5] We highlight identifier place holders (variables) that can only be resolved when executing the operation with a "\$" prefix.

Algorithm 10: getWeavingStrategy

```
Data: partial mashup model $pm$, composition pattern $cp$, object $object$ that triggered the recommendation
Result: weaving strategy $WS$, i.e., a sequence of abstract mashup operations; updated mashup model $pm'$
1 $WS$ = array();
2 $BS$ = getBasicStrategy($cp, object$);
3 foreach $instr \in BS$ do
4     $CtxInstr$ = resolveConflict($pm, instr$);
5     $pm$ = apply($pm, CtxInstr$);
6     append($WS, CtxInstr$);
7 return $\langle WS, pm \rangle$;
```

1.6.2 Contextual Weaving Strategy

Given an object $object$, a pattern cp, and a partial mashup pm, the ***contextual weaving strategy*** WS is derived by applying the mashup operations in the *basic weaving strategy* to the current partial mashup model and thus by weaving the selected cp into pm. The WS is *dynamically* built at runtime by taking into consideration the structure of the partial mashup (the context).

Applying the mashup operations in the basic weaving 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 an instance of type $ctype$, say $comp$, we are in the presence of a conflict: either we decide that we reuse $comp$, which is already there, or we decide to create a new instance of $ctype$. In the former case, we say we apply a *soft* conflict resolution policy, in the latter case a *hard* policy:

Soft: substitute("\$*var*=addComponent(*ctype*)") with "\$$var = comp.id$"
Hard: substitute("\$*var*=addComponent(*ctype*)") with "\$*var*=addComponent(*ctype*)"

Formally, the conflict resolution policy corresponds to a function **resolveConflict**($pm, instr$) $\rightarrow CtxInstr$, where $instr$ is the mashup operation to be applied to pm and $CtxInstr$ is the set of instructions that replace $instr$. Only in the case of a conflict, $instr$ is replaced; otherwise the function returns $instr$ again.

In Algorithm 10 we describe the logic of our pattern weaver. First, it derives a basic strategy BS for the given composition pattern cp and the $object$ from pm (line 2). Then, for each of the mashup operations $instr$ in the basic strategy, 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 $instr$. $CtxInstr$ is then applied to the current pm to compute the updated mashup model pm' (line 5), which is then used as basis for weaving the next $instr$ of BS. The contextual weaving structure WS is constructed as concatenation of all conflict-free instructions $CtxInstr$.

Note that Algorithm 10 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.

1.7 Implementation and Evaluation

We have implemented our prototype system, *Baya* [11], as Mozilla Firefox (`http://mozilla.com/firefox`) extension for Yahoo! Pipes to demonstrate the viability of our interactive recommendation approach. The ***design goals*** behind Baya can be summarized as follows: We didn't want to develop *yet another* mashup environment; so we opted for an extension of existing and working solutions (so far, we focused on Yahoo! Pipes; other tools will follow). Modelers should not be required to *ask* for help; we therefore pro-actively and interactively recommend contextual composition patterns. We did not want the *reuse* to be limited to simple copy/paste of patterns, but knowledge should be *actionable*, and therefore, Baya automatically weaves patterns.

In Baya we have implemented the *model adapters* (see Figure 1.2) in Java (1.6), which are able to convert Yahoo! Pipes's JSON representation into our canonical mashup model and back. All the mining algorithms are also implemented in Java. For the frequent itemset mining we used the tool Carpenter (`http://www.borgelt.net/carpenter.html`), while for graph mining we used the tool MoSS (`http://www.borgelt.net/moss.html`). The resulting patterns are expressed in terms of canonical mashup models, which are then converted to native models (in this case, Yahoo! Pipes JSON representations) by our canonical-to-native model adapter and loaded into the pattern KB.

For testing our mining algorithms, we used a dataset of 970 pipes definitions from Yahoo! Pipes that were retrieved using YQL Console (`http://developer.yahoo.com/yql/console/`). 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 are 11.1, 11.0 and 4.1, respectively, which is an indication that we are dealing with fairly complex pipes.

The results obtained from running our algorithms on the selected dataset show that we are able to discover recurrent practices for building mashups. Table 1.1 reports on the list of pattern types and their Upper Threshold for *minsupp* (UTm). The UTm tells us what is the upper threshold for the *minsupp* values at which we start finding patterns of a given type and for a given dataset. In the cases where we use more than one type of *minsupp* (such as in the component co-occurrence pattern where we use $minsupp_{df}$, $minsupp_{dm}$ and $minsupp_{par}$), the *minsupp* we consider is the one corresponding to the pattern that is first computed in the algorithm. For our dataset, in Table 1.1 we can see that we are always able to find parameter value patterns for some component types. For example, this is the case of Yahoo! Pipes' component *YQL* that has the parameter *raw* with a default value *Results only* that is always kept as-is by the users. From the table we can also notice that the connector and component co-occurrence patterns have the same UTm value. This is because in both cases their corresponding algorithms compute first the frequent dataflow connectors and thus the reference minimum support for the UTm is $minsupp_{df}$. Finally, for the Multi-component pattern we have a UTm of 0.021, a relatively low value, when we consider patterns with at least 4 components. However, considering that here we are talking about complex patterns with at least 4 components that, furthermore, include dataflow connectors, data mappings and parameter value assignments, we can say that, even with a relatively low support value, these patterns still captures recurrent modeling practices for fairly complex settings.

Pattern type	**UTm**
Parameter value pattern	1
Connector pattern	0.257
Connector co-occurrence pattern	0.072
Component co-occurrence pattern	0.257
Component embedding pattern	0.124
Multi-component pattern	0.021

Table 1.1 Summary of pattern types with their corresponding UTm.

The discovered patterns are transformed and stored in a knowledge base that is optimized for fast pattern retrieval at runtime. The implementation of the persistent pattern KB at server side, is based on MySQL (`http://www.mysql.com/`). Via a dedicated Java RESTful API, at startup of the recommendation panel the KB loader synchronizes the server-side KB with the client-side KB, which instead is based on SQLite (`http://www.sqlite.org`). The pattern matching and retrieval algorithms are implemented in JavaScript and triggered by events generated by the event listeners monitoring the DOM changes related to the mashup model.

The weaving algorithms are also implemented in JavaScript. Upon the selection of a recommendation from the panel, they derive the contextual weaving strategy that is necessary to weave the respective pattern into the partial mashup model. Each of the instructions in the weaving strategy refers to a modeling action, where modeling actions are implemented as JavaScript manipulations of the mashup model's JSON represenation. Both the weaving strategies (basic and contextual) are encoded as JSON arrays, which enables us to use the native `eval()` command for fast and easy parsing of the weaving logic.

Figure 1.4 illustrates the performance of the interactive recommendation algorithm of Baya as described in Algorithm 9 in response to the user placing a new component into the canvas, a typical modeling situation. Based on the object-action-recommendation mapping, the algorithm retrieves parameter value, connector, component co-occurrence, and multi-component patterns. As expected, the response times of the simple queries can be neglected compared to the one of the similarity search for multi-component patterns, which basically dominates the whole recommendation performance. During the performance evaluation for Baya, we have also observed that the time required for weaving a pattern is negligible with respect to the total time required for the pattern recommendation and weaving.

1.8 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

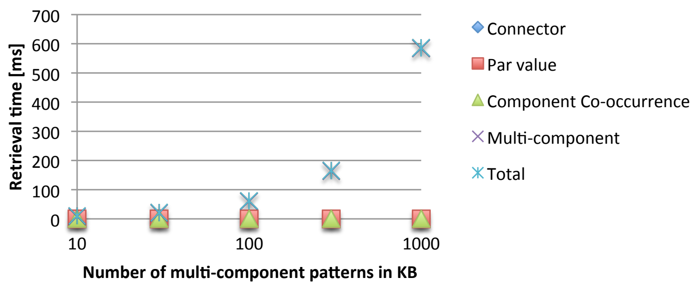


Fig. 1.4 Recommendation types and times in response to a new component added to the canvas

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 a concrete service for an abstract activity at runtime, we aim at interactively assisting developers at design time with domain knowledge in the form of modeling patterns.

In the context of ***web mashups***, Carlson et al. [2], 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. [3] 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. [9] 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. [5] 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 [1] use automatic planning in ***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. [12] 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. [8] 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 models into a keyword vector space. Gschwind et al. [7] allow users to use the control flow patterns introduced by Van der Aalst et al. [14], just like other modeling elements. The system does not provide interactive recommendations and rather focuses on the correct insertion of patterns.

In summary, assisted mashup and service composition approaches either focus on single components or connectors, or they aim to auto-complete compositions starting from user goals by using AI Planning techniques. The BPM approaches do focus on patterns, but most of the times pattern similarity is based on label/text similarity, not on structural compatibility. In our work, we consider that if components have been used together successfully multiple times, very likely their joint use is both syntactically and semantically meaningful. Hence, there is no need to further model complex ontologies or composition rules. Another key difference is that we leverage on the *interactive recommendation* of composition patterns to assists users step-by-step based on their actions on the design canvas. We do not only tell users which patterns may be applied to progress in the mashup composition process, but we also *automatically weave* recommended patterns on behalf of the users.

1.9 Conclusions

With this work, we aim to pave the road for assisted development in web-based composition environments. We represent *reusable knowledge* as patterns, explain how to automatically *discover* patterns from existing mashup models, describe how to *recommend* patterns fast, and how to *weave* them into partial mashup models. We therefore provide the *basic technology* for assisted development, demonstrating that the solutions proposed indeed work in practice.

As for the discovery of patterns, it is important to note that even patterns with very low support carry valuable information. Of course, they do not represent generally valid solutions or complex best practices in a given domain, but still they show *how* its constructs have been used in the past. This property is a positive side-effect of the sensible, a-priori design of the pattern structures we are looking for. Without that, discovered patterns would require much higher support values, so as to provide evidence that also their pattern structure is meaningful. Our analysis of the patterns discovered by our algorithms shows that, in order to get the best out them, domain knowledge inside the mashup models is crucial. Domain-specific mashups, in which composition elements and constructs have specific domain semantics, are a thread of research we are already following. As a next step, we will also extend the canonical model toward more generic mashup languages, e.g., including UI synchronization.

The results of our tests of the pattern recommendation approach even outperform our own expectations, also for large numbers of patterns. In practice, however, the number of really meaningful patterns in a given modeling domain will only unlikely grow beyond several dozens. The described recommending approach will therefore work 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. Recommendation retrieval times of fractions of seconds and negligible pattern weaving times will definitely allow us – and others – to develop more sophisticated, assisted composition environments. This is, of course, our goal for the future – next to going back to the users of our initial study and testing the effectiveness of assisted development in practice.

Acknowledgment. This work was supported by the European Commission (project OMELETTE, contract 257635).

References

1. S. Beauche and P. Poizat. Automated service composition with adaptive planning. In *ICSOC'08*, pages 530–537. Springer-Verlag, 2008.
2. M. P. Carlson, A. H. Ngu, R. Podorozhny, and L. Zeng. Automatic mash up of composite applications. In *ICSOC'08*, pages 317–330. Springer, 2008.
3. H. Chen, B. Lu, Y. Ni, G. Xie, C. Zhou, J. Mi, and Z. Wu. Mashup by surfing a web of data apis. *VLDB'09*, 2:1602–1605, August 2009.
4. A. De Angeli, A. Battocchi, S. Roy Chowdhury, C. Rodríguez, F. Daniel, and F. Casati. End-user requirements for wisdom-aware eud. In *IS-EUD'11*. Springer, 2011.
5. H. Elmeleegy, A. Ivan, R. Akkiraju, and R. Goodwin. Mashup advisor: A recommendation tool for mashup development. In *ICWS'08*, pages 337–344. IEEE Computer Society, 2008.
6. O. Greenshpan, T. Milo, and N. Polyzotis. Autocompletion for mashups. *VLDB'09*, 2:538–549, August 2009.
7. T. Gschwind, J. Koehler, and J. Wong. Applying patterns during business process modeling. In *BPM'08*, pages 4–19. Springer, 2008.
8. T. Hornung, A. Koschmider, and G. Lausen. Recommendation based process modeling support: Method and user experience. In *ER'08*, pages 265–278. Springer, 2008.
9. A. V. Riabov, E. Boillet, M. D. Feblowitz, Z. Liu, and A. Ranganathan. Wishful search: interactive composition of data mashups. In *WWW'08*, pages 775–784. ACM, 2008.
10. S. Roy Chowdhury, F. Daniel, and F. Casati. Efficient, Interactive Recommendation of Mashup Composition Knowledge. In *ICSOC'11*, pages 374–388. Springer, 2011.
11. S. Roy Chowdhury, C. Rodríguez, F. Daniel, and F. Casati. Baya: Assisted Mashup Development as a Service. In *WWW'12*, 2012.
12. S. Smirnov, M. Weidlich, J. Mendling, and M. Weske. Action patterns in business process models. In *ICSOC-ServiceWave'09*, pages 115–129. Springer-Verlag, 2009.
13. P. Tan, S. M, and K. V. *Introduction to Data Mining*. Addison-Wesley, 2005.
14. W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14:5–51, July 2003.

Appendix I

Complementary Assistance Mechanisms for End User Mashup Composition

Soudip Roy Chowdhury, Olexiy Chudnovskyy,Matthias Niederhausen, Stefan Pietschmann, Paul Sharples, Florian Daniel, and Martin Gaedke. Complementary Assistance Mechanisms for End User Mashup Composition. WWW 2013 Companion, May 2013.

Complementary Assistance Mechanisms for End User Mashup Composition

Soudip Roy Chowdhury[1], Olexiy Chudnovskyy[2],Matthias Niederhausen[3],
Stefan Pietschmann[3], Paul Sharples[4], Florian Daniel[1], and Martin Gaedke[2]
[1]{rchowdhury,daniel}@disi.unitn.it, [2]{olexiy.chudnovskyy,martin.gaedke}@cs.tu-chemnitz.de,[3]{matthias.niederhausen,stefan.pietschmann}@t-systems-mms.com,
[4]p.sharples@bolton.ac.uk

ABSTRACT

End user development becomes an increasingly important topic in the field of web mashups. Despite several efforts for simplifying the composition process, the learning curve for using existing mashup editors remains rather steep. In this paper, we describe how this barrier can be lowered by means of an *assisted development* approach that seamlessly integrates automatic composition and interactive pattern recommendation techniques into a mashup tool specifically tailored to the needs of end users. We showcase the use of such an assisted development environment in the context of the open-source mashup platform Apache Rave. The results of our user studies demonstrate the benefits of our approach for end user software development.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques; D.2.6 [**Software**]: Software Engineering—*Programming Environments*

Keywords

assisted mashup development, automated compostion, interactive pattern recommendation, end user development, crisis mashup

1. INTRODUCTION

Mashup tools, such as Yahoo! Pipes (`http://pipes.yahoo.com/pipes/`) or Apache Rave (`http://rave.apache.org/`), offer simple, visual metaphors that aim to enable end users without programming skills to design own composition logics by re-using existing components/widgets. Despite the popularity of mashup tools in research, their common claim for end user suitability is only weakly supported. In practice, building a mashup remains a challenging task for non-programmers and even for less-skilled developers. First, they simply lack knowledge about which components are available and which to use in the design. Second, they lack the necessary technical skills to configure components. Third, they lack algorithmic thinking and, hence, are not able to define a consistent composition logic that integrates multiple components.

In order to aid such users in the design of mashups, research has proposed two distinct approaches: ***goal-oriented solutions*** [4, 6, 9] aim to assist end users by automatically deriving compositions that satisfy user-specified goals; ***pattern-based development*** [3, 5] aims to recommend composition patterns in response to modeling actions, e.g., to auto-complete partial mashup models. Goal-oriented solutions strive for simple interactions with users to elicit the goal, i. e., intent of a composition without requiring users to actually model the mashup, while pattern-based approaches interactively assist them throughout the modeling process. None of the two individually may thus provide suitably assistance to an user in developing own mashups. Previous works therefore motivate the joint use of goal-based and pattern-based approaches for mashup composition [7], yet the authors don't provide detailed insight into how to realize such a system in practice.

Our paper presents an example of such *hybrid* assistance solution, built on top of Apache Rave, an open-source, widget-based mashup platform, and combining the simplicity of a dialog-based automatic composer with step-by-step assistance by an interactive pattern recommender [8]. In this demo we show how these two techniques *complement* each other well in assisting end users. We further show the usability of our system in an end user development scenario (emergency management). The user studies performed with our system provide evidence of the effectiveness of the hybrid system for assisted mashup development.

2. CHALLENGES AND CONTRIBUTIONS

To identify the challenges and requirements for end user assistance, let us introduce a real-world scenario, which also provides the context for our demonstration. In August 2002, a devastating flood caused by heavy rains hit the east of Germany and several other parts of Europe. Such a crisis situation demands for IT systems that support information seekers as well as humanitarian activity coordinators. The former want to quickly get an overview of the overall situation to understand the impact of an emergency incidents. This requires them to aggregate and filter information from different data sources (e. g., news articles, social streams, etc.). The latter need tools that help them to coordinate rescue tasks, calculate risks, and communicate with both their teams and the local authorities.

These situational requirements underline the necessity of

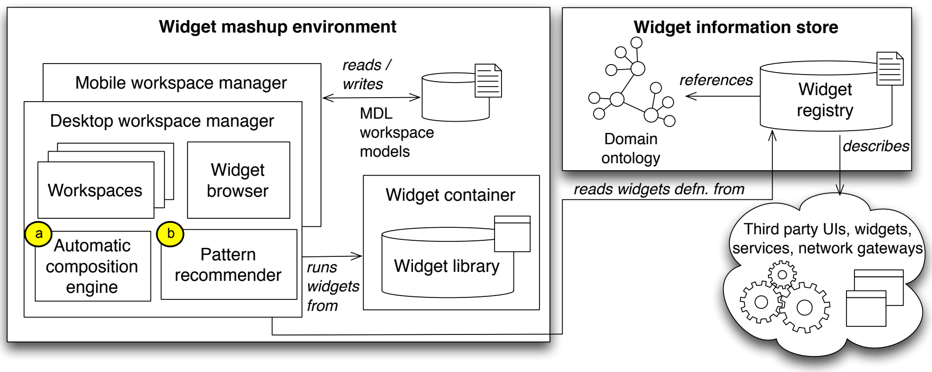


Figure 1: Functional architecture of the OMELETTE assisted development approach

a mashup platform that not only supports the quick development of such "long tail" applications but can also serves users with various technical backgrounds. However, especially in such time-critical situations, end users face several challenges during composition. First, while the intention of building a mashup is clear to end users, they do not necessarily know which kind of widgets they need. Second, if they do know, they still have no understanding of whether these widgets are available or how to compensate their absence. Third, in case both the functionalities needed and offered are clear, users have to find the corresponding widgets. This can be a cumbersome and error-prone adventure for different reasons, but mostly due to insufficient or unsuitable widget descriptions (e.g., meta-data). Finally, end users typically lack knowledge about how to define the data and/or control flow in a composition, especially when it comes to interoperability problems at the widget interface and data layer.

In this demonstration, we show our approach to address these needs in crisis situations like the one explained before. We have designed two complementary assistance mechanisms, namely an *automatic composition engine* and a *pattern recommender*, that help users in designing their mashups fast and reliably.

The contributions of this paper are as follows:

- We describe our *goal-oriented dialog system* that enables the automatic composition of workspaces to less skilled users.
- We describe our *pattern-based recommendation system* that enables more skilled users to refactor and/or extend mashup designs in a step-by-step fashion.
- We explain the *implementation* and integration of the respective algorithms into the open source mashup platform Apache Rave.
- We report on a *user study* conducted with 44 participants, demonstrating the benefit of combining both assistive techniques in one environment.

In the following, we present the realization of these contributions in the context of the EU FP7 project OMELETTE (`http://www.ict-omelette.eu/`). After that, we provide an overview of the demonstration workflow and close this paper with our findings from the user studies and ideas for future work.

3. OMELETTE APPROACH TO ASSISTED MASHUP COMPOSITION

Figure 1 gives an overview of the OMELETTE mashup architecture. Therein, widgets represent full-fledged application modules integrating both business logic and UI. The widget mashup environment allows end users to compose mashups (so-called workspaces) by placing one or more widgets on the composition canvas. The technical complexity of defining the data flow logic (or configuration parameters) are abstracted from the user and handled automatically by the platform and the widget implementations. To facilitate assisted development, OMELETTE extends the mashup engine of Apache Rave and includes additional functional blocks, e. g., inter-widget communication, widget information store, widget registry, etc.

Even though mashup creation sounds very simple, end users need support to cope with the challenges discussed in the previous section. Therefore, we introduce *two* tools to support users realizing the desired applications without being overwhelmed by the technical details of the underlying platform and widget specifications. The first tool, the **Automatic Composition Engine** (ACE), targets novices who have no or very little experience in mashup development and need to be guided through the composition process. The second tool, the **Pattern Recommender** (PR), addresses those users who are already familiar with the composition environment, but need help in finding appropriate building blocks.

3.1 Automatic Composition Engine

The ACE allows end users to focus on the *goal* of the composition instead of the individual building blocks and their accurate "wiring". The ACE (cf. Figure 1.a) extends the mashup environment with a dialog-based interface that enables end users to specify their goals in an interactive manner. The dialog takes place in form of a question-answer game, during which the system elicits and refines user goals. Eventually, this process results in the automatic composition of a workspace derived from the identified goals.

The basis for this mechanism is an extensible knowledge base and a rule engine guiding the dialog with the user and ensuring the goal of a composition is clear at the end. The knowledge base comprises a domain ontology with facts about the application domains (e. g., project management or trip organization) and a set of functions defining the be-

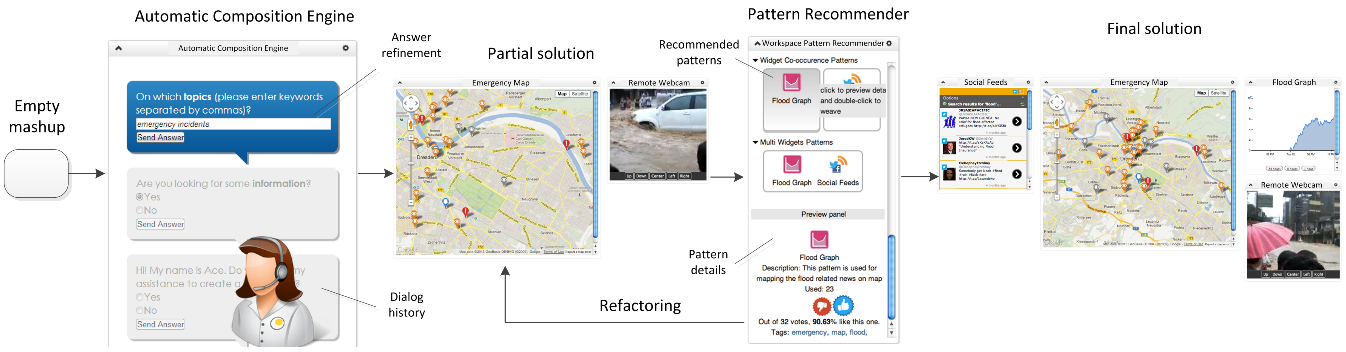

Automatic Composition Engine
Empty mashup
Automatic Composition Engine
On which topics (please enter keywords separated by commas)?
emergency incidents
Send Answer
Are you looking for some information?
Yes
No
Send Answer
Hi! My name is Ace. Do … my assistance to create a …?
Yes
No
Send Answer
Answer refinement
Dialog history
Partial solution
Emergency Map
Remote Webcam
Refactoring
Pattern Recommender
Recommended patterns
Workspace Pattern Recommender
Widget Co-occurence Patterns
Flood Graph
click to preview deta and double-click to weave
Multi Widgets Patterns
Flood Graph
Social Feeds
Preview panel
Pattern details
Flood Graph
Description: This pattern is used for mapping the flood related news on map
Used: 23
Out of 32 votes, 90.63% like this one.
Tags: emergency, map, flood,
Final solution
Social Feeds
Emergency Map
Flood Graph
Remote Webcam

Figure 2: Hybrid recommendation system

havior of the dialog agent. The first function dedicated to ***question building*** is language-aware and responsible for the conversation flow with the user. The rule-based definition of this function enables dynamic and flexible conversations taking application context into account, e. g, the availability and capabilities of mashup components. Based on the user responses, the ***evidence collection*** function produces an overlay model of the domain ontology. The ***widget selection*** function defines a mapping between possible sets of collected evidences and a SPARQL search query to be issued to the widget registry, containing semantic descriptions of functional and non-functional aspects of the available widgets. The current implementation filters keyword annotations from the domain knowledge against widgets' textual attributes, such as title, description, tags, and category. To guarantee the best interoperability between the resulting widgets, they are filtered based on the information about their inter-widget-communication capabilities. Finally, the ***workspace configuration*** function derives a set of configuration parameters to be applied to widgets and workspace based on the evidences collected during the dialog.

In summary, the dialog agent helps users to select and configure widgets out of a large number of potentially incompatible components in an interactive and natural fashion. The more widgets with similar functionality and from different vendors are registered in the platform, the better results are achieved by ACE.

3.2 Pattern Recommender

The design goals behind the PR – as explained in our prior work [1] – can be summarized as follows: support less skilled users in *refactoring* mashups created by the ACE and providing autonomous assistance for skilled developers in building mashups from scratch. The PR helps users reuse existing composition knowledge: the knowledge behind the PR's recommendations is harvested from existing workspace models. Extracted patterns are stored in a knowledge base (KB), which is structured to minimize database join operations for pattern retrieval at runtime. Currently, the PR supports two composition pattern types: ***widget co-occurence*** and ***multi-widget*** patterns. During composition, the PR reacts to user modeling *actions* (adding, deleting, or selecting a widget, etc.) on widgets (the *object* of an action) in the workspace. Upon each interaction, the *action* and its *object* are captured by the *recommendation engine* via suitable event listeners. With this information, the PR engine queries the *client-side KB* for recommendations, where an *object-action-recommendation mapping* tells the engine which types of recommendations are to be retrieved. The list of patterns retrieved from the KB are then filtered and ranked based on the partial mashup model in the workspace and rendered in the *recommendation panel*. The panel is the user interface of the PR and allows users to select a recommended pattern or browse its details. Upon selection of a pattern, the PR automatically weaves it into the current workspace model, resolving possible model conflicts.

4. IMPLEMENTATION

OMELETTE's assisted development approaches are implemented by extending two active Apache Software Foundation projects: Apache Rave and Apache Wookie (`http://wookie.apache.org/`). While Rave is used as the core mashup engine in OMELETTE, Wookie serves as a repository and runtime container for W3C widgets, accessible by both assistance mechanisms for retrieving widget information. Both assistance tools were realized as W3C widgets and are shown in action in Figure 2.

Apart from its client-side UI, **ACE**'s answer processing and evidence collection take place on the server side via a dedicated RESTful interface. The conversation and question building strategies are specified using production rules, which are executed on the server side by the JBoss Drools engine (`http://www.jboss.org/drools/`). To find widgets that best fit a user's needs, the server part of ACE uses a dedicated semantic registry (widget registry cf. Figure 1) based on the WebComposition/DataGridService [2].

The **PR** widget contains recommendation and weaving algorithms implemented in JavaScript. The client-side pattern KB runs on an in-browser SQLite (`http://www.sqlite.org/`) implementation. This eliminates performance overheads of client-server communication for retrieving recommendation patterns at runtime. Client and server-side pattern KBs are synchronized when loading the PR into a user's workspace. From then on, all queries triggered by the PR to retrieve patterns from the KB are directed to the client-side only. JavaScript event listeners capture the triggering events for pattern retrieval, i. e., DOM modifications (e.g., adding a widget, deleting a widget) of the workspace model.
Both the ACE and the PR rely on three new APIs introduced into Apache Rave to provide the necessary integration points between the mashup platform and the assistance tools. The first API allows the pattern mining algorithm of PR's server-side component to fetch all workspace models from the workspace repository. The second one is used by the recommendation algorithm implemented in PR to re-

trieve information about the set of widgets present in the current workspace model. Finally, the third one is used by ACE and PR to populare workspaces with new widgets.

5. DEMONSTRATION STORYBOARD

In the demo session, we plan to showcase the two assistive techniques described above using the scenario introduced in the beginning of this paper. We will show how workspaces can be created ad hoc by end users in the case of an emergency situation, and how such workspaces can be rapidly extended with the help of the assistance provided by the OMELETTE approach.

The demo will start from a blank workspace. By interacting with the dialog system of the ACE, the user will express his/her composition intentions, e. g., to gather information about emergency incidents and getting public web cam footage to gauge the impact on the ground zero. This will lead ACE to automatically populate the workspace with suitable widgets and their implicit wiring, without any further user interaction.

The second part of our demo will involve the extension of this workspace model with other widgets (e.g., telco widgets) that may be of interest to an emergency coordinator in the given scenario. We will show how the interactive recommender helps users to refactor an existing workspace with a new set of widgets in a step-by-step manner.

Finally, we will explain the architecture behind our assisted platform and share the lessons learned during the implementation and user studies.

A screencast of our approach at work is available at `http://www.ict-omelette.eu/assisted-composition`.

6. EVALUATION AND FUTURE WORK

In order to evaluate our approach with a potential user group, we have conducted a user study including both the ACE and the PR. The study was conducted in China and Germany. In total, 44 participants attended the study, with only 11 of them having had previous experiences with widgets or configuring portal interfaces. Participants were equally distributed in test and control groups.

For evaluating the **ACE**, users were given the task to create a simple mashup that required them to build a workspace with at least three widgets. The required widgets were not explicitly named, but rather described by their functionality. The control group was assigned the same task, but had to use Apache Rave's widget store to search for suitable widgets. Interestingly, the study shows that participants using the ACE took more time on average than the control group (261 vs. 159 seconds). One decisive factor for this is the tool's learning curve: While all participants had the chance to try out the widget store before, the ACE was newly introduced. Even further, a few usability issues led some users far astray (hence the high variance), requiring them to start over multiple times.

The **PR** was evaluated in a similar fashion. Here, participants had to modify a given workspace with additional functionalities. The test group used the PR, whereas the control group again used the widget store to find right widgets. As the results show, the PR significantly reduced the overall task completion time (57 vs. 137 seconds). While the difference of the mean development time between the groups seems rather big, it must be noted that the PR has the distinct advantage of not requiring the user to leave his workspace in search of widgets.

For user satisfaction, there was a huge gap between user groups: while 64% of Chinese users said that they found our assistance mechanism to be useful and would use it again, only 36% of German users did so. However, 61% of all users agreed that this feature was important or even essential for a mashup environment.

Overall, the study shows that users with little experience in mashups prefer being guided through the processes of creating and extending their workspace. While no significant increase in efficiency could be verified for the process of creating a new mashup from scratch, results show that the recommendation of additional widgets based on an existing workspace provides significant benefits for users.

As a follow up to this user study, we are currently addressing usability issues of Apache Rave and of our assistance mechanisms. We are also working on a more natural goal elicitation system for the ACE to give users more freedom in creating new workspaces. The improvements on the ACE are going to be part of another evaluation, the results thereof we plan to present at the demo session. Future work includes improvement of composition patterns coverage in the PR's pattern KB as well as providing more explanations with each recommendation step to help users understand and decide whether to follow a recommendation or not.

Acknowledgment. This work was supported by the European Commission (project OMELETTE, contract 257635). Authors thank Vadim Chepegin from TIE Kinetix b.v. for his contributions to the design and implementation of the ACE.

7. REFERENCES

[1] S. R. Chowdhury, C. Rodríguez, F. Daniel, and F. Casati. Baya: assisted mashup development as a service. In *WWW'12 (Companion Volume)*, pages 409–412.

[2] O. Chudnovskyy and M. Gaedke. Development of Web 2.0 Applications using WebComposition / Data Grid Service. *Service Computation'10*, pages 55–61.

[3] O. Greenshpan, T. Milo, and N. Polyzotis. Autocompletion for mashups. *VLDB'09*, 2:538–549.

[4] M. Henneberger, B. Heinrich, F. Lautenbacher, and B. Bauer. Semantic-Based Planning of Process Models. In *Multikonferenz Wirtschaftsinformatik'08.*

[5] A. H. H. Ngu, M. P. Carlson, Q. Z. Sheng, and H.-y. Paik. Semantic-based mashup of composite applications. *IEEE Trans. Serv. Comput.*, 3(1):2–15, Jan. 2010.

[6] S. Pietschmann, C. Radeck, and K. Meißner. Semantics-based discovery, selection and mediation for presentation-oriented mashups. In *MASHUPS'11.*

[7] C. Radeck, A. Lorz, G. Blichmann, and K. Meißner. Hybrid recommendation of composition knowledge for end user development of mashups. In *ICIW'12*, pages 30–33.

[8] S. Roy Chowdhury, F. Daniel, and F. Casati. Efficient, Interactive Recommendation of Mashup Composition Knowledge. In *ICSOC'11*, pages 374–388.

[9] V. Tietz, G. Blichmann, S. Pietschmann, and K. Meißner. Task-based recommendation of mashup components. In *ICWE'11*, pages 25–36.

Appendix J

Interactive Recommendation and Weaving of Mashup Model Patterns for Assisted Mashup Development

*** To be submitted to ACM Transactions on the Web**

Interactive Recommendation and Weaving of Mashup Model Patterns for Assisted Mashup Development

Soudip Roy Chowdhury, University of Trento, Italy
Florian Daniel, University of Trento, Italy
Fabio Casati, University of Trento, Italy

With this article, we give an answer to one of the open problems of tool-based mashup development, i.e., the *lack of mashup and modeling knowledge* users may face when operating a mashup tool. Mashup tools have undoubtedly contributed to the simplification of mashup development, but mashups can nevertheless be complex software artifacts. Developing a mashup that integrates multiple resources from the Web can easily become non-trivial and require 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 in general neither intuitive nor standardized across different mashup tools.

We show how it is possible to effectively *assist* the users of mashup tools in their development task with *contextual, interactive recommendations* of composition knowledge in the form of *mashup model patterns*. We study a set of recommendation algorithms with different levels of performances and describe a flexible pattern weaving approach for the one-click reuse of patterns. We report on the implementation of two pattern recommender plug-ins for two different mashup tools and demonstrate via suitable user studies that recommending contextual mashup model patterns significantly reduces development times in both mashup environments.

Categories and Subject Descriptors: H.m [**Information Systems**]: Miscellaneous; D.1 [**Software**]: Programming Techniques; D.2.6 [**Software**]: Software Engineering—*Programming Environments*

General Terms: Design, Algorithms, Experimentation, Performance

Additional Key Words and Phrases: recommendation, weaving, assisted mashup development, user studies

ACM Reference Format:
ACM Trans. Web V, N, Article A (January YYYY), 35 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Mashing up, i.e., composing, a set of services, for example, into an application 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. Existing mashup tools provide visual programming languages that help users in designing such mashup applications by integrating content from the

Author's addresses:
Soudip Roy Chowdhury, University of Trento, Via Sommarive 5, 38123 Povo (TN), Italy.
email id: rchowdhury@disi.unitn.it; alternate email id: soudeep@gmail.com
Florian Daniel, University of Trento, Via Sommarive 5, 38123 Povo (TN), Italy. email id: daniel@disi.unitn.it
Fabio Casati, University of Trento, Via Sommarive 5, 38123 Povo (TN), Italy. email id: casati@disi.unitn.it

Web. These users, however, might lack appropriate awareness of which composable elements a mashup platform 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 a modeling language.

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 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. In order to better understand the problem that we address in this paper, let's have a look at how a mashup is, for instance, composed in Yahoo! Pipes `http://pipes.yahoo.com/pipes/`, one of the most well-known and used mashup platforms as of today. Let us assume we want to develop a simple pipe that fetches a set of news feeds from Google News website, 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 locates them on a Yahoo! Map based upon the geo-location associated with each news item.

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://www.geonames.org/`, which takes a news RSS feed as an input, analyzes its content, and inserts geo-coordinates, i.e., longitude and latitude, into each news item (where applicable). Doing so requires setting few 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 cre-

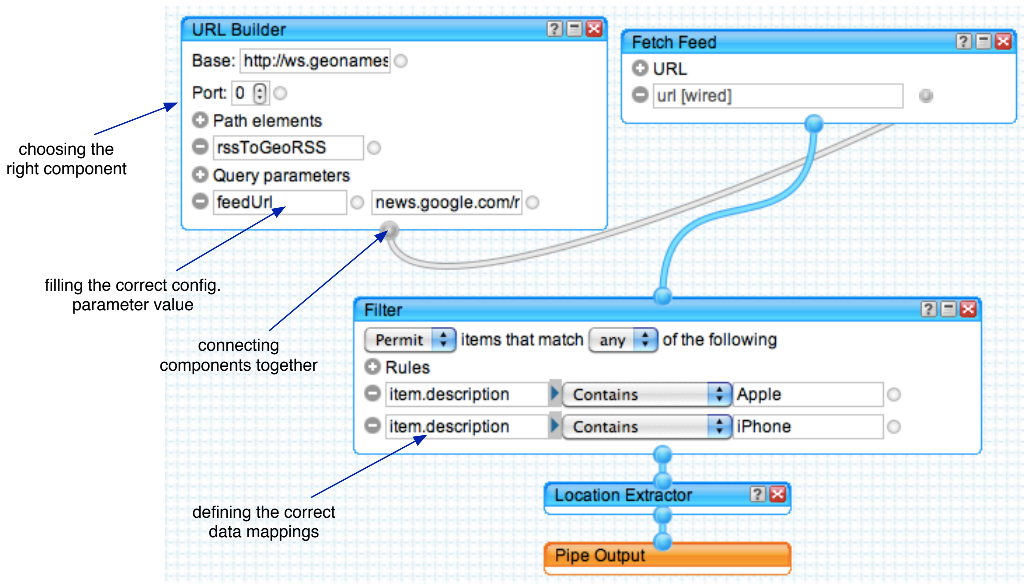

URL Builder
Base: http://ws.geonames
Port: 0
Path elements
rssToGeoRSS
Query parameters
feedUrl
news.google.com/r
Fetch Feed
URL
url [wired]
Filter
Permit items that match any of the following
Rules
item.description Contains Apple
item.description Contains iPhone
Location Extractor
Pipe Output
choosing the right component
filling the correct config. parameter value
connecting components together
defining the correct data mappings

Fig. 1. Screenshot of the Yahoo! Pipes mashup environment showing typical composition steps

ated URL is fed into the *Fetch Feed* component, which fetches the geo-enriched news feed from the Google News site. In order to filter out the news items that we are really interested in, we need to use the *Filter* component, which requires the setting of proper filter conditions via the *Rules* input field. Feeding the filtered feed into the *Location Extractor* component causes the plot of the news items on a Yahoo! Map. Finally, the *Pipe Output* component specifies the end of the pipe.

If we analyze the development steps above, we can easily understand that developing even such a simple composition is out of reach for people without sufficient programming knowledge. As pointed out in the Figure 1, the *URL Builder*, for example, requires the correct setting of it's configuration parameter. Then, components need to be correctly connected in order to define a consistent data-flow logic i.e., the output parameter of a component must be mapped correctly to the input parameter/s of other component/s. But more importantly, plotting news onto a map requires knowing that this can be done by first enriching a feed with geo-coordinates, then by fetching the actual feed, and then by plotting the fetched items on a map. Understanding all these programming concepts is *neither trivial nor intuitive* for a less-skilled developer.

To have a more detailed understanding of the problem space we performed an initial user study [De Angeli et al. 2011] with few end users (10 university accountants), who have little technical expertise. Inspired by the result of this study on how end users would like to be assisted in mashup development, we have started our research explorations toward the interactive, contextual recommendation of composition knowledge in order to assist a modeler in each step of his/her development task, e.g., by suggesting a candidate next component or a whole chain of tasks. The vision is that of developing an assisted, web-based mashup environment (an evolution of our former work [Daniel et al. 2009]) 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 a few of the core research challenges of this vision, i.e., how to design an interactive development recommendation system that supports the reuse of existing composition knowledge for developing mashups in a step-by-step manner. That required us to find answers of the following ***research questions***:

— How to interactively reuse existing composition knowledge for developing new mashups.
— How to represent the composition knowledge that captures the typical modeling steps in mashup designs and that can be recommended to users as reusable knowledge.
— How to support interactive and contextual retrieval of composition knowledge at runtime.
— How to automate the application (weaving) of the modeling edits (addition or deletion of components, connectors etc.), captured in a composition knowledge, to the current composition context.
— How demonstrate and evaluate the usefulness of our interactive, contextual recommendation approach to its target users.
— How to assess the usability and the accuracy of our recommendation algorithms for the target user groups.

Working towards finding solutions for these research questions, in this paper we report on the whole research efforts that we have made so far which includes the summary of our already published research results and new research progresses and a detailed analysis of results. More specifically, in this paper we describe the following ***contributions***:

— We demonstrate the *conceptual model* of our interactive recommendation approach that we introduced in [Roy Chowdhury et al. 2010][1].
— We show *types* of composition patterns that capture the typical modeling steps inside a mashup platform[1].
— We summarize algorithms for the efficient querying and ranking of interactive recommendations based on exact and similarity based matching algorithms proposed in our prior work ([Roy Chowdhury et al. 2011])[1].
— We describe a novel development profile based *recommendation filtering* approach and related set of algorithms[2].
— We study the *performance* of the conceived algorithms and report on their accuracy measures[3].
— We describe our *pattern weaving algorithm* together with the respective weaving strategies and policies, which help to automatically apply a selected pattern to a mashup model under development[3].
— We detail the *architecture* for the assisted development environment that we introduced in [Daniel et al. 2012] and also discussed new architecture components that support development profile based recommendation filtering algorithms[3].
— We describe the *implementation* of our two prototype tools: (*Baya* [Roy Chowdhury et al. 2012]) pattern recommendation and weaving plug-in for Yahoo! Pipes and *Pattern recommender: an extension of Baya* for Apache Rave mashup platform (`http://rave.apache.org/`)[3].
— We report on the results of two *user studies* proving the effectiveness of the approach[2].

In Section 2 we define the preliminaries and the types of composition patterns that we use through out the paper. In Section 3 we discuss in detail our interactive recommendation approach starting from a conceptual model to a set of pattern retrieval algorithms and their detailed performance and accuracy results. In Section 4 we discuss about our pattern weaving algorithms. In Section 5 we show the detailed functional architecture of our system followed by implementation details of our two prototype systems. Section 6 reports the results of two user studies that we performed to assess the usefulness of our recommendation approach for two different mashup platforms and different target user groups. In Section 7 we discuss about the related works, and in Section 8 we concluded our discussion by recapping the lessons we learned and by providing hints of our future work.

2. ASSISTED MASHUP DEVELOPMENT: PRELIMINARIES AND PROBLEM STATEMENT

In our approach, we assist the design of data mashup. Before discussing about our development assistance approach, let us first formalize the model of such mashups. This formalization will help us to define types of composition patterns that capture the knowledge about the modeling steps involved in developing such mashups. These composition patterns are then be used as the knowledge behind our development assistance approach.

2.1. Reference mashup models

Without loss of generality (in terms of mashup models that can be supported), in this paper we focus on data mashups. Data mashups are simple in terms of modeling constructs and expressive power, and therefore, the structure and the complexity

[1] We summarize our previous published work.
[2] We describe entirely *new* contributions.
[3] We summarize our earlier work with major *new* contributions.

of mashup patterns are limited. The model, we define in this paper, is inspired by the modeling constructs in mashup tools like Yahoo! Pipes (`http://pipes.yahoo.com`), JackBe Presto (`http://www.jackbe.com`) and MyCocktail (`http://www.ict-romulus.eu/MyCocktail/`); in our future work, we will also focus on mashup models that take into account control flow based composition patterns and user interfaces.

We define a ***data mashup*** as a tuple $m = \langle name, id, C, DF, RES \rangle$, where $name$ is the name of the mashup, id a unique identifier, C is the set of components, DF is a set of data flow connectors propagating data among components (e.g., data sources or operators), and RES is a set of result parameters of the mashup. Specifically:

— $C = \{c_k | c_k = \langle name_k, id_k, type_k, IP_k, IN_k, DM_k, VA_k, OP_k, OUT_k \rangle\}$ is the set of ***components***, where c_k is an instance of a component characterized by a name label $name_k$, a unique id id_k, and a type $type_k$. IP_k, IN_k, OP_k and OUT_k, are sets of input ports, input parameters, output ports, and output attributes (the attributes of the items in the output data flow), and:
  — $DM_k \subseteq IN_k \times (\bigcup_{c \in C} c.OUT)$ is the set of ***data mappings*** that map attributes of the input data flows of c_k to the input parameters of c_k.
  — $VA_k \subseteq IN_k \times (STR \cup NUM)$ is the set of ***value assignments*** of the input parameters IN_k; values are strings (STR) or numbers (NUM).

— $DF = \{df_j | df_j = \langle srccid_j, srcop_j, tgtcid_j, tgtip_j \rangle\}$ is a set of ***data flow connectors*** that, each, assign the output port $srcop_j$ of a source component with identifier $srccid_j$ to an input port $tgtip_j$ of a target component identified by $tgtcid_j$, such that $srccid \neq tgtcid$.

— $RES \subseteq \bigcup_{c \in C} c.OUT$ is the set of ***outputs*** computed by the mashup.

We exemplify our approach in the context of Yahoo! Pipes, which is well known and comes with a large body of readily available mashup models that we can analyze. Pipes also fits into the mashup model described above and, thus, the examples we provide in this paper are directly applicable in the context of Yahoo! Pipes. Patterns $ptype$ in Pipes correspond to a simplified version of the mashup model i.e., $ptype$=$\langle C, DF, RES \rangle$ with attributes as specified above.

2.2. Composition pattern types

Considering the typical modeling steps 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 set of ***pattern types***. Visual representation of these pattern types is represented at Figure 2.

Parameter value pattern. The parameter value pattern represents a set of recurrent value assignments VA for the input fields IN of a component c (identifiers are system-generated):

$ptype^{par} = \langle \{c\}, \varnothing \, \varnothing \rangle$;

$c = \langle name, 0, type, \varnothing, IN, \varnothing, \varnothing, VA, \varnothing \rangle$

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

Connector pattern. The connector pattern represents a recurrent connector df_{xy}, given 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:

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

$c_x = \langle name_x, 0, type_x, \varnothing, \varnothing, \varnothing, \varnothing, \{op_x\}, OUT_x \rangle$;

$c_y = \langle name_y, 1, type_y, \{ip_y\}, IN_y, DM_y, \varnothing, \varnothing, \varnothing \rangle$.

This pattern helps connecting 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 and comes with their connector, parameter values, and c_y's data mapping logic:
$ptype^{com} = \langle \{c_x, c_y\}, \{df_{xy}\}, \varnothing \rangle$;
$c_x = \langle name_x, 0, type_x, \varnothing, IN_x, \varnothing, VA_x, \{op_x\}, OUT_x \rangle$;
$c_y = \langle name_y, 1, type_y, \{ip_y\}, IN_y, DM_y, VA_y, \varnothing, OUT_y \rangle$.
This pattern helps developing 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:
$ptype^{emb} = \langle \{c_x, c_y, c_z\}, \{df_{xy}, df_{xz}, df_{zy}\}, \varnothing \rangle$;
$c_x = \langle name_x, 0, type_x, \varnothing, \varnothing, \varnothing, \varnothing, \{op_x\}, OUT_x \rangle$;
$c_y = \langle name_y, 1, type_y, \{ip_y\}, IN_y, DM_y, VA_y, \varnothing, \varnothing \rangle$;
$c_z = \langle name_z, 2, type_z, \{ip_z\}, IN_z, DM_z, VA_z, \{op_z\}, OUT_z \rangle$.
This pattern helps, for instance, modeling loops, a task that is usually not trivial to non-experts.

Multi-component pattern. The multi-component pattern represents recurrent model fragments that are generically composed of multiple components. This pattern captures information about the full model fragment, along with its constituent set of components and connectors:
$ptype^{mul} = \langle C, DF, \varnothing \rangle$;
$C = \{c_i | c_i.id = i; i = 0, 1, 2, ...\}$.
This pattern helps understanding domain knowledge and best practices as well as keeping agreed-upon modeling conventions.

This list of pattern types is not exhaustive, but it contains the most representative steps in mashup designs. These patterns help understanding domain knowledge and best practices as well as keeping agreed-upon modeling conventions. To read more about the basic intuitions behind coming up with this set of composition patterns, one can refer to our earlier work [Roy Chowdhury et al. 2011].

2.3. Sources of reusable composition patterns

Composition patterns, as described in the previous section, may come from different sources such as usage examples or tutorials of the modeling tool (developer knowledge), best modeling practices (domain expert knowledge), or recurrent model fragments (community composition knowledge) in a given repository of mashup models. In our research, so far, we have experimented with composition patterns coming from two of the above mentioned sources, i.e., community composition knowledge [Roy Chowdhury et al. 2010] and domain expert knowledge [Roy Chowdhury et al. 2013]. In [Roy Chowdhury et al. 2012] we demonstrate how we enable reuse of composition patterns that are automatically discovered from the existing Yahoo! Pipes model repository and in [Roy Chowdhury et al. 2013] we show how our reuse approach is extended with composition patterns that are manually provided by domain experts in Apache Rave. The intention behind these experimentations is to verify the decoupling of the reuse approach from the knowledge discovery approach and to determine the scalability of our knowledge reuse approach across different sources for composition patterns.

2.4. Problem statement

Given the types of composition patterns as described in the Section 2.2, the problems that we want to address in this paper are (i) how to interactively and contextually recommend these composition patterns inside mashup tools in order to guide users with

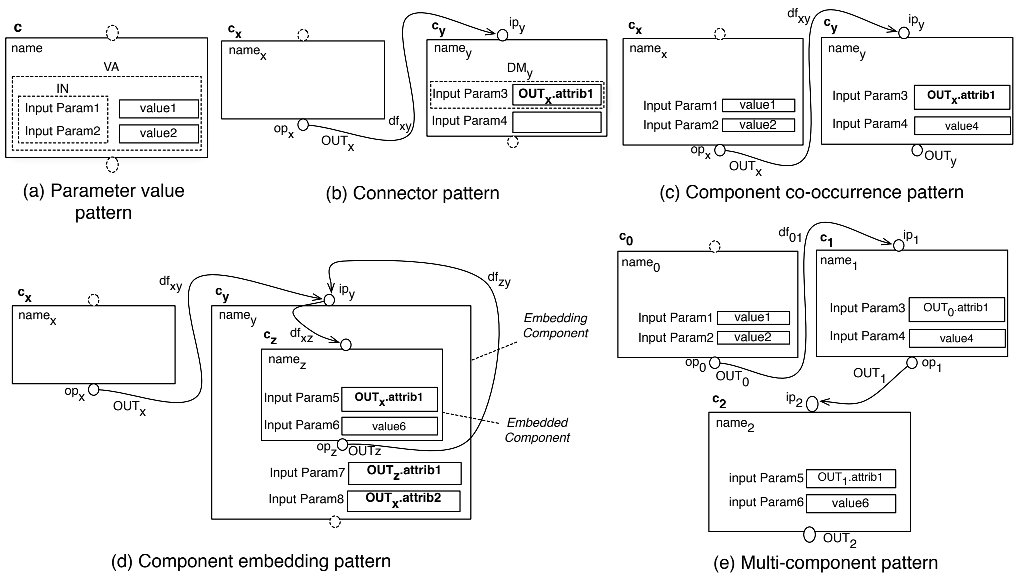

(a) Parameter value pattern
(b) Connector pattern
(c) Component co-occurrence pattern
(d) Component embedding pattern
(e) Multi-component pattern
Embedding Component
Embedded Component

Fig. 2. Composition patterns to aid stepwise, automated mashup modeling

the next modeling steps, and once a user accepts a composition pattern recommendation as provided by our system (ii) how to automatically apply (weave) the selected recommendation inside his/her current mashup design by automatically executing a set of mashup operations on behalf of the user.

3. INTERACTIVE RECOMMENDATION OF COMPOSITION KNOWLEDGE

The research questions that we addressed in this section is how to aid users of a mashup tool by interactively recommending him/her a set of composition patterns that we have identified in the Section 2.2. In particular in this section we focus on defining the conceptual model for our overall interactive assistance approach, followed by a set of recommendation algorithms that are designed to address the research questions introduced in Section 1.

3.1. Conceptual model

The primary goal of the *interactive pattern recommender* is to assist users in designing mashups in a step-by-step manner. The assistance comes in the form of reusable composition knowledge that is represented as composition patterns. The structure and types of composition patterns, that capture the typical design steps in a mashup environment, are the core behind the development assistance supported by our recommendation algorithms.

Figure 3 depicts the conceptual model of our interactive pattern recommendation approach. The “white boxes” in the right hand side of Figure 3 represent the condition or context under which a recommendation is triggered by the recommendation algorithm. For example, applying a modeling action *fill* to an object of type *parameter field* triggers a recommendation that consists of a parameter value pattern. The *object-action-recommendation rule* determines the type/s of recommendations to be invoked based on the current partial composition state, modeling action and the object of the action inside the current partial composition. “Gray boxes” in the Figure 3 represent the concepts related to the recommendation. A recommendation can be to complete a

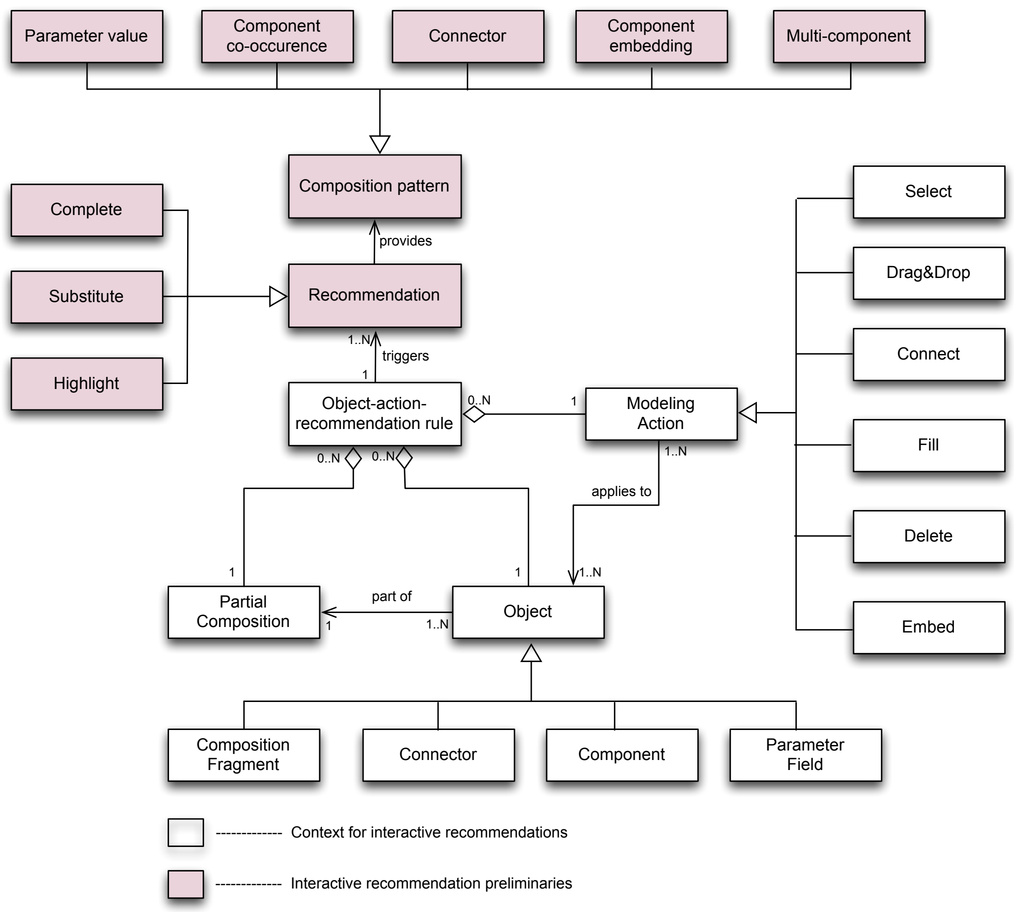


Fig. 3. Conceptual model of interactive pattern recommendation approach

partial composition with a composition pattern or to substitute an existing componen-t/s in the current composition with a similar one from the composition pattern, or the recommendation can highlight compatible components in the current modeling canvas. Composition pattern types are recommended to the users in order to help them to proceed with the development steps.

3.2. Representing and storing composition knowledge: the knowledge base

The core of the interactive recommender is the pattern knowledge base *KB* that stores composition patterns that are decomposed into their constituent parts, so as to enable the incremental recommendation approach. Figure 4 illustrates the structure of the pattern KB. This schema enables the fast retrieval of the composition patterns with a one-shot query over a single table. 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 avoid expensive database join operations or to defer them to the moment in which we really need to retrieve all details of a pattern. In order to retrieve, for example, the representation of a component co-occurrence pattern, it is therefore

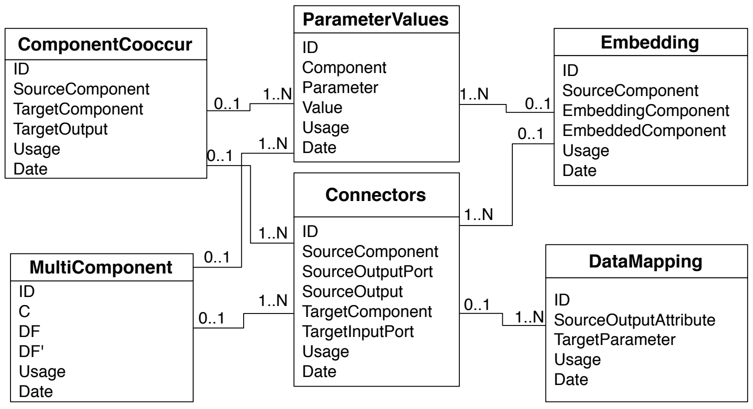


Fig. 4. Knowledge base structure for composition patterns

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.

3.3. Retrieving and recommending composition knowledge

As our conceptual model in Figure 3 shows that the tuple of $\langle object, action, pm \rangle$ is used as a query q by our recommendation algorithm to retrieve composition patterns from the KB, while the *object-action-recommendation* (OAR) rule determines which pattern types to be retrieved against the query (a specific type of *modeling action* on an *Object* type always retrieves a specific type of recommendation). For the retrieval of the contextual composition pattern types, we use a novel exact/approximate pattern matching algorithm that uses structural features of mashup designs during the matching process. The retrieved patterns are then filtered based on the development profiles of a user. The details of our pattern recommendation algorithms are explained in the following subsections.

3.3.1. Exact/approximate matching of composition patterns. 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.

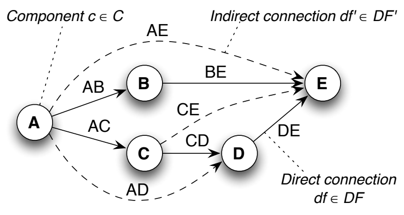
Algorithm 1 details this approach and summarizes the logic implemented by the recommendation engine. In line 3, we retrieve the types of recommendations that can be given (*getSuitableRecTypes* function), given an $object$-$action$ combination. Then, for each recommendation type, we either query for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

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 $recType \in recTypeToBeGiven$ do
5      if $recType \in \{ParValue, Connector, DataMapping, CompCooccur\}$ then
6          $Patterns$ = $Patterns \cup$ queryPatterns($object, KB, recType$) ;   // exact query
7      else
8          $Patterns$ = $Patterns \cup$
           getSimilarPatterns($object, KB.MultiComponent, CompSim, T_{sim}$) ;   // similarity
           search
9  foreach $pat \in Patterns$ do
10     if $rank(pat.cp, pat.sim, pm) \geq T_{rank}$ then
11         append($R, \langle pat.cp, rank(pat.cp, pat.sim, pm) \rangle$) ;   // rank, threshold, remember
12 orderByRank($R$);
13 groupByType($R$);
14 truncateByGroup($R, n$);
15 return $R$;
```

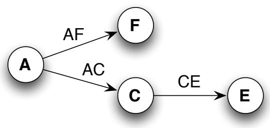
ALGORITHM 2: getSimilarPatterns

```
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 $cp \in CP$ 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, DF, DF' from obj
7      $obj.DF = obj.DF - cp.DF - cp.DF'$;
8      $obj.DF' = obj.DF' - cp.DF' - cp.DF$;
9      $approxSim = 0$;   // will contain the best similarity for approximate matches
10     foreach $c \in obj.C$ do
11         $SimC$ = getSimilarComponents($c, CompSim$) ;   // get set of similar components
12         foreach $simc \in SimC$ 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 $sim \geq T_{sim}$ then
19         $Patterns = Patterns \cup \langle cp, sim \rangle$ ;   // remember patterns with sufficient sim
20 return $Patterns$;
```



getStructure*(cp) = <C,F,F'>* with
C = {A,B,C,D,E},
DF = {AB,AC,BE,CD,DE}, and
DF' = {AE,AD,CE}

(a) An example composition pattern *cp*



getStructure*(object) = <C,F,F'>* with
C = {A,C,E,F},
DF = {AF,AC,CE}, and
DF' = {AE}

(b) An example *object* of a query *q*

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

(c) Component similarity matrix *CompSim*

	Exact match	Approximate match	Weight	meaning
simCC(*object,cp*)	3/4 = 0.75	1/1 = 1.00	0.50	match components
simDFDF(*object,cp*)	1/3 = 0.33	1/1 = 1.00	0.20	match connectors
simDFDF'(*object,cp*)	1/3 = 0.33	-	0.10	allow insertions
simDF'DF(*object,cp*)	0/1 = 0.00	-	0.10	allow deletions
simDF'DF'(*object,cp*)	1/1 = 1.00	-	0.10	allow substitutions

$\mathbf{sim}_{\mathbf{exact}}$ = 0.75*0.5 + 0.33*0.2 + 0.33*0.1 + 1.00*0.1 = **0.57**

$\mathbf{sim}_{\mathbf{approx}}$ = 1.00*0.5 + 1.00*0.2 = **0.70**

sim = $\mathrm{sim}_{\mathrm{exact}}$ + 0.5*$\mathrm{sim}_{\mathrm{approx}}$ /4 = **0.66**

(d) Pattern similarity calculation

Fig. 5. Pattern pre-processing and example of component similarity matrix $CompSim$. Components are identified with characters, connectors with their endpoints.

ALGORITHM 3: getSimilarity

```
Data: query object $object$, complex pattern $cp$
Result: $similarity$
initialize $w_i$ for $i \in 1..5$ with $\sum_i w_i = 1$;
$sim_1 = |object.C \cap cp.C| / |object.C|$ ;                  // matches components
$sim_2 = |object.DF \cap cp.DF| / |object.DF|$ ;               // matches connectors
$sim_3 = |object.DF \cap cp.DF'| / |object.DF|$ ;    // allows insertion of a component
$sim_4 = |object.DF' \cap cp.DF| / |object.DF'|$ ;    // allows deletion of a component
$sim_5 = |object.DF' \cap cp.F'| / |object.DF'|$ ; // allows substitution of a component
$similarity = \sum_i w_i * sim_i$;
return $similarity$;
```

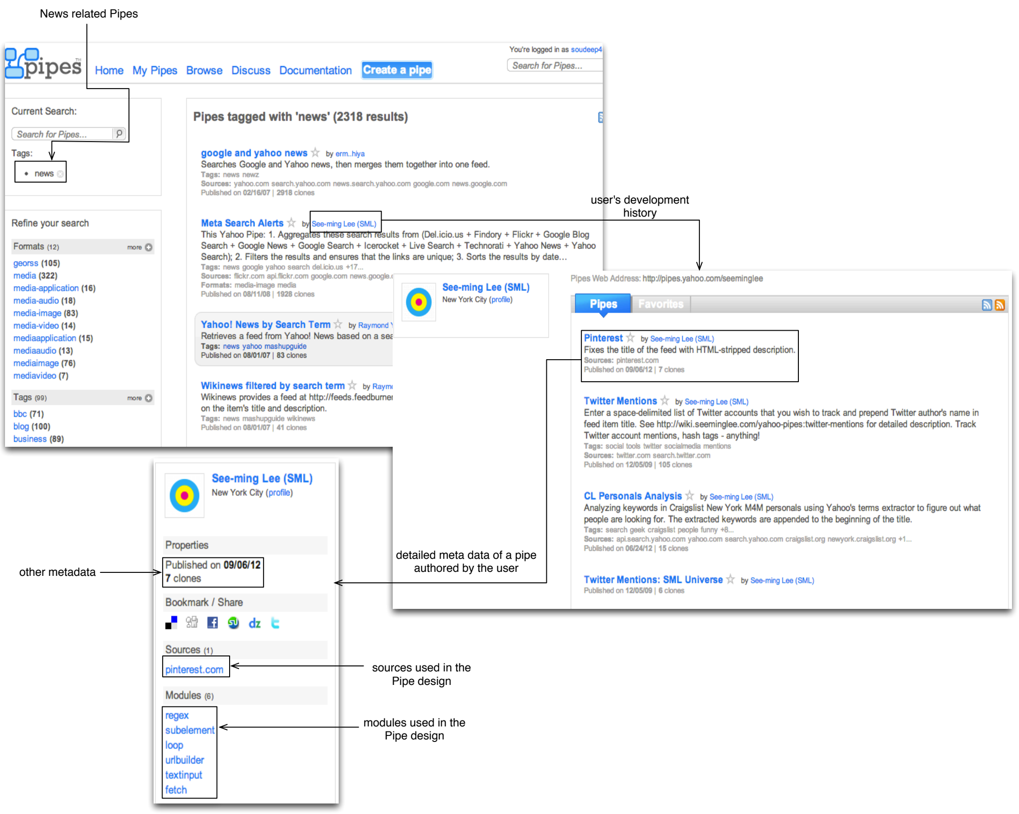
As for the retrieval of ***similar patterns***, our goal was to help modelers, not to disorient them. This led us to the identification of the following principles for the identification of “similar” patterns: preference should be given to exact matches of components and connectors in $object$, candidate 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. The original, graph-like structure of patterns are pre-processed to another pattern representation (cf. Figure 5) that increases the efficiency of our pattern search retrieval algorithm by saving us from the expensive graph traversal approach at runtime. Figure 5(a) illustrates the pre-processing logic: each complex 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 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 *Multi-component Pattern* entity introduced in Figure 4. 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 reflect semantic similarity among components (e.g., two *flight search* services); syntactic differences are taken into account by the pattern structures. Figure 5(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 2. 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 $sim = 0.57 + 0.35/4 = 0.66$. Figure 2(c) demonstrates these similarity calculation steps that are performed by our recommendation algorithm.

3.3.2. Automated filtering of composition patterns based upon user's development profile. A Close investigation of users' development histories in Yahoo! Pipes reveals that users tend to use the same set of modules/data sources across all applications he/she develops. Shani and Gunawardana [2011] suggested to recommend items that the user already knows and likes in order to gain users *trust* on the recommendation system. Our earlier user study [De Angeli et al. 2011] also concluded that users like to get personalized recommendations that take into consideration their development preferences. Being motivated by these observations, in this paper, we hypothesized that filtering of recommendations by considering user's preferences improves the overall accuracy of the system. The user's preferences can be derived by analyzing his/her past development history (cf. Figure 6), which includes the number of modules or data-sources he/she used, the set of tags he/she used for describing his applications etc. However, in this work we develop users' development profiles that only captures information about the number of modules/data sources that users have used in their published mashup designs. We derived these user development profiles in order to apply collaborative-filtering algorithms as a filtering technique for our recommendation algorithm. In the state of the art recommendation methods (e.g., [Sarwar et al. 2000]), which leverages user profiles for personalizing recommendations, the user-item matrix is used as a


News related Pipes
user's development history
other metadata
detailed meta data of a pipe authored by the user
sources used in the Pipe design
modules used in the Pipe design
pipes
Home
My Pipes
Browse
Discuss
Documentation
Create a pipe
Search for Pipes...
Pipes
Favorites

You're logged in as soudeep4

Current Search:

Tags: • news

Refine your search

Formats (12) more

georss (105)
media (322)
media-application (16)
media-audio (18)
media-image (83)
media-video (14)
mediaapplication (15)
mediaaudio (13)
mediaimage (76)
mediavideo (7)

Tags (99) more

bbc (71)
blog (100)
business (89)

Pipes tagged with 'news' (2318 results)

google and yahoo news by erm..hiya
Searches Google and Yahoo news, then merges them together into one feed.
Tags: news newz
Sources: yahoo.com search.yahoo.com news.search.yahoo.com google.com news.google.com
Published on 02/16/07 | 2918 clones

Meta Search Alerts by See-ming Lee (SML)
This Yahoo Pipe: 1. Aggregates these search results from (Del.icio.us + Findory + Flickr + Google Blog Search + Google News + Google Search + Icerocket + Live Search + Technorati + Yahoo News + Yahoo Search); 2. Filters the results and ensures that the links are unique; 3. Sorts the results by date...
Tags: news google yahoo search del.icio.us +17...
Sources: flickr.com api.flickr.com google.com news.google.
Formats: media-image media
Published on 08/11/08 | 1928 clones

Yahoo! News by Search Term by Raymond
Retrieves a feed from Yahoo! News based on a sea
Tags: news yahoo mashupguide
Published on 08/01/07 | 83 clones

Wikinews filtered by search term by Raym
Wikinews provides a feed at http://feeds.feedburne
on the item's title and description.
Tags: news mashupguide wikinews
Published on 08/01/07 | 41 clones

See-ming Lee (SML)
New York City (profile)

Pipes Web Address: http://pipes.yahoo.com/seeminglee

Pinterest by See-ming Lee (SML)
Fixes the title of the feed with HTML-stripped description.
Sources: pinterest.com
Published on 09/06/12 | 7 clones

Twitter Mentions by See-ming Lee (SML)
Enter a space-delimited list of Twitter accounts that you wish to track and prepend Twitter author's name in feed item title. See http://wiki.seeminglee.com/yahoo-pipes:twitter-mentions for detailed description. Track Twitter account mentions, hash tags - anything!
Tags: social tools twitter socialmedia mentions
Sources: twitter.com search.twitter.com
Published on 12/05/09 | 105 clones

CL Personals Analysis by See-ming Lee (SML)
Analyzing keywords in Craigslist New York M4M personals using Yahoo's terms extractor to figure out what people are looking for. The extracted keywords are appended to the beginning of the title.
Tags: search geek craigslist people funny +8...
Sources: api.search.yahoo.com yahoo.com search.yahoo.com craigslist.org newyork.craigslist.org +1...
Published on 06/24/12 | 15 clones

Twitter Mentions: SML Universe by See-ming Lee (SML)
Published on 12/05/09 | 6 clones

See-ming Lee (SML)
New York City (profile)

Properties
Published on 09/06/12
7 clones

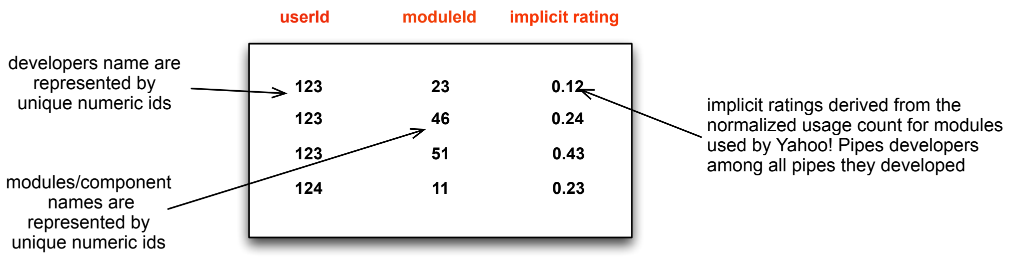
Bookmark / Share

Sources (1)
pinterest.com

Modules (6)
regex
subelement
loop
urlbuilder
textinput
fetch

Fig. 6. Screenshot depicting the development profile information for a user in Yahoo! Pipes

basis for calculating personalized recommendation. This user-items matrix are calculated based upon the explicit rating of an item provided by an user. However, in an application composition domain, such as Yahoo! Pipes, such an explicit rating information is not available. As an alternative to this, collecting users' implicit ratings by analyzing their development behavior [Hu et al. 2008], is the solution we explored in our approach. Hence, we designed a method to capture users' preferences on module and data sources as *implicit rating* metrics. We further normalized these implicit rating value so that it ranges between $(0, 1)$. Rating value 0 means that a user has never


developers name are represented by unique numeric ids
modules/component names are represented by unique numeric ids
implicit ratings derived from the normalized usage count for modules used by Yahoo! Pipes developers among all pipes they developed

userId	moduleId	implicit rating
123	23	0.12
123	46	0.24
123	51	0.43
124	11	0.23

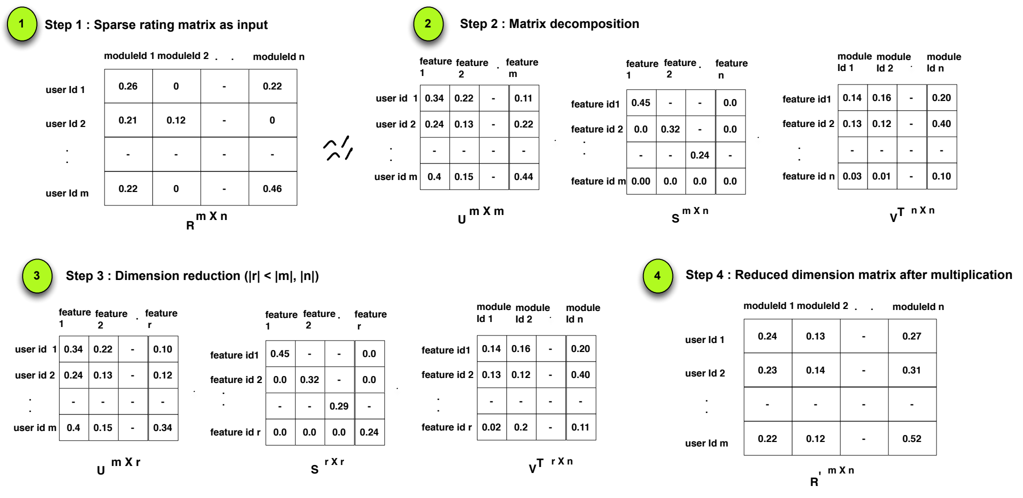
Fig. 7. Snapshot of the extracted user profile data stored in our knowledge base.

used a particular module/data source in any of his designed mashup and the rating value 1 means that the current users uses the particular module or data source in highest number of time, compared to all users who have ever used that module or data sources in their mashups.

3.3.3. User profile generation. To verify our claims on the personalized recommendation, we collected development profiles information for the users of Yahoo! Pipes. Figure 6 shows how we crawled Yahoo! Pipes existing pipes catalog to search for users' development histories. As shown in the figure, we first crawled all pipes in a given application domain e.g., Pipes that are annotated with *news* tag, then for each of the crawled news pipes we extracted the author's preference information i.e., number of modules/components and data sources they have used in their pipes by crawling their development profile page containing information about pipes they have developed. By following this procedure, totally we collected development profiles information for **441** unique Yahoo! Pipes users. The development profile for each user contains two matrices, one of which (cf. Figure 7) captures the association between the users (#441) and the modules (#51) and the other matrix contains the association between the users (#441) and the data-sources (#4781). From these association metrics we derived an implicit rating matrix for the users-modules and the users-data-sources.

This implicit rating data is found to be highly sparse (most of the users never used one or more modules/sources in any of his/her applications) in our dataset. The weakness of other collaborative filtering techniques (e.g., pearson nearest neighbor [Su and Khoshgoftaar 2009]) for large, sparse dataset led us to explore a variant of *singular value decomposition* (SVD) algorithm alternating least square (ALS) [Koren et al. 2009] a well known matrix factorization technique that efficiently handles implicit rating dataset with high sparsity. [Berry et al. 1995] point out that the reduced orthogonal dimensions resulting from SVD are *less noisy* than the original data and captures the latent associations between the items based upon their feature associations at different dimensions.

3.3.4. Singular value decomposition. Singular value decomposition (SVD) is a well-known matrix factorization technique that discovers latent features from an user-item


1
Step 1 : Sparse rating matrix as input
$R^{m X n}$
$\approx$
2
Step 2 : Matrix decomposition
$U^{m X m}$
$S^{m X n}$
$V^{T\ n X n}$
3
Step 3 : Dimension reduction (|r| < |m|, |n|)
$U^{m X r}$
$S^{r X r}$
$V^{T\ r X n}$
4
Step 4 : Reduced dimension matrix after multiplication
$R'^{\ m X n}$

	moduleId 1	moduleId 2	. .	moduleId n
user Id 1	0.26	0	-	0.22
user Id 2	0.21	0.12	-	0
:	-	-	-	-
user Id m	0.22	0	-	0.46

	feature 1	feature 2	.	feature m
user id 1	0.34	0.22	-	0.11
user id 2	0.24	0.13	-	0.22
:	-	-	-	-
user id m	0.4	0.15	-	0.44

	feature 1	feature 2	.	feature n
feature id1	0.45	-	-	0.0
feature id 2	0.0	0.32	-	0.0
:	-	-	0.24	-
feature id m	0.00	0.0	0.0	0.0

	module Id 1	module Id 2	.	module Id n
feature id1	0.14	0.16	-	0.20
feature id 2	0.13	0.12	-	0.40
:	-	-	-	-
feature id n	0.03	0.01	-	0.10

	feature 1	feature 2	.	feature r
user id 1	0.34	0.22	-	0.10
user id 2	0.24	0.13	-	0.12
:	-	-	-	-
user id m	0.4	0.15	-	0.34

	feature 1	feature 2	.	feature r
feature id1	0.45	-	-	0.0
feature id 2	0.0	0.32	-	0.0
:	-	-	0.29	-
feature id r	0.0	0.0	0.0	0.24

	module Id 1	module Id 2	.	module Id n
feature id1	0.14	0.16	-	0.20
feature id 2	0.13	0.12	-	0.40
:	-	-	-	-
feature id r	0.02	0.2	-	0.11

	moduleId 1	moduleId 2	. .	moduleId n
user Id 1	0.24	0.13	-	0.27
user Id 2	0.23	0.14	-	0.31
:	-	-	-	-
user Id m	0.22	0.12	-	0.52

Fig. 8. Example of the matrix-factorization steps in SVD algorithm.

rating matrix [Koren et al. 2009]. In its basic form, SVD characterizes both items and users by vectors of features inferred from the rating matrix. High correlation between item features and user features leads to a specific rating pattern, and during the matrix factorization SVD algorithm learns this pattern and accordingly it predicts the rating for an item for a particular user. Technically, SVD factors an $m \times n$ matrix R into three matrices:

$$\boxed{R = U \cdot S \cdot V^T} \tag{1}$$

where, U and V^T are two orthogonal matrices of size $m \times r$ and $r \times n$ respectively; r is the rank of the matrix R. In Appendix B with a help of an example we show the logic of the algorithms behind SVD. In our application scenario, if we consider matrix R as the rating matrix which captures the implicit ratings (as described in Subsection 3.3.2) of users on modules, then it can be decomposed to 3 reduced matrices in which matrix U contains the association metrics between users $\times$ features, matrix S contains the association metrics between features $\times$ features and matrix V^T contains the association metrics for feature $\times$ modules. S is a diagonal matrix of size $r \times r$ having all singular values of matrix R as its diagonal entries. All the entries of matrix S are positive and stored in decreasing order of their magnitude. By SVD calculation we aim at finding r independent feature vectors that represent the whole feature space. By using SVD's dimension reduction technique shown in Figure 8, we find a the best lower rank approximation ($R^{'}$) of the original matrix R. For example, if every user who used module1 also used module2; then the features of these two modules are linearly dependent and would contribute to a single combined feature and thus we reduce the dimension of matrix S. Figure 8 demonstrates how do we use the matrix factorization property of SVD algorithm to remove the sparsity in our user-module and user-data source ranking data. As shown in Figure 8, we start with the sparse rating matrix (Step 1), then by using the SVD algorithm we factorize the input rating matrix to produce 3 matrices U, S and V^T (Step 2). With the help of the ALS algorithm as discussed in the Section 3.3, we then do the dimension reduction steps (Step 3) and finally we calculate the dot product of the reduced matrices to produce final non-sparse rating matrices (Step 4).

In our experiment to tackle the sparsity of our implicit ratings data, we implemented the ALS variant of SVD algorithm as defined in the paper [Zhou et al. 2008]. To find the optimal values for the free parameters in the ALS algorithm we used trial and error method. The best performance is achieved for our dataset with the following settings: number of hidden features n_f= 30, λ regularization metric to prevent over fitting = 0.05f, and the number of iterations is set to 20. To calculate the accuracy for the rating prediction part of our recommendation algorithm, we used a standard accuracy measure metric, i.e., root mean square error (RMSE).

$$\boxed{RMSE = \sqrt{1/|S_{test}| \sum_{(m,u) \in S_{test}} (R_{m,u} - P_{m,u})^2}} \tag{2}$$

in which $R_{m,u}$ denotes the rating of a module m for a user u in the input rating matrix for the ALS algorithm and $P_{m,u}$ denotes the predicted rating for the same module for the same user as produced by the ALS algorithm. As a test data set S_{test} for the accuracy measure calculation, we consider those entries in the original rating matrix (before running the ALS algorithm) that have non-zero rating value i.e., for which we already know the rating. The calculated RMSE measures for the prediction of modules and data source ranking by the ALS based implementation are **0.1206** and **0.2711** respectively. RMSE values close to 0 (i.e., ¡ 0.5) determine the good quality of our rating

ALGORITHM 4: getPesonalizedRecommendations

```
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, user-module rating matrix $R'$, current user id $uid$
Result: recommendations $R = [\langle cp_i, rank_i\rangle]$ with $rank_i \geq T_{rank}$
$R$ = array();
$Patterns$ = set();
$recTypeToBeGiven$ = getSuitableRecTypes($object, action, OAR$);
foreach $recType \in recTypeToBeGiven$ do
    if $recType \in \{ParValue, Connector, DataMapping, CompCooccur\}$ then
        $Patterns$ = $Patterns \cup$ queryPatterns($object, KB, recType$) ;        // exact query
    else
        $Patterns$ = $Patterns \cup$ getSimilarPatterns($object, KB.MultiCompPattern, CompSim, T_{sim}$) ;
        // similarity search
foreach $pat \in Patterns$ do
    if $rank(pat.cp, pat.sim, pm) \geq T_{rank}$ then
        append($R, \langle pat.cp, rank(pat.cp, pat.sim, pm)\rangle$) ;        // rank, threshold, remember
foreach $r \in R$ do
    foreach $module \in r.pat.cp$ do
        $r.rank$ += getPersonalizedRank($uid, module, M$) ;    // rank a pattern based upon user ratings for its
        constituent modules
sortByRank($R$);
groupByType($R$);
truncateByGroup($R, K$);        // filter top-K patterns based upon their rank value
return $R$;
```

prediction algorithm i.e., it says that the predicted ratings, for those items for which we already knew the rating before running the algorithm, are closer to the expected ratings.

These predicted rating matrices, which already incorporate users' preference information, are then used as an input by our recommendation algorithm. Algorithm 4 shows how we incorporate this rating matrix information in the filtering of the retrieved composition patterns based upon user's development preferences. As shown in lines (12-14) in Algorithm 4, each retrieved pattern is assigned a rank based on the sum of rating values for it's constituent modules for the given user. These ratings information are retrieved from the non-sparse matrix produced by of the ALS algorithm. Patterns are then sorted based upon the assigned rank value (line 15) and finally the top-k patterns are selected from the sorted list and are recommended to the user.

3.3.5. Evaluation

— **Test setting**. 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 on SQLite (`http://www.sqlite.org`) for storing data on the client's in-browser memory. Given that SQLite does not support set data types, we serialize the representation of complex patterns $\langle C, DF, DF'\rangle$ 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 4, 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,

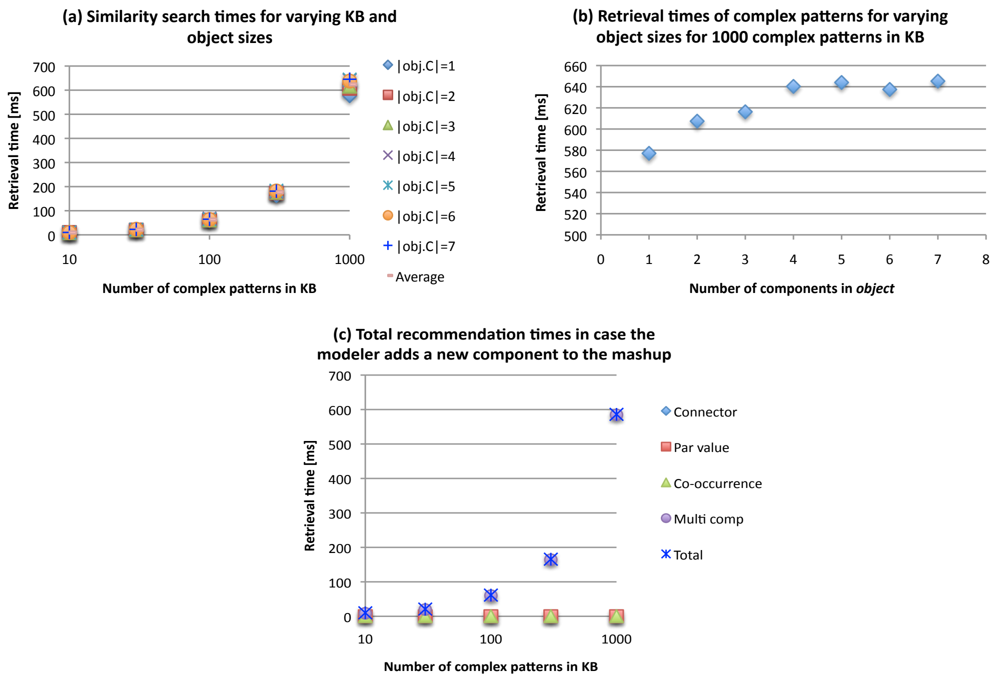


Fig. 9. Performance evaluation of the client-side knowledge recommender.

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 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 $|obj.C| \in \{1..7\}$.

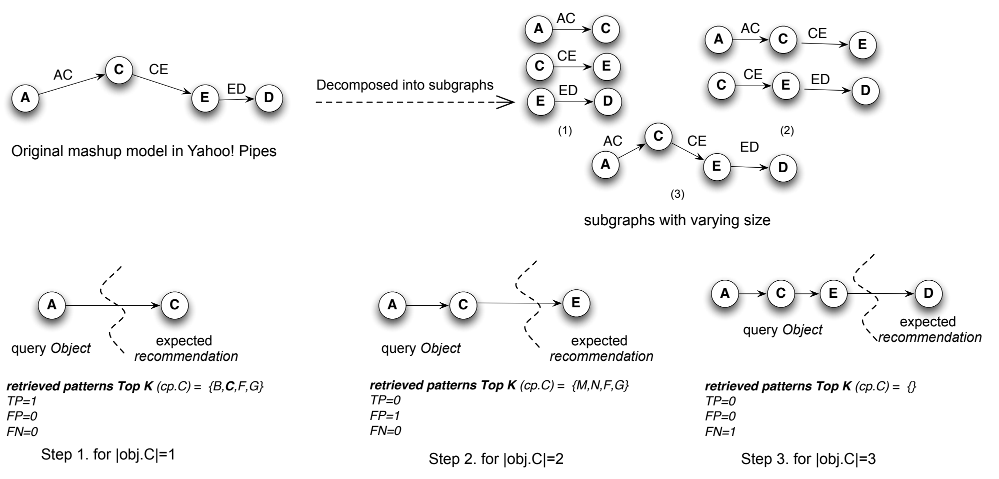
— **Performance of recommendation algorithms**. In Figure 9, we illustrate the tests we performed and the respective ***results***. The first test in Figure 9(a) studies the performance in terms of pattern retrieval times of Algorithm 2 for different *KB sizes*; the figure plots the retrieval times for 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 9(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 9(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.

— **Recommendation Accuracy**. To evaluate the accuracy of our recommendation algorithms (with and without users' development preference based filtering), we follow the widely accepted recommendation system evaluation metrics such as precision and recall metrics. To calculate these metrics we had to calculate the *true positive*, *false positive* and *false negative* metrics for our test results. If the expected modules are found in the top-k list as returned by the recommendation engine then we mark the result as *true positive* (TP), if the expected module is not found in the recommended top-k list then we mark the result as *false positive* (FP), and if algorithm doesn't return any recommendation for a step then we mark the result as *false negative* (FN). The precision, recall and the F_1 measures are calculated by using the standard formula.

$$\boxed{\begin{aligned}\text{precision} &= \frac{|TP|}{|TP| + |FP|}\\ \text{recall} &= \frac{|TP|}{|TP| + |FN|}\end{aligned}} \tag{3}$$

For our test purpose we selected 100 working pipes models, from the crawled pipes set that we used for building the user development profiles. For the test purpose we made sure that those pipes are not considered as input while building our pattern KB. The reasons for doing so were: firstly we wanted to filter the recommendations based upon the user preferences that we have already built and secondly we didn't want to recommend patterns to the same pipe that we might have used while extracting the pattern information. Doing so also verified the coverage of our pattern KB to a certain extent. The strategy that we followed to perform the evaluation test is described in Figure 10. We decomposed a subset of 100 working pipes (pipes that were not considered as input for our pattern mining algorithm) into subgraphs and then used them as query for our recommendation algorithm. Based upon the retrieved results we calculate the precision and recall measures for our algorithms. We use the same evaluation strategy for both of our recommendation algorithms, with and without preference based filtering algorithm and the results of the evaluations: varying object size and with fixed k=10 is shown in Figure 11. In all of our tests, the recommendation algorithm variant that includes user preference based filtering approach *out performs*



Original mashup model in Yahoo! Pipes
Decomposed into subgraphs
subgraphs with varying size
query Object
expected recommendation
retrieved patterns Top K (cp.C) = {B,C,F,G}
TP=1
FP=0
FN=0
Step 1. for |obj.C|=1
retrieved patterns Top K (cp.C) = {M,N,F,G}
TP=0
FP=1
FN=0
Step 2. for |obj.C|=2
retrieved patterns Top K (cp.C) = {}
TP=0
FP=0
FN=1
Step 3. for |obj.C|=3

Fig. 10. Evaluation strategy for calculating the accuracy of our recommendation algorithms

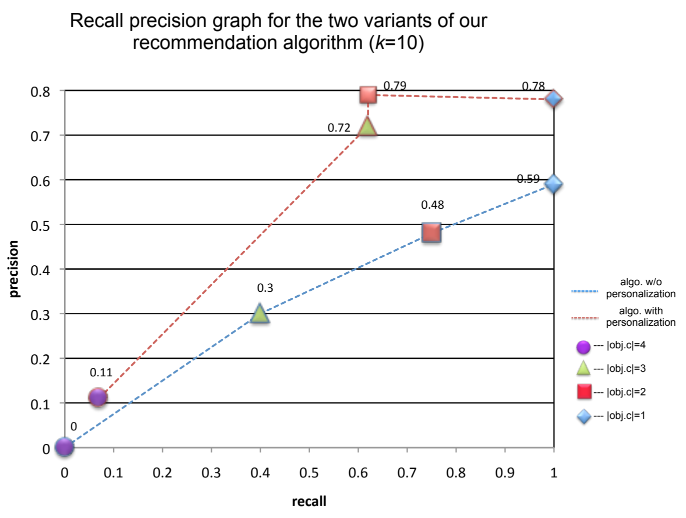


Fig. 11. Accuracy measures for the top-10 recommendations by our algorithms

the naive version. This verifies our claim of incorporating the personalization aspects in the recommendation algorithm to improve the accuracy of the overall system.
In the test result as shown in Figure 11, one may observe that the accuracy falls sharply once we go beyond object size 3, this is due to the fact that in our pattern KB we hardly have patterns with size greater than 4. This limitation can be addressed by extending the coverage of our pattern knowledge base. One may also observe that the precision and recall of both algorithms fall sharply once we go below 7 for the k value (cf. Figure 12). One may also observe in Figure 11, that the recall values calculated for both our algorithms with object size =1 and varying k, are always 1. This is due to the fact that in Yahoo! Pipes there are a limited number of modules (#51), so for

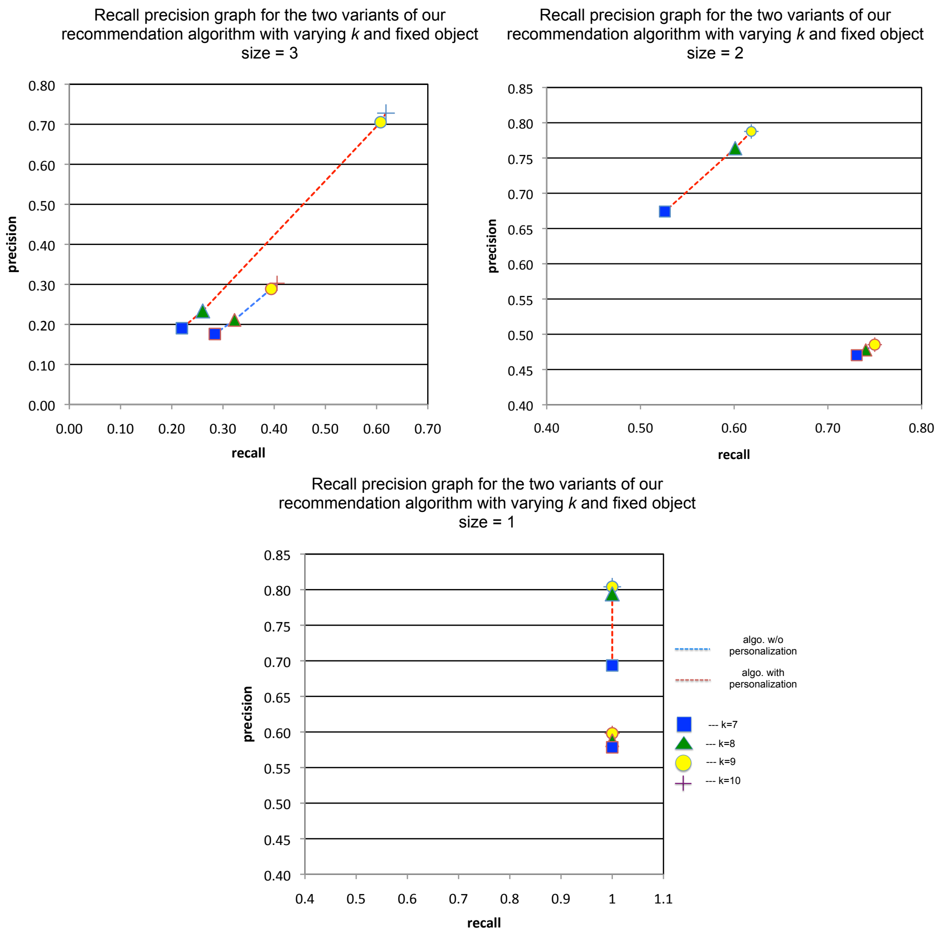


Fig. 12. Accuracy measures for our algorithms under different parameters settings

each object (each modules in this case) in the test pipes model, we have at least one pattern to be recommended in our KB.

4. AUTOMATED WEAVING OF COMPOSITION PATTERNS

We don't limit our system functionality to only recommending patterns that can be applied to the current composition context, but we also help users in progressing their development task by applying a selected pattern in the current model on behalf of the user. We call this functionality automated *weaving* of composition patterns. In order to assist the modeler in this task, we desinged pattern weaving algorithms that enable our system to take an active role in the modeling process and applying a selected pattern to the partial mashup model in the modeling canvas on behalf of the user. Doing so is not trivial at all and requires solving a set of peculiar modeling issues automatically.

Weaving a given pattern cp into a partial mashup model pm is not straightforward and it 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. We have defined the below strategies and algorithms in order to achieve the automated weaving functionality in Baya.

4.1. Basic Weaving Strategy

Given an $object$ and the pattern cp of a recommendation, the ***basic weaving strategy*** BS provides the sequence of mashup operations that are necessary to weave cp into the $object$. The basic weaving strategy tells how to expand $object$ into cp ($object$ being a part of cp) without taking into consideration pm. The basic weaving strategy is *static* for each pattern type. The key ingredient of the BS is therefore the basic mashup operations available to express the strategy. In Table I, we define the set of ***mashup operations*** that resemble the operations a developer can typically perform in the modeling canvas when designing a mashup. Mashup operations modify the partial mashup pm and produce an updated version pm'. All operations assume that 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! Pipe's mashup models. In Table II, we provide the basic strategies for the patterns introduced in Section 2.2 in the form of a function getBasicStrategy($cp, object$) $\rightarrow BS$. For instance, the basic weaving strategy for a component co-occurrence pattern of type $ptype^{comp}$ is as follows (we assume $object$=$comp_x$ with $comp_x$.type=c_x.type, where c_x is the first component of the selected pattern): To weave the pattern, a new component c_y must be added to the composition. This step is denoted by the instruction \$newcid[4]=addComponent($c_y.type$), which is followed by connecting the existing component $comp_x$with the new component (addConnector($\langle comp_x$.id, $comp_x$.op,\$newcid,c_y.ip$\rangle$)). Then, we apply the respective data mappings of c_y and value assignments for the parameters of both c_x and c_y.

4.2. 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 we decide to create a new instance of $ctype$. In the former case, we say, we apply a *soft* conflict resolution policy, in the latter case a *hard* policy. In Table III, we describe our hard conflict resolution policy for the conflicts that may arise when weaving a pattern, while the soft conflict policy can be found at Appendix A. Given an $object$, a pattern cp, and a partial mashup pm, the ***contextual weaving strategy*** WS is the sequence of mashup operations that are necessary to weave cp into pm.

The core logic of our pattern weaver is expressed in Algorithm 5. 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 $instr$. $CtxInstr$ is then applied to the current pm to compute the updated mashup model pm' (line 5), which is then used

[4]We highlight identifier place holders (variables) that can only be resolved when executing the operation with a "\$" prefix.

Table I. Data-flow based mashup operations for the definition of basic strategies

addComponent($ctype$) $\rightarrow cid'$: produces a pm' with a new component of type $ctype$ added to pm; the operation returns cid', i.e., the identifier of the newly created component.
deleteComponent(cid): produces pm' with the component identified by cid and all references to it or elements thereof (e.g., connectors with other components, data mappings) deleted from pm.
assignValues(cid, VA): produces pm' with the value assignments VA added to component cid.
deleteAllValues(cid): produces pm' with all input parameters of component cid emptied.
deleteValue(cid, in): produces pm' with the input parameter in for component cid emptied.
addConnector(df_{xy}): produces pm' with the output port op_x of the component with identifier cid_x connected to the input port in_y of the component identified by cid_y (remember $df_{xy} = \langle cid_x, op_x, cid_y, ip_y \rangle$).
deleteConnector(df_{xy}): produces pm' with data flow df_{xy} and the possible data mapping defined in the target component deleted from pm.
assignDataMappings(cid, DM): produces pm' with a data mapping DM for component cid.
deleteAllDataMappings(cid): produces pm' with data mappings deleted from component cid.
deleteDataMapping(cid, in): produces pm' with the data mapping for the input parameter in deleted from the component identified by cid.
embedComponent($hostid, embid$): produces pm' with the component with identifier $embid$ embedded in the component with identifier $hostid$.

ALGORITHM 5: 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 $pm'$
$WS$ = array();
$BS$ = getBasicStrategy($cp, object$);
foreach $instr \in BS$ do
    $CtxInstr$ = resolveConflict($pm, instr$);
    $pm$ = apply($pm, CtxInstr$);
    append($WS, CtxInstr$);
return $\langle WS, pm \rangle$;
```

as basis for weaving the next $instr$ of BS. The contextual weaving structure WS is constructed as concatenation of all conflict-free instructions $CtxInstr$. Note that Algorithm 5 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.

Example. Let's see a concrete example of how the contextual weaving strategy is built starting from the basic strategy, the conflict resolution policy, the partial mashup, and the pattern to be woven. We use the modeling situation illustrated in Figure 1. Let us assume an intermediate modeling step, in which the modeler's last modeling action was placing the *Fetch Feed* component and connecting it with the output of the *URL Builder* component. Let us further assume that our pattern recommender recommends a set of patterns in response to this modeling action. Among the recommended set of composition patterns suppose the modeler wants to accept a component co-occurrence recommendation that suggest to add a *Filter* component with the existing *Fetch Feed* component in the current composition context, similar to the scenario as shown in Figure 1. Applying this pattern to the partial model in the canvas requires: (i) adding

Table II. Function getBasicStrategy$(cp, object) \rightarrow BS$

Object	**Basic Strategy**
Parameter value pattern $ptype^{par}$	
comp with *comp.type*=*c.type*	assignValues$(comp.id, VA)$;
Connector pattern $ptype^{con}$	
$comp_x, comp_y$ with $comp_x.type$=$c_x.type$ and $comp_y.type$=$c_y.type$	addConnector$(comp_x.id, c_x.op, comp_y.id, c_y.ip)$; assignDataMappings$(comp_y.id, c_y.DM)$;
Component co-occurence pattern $ptype^{com}$	
$comp_x$ with $comp_x.type$= $c_x.type$	$\$newcid$=addComponent$(c_y.type)$; addConnector$(\langle comp_x.id, c_x.op, \$newcid, c_y.ip\rangle)$; assignDataMapping$(\$newcid, c_y.DM)$; assignValues$(comp_x.id, c_x.VA)$; assignValues$(\$newcid, c_y.VA)$;
Component embedding pattern $ptype^{emb}$	
$comp_x, comp_y, df_{xy}$ with $comp_x.type$= $c_x.type$ and $comp_y.type = c_y.type$	\$embcid=addComponent$(c_z.type)$; addConnector$(\langle comp_x.id, c_x.op, \$embcid, c_z.ip\rangle)$; addConnector$(\langle \$embcid, c_z.op, comp_y.id, c_y.ip\rangle)$; embedComponent$(comp_y.id, \$embcid)$; assignDataMappings$(comp_y.id, c_y.DM)$; assignDataMappings$(\$embcid, c_z.DM)$; assignValues$(comp_x.id, c_x.VA)$; assignValues$(comp_y.id, c_y.VA)$; assignValues$(\$embcid, c_z.VA)$;
Multi component pattern $ptype^{mul}$	
comp with $comp.type \in$ Types(C)	$\forall c_i \in (C - \{comp\})\ \$newcid[i] = addComponent(c_i.type)$; $compidx = i$ with $c_i.type$=$comp.type$; $\$newcid[compidx] = comp.id$; $\forall f^{xy} \in DF$ addConnector$(\langle \$newcid[srccid], srcop, \$newcid[tgtcid], tgtip\rangle)$; $\forall i \in \$newcid$ assignDataMappings$(\$newcid[i], c_i.DM)$; $\forall i \in \$newcid$ assignValues$(\$newcid[i], c_i.VA)$;

a new *Filter* component, (ii) connecting the *Filter* component with the output of the *Fetch Feed* component, (iii) applying the data mapping stored in the pattern to the newly created *Filter* component, (iv) resolving the conflict among the values of the *URL parameter* of the *Fetch Feed* component in the partial mashup and in the pattern, (v) assigning parameter values to the *Filter* component. While these steps can be done also manually, but our goal is to perform them automatically.

Applying Algorithm 5 to this weaving situation produces the contextual weaving strategy in Figure 13, which resembles the modeling steps described above in terms of the basic mashup operations introduced in Table I. Line 1 adds the new *Filter* component and stores the respective identifier in the variable $\$newcid$. Line 2 connects the new component to the *Fetch Feed* component. In order to do so, the pattern weaver retrieves the id of the *Fetch Feed* component from the JSON representation of the partial mashup model (in our test with Yahoo! Pipes this specifically produced the id "sw-100"; for different runs, this identifier will change) and invokes the function $addConnector$, passing the id "sw-100", the type of the output port "_OUTPUT" for *Fetch Feed*, the id of the newly created Filter component, and the type of the respective input port ("_INPUT") for Filter component to it. The output and input port types are stored in the pattern, they are replaced with their ids at runtime.

Line 3 assigns the data mappings to the *Filter* component in the form of three name-value pairs. the name identifies the input field (e.g., "conf.Rule[0].field"), while the

Table III. Hard conflict resolution policy resolveConflict($pm, instr$) $\rightarrow CtxInstr$

Basic instruction $instr$	**Conflict with** pm	**Contextual instr.** $CtxInstr$
assignValues(cid, VA);	We want to apply only the new value assignment, independently of possible existing value assignments.	deleteAllValues(cid); assignValues(cid, VA);
addConnector(df_{xy});	The connector df_{xy} already exists.	—
addConnector(df_{xy});	A connector $df_{zy} \neq df_{xy}$ from a component $comp_z$ to the same input port ip_y of df_{xy} already exists, and in_y allows only one connector in input.	deleteConnector(df_{zy}); addConnector(df_{xy});
$var=addComponent(*ctype*);	A component *comp* of type *ctype* already exists, and we don't want to reuse existing components.	$var=addComponent(*ctype*);
assignDataMappings(*cid*, *DM*)	We want to apply only the new data mapping to the component.	deleteAllDataMappings(*cid*); assignDataMappings(*cid*, *DM*);
embedComponent(*hostid*, *embid*);	A component with identifier *oldid* has already been embeded into the component *hostid*.	deleteComponent(*oldid*); embedComponent(*hostid*, *embid*);

1 $newcid=addComponent("Filter");
2 addConnector(⟨"sw-100","_OUTPUT",$newcid,"_INPUT"⟩);
3 assignDataMapping($newcid,{⟨"conf.Rule[0].field","item.description"⟩,
⟨"conf.Rule[1].field","item.description"⟩,⟨"conf.Rule[2].field","item.description"⟩});
4 deleteAllValues("sw-100");
5 assignValues("sw-100",⟨"conf.URL","url[wired]"⟩);
6 assignValues($newcid,{⟨"conf.MODE","permit"⟩,
⟨"conf.COMBINE","any"⟩,
⟨"conf.RULE[0].op","contains"⟩ ,⟨"conf.RULE[0].value","Apple"⟩,
⟨"conf.RULE[1].op","contains"⟩,⟨"conf.RULE[1].value","AppleMac"⟩,
⟨"conf.RULE[2].op","contains"⟩,⟨"conf.RULE[2].value","iPhone"⟩});

Fig. 13. Contextual weaving strategy for weaving a composition pattern

value is the data mapping (e.g., "item.description"). Both values are stored in the pattern.

Lines 4 and 5 assign the value to the *URL parameter* (identified internally via "conf.URL") of the *Filter* component. Actually, the two lines are the result of the resolution of a conflict. The *conflict resolver* therefore expands the *assignValues* function as described in Table III, first deleting the old value and then assigning the new one. Incidentally, in our example the old and the new values are the same; this is not true in general.

Finally, line 6 applies the value assignments to the *Filter* component and thus the pattern is successfully woven into the partial mashup model in the canvas.

5. IMPLEMENTATION

Figure 14 (adapted from the architecture diagram presented in our previous work [Daniel et al. 2012]) details the internals of our pattern recommender and weaving approach. We distinguish between client and server side, where the pattern mining logic and the user's development profile creation logic are located in the server and the recommendation and weaving logic resides in the client. We also distinguish between the online and offline steps involved in our system.

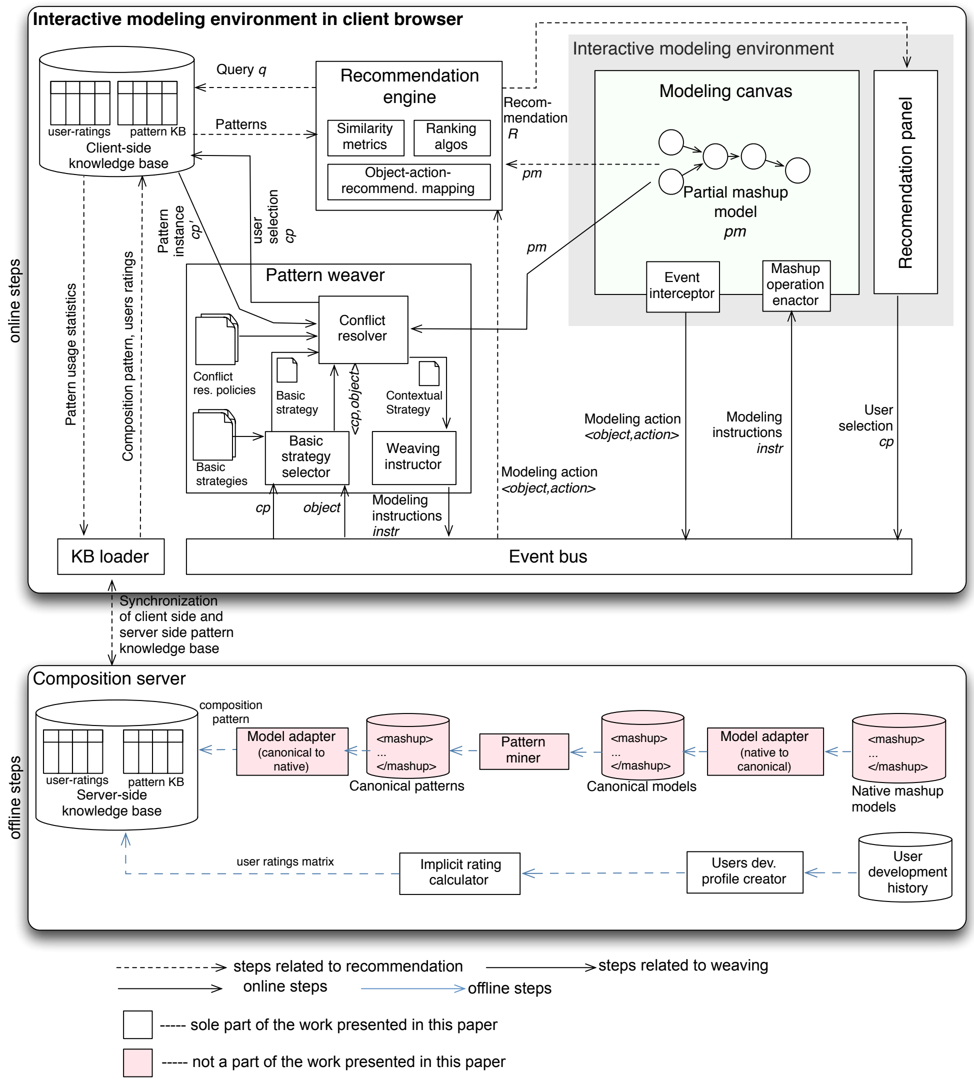


Fig. 14. Functional architecture of the interactive pattern recommendation and automated weaving approach

In the recommendation server, a model adapter imports the native mashup models into the canonical format. The pattern miner then extracts reusable composition knowledge in the form of composition patterns, which is then handed to a second model adapter to convert the canonical patterns into native patterns and load them into a knowledge base (KB). To know more about the pattern-mining algorithms, we encourage the reader to refer to the work [Rodríguez et al. 2013]. The *Development profile creator* creates the development profiles (containing information about the components and sources used by the user in his/her mashups) for users as explained in Section 3.3.2. Based on this information the *implicit rating calculator* then calculates the *preference ratings* for all components and sources for all users of the system. Implicit rating calculator implements matrix-factorization based collaborative filtering algorithm *SVD* to generate the user-component and user-source rating matrices. These matrices are then stored inside the server-side knowledge base along with the pattern KB.

The client-side pattern KB runs on an in-browser database implementation. This eliminates performance overheads of client-server communication for retrieving recommendation patterns at runtime. Client and server-side pattern KBs are synchronized once upon loading the client-side interactive recommender inside the client browser. From then on, all queries triggered by the recommendation engine to retrieve composition patterns from the KB are directed to the client-side only.

The *interactive modeling environment* runs in the client-side browser. It is here where the *pattern recommendation* logic reacts to modeling *actions* performed by the modeler on a construct (the $object$ of the action) in the canvas. For instance, we can *drop* a component onto the canvas, or we can *select* a parameter. Upon each interaction, the $action$ and its $object$ are published on a browser-internal *event bus*, which forwards them to the *recommendation engine*. With this information and the partial mashup model pm the engine queries the *client-side KB* for recommendations, where an *object-action-recommendation mapping* tells the engine which types of recommendations are to be retrieved. The list of patterns retrieved from the KB are then ranked based upon the user's preferences on the constituent components in a pattern. Finally the ranked patterns are rendered in the *recommendation panel*.

Upon the selection of a pattern from the recommendation panel, the *pattern weaver* weaves it into the partial mashup model. The *pattern weaver* first retrieves a *basic weaving strategy* (a set of modelagnostic mashup instructions) and then derives a *contextual weaving strategy* (a set of model-specific instructions), which is used to weave the pattern. Deriving the contextual strategy from the basic one may require the resolution of possible *conflicts* among the constructs of the partial model and those of the pattern to be weaved. The pattern weaver resolves them according to a configurable conflict resolution policy.

5.1. Baya for Yahoo! Pipes

We have implemented the algorithms, policies, and architectural components described in Figure 14 as a Mozilla FireFox add-on for the Yahoo! Pipes mashup environment; the tool is called Baya[5] [Roy Chowdhury et al. 2012]. We didn't want to develop *yet another* mashup environment; so we opted for an extension of an existing mashup tool (as a first step, we focused on Yahoo! Pipes; other tools will follow). In Baya we used community composition knowledge i.e., composition patterns that are mined from the exisiting Yahoo! Pipes models, as the source for the recommendation knowledge.

Baya seamlessly integrates the interactive recommendation panel with the Pipes modeling canvas, allowing the modeler to inspect recommendations and to choose which pattern to weave. The implementation is based on JavaScript for the business logic (e.g., the pattern search, retrieval, and matching algorithms and the pattern weaving algorithm) and XUL (XML User Interface Language, `https://developer.mozilla.org/En/XUL`) for the 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. Mashup operations are written in JavaScript to manipulate the mashup model's JSON representation. We save the modified version in the Pipe's server by using the HTTP Post method of Pipe's client-side environment.

[5]The Baya weaver is a weaverbird that weaves its nest with long strips of leaves.

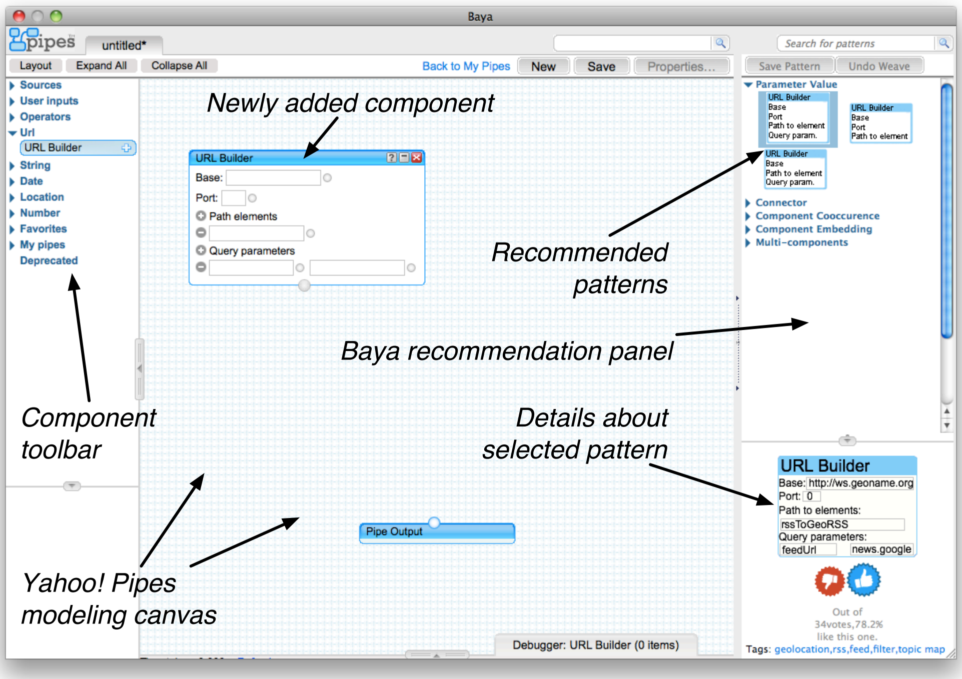
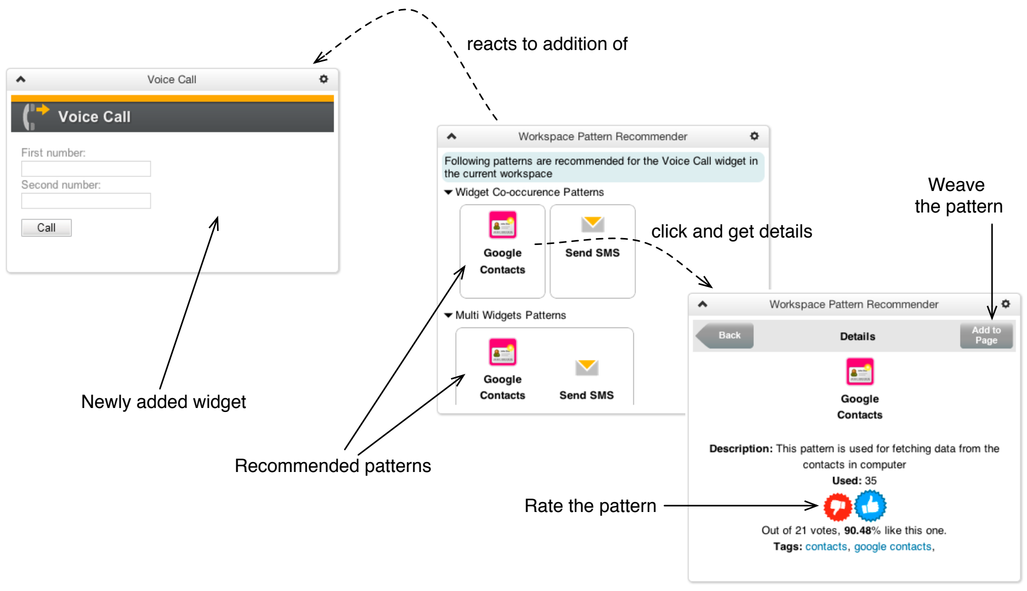


Fig. 15. Screenshot of Baya

Both the weaving strategies (basic and contextual) are encoded as JSON arrays of mashup operations, which enables 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`).

5.2. Extension of Baya for other mashup tools

In the context of the EU project OMELETTE (`http://www.ict-omelette.eu/`), we extended towards extending Baya's recommendation and weaving algorithms to support W3C widget based mashup development in and open-source mashup platform Apache Rave (`http://rave.apache.org/`). To aid users of Apache Rave in building their mashups by reusing existing composition knowledge, we developed a pattern recommender widget (cf. Figure 16). Currently, the pattern recommender widget supports two composition pattern types: ***widget co-occurence*** and ***multi widget*** patterns. For this version of Baya for Apache Rave, we used expert composition knowledge i.e., composition patterns manually provided by domain experts. Our pattern knowledge base contains compositon patterns (i.e., instances of widget co-occurence and multi widgets) that covers different application scenarios that we support in our tool. Similar to Baya, the pattern recommender reacts to user modeling *actions* (adding, deleting, or selecting a widget, etc.) on widgets (the *object* of an action) in the workspace (current mashup design). The pattern recommender widget contains recommendation and weaving algorithms implemented in JavaScript. The client-side pattern KB runs on an in-browser SQLite (`http://www.sqlite.org/`) implementation. JavaScript event listeners capture the triggering events for pattern retrieval, i. e., DOM modifications (e.g., adding a widget, deleting a widget) of the workspace model. The composition patterns are stored as



Following patterns are recommended for the Voice Call widget in the current workspace

Description: This pattern is used for fetching data from the contacts in computer

Out of 21 votes, **90.48%** like this one.

Fig. 16. Screen shot of workspace pattern recommender: an assistance utility that supports step-by-step widget mashups

JSON model in the pattern KB. The pattern recommender widget implementation extends Apache Rave's native APIs for accessing the current mashup model information and for weaving of a selected pattern into the current workspace model.

6. USER STUDIES

To asses the effectiveness of Baya, two user studies were performed during the course of this research work.

6.1. Baya in Yahoo! Pipes

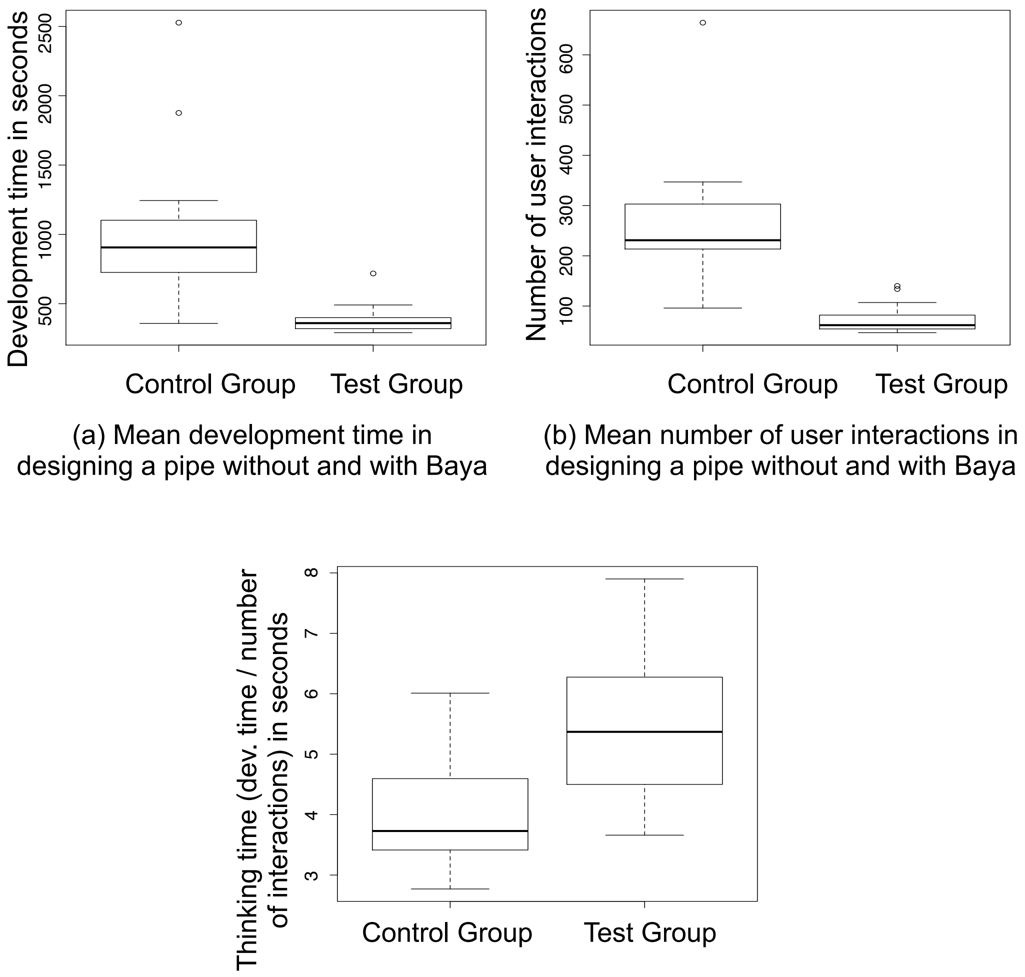
. The first *user study* was performed by us, with the help of Amazon's Mechanical Turk crowd sourcing platform (`https://www.mturk.com/mturk/welcome`). We specifically tested the following hypotheses:

HYPOTHESIS 1. (H1) *Baya speeds up development. That is, the average development time by users with Baya is lower than that by users without Baya.*

HYPOTHESIS 2. (H2) *Pipe design with Baya requires less user interactions (number of clicks) than without Baya.*

HYPOTHESIS 3. (H3) *Pipe design with Baya requires less thinking time (time between two user interactions) to take modeling decisions than without Baya.*

In order to collect data for the evaluation, we elaborated a scenario to be developed in Yahoo! Pipes and designed two different test settings: a *control group* developed the scenario without Baya, a *test group* developed the same scenario with the help of Baya. A google form replica for the questionnaire that we used for control group can be found at `http://goo.gl/9UnZW` and similarly 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 control group, the recommendation panel was disabled. After the development task, partici-



(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. 17. User study with 30 participants split into a control and a test group

pants were asked to fill an on-line questionnaire, probing their satisfaction with their development experience (we used a five-point Likert scale ranging from *strongly agree* - *strongly disagree* to collect feedback). In three days (from May 16 - 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). Each participant had to pass a qualification test with questions about Yahoo! Pipes, in order to assure that we only enrolled people with some minimum level of development knowledge and to prevent junk answers.

The data collected are illustrated in Figure 17. In order to accept or reject our hypotheses, we used Welch's t-test for equal sample sizes and unequal variance:

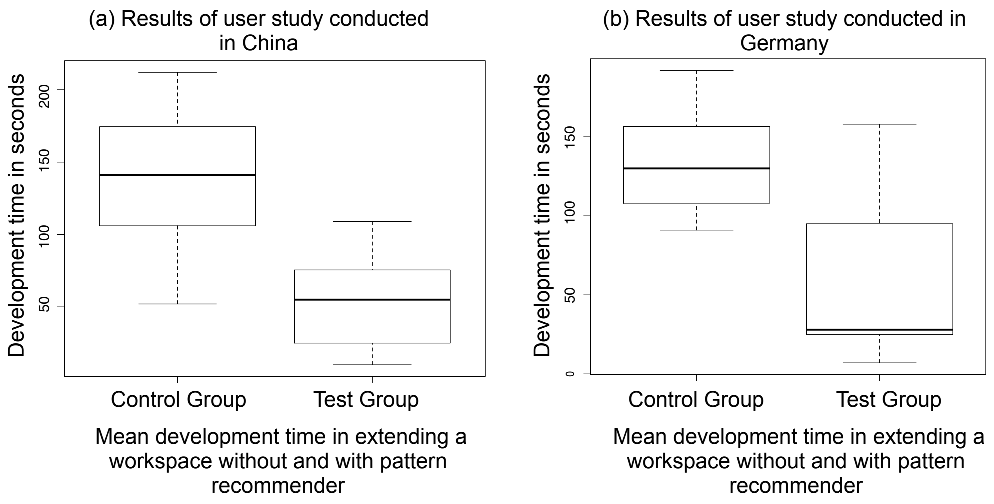


Fig. 18. Results for usability validation test executed in China and in Germany to evaluate the benefit of pattern recommender tool

— **H1**: Figure 17(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, i.e., 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 hypotheses, which proves that *Baya speeds up development*.
— **H2**: Figure 17(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, which is again a very small, and we have to reject the null hypothesis. Hence, *Baya requires less user interactions*.
— **H3**: Figure 17(c) shows the thinking times; $\mu_{th,ctrl} = 4.0s$ and $\mu_{th,test} = 5.5s$, i.e., the control group has lower thinking time (against H3). The null hypothesis is $\mu_{th,test}$ - $\mu_{th,ctrl}$ = 0. The p-value for this hypothesis was 0.00209, which is again a very small probability. Hence, *Baya increases thinking time*.

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% (27% strongly agreed and 73% agreed) agreed that assigning the right parameter values to a module took most of their design time. 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 their time, whereas 73% agreed that the automatic weaving feature saved their time (27% expressed a neutral view on that). As for the tested scenario, 80% of the control group and 73% of the test group strongly agreed that the scenario was non-trivial, a property we required to be able to collect meaningful data.

6.2. Baya in Apache Rave

. To cross-validate the result of our first user study another follow-up study was conducted with our protype system by T-Systems (`http://www.t-systems-mms.com/`) and Huwaei (`http://www.huawei.com/en/`) in the context of the OMELETTE project. The aim of that study was again to evaluate the usefulness of our assisted development approach in an end-user mashup design scenario. The study was conducted in China and in Germany and totally 22 participants took part in that study. The data collected during the study are illustrated in Figure 18. Again we used Welch's t-test for equal sample sizes and unequal variance to verify our hypotheses. In this study, however, due to organizational reasons and because the test was performed without our involvement only **H1** was tested. In this study the control and the test group were asked to extend an existing mashup (workspace) with new design requirements with and without the help of our pattern recommendation tool. This task required them to add at least two new widgets in their existing workspace in order to meet the design requirement. The development time for both the groups were collected in order to test the validity of our hypothesis.

H1: Figure 18(a) shows the collected development times, with $\mu_{dev,ctrl} = 139.8s$ and $\mu_{dev,test} = 54.6s$ for Chinese users. Figure 18(b) shows the collected development times, with $\mu_{dev,ctrl} = 132.1s$ and $\mu_{dev,test} = 58.9s$ for German users. The null hypothesis in both the tests is $\mu_{dev,test}$ - $\mu_{dev,ctrl}$ = 0, i.e., there is no significant difference between the two development times. The p-value for the null hypotheses for China and Germany tests were measured as 0.0001 and 0.0007 respectively, which is very small. Hence, we reject the null hypotheses, which reconfirms that *Baya speeds up development also in Apache Rave*. 67% of all users agreed that this feature was important or even essential for a mashup environment. Few users also reported that Baya's assistance helped them to learn the usage of new widgets, which they were unaware of. We refer our readers to this document (`http://goo.gl/JLjAu`) to read in details about this study.

These two user study results not only back our claims about the usefulness of Baya in the end-user development paradigm, but they also reveal the didactic value of our interactive step-by-step assistance mechanism.

7. RELATED WORK

In the context of *web mashups*, several works aim to assist less skilled developers in the design of mashups. Syntactic approaches [Wong and Hong] suggest modeling constructs based on syntactic similarity (comparing output and input data types), while semantic approaches [Ngu et al. 2010] annotate modeling constructs with semantic descriptions to support suggestions of composition design based on the user's specified composition goal. In programming by demonstration [Cypher et al. 1993], the system aims to auto complete a process definition, starting from a set of user-selected model examples. Goal-oriented approaches [Riabov et al. 2008] aim to assist the user by automatically deriving compositions that satisfy user-specified goals. Hybrid approaches [Elmeleegy et al. 2008] extends both the semantic and the goal based techniques. In this approach an interactive goal recommended suggests a set of composition goals to users based upon his/her current composition context. If the user accepts a system specified goal, the mashup engine auto completes the partial composition with the help of a AI planner to meet his specified goal. Pattern-based development [Deutch et al. 2010] aims at recommending connector patterns (so-called glue patterns) in response to user selected components (so-called mashlets) in order to autocomplete the partial mashup. The limitation of these approaches is that they partly over estimate the skills of less skilled developers, as they either still require advanced modeling skills (which users don't have), or they expect the user to specify complex rules for defining goals (which they are not able to), or they expect domain experts to specify and maintain complex semantic networks describing modeling constructs (which they don't do).

The business process management community more specifically focuses on patterns as a means for knowledge reuse. Among the related works for applying or weaving recommended patterns, the automated pattern application approach [Gschwind et al. 2008] uses structural properties of the current composition model to tell the user which pattern (simple merge, exclusive choice, parallel branch, and similar) among the workflow patterns introduced by Van Der Aalst et al. [Van Der Aalst et al. 2003] are applicable in the current modeling context. The structural properties of the workflow patterns are verified against the current process model structure to check their applicability. These control flow patterns are not able to capture domain knowledge of the underlying mashup applications, and hence are not contextual.

The syntax-based assistance approach proposed in [Mazanek and Minas 2009] recommends the user a set of workflow patterns based on his current process model, and, once a user selects one of the recommended patterns, weaves the pattern by considering the structural compatibility among modeling constructs (e.g., a gateway must be followed by an activity in the current composition). However, this approach is limited to only block-structured models, and also the instance level information of a composition model (e.g., an activity of type A must be followed by an activity of type B, and so on) is not captured in the recommended patterns.

To address the limitations as identified in the existing related works, in our research, we designed a more generic knowledge reuse approach, that can also be applicable to design models that are not strictly block-structured. In addition to the structural compatibility, we also considered the underlying mashup language and application domain (data flow based mashup) while capturing the knowledge to be reused for the assistance. Further in our research we also addressed challenges related to the automated application of a selected recommendation in the current composition context, in a way by auto-completing mashup design steps on behalf of a user. However, unlike existing auto-completion approaches that work at the application level, our automated weaving approach operates at different granularity levels of a model (e.g., adding a component, adding a connector or adding a multi components etc.), also keeping the current composition context into consideration. This provides more flexibility to a user in designing a mashup according to his intent.

8. CONCLUSION AND FUTURE WORK

This paper presents our work in supporting the reuse of pattern-based composition knowledge for mashup development. Our study provides the theoretical foundation of the assisted mashup development and introduces an efficient mechanism (interactive recommendation of composition patterns and automated weaving of composition patterns) to provide contextual development assistance to less skilled users. We designed and implemented two research prototype tools that realize our interactive development recommendation approach for two different mashup platforms. We performed thorough performance and accuracy evaluation tests to demonstrate the efficiency of our recommendation algorithms. We also reported the results of two evaluation tests, which were performed to assess the value of our approach for two different mashup tools and target users. The results of the tests confirm the applicability and the usefulness of our approach in the mashup development domain.

Some of the limitations that we have identified and that we will be addressing in our future work are:

— **Coverage of the pattern KB**. As reported in Figure 11, the accuracy of our recommendation algorithm falls down sharply once we go beyond the query object size of 3. This is due to the low coverage of our pattern KB for patterns with size more than 4. This is partly because while building the knowledge base for the initial prototype system, we only considered a subset of pipes (tagged as *"most popular"* pipes) as an input for the pattern mining algorithm, that we considered as the source for the composition pattern. The high support value required for identifying multi-component

patterns with many components (more than 4) by the pattern mining algorithm also contribute to this low coverage problem.
To address the pattern KB's coverage issue, in our future work we are planning to crawl the *entire mashup repository* of Yahoo! Pipes, which has several thousands of pipes models available. We also wish to explore other sources for composition pattern knowledge (e.g., knowledge contributed by domain experts) in the future design of our system. As a step towards that, we are working towards defining algorithms for finding expert users in a development platform like Yahoo! Pipes. By analyzing these expert users' mashup designs we can further identify knowledge for our recommendation algorithm. This process is not straight-forward and has it's own research challenges. In our future work we are going to address them one-by-one. We also want to cross-verify the effect of different sources of composition knowledge (e.g., knowledge from the expert users vs knowledge mined from the repository) on the quality of recommendations.

— **Usability issues for the consumption of recommendations**. Accepting/rejecting a recommendation as suggested by any recommendation system requires users to understand the recommended pattern (details of what is recommended and why it is recommended). In our tool we show users the details of a recommended patterns in the form of a preview of a pattern and the associated meta data (e.g., how many users used a pattern or liked a pattern). However, we believe, that to enhance the usability and transparency of our recommendation approach, more work is required in terms of representation of recommendations in the UI.
As a solution for this limitation, more work is required to understand the most user friendly and intuitive representation for composition patterns inside the UI. However, this is a highly tool-specific issue and, hence, requires different solutions for different mashup environments. In addition to this, we are also working on improving the usability of our recommendation system by providing more *explanations* (why a pattern is suggested, and what would a pattern do in the current context) about the suggested recommendations. We believe that an explanation along with each recommended steps will increase the transparency of our recommendation system to users.

— **Limiting the novelty of recommendations**. Personalization or preference-based filtering of recommendations certainly increases the relevance of recommendations to a user, but it also limits the novelty of recommendations (only preferred modules/data sources always appear in the top-k recommended list). This may limit the usefulness and the didactic aspects of a recommendation system.
To address this issue, we are working on techniques to introduce *diversity* into recommendations. We hypothesize that a right balance of novelty and personalization in recommendations can make our recommendation system more valuable to users.

— **Scalability of the personalized recommendation algorithm**. New users (for whom the system doesn't have the development preference information) of our assisted development platform poses challenges to the personalized recommendation algorithm. Since the system has no knowledge about the ratings for modules and data-sources for such users, it can't calculate the preference metrics for them. In order to get their development preference information, the system is required to re-execute the implicit-rating matrices calculation for all users. Re-doing this matrix factorization step for a large dataset is an expensive process, and may hinder the overall performance of our recommendation system.
To address the scalability issue of our personalized recommendation algorithm, in our future work we will explore the applicability of incremental singular value decomposition algorithm [Sarwar et al. 2002] and other state-of-the-art collaborative filtering techniques in the context of our recommendation algorithm.

ACKNOWLEDGMENTS

We thank Carlos Rodríguez for providing the access to mashup patterns that are mined from Yahoo! Pipes mashup models. We thank the T-Systems MMS team and Huawei team and especially Matthias Niederhausen and Stefan Pietschmann of T-Systems team for carrying out the user study of Baya in Apache Rave and making the result available to us for the analysis. We thank Saravanan Thirumuruganathan for his extensive help during the data collection for user development-profiles building steps. This work was supported by the European Commission (project OMELETTE, contract 257635).

REFERENCES

BERRY, M. W., DUMAIS, S., O'BRIEN, G., BERRY, M. W., DUMAIS, S. T., AND GAVIN. 1995. Using linear algebra for intelligent information retrieval. *SIAM Review 37*, 573–595.

CYPHER, A., HALBERT, D. C., KURLANDER, D., LIEBERMAN, H., MAULSBY, D., MYERS, B. A., AND TURRANSKY, A., Eds. 1993. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA.

DANIEL, F., CASATI, F., BENATALLAH, B., AND SHAN, M.-C. 2009. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In *ER'09*. Springer, 428–443.

DANIEL, F., RODRIGUEZ, C., ROY CHOWDHURY, S., MOTAHARI NEZHAD, H. R., AND CASATI, F. 2012. Discovery and reuse of composition knowledge for assisted mashup development. In *Proceedings of the 21st international conference companion on World Wide Web*. WWW '12 Companion. ACM, New York, NY, USA, 493–494.

DE ANGELI, A., BATTOCCHI, A., CHOWDHURY, S. R., RODRÍGUEZ, C., DANIEL, F., AND CASATI, F. 2011. End-user requirements for wisdom-aware eud. In *IS-EUD*. 245–250.

DEUTCH, D., GREENSHPAN, O., AND MILO, T. 2010. Navigating in complex mashed-up applications. *Proc. VLDB Endow. 3,* 1-2, 320–329.

ELMELEEGY, H., IVAN, A., AKKIRAJU, R., AND GOODWIN, R. 2008. Mashup advisor: A recommendation tool for mashup development. In *ICWS'08*. IEEE Computer Society, 337–344.

GSCHWIND, T., KOEHLER, J., AND WONG, J. 2008. Applying patterns during business process modeling. In *BPM'08*. Springer, 4–19.

HU, Y., KOREN, Y., AND VOLINSKY, C. 2008. Collaborative filtering for implicit feedback datasets. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*. ICDM '08. IEEE Computer Society, Washington, DC, USA, 263–272.

KOREN, Y., BELL, R., AND VOLINSKY, C. 2009. Matrix factorization techniques for recommender systems. *Computer 42,* 8, 30–37.

MAZANEK, S. AND MINAS, M. 2009. Business Process Models as a Showcase for Syntax-Based Assistance in Diagram Editors. In *MODELS '09*. 322–336.

NGU, A., CARLSON, M., SHENG, Q., AND YOUNG PAIK, H. 2010. Semantic-based mashup of composite applications. *IEEE TSC 3,* 1, 2 –15.

RIABOV, A. V., BOILLET, E., FEBLOWITZ, M. D., LIU, Z., AND RANGANATHAN, A. 2008. Wishful search: interactive composition of data mashups. In *WWW'08*. ACM, 775–784.

RODRÍGUEZ, C., ROY CHOWDHURY, S., DANIEL, F., R. MOTAHARI NEZHAD, H., AND CASATI, F. 2013. *Assisted Mashup Development: On the Discovery and Recommendation of Mashup Composition Knowledge – In Press*. Springer, 683–708.

ROY CHOWDHURY, S., BIRUKOU, A., DANIEL, F., AND CASATI, F. 2011. Composition patterns in data flow based mashups. In *Proceedings of EuroPLoP 2011*. 27–28.

ROY CHOWDHURY, S., CHUDNOVSKYY, O., NIEDERHAUSEN, M., PIETSCHMANN, S., SHARPLES, P., DANIEL, F., AND GAEDKE, M. 2013. Complementary assistance mechanisms for end user mashup composition– submitted. In *WWW ' 13*. WWW '13 Companion.

ROY CHOWDHURY, S., DANIEL, F., AND CASATI, F. 2011. Efficient, Interactive Recommendation of Mashup Composition Knowledge. In *ICSOC'11*. 374–388.

ROY CHOWDHURY, S., RODRÍGUEZ, C., DANIEL, F., AND CASATI, F. 2010. Wisdom-aware computing: On the interactive recommendation of composition knowledge. In *WESOA'10*. Springer, 144–155.

ROY CHOWDHURY, S., RODRÍGUEZ, C., DANIEL, F., AND CASATI, F. 2012. Baya: assisted mashup development as a service. In *Proceedings of the 21st international conference companion on World Wide Web*. WWW '12 Companion. ACM, New York, NY, USA, 409–412.

SARWAR, B., KARYPIS, G., KONSTAN, J., AND RIEDL, J. 2002. Incremental singular value decomposition algorithms for highly scalable recommender systems. In *Fifth International Conference on Computer and Information Science*. 27–28.

SARWAR, B. M., KARYPIS, G., KONSTAN, J. A., AND RIEDL, J. T. 2000. Application of dimensionality reduction in recommender system – a case study. In *IN ACM WEBKDD WORKSHOP*.

SHANI, G. AND GUNAWARDANA, A. 2011. Evaluating recommendation systems. In *Recommender Systems Handbook*. 257–297.

SU, X. AND KHOSHGOFTAAR, T. M. 2009. A survey of collaborative filtering techniques. *Adv. in Artif. Intell. 2009*, 4:2–4:2.

VAN DER AALST, W. M. P., TER HOFSTEDE, A. H. M., KIEPUSZEWSKI, B., AND BARROS, A. P. 2003. Workflow patterns. *Distrib. Parallel Databases 14*, 5–51.

WONG, J. AND HONG, J. I. Making mashups with marmite: towards end-user programming for the web. In *CHI'07*. 1435–1444.

ZHOU, Y., WILKINSON, D., SCHREIBER, R., AND PAN, R. 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-Verlag, Berlin, Heidelberg, 337–348.

Online Appendix to: Interactive Recommendation and Weaving of Mashup Model Patterns for Assisted Mashup Development

Soudip Roy Chowdhury, University of Trento, Italy
Florian Daniel, University of Trento, Italy
Fabio Casati, University of Trento, Italy

A. SOFT CONFLICT RESOLUTION POLICY

During weaving steps, we are in the presence of a *conflict* if we want to add a new construct to the partial mashup model pm, but the partial mashup model already contains the same or a similar construct. Unlike *hard* conflict resolution policy which resolves the conflict by re-assigning the constructs with the new values from the patterns, *soft* conflict resolution policy aims to maximize reuse.

Table IV. Soft conflict resolution policy for the function resolveConflict($pm, instr$) $\rightarrow CtxInstr$

Instruction $instr$	**Conflict with** pm	**Contextual instructions** $CtxInstr$	Description
assignValues(cid, VA);	We want to preserve possible value assignments, if they are not in conflict with any of the values in VA.	assignValues(cid, VA);	We keep existing parameter values. Yet, new parameter values prevail over old ones, and the assignValue() operation simply overwrites the old value assignments.
addConnector(df_{xy});	The connector df_{xy} already exists.	—	No need to add the connector again.
addConnector(df_{xy});	A connector $df_{zy} \neq df_{xy}$ from a component $comp_z$ to the same input port ip_y of df_{xy} already exists, and in_y allows only one connector in input.	deleteConnector(df_{zy}); addConnector(df_{xy});	Before adding the new connector, we delete the old one.
\$var[1]=addComponent(*ctype*);	A component *comp* of type *ctype* already exists, and we want to reuse existing components.	**\$var**=$comp.id$;	We reuse the existing instance of the required component type, without creating a new one.
assignDataMappings(cid, DM)	We want to preserve possible data mappings data are not in conflict with the data mappings in DM.	assignDataMappings(cid, DM);	We keep existing data mappings and apply the ones in DM. New data mappings overwrite old ones, preserving those without conflict.
embedComponent($hostid$, *embid*);	A component with identifier *oldid* has already been embedded into the component $hostid$.	deleteComponent(*oldid*); embedComponent($hostid$, *embid*);	Before embedding the new component, we need to delete the already embedded component.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

[1] We use the notation **$var** to denote placeholders for variables, in order to keep the policy independent of variable names, such as **newcid** in the above basic weaving strategy.

B. SINGULAR VALUE DECOMPOSITION- A RUNNING EXAMPLE

SVD is based on a theorem from linear algebra which says that a rectangular matrix R can be broken down into the product of three matrices - an orthogonal matrix U , a diagonal matrix S, and the transpose of an orthogonal matrix V . i.e.,

$R = \mathrm{U} \cdot \mathrm{S} \cdot V^T$

where, $U^T U = I$, $V^T V = I$; the columns of U are orthonormal eigenvectors of RR^T, the columns of V are orthonormal eigenvectors of $R^T R$, and S is a diagonal matrix containing the square roots of eigenvalues from U or V in descending order.

To explain the steps involved in the SVD matrix factorization technique, let us start with a small rating matrix R.

$$R = \begin{bmatrix} 0.2 & 0.4 & 0 \\ 0.1 & 0 & 0.5 \\ 0.2 & 0.3 & 0.6 \end{bmatrix}, \text{ when } R^T = \begin{bmatrix} 0.2 & 0.1 & 0.2 \\ 0.4 & 0 & 0.3 \\ 0 & 0.5 & 0.6 \end{bmatrix}.$$

As one may observe that the matrix R has lot of zero values (sparse matrix) in it. The main motivation behind applying the SVD algorithm is to reduce the sparsity of the original matrix R; by factorizing it to sub-matrices and by reducing its dimension to get a better lower rank approximation of the original matrix R. The dot product of R and R^T is

$$RR^T = \begin{bmatrix} 0.2 & 0.02 & 0.16 \\ 0.02 & 0.26 & 0.32 \\ 0.16 & 0.32 & 0.49 \end{bmatrix}$$

The eigen values of RR^T are $\lambda = 0.756$, $\lambda = 0.196$ and $\lambda = 0.001$. The column vectors of U are taken from the orthonormal eigenvectors of RR^T, and ordered right to left from largest corresponding eigenvalue to the least. And hence the value for U matrix is calculated as,

$$U = \begin{bmatrix} -0.253 & 0.879 & -0.405 \\ -0.534 & -0.476 & -0.699 \\ -0.806 & 0.040 & 0.590 \end{bmatrix}$$

To calculate the value for the V^T matrix, we follow to same procedure as for the calculation of U, i.e., we do a dot product of matrices $R^T R$ and then we find the orthonormal eigenvectors for RR^T. Finally we transpose this eigenvector to get the value for the V^T matrix. The calculation steps are explained below.

$$R^T R = \begin{bmatrix} 0.09 & 0.14 & 0.17 \\ 0.14 & 0.25 & 0.18 \\ 0.17 & 0.18 & 0.61 \end{bmatrix}$$, where the eigen value for $R^T R$ are calculated as $\lambda = 0.756$, $\lambda = 0.001$ and $\lambda = 0.196$. The orthonormal eigen vector V is:

$$V = \begin{bmatrix} -0.306 & -0.901 & -0.307 \\ -0.396 & 0.414 & -0.820 \\ -0.866 & 0.129 & 0.483 \end{bmatrix}$$

and the transpose matrix is:

$$V^T = \begin{bmatrix} -0.306 & -0.396 & -0.866 \\ -0.901 & 0.414 & 0.129 \\ -0.307 & -0.820 & 0.483 \end{bmatrix}$$

For S matrix value, we take the square roots of the non-zero eigenvalues and populate the diagonal with them, putting the largest in s_{11} , the next largest in s_{22} and so on until the smallest value ends up in s_{mm} . The non-zero eigenvalues of U and V are always the same, so thats why it doesnt matter which one we take them from. By following this process we can get the value of singular value matrix S as:

$$S = \begin{bmatrix} 0.867 & 0 & 0 \\ 0 & 0.443 & 0 \\ 0 & 0 & 0.036 \end{bmatrix}$$

In order to illustrate the effect of dimensionality reduction on this data set, we'll restrict S to the first two singular values:

$$S' = \begin{bmatrix} 0.867 & 0 \\ 0 & 0.443 \end{bmatrix}$$

For the matrix multiplication in the reduced dimension, we have to eliminate the corresponding column vectors of U and corresponding row vectors of V^T to give us an approximation of R using 2 dimensions instead of the original 3. The result of this dimension reduction step is: $R' = U' \cdot S' \cdot V^{T'}$

$$= \begin{bmatrix} -0.253 & 0.879 \\ -0.534 & -0.476 \\ -0.806 & 0.040 \end{bmatrix} \cdot \begin{bmatrix} 0.867 & 0 \\ 0 & 0.443 \end{bmatrix} \cdot \begin{bmatrix} -0.306 & -0.396 & -0.866 \\ -0.901 & 0.414 & 0.129 \end{bmatrix}$$

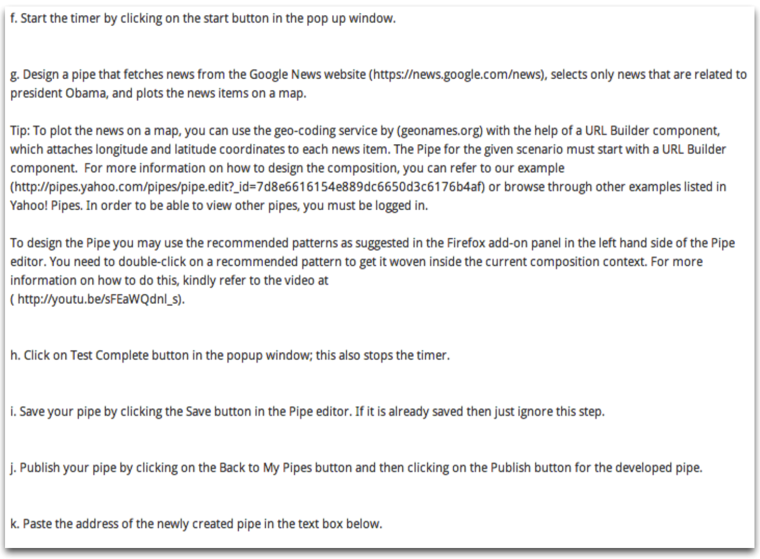
$$= \begin{bmatrix} 0.283 & 0.248 & 0.240 \\ 0.332 & 0.096 & 0.374 \\ 0.198 & 0.284 & 0.608 \end{bmatrix}$$

One can notice that in the resultant matrix R' the sparsity is reduced for the lower rank approximation. Hence by this process we get the best lower rank approximation of the original sparse rating matrix R.

C. QUESTIONNAIRE FOR THE USER STUDY

Figure 19 depicts the snapshot of the tasks that we had asked our test group participants in Mechanical Turk to perform. One may observe that the test group participants are asked to download our assistance tool Baya and they had given the option to use Baya's interactive recommendations in their designs. The control group participants were also asked to download Baya, but in that version of Baya we de-activated the recommendation feature.

Figure 20 shows a snapshot of the questionnaire form that we used to collect data from the users during our study. To collect the feedback from users we used such questionnaire form containing questions and corresponding multiple choice answers to capture users' views and criticism on our tool (for the answers we used a five-point Likert scale ranging from *strongly agree* - *strongly disagree*) and a free-form text field for collecting more detailed feedback from users.



f. Start the timer by clicking on the start button in the pop up window.

g. Design a pipe that fetches news from the Google News website (https://news.google.com/news), selects only news that are related to president Obama, and plots the news items on a map.

Tip: To plot the news on a map, you can use the geo-coding service by (geonames.org) with the help of a URL Builder component, which attaches longitude and latitude coordinates to each news item. The Pipe for the given scenario must start with a URL Builder component. For more information on how to design the composition, you can refer to our example (http://pipes.yahoo.com/pipes/pipe.edit?_id=7d8e6616154e889dc6650d3c6176b4af) or browse through other examples listed in Yahoo! Pipes. In order to be able to view other pipes, you must be logged in.

To design the Pipe you may use the recommended patterns as suggested in the Firefox add-on panel in the left hand side of the Pipe editor. You need to double-click on a recommended pattern to get it woven inside the current composition context. For more information on how to do this, kindly refer to the video at (http://youtu.be/sFEaWQdnl_s).

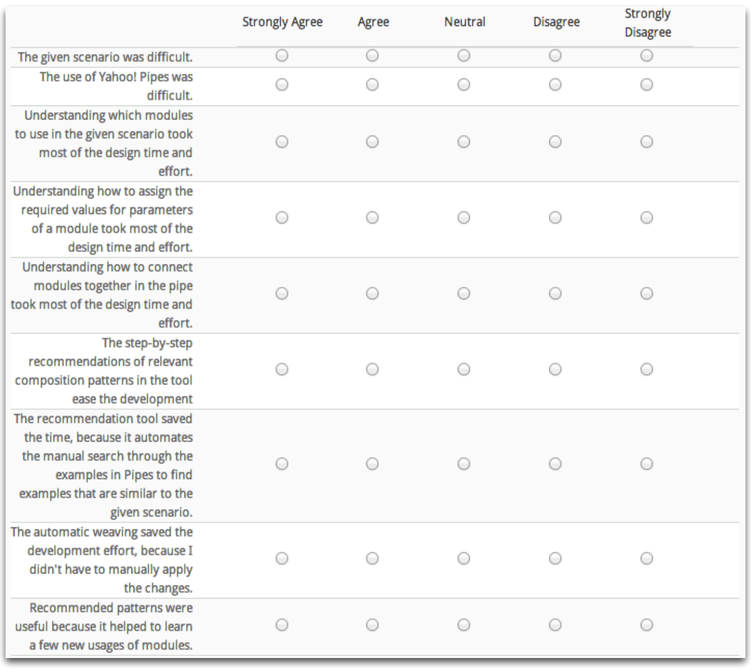
h. Click on Test Complete button in the popup window; this also stops the timer.

i. Save your pipe by clicking the Save button in the Pipe editor. If it is already saved then just ignore this step.

j. Publish your pipe by clicking on the Back to My Pipes button and then clicking on the Publish button for the developed pipe.

k. Paste the address of the newly created pipe in the text box below.

Fig. 19. Screen shot capturing a snapshot of the task details that we asked our users (test group) to perform



	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
The given scenario was difficult.	○	○	○	○	○
The use of Yahoo! Pipes was difficult.	○	○	○	○	○
Understanding which modules to use in the given scenario took most of the design time and effort.	○	○	○	○	○
Understanding how to assign the required values for parameters of a module took most of the design time and effort.	○	○	○	○	○
Understanding how to connect modules together in the pipe took most of the design time and effort.	○	○	○	○	○
The step-by-step recommendations of relevant composition patterns in the tool ease the development	○	○	○	○	○
The recommendation tool saved the time, because it automates the manual search through the examples in Pipes to find examples that are similar to the given scenario.	○	○	○	○	○
The automatic weaving saved the development effort, because I didn't have to manually apply the changes.	○	○	○	○	○
Recommended patterns were useful because it helped to learn a few new usages of modules.	○	○	○	○	○

Fig. 20. Screen shot capturing a snapshot of questionnaire that we used in our user study