Discovery and Reuse of Composition Knowledge for Assisted Mashup Development

Florian Daniel
University of Trento
Via Sommarive 5
38123 Povo (TN), Italy
daniel@disi.unitn.it

Carlos Rodríguez
University of Trento
Via Sommarive 5
38123 Povo (TN), Italy
crodriguez@disi.unitn.it

Soudip Roy Chowdhury
University of Trento
Via Sommarive 5
38123 Povo (TN), Italy
rchowdhury@disi.unitn.it

Hamid R. Motahari Nezhad
Hewlett Packard Labs
Palo Alto, USA
hamid.motahari@hp.com

Fabio Casati
University of Trento
Via Sommarive 5
38123 Povo (TN), Italy
casati@disi.unitn.it

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 – if sensibly structured – even complex recommendations can be efficiently queried and weaved also inside the client browser.

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

Keywords

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

Copyright is held by the author/owner(s).

WWW2012, April 16-20, 2012, Lyon, France.
.

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.

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.

For instance, Figure 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

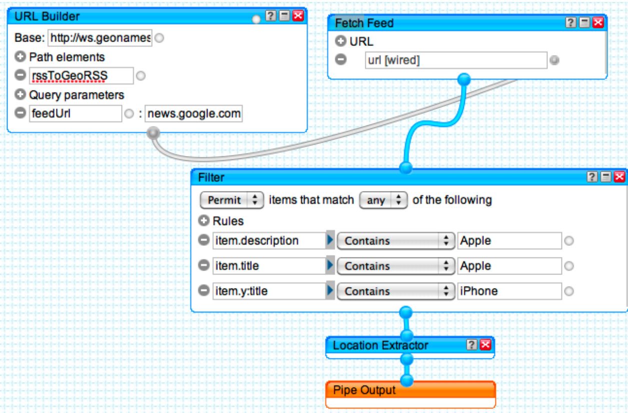

Figure 1: A typical pattern in Yahoo! Pipes

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. Such knowledge may come from a variety of possible sources. In this paper, 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/composition 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 paper, 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 paper, 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.
- We describe an *architecture* of our knowledge recommender that can be used to equip any mashup environment with interactive assistance for its developers.
- We define a set of *mashup pattern types* that resemble the modeling actions of typical mashup environments.
- 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 develop a *pattern weaving algorithm* that automatically applies patterns to mashup models, allowing the developer to progress in his development task.

In the next section, we introduce the canonical mashup model and formulate our problem statement. In Section 3 we describe the architecture of our recommendation platform and characterize the patterns we are interested in. In Sections 4, 5 and 6 we, respectively, describe the mining, recommendation, and weaving algorithms. In Section 7 we overview related work. Then we conclude the paper.

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 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 paper 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.

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., $IP = \varnothing$.
- *Data processing* components consume data in input and produce processed data in output. Therefore: IP, $OP \neq \varnothing$. Filter components, operators, and data transformers are examples of data processing components.

[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.

- *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., $OP = \varnothing$.

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, $src \in \{$"*Pipes*", "*Wires*", "*myCocktail*", ...$\}$ keeps track of the source platform of 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, $E_k = \varnothing$.

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

[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*.

Without loss of generality, throughout this paper 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). Hence, $GP, RES = \varnothing$ and a pipe corresponds to a restricted canonical mashup of the form $m = \langle name, id, \text{"}Pipes\text{"}, C, \varnothing, DF, \varnothing\rangle$ 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.

2.2 Problem Statement

Given the above canonical mashup model, the problem we want to solve in this paper is understanding (i) which *kind of knowledge* can be extracted from canonical mashup models, (ii) how to *develop mining algorithms* that are able to discover such knowledge, (iii) how to *recommend discovered patterns* for reuse in concrete mashup tools, and (iv) how to *weave patterns* into a partial mashup under development.

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.

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. 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:

$ptype^{par} = \langle \{c\}, GP, \varnothing, \varnothing \rangle$;
$c = \langle name, 0, type, \varnothing, IN, \varnothing, \varnothing, VA, \varnothing, \varnothing \rangle$[4];
$GP \neq \varnothing$ if VA also assigns global parameters to IN;
$GP = \varnothing$ 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:

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

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:

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

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:

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

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:

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

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, that are not yet captured by the other pattern types. 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.

3.2 Architecture

Figure 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 server, a *model adapter* imports the native mashup models into the canonical format. The *pattern miner* then extracts reusable composition knowledge, which is loaded by the *data transformer* into a knowledge base (KB) that 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

[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.

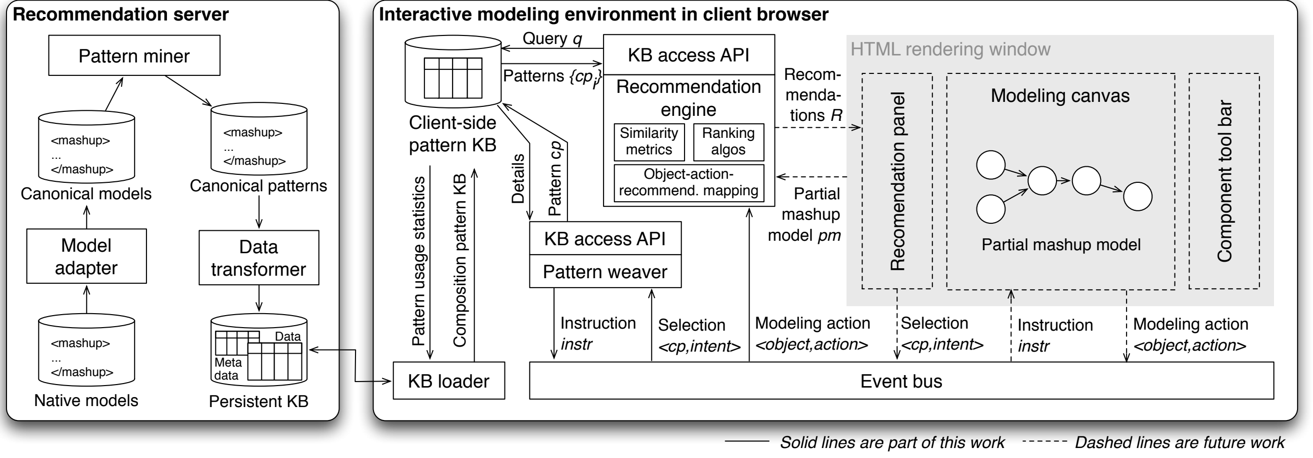

Figure 2: Functional architecture of the composition knowledge discovery and recommendation approach

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.

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. We do not simply interpret mashups as graphs and apply standard graph mining algorithms [12]. Instead, we prefer looking better into the details of each individual pattern and implementing dedicated mining algorithms for each pattern, 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 algorithms make use of standard statistics and frequent itemset mining [12].

4.1 Mining algorithms

For each of the pattern types identified in Section 3.1, we have implemented a respective pattern mining algorithm, the details of which we provide in `http://goo.gl/Dis5V`. Unfortunately – for space reasons – we are not able to discuss all algorithms here. In the following we therefore specifically focus on two pattern types, i.e., connector patterns and component co-occurrence patterns, which however allow us to exhaustively explain our approach and the logic of the algorithms.

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 1 explains 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 2; 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 3) 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 [12], 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 3.1; the rest of the pattern comes from the component definitions.

Component co-occurrence pattern. The component co-occurrence pattern is an extension of the connector pattern; in addition to the connector and data mapping, it also contains the parameter value assignments of the two components involved in the connector. As shown in Algorithm 4, the respective mining logic is similar to 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 filter out those combinations of VA_x, VA_y and DM_y that co-occur for the given connector and that have a support greater than $minsupp_{df}$.

Algorithm 1: 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. $\quad DB[df_{xy}]$ = getConnectorInstances(M, df_{xy});
   `// create database for frequent itemset mining`
6. $\quad DBDM_y$ = array():
7. $\quad$ **foreach** $dfi_{xy} \in DB[df_{xy}]$ **do**
8. $\quad\quad c_y$ = target component of dfi_{xy};
9. $\quad\quad$ append($DBDM_y$, $c_y.DM$);
10. $\quad FI_{dy}$ = mineFrequentItemsets($DBDM_y$, $minsupp_{dm}$);
    `// construct the connector patterns`
11. $\quad$ **foreach** $DM_y \in FI_{dy}$ **do**
12. $\quad\quad Patterns = Patterns \cup \{\langle df_{xy}, DM_y\rangle\}$
13. **return** $Patterns$;

Algorithm 2: 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. $\quad$ 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. $\quad$ **if** *computeSupport*(df_{xy}, DB_{df}) $\geq minsupp_{df}$ **then**
7. $\quad\quad F_{df} = F_{df} \cup \{df_{xy}\}$;
8. **return** F_{df};

4.2 Implementation and Evaluation

We implemented a *model adapter* (see Figure 2) in Java (1.6), which is able to convert Yahoo! Pipes's JSON representation into our canonical mashup model. All the mining algorithms are also implemented in Java. For the frequent itemset mining we used ARMiner (`http://www.cs.umb.edu/~laur/ARMiner/`), which implements a set of tools for association rule mining. The output of the algorithms is expressed as XML documents, with a schema that is aligned with the patterns introduced in Section 3.1 and and the pattern KB.

For our experiments we used a dataset of 303 pipes definitions from the repository of Pipes. We selected pipes from the list of "most popular" pipes, in that popular pipes are more likely to be functional and useful. The average number of components, connectors and input parameters are 12.7, 13.2 and 3.1, respectively, which is an indication that we are dealing with fairly complex mashup compositions.

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 shows a summary of the patterns discovered by the two algorithms introduced before. We used a minimum support threshold between 0.050 and 0.075 for finding patterns, but, clearly, this is a configuration parameter subject to tuning. In the table, we report the average support of the discovered patterns. For example, given that Yahoo! Pipes is particularly strong for processing Atom and RSS feeds, it is common for our algorithms to find

Algorithm 3: 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. $\quad$ append($DB_{xy}], m_i.DF \cap \{df_{xy}\}$); ; `// fill with instances of the reference connector type`
4. **return** DB_{xy};

Algorithm 4: mineComponentCooccurrences

Data: repository of mashup models M, minimum support of data flow connectors ($minsupp_{df}$), data mappings ($minsupp_{dm}$) and parameter value assignments ($minsupp_{va}$).
Result: set of component co-occurrence patterns with their corresponding dataflow connectors, data mappings and parameter values $\{\langle df_{xy,i}, VA_{x,i}, VA_{y,i}, DM_{y,i}\rangle\}$

1. F_{df} = getRecurrentConnectors($M, minsupp_{df}$);
2. DB = array(); `// database of recurrent connector instances`
3. $Patterns$ = set(); `// set of component co-occurrence patterns`
4. **foreach** $\$df_{xy} \in F_{df}$ **do**
5. $\quad DB[df_{xy}]$ = getConnectorInstances(M, df_{xy});
   `// create databases for frequent itemset mining`
6. $\quad DBVA_x$ = array();
7. $\quad DBVA_y$ = array():
8. $\quad DBDM_y$ = array():
9. $\quad$ **foreach** dfi_{xy} *in* $DB[df_{xy}]$ **do**
10. $\quad\quad c_x$ = source component of dfi_{xy};
11. $\quad\quad c_y$ = target component of dfi_{xy};
12. $\quad\quad$ append($DBVA_x$, $c_x.VA$);
13. $\quad\quad$ append($DBVA_y$, $c_y.VA$);
14. $\quad\quad$ append($DBDM_y$, $c_y.DM$);
15. $\quad FI_{vx}$ = mineFrequentItemsets($DBVA_x$, $minsupp_{par}$);
16. $\quad FI_{vy}$ = mineFrequentItemsets($DBVA_y$, $minsupp_{par}$);
17. $\quad FI_{dy}$ = mineFrequentItemsets($DBDM_y$, $minsupp_{par}$);
    `// keep only those combinations of value assignments and data mappings that frequently occur together`
18. $\quad Coo$ = set();
19. $\quad$ **foreach** $\langle VA_x, VA_y, DM_y\rangle \in FI_{vx} \times FI_{vy} \times FI_{dy}$ **do**
20. $\quad\quad$ **if** *computeSupport*($\langle VA_x, VA_y, DM_y\rangle, DB[df_{xy}]$) $\geq minsupp_{df}$ **then**
21. $\quad\quad\quad Coo = Coo \cup \{\langle VA_x, VA_y, DM_y\rangle\}$;
    `// construct the component co-occurrence patterns`
22. $\quad$ **foreach** $\langle VA_x, VA_y, DM_y\rangle \in Coo$ **do**
23. $\quad\quad Patterns = Patterns \cup \{\langle df_{xy}, VA_x, VA_y, DM_y\rangle\}$
24. **return** $Patterns$;

patterns of the type"use *textinput*, *urlbuilder*, *fetchfeed*, *sort* components together, connecting them in sequence." These patterns are valid and make sense, yet they lack semantics, mainly because the components in Yahoo! Pipes are *generic*. This lack of semantics is alleviated to some extent by discovering fragments that are as complete as possible: instead of just telling which component types co-occur together, we also need to tell how they are connected, how data is mapped inside components and how the parameter values of components are filled. Among these, and in the context of this experiment, parameter values are the most powerful way to give semantics to mashup constructions.

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, recommendations are to be delivered at high speed, without slowing down the modeler's composition pace. Recommending patterns is

Table 1: Summary of discovered patterns

Connector patterns
[Connectors] Minimum support found: 0.0759
[Connectors] Maximum support found: 0.3234
[Connectors] Number of frequent dataflow connectors: 26
[Data mappings] Minimum support found: 0.0502
[Data mappings] Maximum support found: 0.0970
[Data mappings] Number of frequent data mappings: 166

Parameter values for component co-occurence pattern
Minimum support found: 0.0769
Maximum support found: 0.2308
Number of frequent itemsets: 464

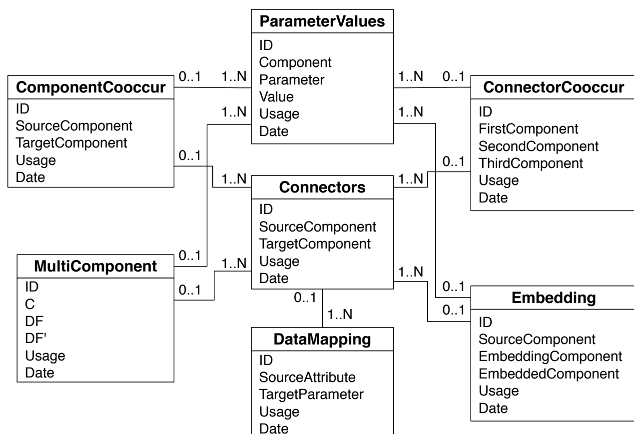

Figure 3: KB structure optimized for Pipes

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.

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 for 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 3, whose structure enables the retrieval of each of the patterns introduced in Section 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 $ComponentCooccur \bowtie ParameterValues$.

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 5: 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;

Algorithm 5 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 *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].

5.3 Implementation and 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 in Firefox 3.6.17 on a common MacBook. The implementation of the *client-side KB* is based on Google

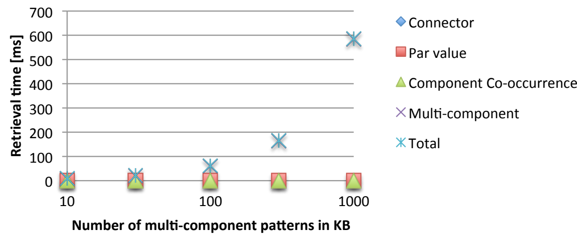

Figure 4: Recommendation types and times in response to a new component added to the canvas

Gears (`http://gears.google.com`), which internally uses SQL Lite (`http://www.sqlite.org`) for storing data on the client's hard drive. We artificially generated several thousands of patterns (of which 1000 multi-component patterns) of different types and sizes and measured the recommendation retrieval times in function of varying *object* sizes [10].

Figure 4 illustrates the performance of Algorithm 5 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.

6. WEAVING PATTERNS

Weaving a given pattern *cp* into a partial mashup model *pm* is not straightforward and requires a thorough analysis of both *pm* and *cp*, in order to understand how to connect the pattern to the constructs already 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*.

6.1 Basic Weaving Strategy

Given the *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 *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.

In Table 2 we define a set of ***mashup operations*** that resemble the operations a developer can typically perform manually in the modeling canvas. 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 are highly platform-specific, in that they need to operate inside the target modeling environment; in our case, our implementation manipulates the internal JSON representation of Pipes.

For instance, the basic weaving strategy for a component

Table 2: Pipes-like basic mashup operations

Operation
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, parameter values) deleted from pm.
assignValues(cid, VA): produces pm' with the value assignments VA added to the component with identifier cid.
deleteAllValues(cid): produces pm' with all input parameters of the component identified by cid emptied.
deleteValue(cid, in): produces pm' with the input parameter in for the component identified by 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 the data mapping DM added to the component identified by cid.
deleteAllDataMappings(cid): produces pm' with all data mappings deleted from the component identified by 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 $hostid$.

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):

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

That is, given a component, we create the other component of the pattern and apply the respective data mappings and value assignments. Note that, the basic strategy is not directly applied to *pm*; it represents an array of basic modeling operations to be further processed.

In `http://goo.gl/Xk7VF` we provide the basic strategies for all the patterns introduced in Section 3.1 in the form of a function **getBasicStrategy**($cp, object$) $\rightarrow BS$.

6.2 Contextual Weaving Strategy

Given an object *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 contextual strategy takes into account the structure of the partial mashup (the context). The contextual strategy is derived by applying a basic strategy to a partial mashup. Therefore, WS is *dynamically* built at runtime.

Applying the mashop operations in the basic strategy may

[5]We highlight in bold variables that cannot be resolved immediately inside the basic strategy and that, hence, are kept as is for later resolution (at the time the individual mashup operations will be executed)

Algorithm 6: 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 ∈ BS do
4     CtxInstr = resolveConflict(pm, instr);
5     pm = apply(pm, CtxInstr);
6     append(WS, CtxInstr);
7 return ⟨WS, pm⟩;
```

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

In `http://goo.gl/9jJtK` we provide the complete soft and hard conflict resolution policies. The policies come in the form of 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 6 we describe how we derive the contextual weaving strategy. The algorithm produces both the list of contextual weaving instructions and the final mashup model pm'. The former can be used to interactively weave *cp* into *pm*, the latter to convert pm' into native formats.

6.3 Implementation and Evaluation

As highlighted in Figure 2, we are still working on the implementation of an own mashup editor; we have therefore implemented the above pattern weaving functionality by tapping into *Yahoo! Pipes*, extracting partial mashup models, weaving patterns, and feeding back the updated models. The implementation runs on the same test system described in Section 5.3 and is entirely based on *JavaScript*. Both weaving strategies are encoded as JSON arrays, which enables us to use the native `eval()` command for fast and easy parsing of the weaving logic. The respective mashup operations manipulate the native JSON format of Pipes.

To test our algorithms, we have manually developed a set of partial mashup definitions in Pipes. In order to access them, as well as to get the structure of the *object*, we have fetched their internal JSON representations from Pipes using Yahoo!'s YQL (`http://developer.yahoo.com/yql/console/`) service. The service provides access to pipes models via queries like `"select PIPE.working from json where url=http://pipes.yahoo.com/pipes/pipe.info?_out=json&_id=91bcac4061c6d893b15b4b5864bf37bd"`, the final number being the *id* of the partial mashup composition. In order to feed pipe models back after the weaving process, we simply reload the model in the Pipes editor, however redirecting the respective HTTP request to our own server by suitably setting the *hosts* file of our test machine. This causes Pipes to fetch the updated model from our web server, not from Yahoo!'s own server.

The two key Pipes-specific issues to be solved in the implementation are the management of *identifiers* and the *layout*. Pipes assigns system-generated ids to model constructs in a composition; in the same spirit, we have developed a unique id generator able to produce compatible ids (e.g., `sw-xxx` for components and `_wx` for wires). Each component has coordinates to place it correctly inside the mashup canvas. As for now, we generate random coordinates based on the average height and width of the different component types and on the average distance between them. In the future, we plan to implement a more sophisticated layout logic.

7. 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 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

[6]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.

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. [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. [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. [13], just like other modeling elements. The system does not provide interactive recommendations and rather focuses on the correct insertion of patterns.

In summary, mashups and service composition 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, 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 in order to mine meaningful patterns.

8. CONCLUSIONS

With this paper, 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., also featuring 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 or 100. 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).

9. 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, `http://goo.gl/QCTJH`, 2011. Springer.

[11] 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.

[12] P. Tan, S. M, and K. V. *Introduction to Data Mining*. Addison-Wesley, 2005.

[13] 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.