Composition Patterns in Data Flow Based Mashups

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

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

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

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

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

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

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

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

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

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

news, lists the news with their corresponding headlines and aggregates similar news under the same headline for better readability purpose.

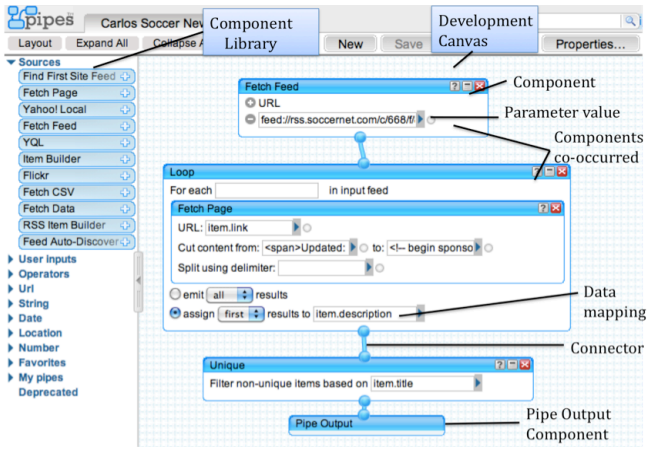

Figure 1 Implementation of the example scenario in Yahoo! Pipes.

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

3. COMPOSTION PATTERNS

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

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

Where

N- denotes the name of a mashup application.

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

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

Where

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroPLoP'11, July 13–17, 2011, Bavaria, Germany

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

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

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

O – denotes the output of a mashup application.

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

C = < I, R, Q >

Where

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

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

Q – denotes the output value for the component c_i.

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

3.1 Frequent Parameter Value

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

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

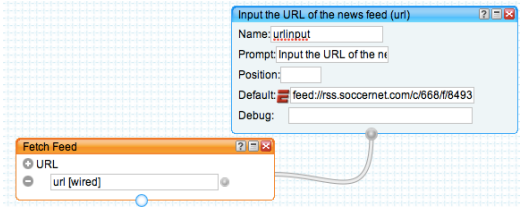

Figure 2 Example of Frequent Parameter Value

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

3.2 Associated Parameters Value

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

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

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

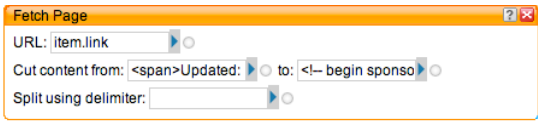

Figure 3 Example of Associated Parameters Value

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

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

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

- *Consequences:*

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

3.3 Components Co-occurrence

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

- *Example:*

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

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

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

- *Forces:*

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

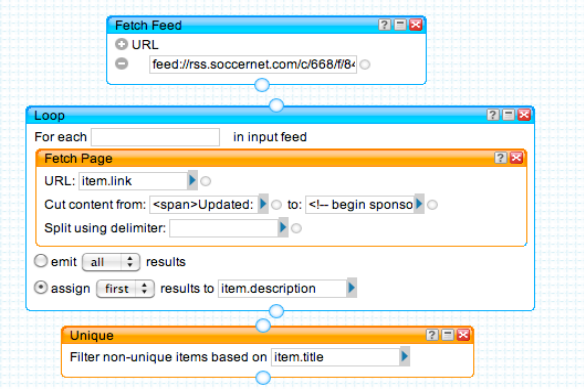

Figure 4 Example of Components Co-occurrence

  - For defining a consistent mashup, not only the co-existence of components in a given composition context but also their inter-dependencies (proper mapping of parameter value from one component to another in order to make the data-flow consistent etc.) have to be defined properly. Developers using such mashup platforms must have the background knowledge about how to satisfy these criteria while defining the data-flow logic for a mashup application.

  - Making a simple mistake in the intermediate steps during the mashup design may lead the whole application to become erroneous. At the later stage of the development, identifying such mistakes, which have occurred in the earlier steps, become difficult.

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

understanding the possible options for components, which the user may use in his composition.

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

3.4 Data-mapping

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

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

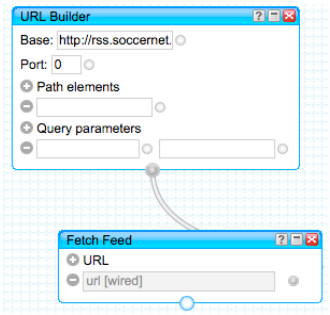

Figure 5a

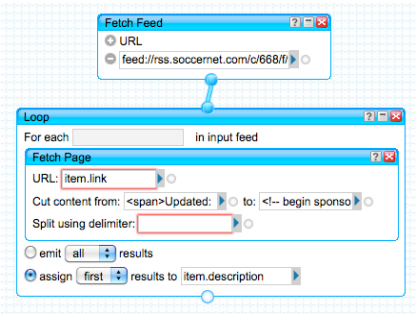

Figure 5b

Figure 5 Examples of Data-mapping patterns

- *Problem:* Defining the proper data-mapping logic requires developer to know the technical details about the data-flow logic. If the user makes a mistake in defining the data-mapping logic between the components then the whole composition logic becomes erroneous.
- *Forces:*
    - A user needs to know the type information of the input parameters for the target component as well as the type information for the output parameter of the source component. The type of these two parameters must match for the data mapping between the components.
    - When the output of one component is used as an input parameter value by more than one component, then the type of the output of the source component must match with the input of all the target components.
    - Mapping a specific value from the output set of a component to the input parameter value of another component, for example in Figure 5b, the mapping of *item.link* from the output list *Item* of *Fetch Feed* component to the *URL* parameter of *Fetch Page* is another data-mapping example. However we can observe in this example that knowing this kind of finer mapping details involves technical knowledge as well as knowledge on the data-flow logic.
    - Learning the possible relevant options from the previous application examples is not very easy for the end-users. Also this learning process requires time and expertise.
- *Solution:* To solve the problem as explained above, in data-mapping, we capture the association rule capturing the information about how the output of one component (source component) are mapped to the input parameter (configuration parameter) value of another component (Target component). The data-mapping information is captured in terms of association rules between the parameter values of the components. Let us assume, for a given pair of components (c_i ,c_{i+1}), output object q_i of c_i (Source Component) is mapped to configuration parameter P_j of c_{i+1} (Target Component). Furthermore let us assume that q_i contains set of N values as $\{ß_1, ß_2 \ldots ß_N\}$ and out of that a subset $\{ ß_{j,\ldots} ß_K \}$, where 1<=i and K<=N, can be mapped to P_i , in a given composition context. The association relation that captures the relation of $\{\{c_i, q_i[ß_{j,\ldots} ß_K]\} \rightarrow \{c_{i+1,} P_j\}\}$ with corresponding support and confidence value is stored as *data-mapping*.
- *Consequences:*
    - Given two components this pattern will help users to know in how many ways they can be connected with each other via data-mapping.
    - If the number of components increases, the possible options for their data mapping with each other increase. The viable options for the possible data mappings also become exponentially high.

3.5 Associated Composition Fragments

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

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

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

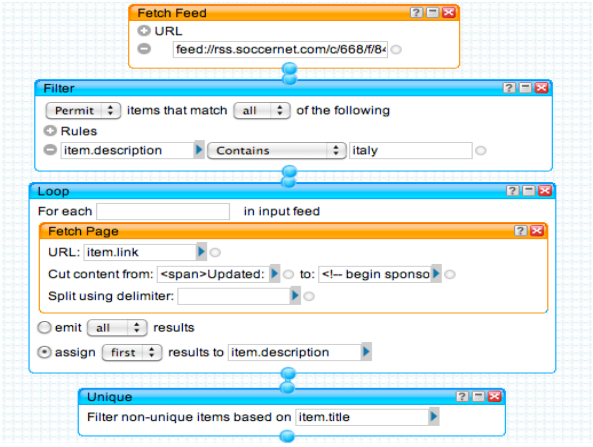

Figure 6 Example of Associated Composition Fragments (containing component/s, connector/s as a part of meaningful compositions)

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

knowledge in order to help end-users in their composition tasks.

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

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

4. DISCUSSION

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

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

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

5. LITERATURE REVIEW

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

6. Conclusion

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

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

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

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

7. ACKNOWLEDGMENTS

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

8. REFERENCES

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

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

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

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

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

[6] Jeffrey Wong and Jason I. Hong. 2007. Making mashups with marmite: towards end-user programming for the web. In Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '07). ACM, New York, NY, USA, 1435-1444

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

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

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

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

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

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

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

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

Appendix A

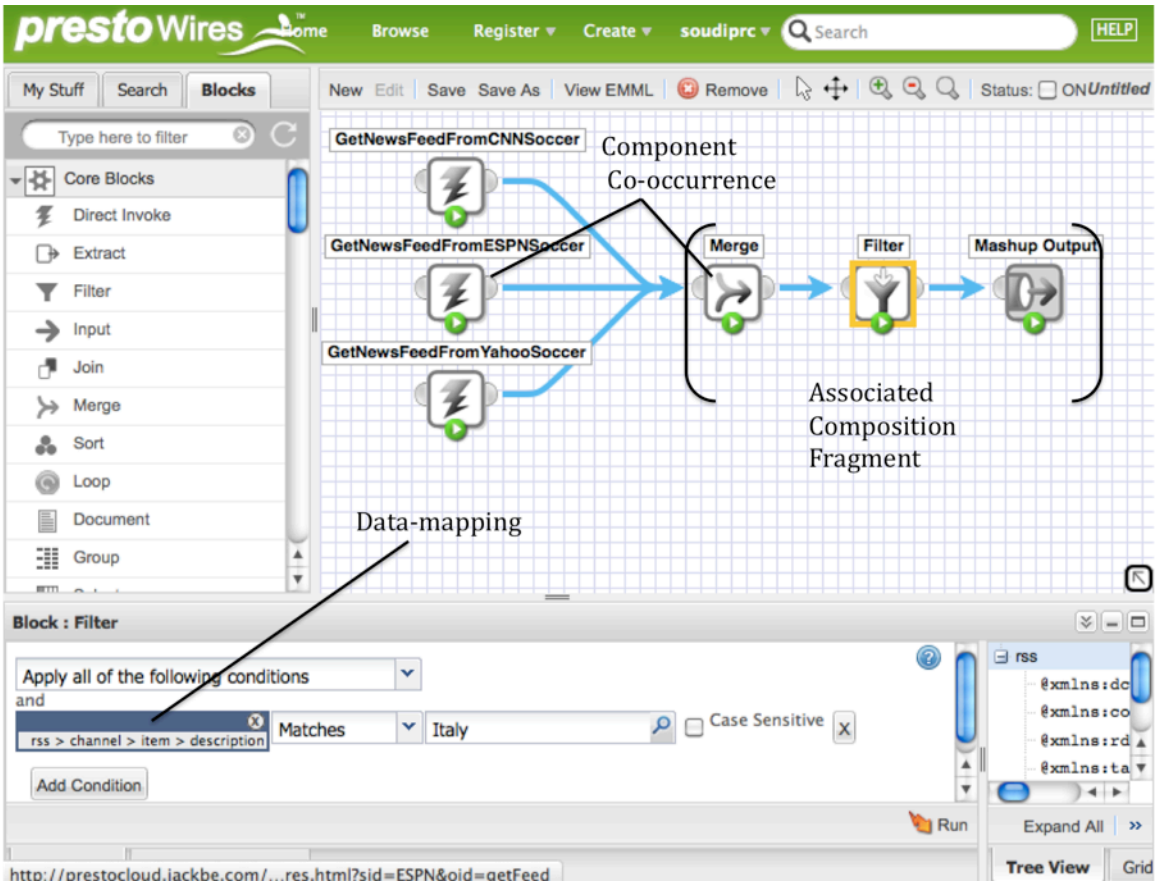

Figure 7 Composition Patterns in Presto Wires

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

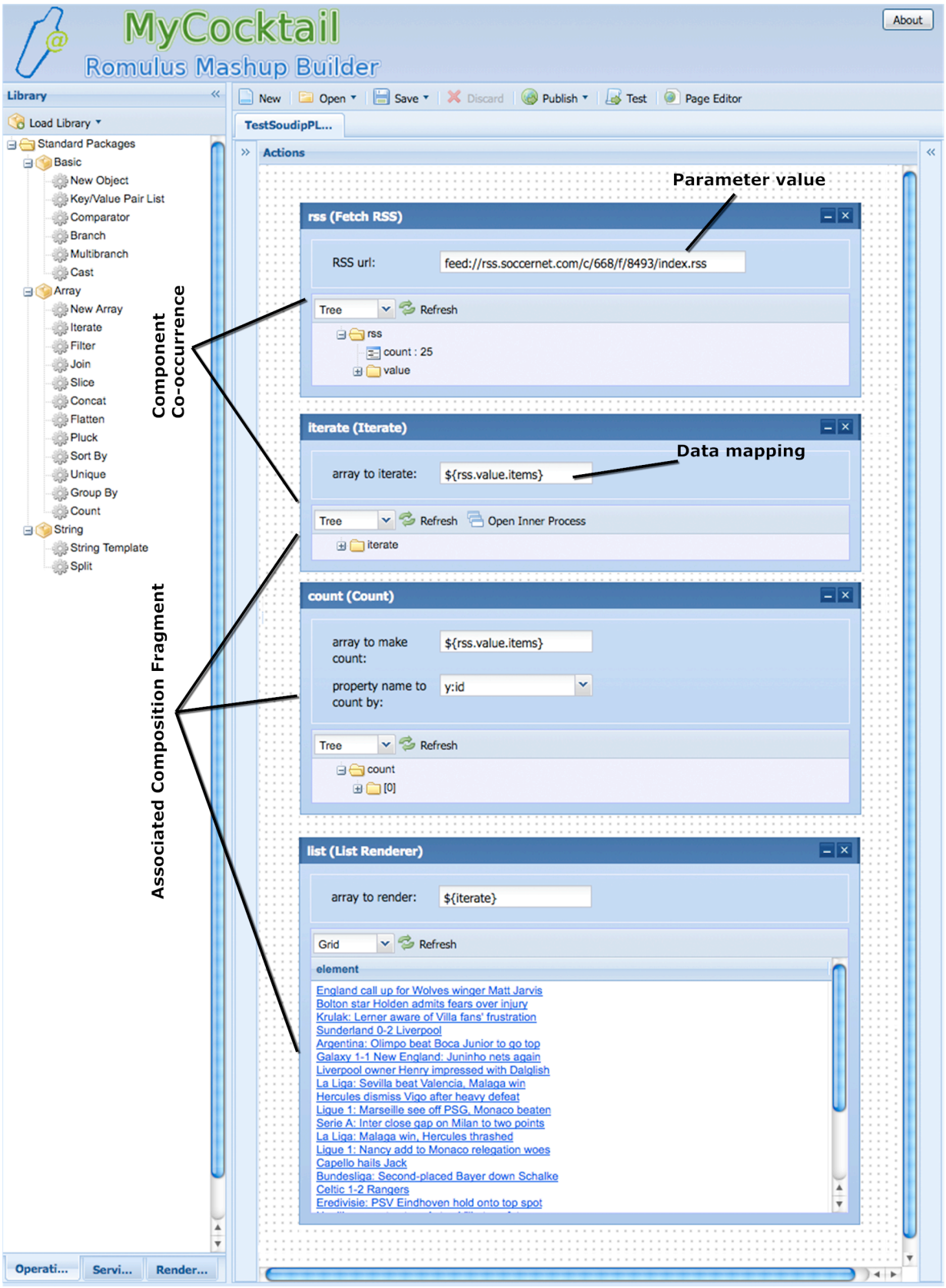

Figure 8 Composition Patterns in MyCocktail Mashup Builder