MashStudio: An On-the-fly Environment for Rapid Mashup Development

Jianyu Yang, Jun Han, Xu Wang, and Hailong Sun

School of Computer Science and Engineering, Beihang University,
Beijing, 100191 China
{yangjy,hanjun,wangxu,sunhl}@act.buaa.edu.cn

Abstract. Mashup is a new application development pattern that integrates different resources such as data, service or api to construct application. Despite the emergence of mashup platforms like YahooPipes or Popfly, current platforms aim toward users with some programming knowledge. When facing a lot of components in the platform, common users without any programming knowledge always don't know how to begin. MashStudio aims to provide interactive assistance at every step during the development, in order to significantly improve the development efficiency and quality. To achieve this goal, we created a repository with collected mashup process (the composition logic information of a mashup) and analyzed these processes to find meaningful mashup process fragments, which are reusable and will be interactively recommended to users when building mashup. Moreover, MashStudio can provide the function of "just in time compilation" to help user find mistakes in time instead of after accomplished the whole mashup. Finally, the experimental evaluation demonstrates the efficiency and utility of our method.

Keywords: mashup, end user, recommendation, process fragments reuse

1 Introduction

As a new application development pattern, mashup supports the integration of different resources provided by third parties and available from the Internet to help end user rapidly build personalized application. Meanwhile, with the rapid development of the Web 2.0 technologies, more and more opened user interfaces or web services are published in the Internet. Agenzy[1], a website that combines 8 open APIs provided by Amazon, Brightcove, LinkShare and contrasts products from different suppliers, is a typical application built by mashup technologies. So far, there are about 5800 APIs and 6600 mashup applications listed in ProgrammableWeb[2], from where users can register new mashups and access these data through the Internet freely. YahooPipes[3] provides an online development tool to help users build their own applications on the web, and almost 36,000

[1] http://www.agenzy.com/
[2] http://www.programmableweb.com/
[3] http://pipes.yahoo.com/

mashups have been created with this tool. Moreover, with the development of mobile Internet, mobile terminals become more powerful than before, and more people are willing to use mobiles for its portability, people can access Internet with mobile at anytime and anywhere. All these make mobile terminal become a new platform for all kinds of applications.

Currently, many existing popular mashup editors like YahooPipes, Microsoft Popfly[4], IBM Damia[5] can further make it easy for end users to build mashup applications through simple direct drag and drop manipulation or UI operator components. Yet, although these advances in graphical development environment, building a mashup application with the above editors is still not easy for end users, especially for the users who just begin to use these editors and don't familiar with the details. Facing with many different modules in the editor, users have to perform some unpleasant steps when building mashup applications, such as finding a suitable module for their situational needs, reading its specification and figuring out the parameters. Even so, users may still make some mistakes like connecting unmatched modules together, and they always don't realize this situation during development. Finally, when accomplished the mashup, they may found that the mashup application can't give the right results as their expectations, but locating the mistakes in the complex mashup process is a difficult task for end users.

To figure out the above problems, we build an on-the-fly development environment, named MashStudio, that can provide design-time assistance by recommending connectable components or process fragments to end-user and making in time response to give correctness verification in every step. By leveraging the power of collective intelligence, MashStudio collected real mashups from the Internet, we created a repository to analyze the internal structure of the collected mashups and compute the connectable components based on a probabilistic model. Compare with other mashup tools, MashStudio has the following four features which make it more efficient and convenient for end-user.

1. Unicursal Development. To simplify the development and improve development efficiency, MashStudio provide design-time assistance at every step. After dragging and dropping a component, the user just needs to click current component, MashStudio will interactively return a list of recommended items, and the alternative items will be classified and ranked for the users' convenience. The user just needs to click the required item, it will be automatically connected to current partial mashup process. The whole development procedure just like painting a unicursal, what the users need to do is no more than "select" and "click".

2. On-the-fly Development. To detect potential errors without delay, MashStudio don't distinguish design-time and run-time. That is to say, MashStudio can get the intermediate state and validate the correctness of the user's action in real time, and it may alert user whenever user makes mistakes. If not the on-the-fly environment, when the user accomplished the whole mashup, he

[4] http://www.popfly.com/Overview/
[5] http://www.damia.alphaworks.ibm.com/

may find that the mashup can't execute at all, but locating this bug and repairing this complex mashup process may be not that easy.

3. Process Fragment Reuse. As we all know, reusing existing mashups that basically match user's need is much easier than creating the whole mashup application from scratch. On the other hand, reusing process fragment and considering the fragment as a single component will significantly reduce the complexity of the mashup process, meanwhile the development efficiency will been significantly improved.

4. Develop on PC and Use on Mobile. With the development of mobile Internet and mobile devices, people can access Internet at anytime and anywhere, which brings great convenience for people's lives. So we combine the traditional personal computer and mobile device together, users can build their own mashup applications on PC and access them from their mobile devices freely. MashStudio provides programming interfaces for third-party devices. When accessing mashup application from mobile, MashStudio's background engine will execute this mashup and return the result.

The rest of this paper is organized as follows. Section 2 describes a typical scenario to illustrate the features mentioned above. Section 3 gives a detailed description of our approach. In Section 4, we will evaluates MashStudio with experiments. Related works are summarized in Section 5. Finally, we will conclude with future works in Section 6.

2 Motivation Scenario

We describe a typical scenario to illustrate how MashStudio provide assistances for end users. In this scenario, Tom is a sports fan, he wants to create a mashup application that aggregates sports news RSS information and shows the news on google map. Without MashStudio, Tom has to pick up components such as "fetch feed", "google map", "extract location" and "union", and then connects them together to build mashup application. When building the mashup, Tom has to read their specifications to figure out all these components first. However, with MashStudio, the building procedure is much easier:

(1) Tom drags and drops a "google map" component on the edit panel, MashStudio recommends a list of components that can be connected to it.
(2) In the recommended list, Tom finds a "breaking news" component that supports sports information from several famous sports websites. Moreover, this news component also provides coordinate information for map display.
(3) Tom selects the "breaking news" component, then the "breaking news" component automatically connects to "google map".
(4) Tom just wants to see the latest news, so he needs to modify this mashup. He clicks the output point of "breaking news" component, MashStudio recommends a component list.
(5) Tom selects a "sort filter" component to sort the sports news, but mistakenly configures the component parameter to sort news by wrong time for-

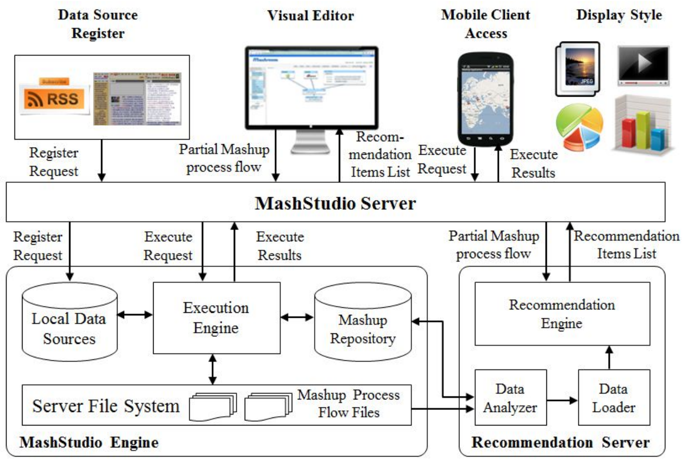

Fig. 1. System Architecture

mat. Because of this component doesn't match "breaking news" component. MashStudio validates Tom's action and alerts him.
(6) Tom reselects "sort by date", and MashStudio automatically connects it to "breaking news".
(7) Then, Tom clicks the "sort filter" component, and selects "truncate" to keep the latest 20 sports news. Then he saves this mashup and access the news from his mobile freely.

In step 2, without MashStudio, Tom has to select interested news sources by himself, and connects them together in appropriate order, then extracts location information. Now MashStudio can help Tom accomplish all these tasks by recommending a single component based upon the information already exist in editor panel and the mashup process repository from real users. Because of replacing the complex task with a single component, the mashup process becomes much simpler. Moreover, MashStudio provides an on-the-fly environment to realize real-time verification. In step 5, Tom can find the mistake in time, rather than waiting until he accomplished the whole mashup.

3 The MashStudio Platform

MashStudio can provide design-time assistance by recommending components based upon the partial process already exist in editor panel and the mashup process repository collecting from real users. So far, we collected 6000 real mashup processes from YahooPipes and analyzed their internal structure to extract meaningful knowledge as the basis for recommendation. The following sections first describe the overall architecture of MashStudio, then explain the details of how to deal with collected data and the recommendation strategy.

3.1 System Architecture

The structure of MashStudio is depicted in Figure 1. The main components of MashStudio include Visual Editor, Data Source Register, Mobile Client Access, Execution Engine, Mashup Repository, and Recommendation Server.

Visual Editor: MashStudio provide a visual drag and drop development environment, the end user can use the visual editor through their browser software to develop their personal mashup applications and save them in server.

Data Source Register: MashStudio provides many kinds of data sources to users, but this may not meet user's need. Therefore, MashStudio provides this browser plugin to scrape the information on the screen when the user browses web pages.

Mobile Client Access: MashStudio provides access APIs for mobile clients, so users can trigger a request from their mobile phones to execution engine and get the execution results at anytime and anywhere.

Execution Engine: The execution engine can analyze the mashup process, and then access and integrate data in accordance with the process logic. Moreover, whenever users take any action in Visual Editor, the action will be sent to execution engine. The engine will put this change into practice in realtime and return the intermediate results immediately.

Mashup Repository: When the user accomplished the mashup application, the process file of this application will be saved to mashup repository. This repository also provides initial materials for recommendation server to support the recommendation engine.

Recommendation Server: As previously mentioned, MashStudio needs to provide design-time assistance for user, so the response time is very import for user experience. The recommendation server loads the initial data in mashup repository and analyzes them off-line. The recommendation engine is the core of the recommendation server, and it performs a recommendation algorithm to realize rapid recommendation based upon existing partial process.

3.2 The Execution Engine

The execution engine provides support for the on-the-fly environment. It extracts logic information based upon the input mashup process coming from the MashStudio Web Server, and then executes the operations in the process according to the logic information. The requests to the engine may come from Visual Editor or Mobile Client, and the purposes of the requests are different. The Visual Editor needs to get the intermediate data in the editing process to provide assistance for modeling, therefore, the request is in debug mode. However, Mobile Client needs to obtain the final implementation results of the mashup process, so its request is in the execution mode.

The process file to Visual Editor contains UI information and execution logic. First, the execution engine preprocesses the process information to extract logic information and establish the run-time object (MashStudio Execution Object, MEO for short). In fact, MEO is a tree consisted of operator nodes and data flow edge, and it is transferred with XML file. Then the execution engine maps the operator nodes in MEO to a collection of object instances in the memory and analyzes the data flow.

Definition 1. *Data Flow Object (DFO for short): DFO is intermediate data in the flow, and it stands for the edge of the flow. DFO = (Id, Status, Value), where Id is the unique identifier for DFO, Status is the state of the DFO and contains the following three states: 1. Not running, 2. Normal, 3. Exception, Value is the data value of DFO and Value is effective only when the status is 2.*

Definition 2. *Data Operator Object (DOO for short): DOO is executable node in the flow, and it consists of operator nodes that obtain data and process data. DOO = (Id, InFlowSet, OutFlowSet, Type), where Id is the unique identifier for DFO, InFlowSet is a set of input DFO, OutFlowSet is a set of output DFO, Type is the operator object specific type.*

The implementation condition of the DOO depends on the status of the DFOs in InFlowSet. Some DOOs, such as Sort, can't achieve the implementation condition until all the DFOs in InFlowSet have reached status 2, while some others just need status 2 or status 3. As defined above, we can convert the flow into graph G = (Vo, Ef, Vn, Ve)

where $Vo = \{o_i\ (oId,\ oInFlowSet,\ oOutFlowSet,\ oType) \mid oInFlowSet,\ oOutFlo \in Ef,\ 1 \leq i \leq n\}$and Vo is a set of DOO, $Ef = \{f_i \mid f_i \in DFO,\ 1 \leq i \leq n\}$, $V_n = \{o_i \mid o_i \in Vo, 1 \leq i \leq n\}$ and Vn stands for a set of not running DOOs, Ve = $\{o_i \mid o_i \in Vo, 1 \leq i \leq n\}$ and Ve stands for a set of running DOOs. Then the execution engine traverses the collection Vo to get the execution path, and the core algorithm of execution engine is described as algorithm 1.

Algorithm 1 searches not-running nodes to check if they satisfy execution condition, and then call the function of execute(O_i,type(O_i)) to execute them until the above process is over. In execution mode, we just need to return the final results of the implementation. But in debug mode, we need to gain the intermediate results and states, so we return DFO collection Ef.

3.3 The Data Analyzer

The data analyzer aims to find meaningful mashup process fragments, which can be recommended as a whole. Elmeleegy et al. in [1] recommend components based upon co-occurrence, because several components always occur together means that they have special relevance.

We collected 6000 mahup processes from YahooPipes. The data analyzer needs to pre-process these mashups to convert them into our weighted directed graph. Figure 2 shows two mashup applications' structure, and the parts in the dashed line both occur in the two structures. Our goal is to find frequent co-occurrence parts, which we called "clique", in the 6000 mashup processes. First, each kind of component has the unique identifier id, and then we convert the 6000 mashup processes into a weighted directed graph. Figure 3 demonstrates how to convert structure 1 and structure 2 into a weighted directed graph, the attributes

Algorithm 1 Execution Engine

```
Input:
    a graph G = (Vo, Ef, Vn, Ve);
 1: start:
 2: for each $O_i \in V_n$ do
 3:   if satisfyExecutionConditions($O_i$,type($O_i$),InFlowSet($O_i$)) then
 4:     $V_n = V_n - O_i$;
 5:     $V_e = V_e \bigcup O_i$;
 6:   end if
 7: end for
 8: if size($V_e$) > 0 then
 9:   while size($V_e$) > 0 do
10:     $V_e \leftarrow V_e - O_i$;
11:     execute($O_i$,type($O_i$)); // execute not-running DOO nodes;
12:     setDFO($O_i$,OutFlowSet($O_i$)); // update status and value of DFO in Ef;
13:   end while
14:   goto start;
15: end if
16: Output G;
```

on the edges show which structure the record comes from. For example, the attribute $< 1, 2 >$ at the edge of node 1 to node 4 shows that this edge both occurred in structure 1 and structure 2. Then, we process the weighted directed graph in accordance with the following five steps to discover meaningful clique. The five steps can be summed up to algorithm 2.

Definition 3. *The complex weighted directed graph G = (V, E, A), where V is the set of nodes, E is the set of edges, and A is the attributes set at edge set E. For each $e \in E$, A(e): $v_1 \rightarrow v_2, < s_1, s_2, s_3 >$. A(e) means edge e is from node v_1 to node v_2 and edge e occurs in structure $s_1s_2s_3$, and Num(A(e)) is the occurrence number of edge e. For each $v \in V$, Output(v) $= < e_1, e_2, e_3 >$ means node v has output edges $e_1e_2e_3$ in the weighted directed graph, and Input(v) $=$ $< e_4, e_5, e_6 >$ means node v has input edges $e_4e_5e_6$.*

(1) In this step we set a threshold T, for each $v_i \in V$, BFSGen(G,T,v_i) will execute the bread-first search from v_i to find components that always occur together. If the times that these components occur in the same mashup process is more than T and these components are connected, BFSGen will return them as a domain. Finally BFSGen extracts a domain list $D =< d_1, d_2, \ldots >$, $di \in D$ is a connected domain.
(2) For each $d_i \in D$, we search the leaf nodes V_{leaf} in domain d_i. For every node v in domain d_i, $v \in V_{leaf}$ if only Num(Output(v)) = 0.
(3) In every domain $di \in D$, ReverseTraverse(d_i,v) starts from each leaf node $v \in V_{leaf}$, then traverse the domain d_i along the edges pointing to current node. Each traversal, we can gain a sub-domain, and these sub-domains may contain repeated parts.

(4) In this step, we take the attributes at the edges into account. For each edge e in the sub-domain of step c, if more than half of the occurrences in $A(e) :< s_1, s_2, \ldots >$ are the same, then the edges belong to the same type. After the treatment of ExtractConnectedArea(sub-domain), we divide the sub-domain into different connected areas according to the edges' type.

(5) The set of connected areas obtained from previous step may exist duplicated items. After the operation of duplicate removal, we get the final outputs.

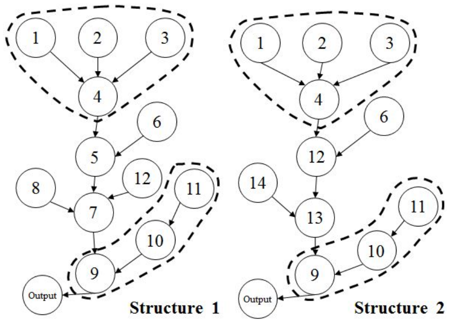

Fig. 2. Mashup Structure

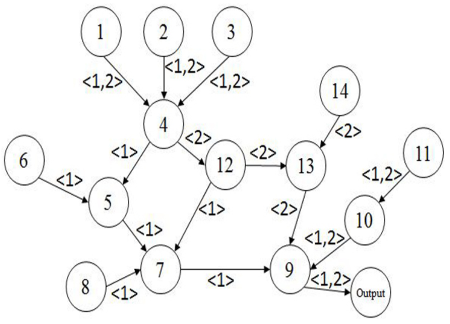

Fig. 3. Weighted Directed Graph

Algorithm 2 Clique Search

```
Input:
     a weighted directed graph G = (V,E,A) , a threshold T;
 1: Init a domain array D, a leaf node set $V_{leaf}$, a clique array Out;
 2: for each $v_i \in V$ do
 3:    connected domain $di \leftarrow BFSGen(G, T, v_i)$;
 4:    add $d_i$ to D;
 5: end for
 6: for each $d_i(V, E, A) \in D$ do
 7:    for each $v_i \in V$ do
 8:       if Num(Output($v_i$))=0 then
 9:          add $v_i$ to set $V_{leaf}$;
10:       end if
11:    end for
12:    for each $v_j \in V_{leaf}$ do
13:       sub-domain(V,E,A)←ReverseTraverse($d_i, v_j$);
14:       areasList←ExtractContectedAreas(sub-domain(V,E,A));
15:    end for
16: end for
17: Out←DuplicateRemoval(areasList);
18: Output Array Out;
```

3.4 The Data Loader

To save the recommendation response time, the Data Loader needs to preload the weighted directed graph which we constructed in the section of the data analyzer, as well as the cliques that we have obtained from algorithm 2. In this section we will also accomplish some statistical works and load the statistical results into the memory in advance, in order to further reduce response time delay of the recommendation engine.

3.5 Recommendation Engine

Given the current partial mashup, the recommendation engine will return a list of ordered cliques leveraging on the conditional probability calculation. Due to the preprocessing in the previous section, the candidate components and basic statistical results can be fast retrieved from the data loader. The cliques list L in the data loader is $\{S_1, S_2, \ldots S_k\}$. Now we suppose that the current component is C and the recommendation engine needs to recommend candidate component C^* connected to C, in the meanwhile, the input components connected to C is $\{I_1, I_2, \ldots I_i\}$, the output components connected to C is $\{O_1, O_2, \ldots O_j\}$. If $C \in S_i$, that is, C occurred in S_i as a node. Then recommendation engine return the components connected to C in clique S_i. If $C \notin S_i$, we calculate the probability that candidate C^* can be connected to component C as the following inclusion and exclusion principle formula.

$$\begin{aligned} & P\left(C^* | \bigcup_{m=1}^{i+j+1} A_m\right) \\ = & \sum_{m=1}^{i+j+1} P\left(C^* | A_m\right) - \sum_{1 \leq m1 < m2 \leq i+j+1} P\left(C^* | A_{m1} \cap A_{m2}\right) \\ & + \ldots + P\left(C^* | \bigcap_{m=1}^{i+j+1} A_m\right) \\ = & \sum_{m=1}^{i+j+1} (-1)^{m-1} [\sum_{sub \subset \{1,2,\ldots,i+j+1\}, |sub|=m} P\left(C^* | A_{sub}\right)] \end{aligned} \tag{1}$$

Suppose $\{A_1, A_2, \ldots A_{i+j+1}\}$ are component collection that contains current component C, input set I connected to C and output set O connected to C.

After obtaining the probability of the candidate C^*, we search the candidate clique Sc in the clique collection $\{S_1, S_2, \ldots S_k\}$ that contains the candidate C^*. There may be several alternative cliques Sc that all contain the candidate C^*. Suppose Sc is consisted of $\{B_1, B_2, \ldots B_n\}$, we use the formula of P $\left(C^* | \bigcup_{m=1}^{i+j+1} A_m\right) *P(\bigcup_{q=1}^{n} B_q | C^*)$ to measure the probability that S_c should be recommended to user given the partial mashup. P$(\bigcup_{q=1}^{n} B_q | C^*)$ has been calculated during the preprocessing in the data loader. Given the partial mashup, recommendation engine uses algorithm 3 to obtain the ordered candidate cliques.

4 Experiments and Results

In this section we will set up several experiments to test the recommendation performance, recommendation accuracy, rank quality, recommendation coverage

Algorithm 3 Mashup Recommendation

```
Input:
    a weighted directed graph WDG and cliques collection L;
    current component C and its input components collection I={$I_1, I_2, \ldots I_i$} output
    components collection O={$O_1, O_2, \ldots O_j$};
 1: Init a component collection S, a clique array Out1, a clique array Out2;
 2: S←FindConnectedComponentsInWDG(WDG,C);
 3: for each $clique \in L$ do
 4:     if isContained (C, clique) then
 5:         $Out1 \leftarrow Out1 + clique$;
 6:     end if
 7: end for
 8: for each $component \in S$ do
 9:     if not isIncluded (component,I,O) then
10:         for each $clique \in L$ do
11:             $clique.probability \leftarrow calculate(component, I, O, C, clique)$;
12:             $Out2 \leftarrow Out2 + clique$;
13:         end for
14:     end if
15: end for
16: sortByProbability(Out2);
17: Obj.array1= Out1;
18: Obj.array2= Out2;
19: Output Obj;
```

and diversity to show that MashStudio can provide an effective environment for rapid mashup development.

4.1 Data Sets and Experiment Setup

We collect 6000 real mashup processes from YahooPipes and divide them into two set. One set contains random selected 500 processes and is used as test set, and the rest processes are used as training set. We use the training set as the repository in MashStudio. For each mashup in the test set, we divide the mashup into two parts, and one part is used as experiment input, then we compare the the other part with the candidates that recommended by MashStudio.

The next experiments are run on a DELL desktop computer OPTIPLEX 790 with 3.10GHz Intel Core i5-2400 CPU and 4GB RAM.

4.2 Recommendation Performance

We have described our recommendation algorithm in section 3.4, the recommendation performance is related to the number of the cliques that we discovered in section 3.3, and the number of the cliques is relevant to the mashup repository scale. Figure 4 shows the average running time condition with the growth of the repository scale. Because we have accomplished some statistical works and loaded the statistical results into the memory in advance, the run time is reduced a lot, but further reduce the run time on large scale repository is necessary.

4.3 Recommendation Accuracy

This experiment is carried out to evaluate the recommendation accuracy. We use the test set to evaluate the accuracy. We divide the 500 test mashup into 5 set and every set contains 100 test mashups. As described in section 4.3, we extract partial mashup process from each test mashup as input and use the rest part RP to test our method. We use the following formula to calculate accuracy:

$$\text{Accuracy} = \frac{1}{M} \sum_{i=1}^{M} \frac{N_i}{L} \tag{2}$$

M is the set size (here is 100), L is the length of recommended items list and Ni is the number of the recommended items that are similar to RP. Then we compare the recommended result of MashStudio with random selection. Figure 5 shows that the recommended results of MashStudio are better than random recommendation. Set 4 is significantly less than the others, because MashStudio uses the repository to calculate current user's recommended list, if test set 4 contains specific users who are different from others, it may result in this condition.

4.4 Rank Quality

Rank quality is important for user experience. We carry out the same experiment as section 4.3, we divide the 500 test mashup into 5 set, then extract partial mashup process from each test mashup as input and use the rest part RP to test rank quality. By counting the RP's ranking in the recommended list to evaluate the rank quality. If RP doesn't exist in the list, we consider its ranking as the list length. We use the average rank score formula to evaluate rank quality:

$$\text{RS} = \frac{1}{M} \sum_{i \in testset} \frac{R_i}{L} \tag{3}$$

M is the size of the test set, L is the recommended item list length, Ri is the ranking of the RP in the recommended list (In fact, the recommended item may not exactly the same as RP, we measure the similarity between recommended item and RP with function $S(\alpha, \beta)$, if the similarity is bigger than a threshold, we consider that they are the same). So the smaller RS value means high quality rank. Figure 6 demonstrated that compared with random recommendation, MashStudio improved the rank quality 40.7% $\sim$ 52.7%.

4.5 Recommendation Diversity

In fact, even a recommendation with high accuracy can't ensure that the user will be satisfied with the recommended results. A good recommendation system

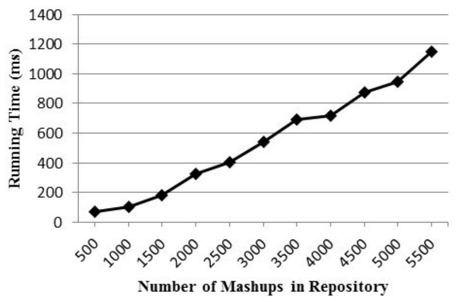

Fig. 4. Recommendation Performance

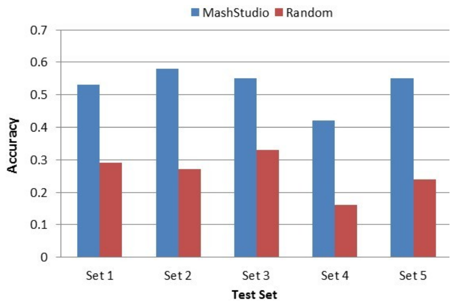

Fig. 5. Recommendation Accuracy

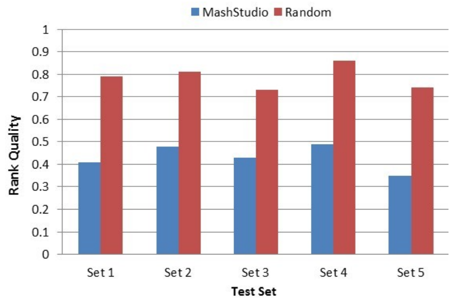

Fig. 6. Rank Quality

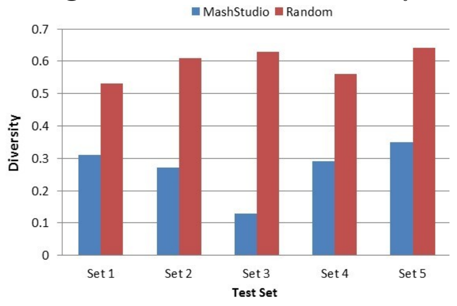

Fig. 7. Recommendation Diversity

should supply something new and useful to user. So we bring diversity to measure how many different kinds of items the recommendation system can supply.

$$\text{Div} = \frac{1}{L*(L-1)}\sum_{\alpha\neq\beta}S(\alpha,\beta) \tag{4}$$

L is the recommendation list length, α and β are different items in the recommendation list, and $S(\alpha,\beta)$ evaluates the similarity between α and β by calculating the different components in them and normalizes the result. Note that $S(\alpha,\beta)$ and $S(\beta,\alpha)$ are equal and we count it twice in the accumulation. Figure 7 illustrates that although MashStudio provides variety of recommended results to some extent, it is still obviously less than random selection. This is because MashStudio always tends to recommend candidates related with current input, and these candidates may have more common parts, while random selection never considers this. In the future, we may make an effort to recommend accurate and diverse results to users, in order to help them create mashup applications with more abundant functions.

5 Related Work

So far, many mashup development tools[2] have been built to provide assistance for non-programmer users, such as YahooPipes, Microsoft PopFly, Intel Mash

Maker and IBM DAMIA. YahooPipes and Microsoft PopFly offer a pipes-like data flow to quickly assemble data feed or other kinds of data sources. Intel Mash Maker[6] provides mashup function as a browser plugin. Mash Maker enables user to create mashup applications when browsing the websites. IBM DAMIA[3][4] is an information integration platform mainly for enterprise users. It offers support to quickly assemble data feeds from the Internet and a variety of enterprise data sources. Other similar platforms also provide visual functions to build mashup applications in various ways, such as Kongdenfha et al.[5] proposed a spreadsheet based web mashup, and Wang et al.[6] use nested tables to construct mashup applications. Recommendation system has been proven an effective way to provide better user experience in many domains like shopping and social network system. But few online mashup development tools, such as mentioned above, can provide recommendation function in the process of building workflow.

In the last several years, a lot of works about mashup recommendation have been done to simplify the mashup development process. Some research efforts focus upon semantic model, such as works in [7][8]. They first build a semantic model based upon annotation in the description and parameters, and then recommend compatible components leveraging on the semantic model. But when the data volume is large, the response real time feature is a big problem. Another related research efforts, such as proposed in [9][10][11], are directed toward the problem of finding connections between the components. When the user provides a set of components, they recommend suitable connector (so-called glue pattern) to connect the components to satisfy user's desire. Elmeleegy et al.[1] recommend components based upon co-occurrence in probability and statistics, because several components always occur together means that they have special relevance. To further simplify data mashup composition, Riabov et al.[12] provide an automatic composition engine and allows users to select tags to describe their goals, then the engine will automatically generates candidate mashups for preview. The users iteratively change the desired tags to refine their wishes until they get the satisfied results. The work presented in [13] integrates the APIs by surfing a web of data APIs to construct mashup process.

Although the above recommendation systems simplify the complexity of the mashup creation procedure in a way, they overlooked that the most users neither received any training nor the domain experts. MashStudio introduces a design-time recommendation mechanism to recommend process fragment, rather than single component, to help user rapidly construct mashup application. MashStudio also provides an on-the-fly environment to validate user's action in time. These mechanisms are significantly different from the other mashup platforms.

6 Conclusion and Future Work

This paper proposed a mashup development platform to provide design-time assistance for end user. The key attributes of this paper lie in the following

[6] http://mashmaker.intel.com/

several points. Firstly, we take the mashup process fragment into account, and recommend meaningful fragments to user to improve the development efficiency. Secondly, we provide an on-the-fly environment to validate the correctness of the user's action in real time, in order to help user realize the bug in time rather than accomplishing the whole mashup process. Finally, we introduce the mode of developing mashup on PC and using it on mobile. There is however some future works to be done to improve MashStudio, such as considering the semantic factor to further improve the accuracy of the recommendation; on the other hand, we may make an effort to supply accurate and diverse recommendation to bring something new to user. To further optimize the recommendation performance on large scale repository is also what we want to do in the future.

References

1. Elmeleegy, H., Ivan, A., Akkiraju, R., Goodwin, R.: Mashup advisor: A recommendation tool for mashup development. In: Web Services, 2008. ICWS'08. IEEE International Conference on, IEEE (2008) 337–344
2. Hoyer, V., Fischer, M.: Market overview of enterprise mashup tools. Service-Oriented Computing–ICSOC 2008 (2008) 708–721
3. Simmen, D., Altinel, M., Markl, V., Padmanabhan, S., Singh, A.: Damia: data mashups for intranet applications. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, ACM (2008) 1171–1182
4. Altinel, M., Brown, P., Cline, S., Kartha, R., Louie, E., Markl, V., Mau, L., Ng, Y., Simmen, D., Singh, A.: Damia: a data mashup fabric for intranet applications. In: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment (2007) 1370–1373
5. Kongdenfha, W., Benatallah, B., Vayssière, J., Saint-Paul, R., Casati, F.: Rapid development of spreadsheet-based web mashups. In: Proceedings of the 18th international conference on World wide web, ACM (2009) 851–860
6. Wang, G., Yang, S., Han, Y.: Mashroom: end-user mashup programming using nested tables. In: Proceedings of the 18th international conference on World wide web, ACM (2009) 861–870
7. Carlson, M., Ngu, A., Podorozhny, R., Zeng, L.: Automatic mash up of composite applications. Service-Oriented Computing–ICSOC 2008 (2008) 317–330
8. Melchiori, M.: Hybrid techniques for web apis recommendation. In: Proceedings of the 1st International Workshop on Linked Web Data Management, ACM (2011) 17–23
9. Greenshpan, O., Milo, T., Polyzotis, N.: Autocompletion for mashups. Proceedings of the VLDB Endowment **2**(1) (2009) 538–549
10. Greenshpan, O.: Harnessing data management technology for web mashups development. Proceedings of the VLDB Endowment **2**(1) (2010) 96–101
11. Deutch, D., Greenshpan, O., Milo, T.: Navigating in complex mashed-up applications. Proceedings of the VLDB Endowment **3**(1-2) (2010) 320–329
12. Riabov, A., Boillet, E., Feblowitz, M., Liu, Z., Ranganathan, A.: Wishful search: interactive composition of data mashups. In: Proceeding of the 17th international conference on World Wide Web, ACM (2008) 775–784
13. Chen, H., Lu, B., Ni, Y., Xie, G., Zhou, C., Mi, J., Wu, Z.: Mashup by surfing a web of data apis. Proceedings of the VLDB Endowment **2**(2) (2009) 1602–1605