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.

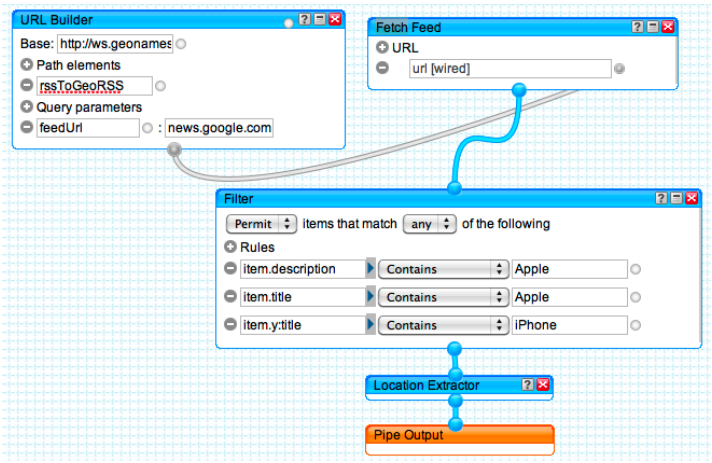

Fig. 1.1 A typical pattern in Yahoo! Pipes

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, $src \in \{$"*Pipes*","*Wires*","*myCocktail*", ...$\}$ 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 al-

low 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.

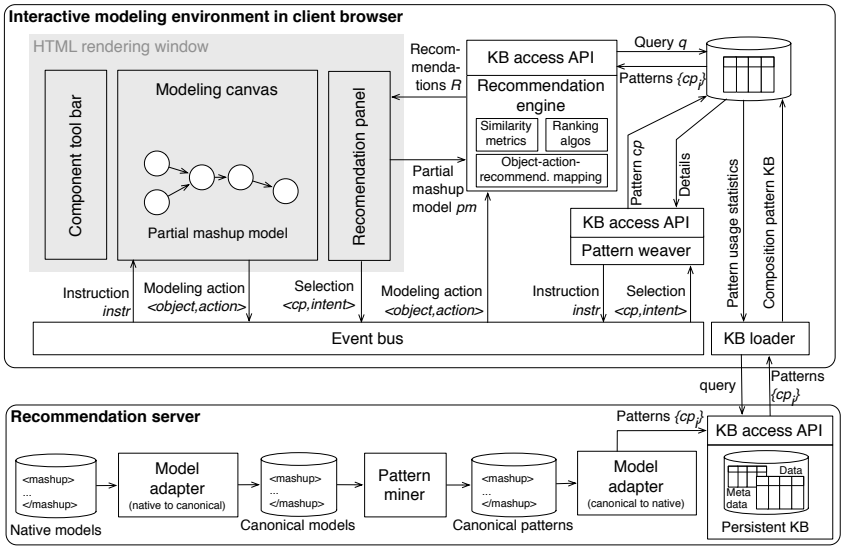

Fig. 1.2 Functional architecture of the composition knowledge discovery and recommendation approach

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 ∈ 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 ∈ FI do
9          Patterns = Patterns ∪ {⟨ct, VA⟩};
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 ∈ 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 ∈ 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 ∈ FI_dy do
12         Patterns = Patterns ∪ {⟨df_xy, DM_y⟩};
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 ∈ 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 ∈ DB_df do
6      if computeSupport(df_xy, DB_df) ≥ minsupp_df then
7          F_df = F_df ∪ {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 ∈ M do
3      append(DB_xy], m_i.DF ∩ {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
1  DB_df = array();
2  foreach m_i ∈ M do
3      append(DB_df, m_i.DF);
4  F_df = mineFrequentItemsets(DB_df, minsupp_df);
5  DB_ci = array();
6  foreach m_i ∈ M do
7      foreach DF_xy ∈ F_df do
8          if DF_xy ∩ m_i.DF = DF_xy then
9              foreach dfi_xy ∈ DF_xy do
10                 append(DB_ci[DF_xy], getConnectorInstances({m_i}, dfi_xy);
   // find data mappings for the frequent dataflow connectors obtained above
11 DBDM_y = array();
12 foreach DF_xy ∈ DB_ci do
13     foreach dfi_xy ∈ DF_xy do
14         c_y = target component of dfi_xy;
15         append(DBDM_y, c_y.DM);
16 FI_dy = mineFrequentItemsets(DBDM_y, minsupp_dm);
   // construct the connector patterns
17 Patterns = set();
18 foreach DM_y ∈ FI_dy do
19     Patterns = Patterns ∪ {⟨DF_xy, DM_y⟩};
20 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 $\{\langle df_{xy,i}, VA_{x,i}, VA_{y,i}, DM_{y,i}\rangle\}$

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 $DB[df_{xy}]$ = getConnectorInstances(M, df_{xy});
// create databases for frequent itemset mining
6 $DBVA_x$ = array();
7 $DBVA_y$ = array();
8 $DBDM_y$ = array();
9 **foreach** dfi_{xy} *in* $DB[df_{xy}]$ **do**
10 c_x = source component of dfi_{xy};
11 c_y = target component of dfi_{xy};
12 append($DBVA_x$, $c_x.VA$);
13 append($DBVA_y$, $c_y.VA$);
14 append($DBDM_y$, $c_y.DM$);
15 FI_{vx} = mineFrequentItemsets($DBVA_x$, $minsupp_{par}$);
16 FI_{vy} = mineFrequentItemsets($DBVA_y$, $minsupp_{par}$);
17 FI_{dy} = mineFrequentItemsets($DBDM_y$, $minsupp_{dm}$);
// keep only those combinations of value assignments and data mappings that occur together in mashup instances
18 Coo = set();
19 **foreach** $\langle VA_x, VA_y, DM_y\rangle \in FI_{vx} \times FI_{vy} \times FI_{dy}$ **do**
20 **if** $computeSupport(\langle VA_x, VA_y, DM_y\rangle, DB[df_{xy}]) \geq minsupp_{pc}$ **then**
21 $Coo = Coo \cup \{\langle VA_x, VA_y, DM_y\rangle\}$;
// construct the component co-occurrence patterns
22 **foreach** $\langle VA_x, VA_y, DM_y\rangle \in Coo$ **do**
23 $Patterns = Patterns \cup \{\langle df_{xy}, VA_x, VA_y, DM_y\rangle\}$;
24 **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
DB_em = array();
foreach m_i ∈ M do
    foreach ⟨c_x, c_y, c_z⟩ ∈ m_i.C × m_i.C do
        if (c_x preceeds c_y) and (c_y embeds c_z) then
            em_xyz = ⟨c_x, c_y, c_z⟩;
            append(DB_em, em_xyz);
// find the frequent component embeddings
F_em = set();
foreach em_xyz ∈ DB_em do
    if computeSupport(em_xyz, DB_em) ≥ minsupp_em then
        append(F_em, em_xyz);
// get dataflows involving the frequent component embeddings
DB_df = array();
F_df = array();
foreach m_i ∈ M do
    foreach em_xyz ∈ F_em do
        if em_xyz ∈ m_i then
            append(DB_df[em_xyz], ⟨m_i.df_xy, m_i.df_xz, m_i.df_yz⟩);
    F_df = mineFrequentItemsets(DB_df, minsupp_df);
// get parameter value and data mapping instances and compute the
   corresponding frequent itemsets
DB_va = array(); DB_dm = array();
foreach m_i ∈ M do
    foreach ⟨df_xy, df_xz, df_yz⟩ ∈ F_df do
        if ⟨df_xy, df_xz, df_yz⟩ ∈ m_i then
            c_x = component instance c_x ∈ m_i corresponding to df_xy;
            c_y = component instance c_y ∈ m_i corresponding to df_xy;
            c_z = component instance c_z ∈ m_i corresponding to df_yz;
            VA_x = c_x.VA; DM_x = c_x.DM;
            VA_y = c_y.VA; DM_y = c_y.DM;
            VA_z = c_z.VA; DM_z = c_z.DM;
            append(DB_va, VA_x ∪ VA_y ∪ VA_z);
            append(DB_dm, DM_x ∪ DM_y ∪ DM_z);
F_va = mineFrequentItemsets(DB_va, minsupp_par);
F_dm = mineFrequentItemsets(DB_dm, minsupp_dm);
// construct the component embedding pattern
Patterns = set();
foreach ⟨EM, DF, DM, VA⟩ ∈ F_em × F_df × F_dm × F_va do
    if computeSupport(⟨EM, DF, DM, VA⟩, M) ≥ minsupp_pc then
        c_x, c_y, c_z = components corresponding to the dataflows df ∈ DF;
        Patterns = Patterns ∪ {⟨{c_x, c_y, c_z}, DF, DM, VA⟩};
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 ∈ 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 ∈ M do
8      foreach fg_i ∈ 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 ∈ DB_mc do
       // get parameter values and data mappings and compute the corresponding
          frequent itemsets
14     DBVA = array();
15     DBDM = array();
16     foreach mf ∈ MC do
17         foreach c_x ∈ 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 ⟨VA, DM⟩ ∈ FI_va × FI_dm do
23         foreach mf ∈ MC do
24             if ⟨VA, DM⟩ ∈ mf then
25                 Patterns = Patterns ∪ {⟨mf.C, mf.DF, VA, DM⟩} ;   // 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 $ComponentCooccur \bowtie ParameterValues$.

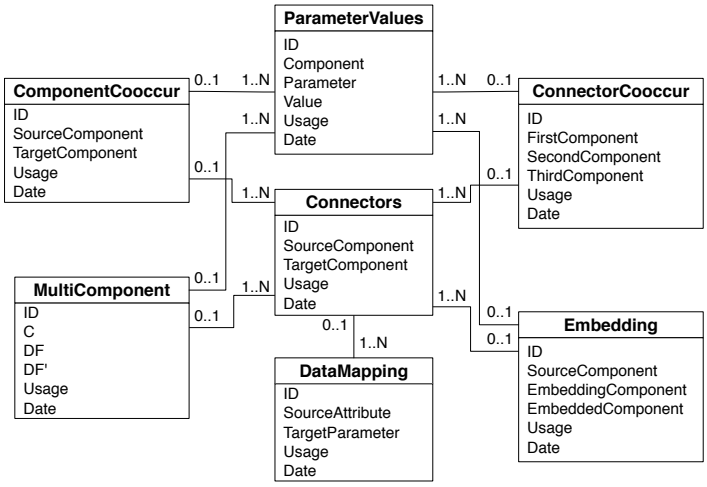

Fig. 1.3 KB structure optimized for Pipes

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 ∈ recTypeToBeGiven do
5      if recType ≠ "Mul" then
6          Patterns = Patterns ∪ queryPatterns(object, KB, recType) ;        // exact query
7      else
8          Patterns = Patterns ∪ getSimilarPatterns(object,
           KB, CompSim, T_sim) ;                                  // similarity search
9  foreach pat ∈ Patterns do
10     if rank(pat.cp, pat.sim, pm) ≥ T_rank then
11         append(R, ⟨pat.cp, rank(pat.cp, pat.sim, pm)⟩) ;   // rank, threshold, remember
12 orderByRank(R);
13 groupByType(R);
14 truncateByGroup(R, n);
15 return R;
```

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 con-

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

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

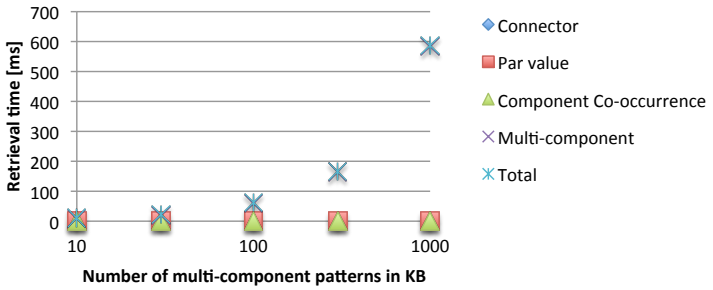

Fig. 1.4 Recommendation types and times in response to a new component added to the canvas

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.