Online Appendix to: Recommendation and Weaving of Reusable Mashup Model Patterns for Assisted Development

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

This online appendix contains the following.
(1) Figure 9
(2) Algorithms 4 & 5
(3) Tables I, II, III, and IV

A. PATTERN KNOWLEDGE BASE STRUCTURE

Figure 9 illustrates the structure of the pattern KB. The schema enables fast retrieval of all the patterns defined in Section 3.2 with a one-shot query over a single table. The KB is partly redundant (e.g., the *MultiComponent* entity also contains components and connectors; we explain the details in Section 4.2), but this is intentional. It allows us to avoid expensive database join operations at recommendation time and to defer them to when a pattern must be woven. For example, in order to retrieve the representation of a component co-occurrence pattern, it is enough to query the *ComponentCooccur* entity for the *SourceComponent* and the *TargetComponent* attributes; only weaving the pattern then requires querying $ComponentCooccur \bowtie DataMapping \bowtie ParameterValues$ for the necessary details.

B. RECOMMENDING PATTERNS

In the following, we describe in more detail the two algorithms to retrieve personalized recommendations and expert recommendations, as introduced in Section 4.3 and Section 4.4, respectively.

B.1. Personalized Recommendations Algorithm

Algorithm 4 shows how we use the inferred user-item rating matrix R' to retrieve personalized recommendations. R' is precomputed offline and, together with the user identifier *uid*, constitues a new input compared to Algorithm 2. In essence, we retrieve all similar patterns using getPatterns (line 2) and then, for each pattern, we sum the personal component ratings encoded in R' of all components of the pattern (lines 4–6), normalize the sum by the number of components (line 7), and keep those patterns that exceed the ranking threshold value T_{rank} (lines 8 and 9).

B.2. Expert Recommendations Algorithm

Algorithm 5 is very similar to Algorithm 4 with two key differences: instead of the inferred user-item matrix R', it uses the expert-item rating matrix E, and it is independent of any particular user identifier *uid*.

C. PATTERN WEAVING

C.1. Mashup Operations

Table I lists all mashup operations identified for dataflow-based mashup environments like Yahoo! Pipes. Mashup operations can be combined in a scripting-like fashion to

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

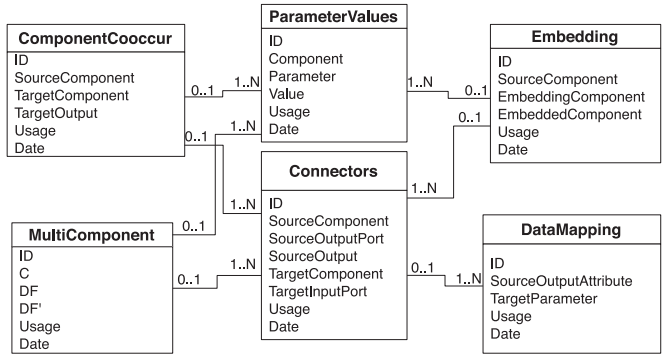

Fig. 9. Knowledge base structure for storage and retrieval of mashup model patterns.

ALGORITHM 4: getPesonalizedRecommendations

Data: $q = \langle object, action, pm \rangle$, KB, OAR, $CompSim$, T_{sim}, T_{rank}, k for top-k threshold, user id uid, inferred user-item rating matrix R'
Result: Recommendations $R = [\langle cp_i, rank_i \rangle]$ with $rank_i \geq T_{rank}$

```
R = array();
Patterns = getPatterns(q, KB, OAR, CompSim, T_sim) ;                    // retrieve patterns
foreach pat ∈ Patterns do
    personalRank = 0;                                                   // initialize personal rating
    foreach component ∈ pat do
        personalRank += getComponentRating(R', component, uid);         // sum individual, personal ratings
    personalRank = personalRank/|pat|;                                  // normalize by number of components in pattern
    if personalRank ≥ T_rank then
        append(R, ⟨pat.cp, personalRank⟩);                              // rank, threshold, remember
OrderGroupTruncate(R, k);
return R;
```

ALGORITHM 5: getExpertRecommendations

Data: $q = \langle object, action, pm \rangle$, KB, OAR, $CompSim$, T_{sim}, T_{rank}, k for top-k threshold, expert-item rating matrix E
Result: Recommendations $R = [\langle cp_i, rank_i \rangle]$ with $rank_i \geq T_{rank}$

```
R = array();
Patterns = getPatterns(q, KB, OAR, CompSim, T_sim) ;                    // retrieve patterns
foreach pat ∈ Patterns do
    expertRank = 0;                                                     // initialize expert rating
    foreach component ∈ pat do
        expertRank += getComponentRating(E, component);                 // sum individual expert ratings
    expertRank = expertRank/|pat|;                                      // normalize by number of components in pattern
    if expertRank ≥ T_rank then
        append(R, ⟨pat.cp, expertRank⟩);                                // rank, threshold, remember
OrderGroupTruncate(R, k);
return R;
```

instruct the pattern weaver as to how to emulate modeling actions inside the modeling canvas so as to weave a given pattern *cp* into a partial mashup model *pm*.

C.2. Basic Weaving Strategy

Table II illustrates the basic weaving strategies for the five identified mashup pattern types, along with the assumptions regarding the object of the query that triggered the recommendation of the pattern. We recall that the basic weaving strategy tells which modeling actions to apply so as to expand the object into the chosen pattern *cp*.

Table I. Dataflow-Based Mashup Operations for the Definition of Weaving Strategies

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

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

Object	**Basic Strategy**
Parameter value pattern $ptype^{par}$	
$comp$ with $comp.type$=$c.type$	assignValues($comp.id, VA$);
Connector pattern $ptype^{con}$	
$comp_x, comp_y$ with $comp_x.type$=$c_x.type$ and $comp_y.type$=$c_y.type$	addConnector($comp_x.id, c_x.op, comp_y.id, c_y.ip$); assignDataMappings($comp_y.id, c_y.DM$);
Component co-occurence pattern $ptype^{com}$	

$comp_x$ with $comp_x.type$= $c_x.type$	$newcid=addComponent($c_y.type$);
	addConnector($\langle comp_x.id, c_x.op,$ $newcid, c_y.ip\rangle$);
	assignDataMapping($newcid, c_y.DM$);
	assignValues($comp_x.id, c_x.VA$);
	assignValues($newcid, c_y.VA$);

Component embedding pattern $ptype^{emb}$	

$comp_x, comp_y, df_{xy}$ with $comp_x.type$= $c_x.type$ and $comp_y.type = c_y.type$	$embcid=addComponent($c_z.type$);
	addConnector($\langle comp_x.id, c_x.op,$ $embcid, c_z.ip\rangle$);
	addConnector($\langle embcid, c_z.op, comp_y.id, c_y.ip\rangle$);
	embedComponent($comp_y.id,$ $embcid$);
	assignDataMappings($comp_y.id, c_y.DM$);
	assignDataMappings($embcid, c_z.DM$);
	assignValues($comp_x.id, c_x.VA$);
	assignValues($comp_y.id, c_y.VA$);
	assignValues($embcid, c_z.VA$);

Multi-component pattern $ptype^{mul}$	
$comp$ with $comp.type \in$ Types(C)	$\forall c_i \in (C - \{comp\})$ $newcid[i] = addComponent(c_i.type)$; $compidx = i$ with $c_i.type$=$comp.type$; $newcid[compidx] = comp.id$; $\forall f^{xy} \in DF$ addConnector($newcid[srccid], srcop,$ $newcid[tgtcid], tgtip\rangle$); $\forall i \in newcid$ assignDataMappings($newcid[i], c_i.DM$); $\forall i \in newcid$ assignValues($newcid[i], c_i.VA$);

C.3. Conflict Resolution Policies

During weaving, we are in the presence of a *conflict* if we want to add a new construct to the partial mashup model pm, but the partial mashup model already contains this construct. The hard conflict resolution policy (Table III) resolves the conflict by creating and using a new construct of the same type, whereas the soft conflict resolution

Table III. Hard Conflict Resolution Policy resolveConflict($pm, instr$) $\rightarrow CtxInstr$

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

Table IV. Soft Conflict Resolution Policy resolveConflict($pm, instr$) $\rightarrow CtxInstr$

Basic instruction *instr*	**Conflict with** *pm*	**Contextual instr.** *CtxInstr*
assignValues(*cid*, *VA*);	We want to preserve possible value assignments, if they are not in conflict with any of the values in *VA*.	assignValues(*cid*, *VA*);
addConnector(df_{xy});	The connector df_{xy} already exists.	—
addConnector(df_{xy});	A connector $df_{zy} \neq df_{xy}$ from a component $comp_z$ to the same input port ip_y of df_{xy} already exists, and in_y allows only one connector in input.	deleteConnector(df_{zy}); addConnector(df_{xy});
$var=addComponent(*ctype*);	A component *comp* of type *ctype* already exists, and we want to reuse existing components.	$var=*comp.id*;
assignDataMappings(*cid*, *DM*)	We want to preserve possible data mappings data are not in conflict with the data mappings in *DM*.	assignDataMappings(*cid*, *DM*);
embedComponent(*hostid*, *embid*);	A component with identifier *oldid* has already been embedded into the component *hostid*.	deleteComponent(*oldid*); embedComponent(*hostid*, *embid*);

policy (Table IV) aims to maximize reuse and therefore reemploys the already existing component when weaving the pattern.

C.4. Weaving Example

Let's see a concrete example of how the contextual weaving strategy is built, starting from the basic strategy, the conflict resolution policy, the partial mashup model, and the pattern to be woven. We use the modeling situation illustrated in Figure 1. Let us assume that the modeler's last modeling action was placing the Fetch Feed component and connecting it with the output of the URL Builder component. Let us further assume that, among the recommended patterns, the modeler accepts a component co-occurrence pattern that suggests to add a Filter component after the existing Fetch Feed component. Applying this pattern to the partial model in the canvas requires: (i) adding a new Filter component, (ii) connecting the Filter component with the output of the Fetch Feed component, (iii) applying the data mapping stored in the pattern to the newly created Filter component, (iv) resolving the conflict among the values of the "URL" parameter of the Fetch Feed component in the partial mashup and in the pattern, and (v) assigning parameter values to the Filter component.

Using Algorithm 3 produces the contextual weaving strategy and shown in Figure 10 that resembles the modeling steps described before in terms of the basic mashup operations introduced in Table I. Line 1 adds the new Filter component and stores the respective identifier in the variable $\$newcid$. Line 2 connects the new component to the Fetch Feed component. In order to do so, the pattern weaver retrieves the id of the Fetch Feed component from the JSON representation of the partial mashup model (in our test with Yahoo! Pipes, this specifically produced the id "sw-100"; for different

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

Fig. 10. Contextual weaving strategy weaving a pattern into a partial mashup model.

runs, this identifier will change) and invokes the function *addConnector*, passing the id "sw-100", the type of the output port "_OUTPUT" for Fetch Feed, the id of the newly created Filter component, and the type of the respective input port ("_INPUT"). The output and input port types are stored in the pattern and replaced with their ids at runtime. Line 3 assigns the data mappings to the Filter component in the form of three name-value pairs. The name identifies the input field (e.g., "conf.Rule[0].field"), while the value is the data mapping (e.g., "item.description"). Both values are stored in the pattern. Lines 4 and 5 assign the value to the "URL" parameter (identified internally via "conf.URL") of the Filter component. Actually, the two lines are the result of the resolution of a conflict. The conflict resolver expands the *assignValues* function described in Table III, first by deleting the old value and then by assigning the new one. Incidentally, in our example the old and new values are the same, but this is not true in general. Finally, line 6 applies the value assignments to the Filter component and the pattern is successfully woven into the partial mashup model in the canvas.