Baya: Assisted Mashup Development as a Service

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

ABSTRACT

In this demonstration, we describe *Baya*, an extension of Yahoo! Pipes that *guides* and *speeds up* development by interactively recommending composition knowledge harvested from a repository of existing pipes. Composition knowledge is delivered in the form of *reusable mashup patterns*, which are retrieved and ranked on the fly while the developer models his own pipe (the mashup) and that are automatically weaved into his pipe model upon selection. Baya mines candidate patterns from pipe models available online and thereby leverages on the *knowledge of the crowd*, i.e., of other developers. Baya is an extension for the Firefox browser that seamlessly integrates with Pipes. It enhances Pipes with a powerful new feature for both *expert developers* and *beginners*, speeding up the former and enabling the latter. The discovery of composition knowledge is provided *as a service* and can easily be extended toward other modeling environments.

Categories and Subject Descriptors

H.m [**Information Systems**]: Miscellaneous; D.1 [**Software**]: Programming Techniques; D.2.6 [**Software**]: Software Engineering—*Programming Environments*

Keywords

Baya, Assisted mashup development, Composition patterns, Pattern mining, Pattern recommendation, Weaving

1. INTRODUCTION

Mashup tools, such as Yahoo! Pipes (`http://pipes.yahoo.com/pipes/`) or JackBe Presto Wires (`http://www.jackbe.com`), simplify the development of composite applications by means of easy development paradigms (e.g., using visual programming metaphors) and hosted runtime environments that do not require the installation of any client-side software. Yet, despite the initial goal of enabling end users to develop own applications and the advances in simplifying technology, mashup development is still a *complex task* that can only be managed by skilled developers.

For instance, Figure 1 illustrates a Yahoo! Pipes model that encodes how to plot news items on a map. The example shows that understanding and modeling the logic for building such a mashup is ***neither trivial nor intuitive***. Firstly, we need to enrich the news feed with geo-coordinates, then,

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

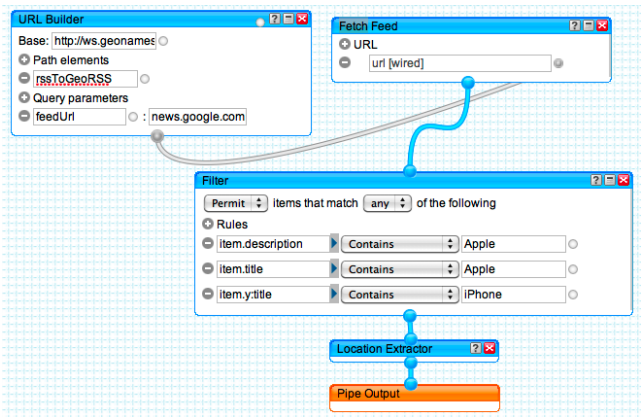

Figure 1: A typical pattern in Yahoo! Pipes

we must fetch the actual news items, and only then we can plot the items on a map. If modeling difficulties arise, it is common practice to manually search the Web for *examples* or *help* on which components to use, on how to fill the respective parameter fields, or on how to propagate data.

In order to aid less skilled developers in the design of mashups like the one above, in programming by demonstration [1], for instance, the system aims to auto-complete a process definition, starting from a set of user-selected model examples. Goal-oriented approaches [4] aim to assist the user by automatically deriving compositions that satisfy user-specified goals. Pattern-based development [3] aims at recommending connector patterns (so-called glue patterns) in response to user selected components (so-called mashlets) in order to autocomplete the partial mashup. Syntactic approaches [7] suggest modeling constructs based on syntactic similarity (comparing output and input data types), while semantic approaches [5] annotate constructs to support suggestions based on the meaning of constructs. The limitations in these approaches lie in the fact that they overlooked the perspectives for end user development, as they either still require advanced modeling skills (which users don't have), or they expect the user to specify complex rules for defining goals (which they are not able to), or they expect domain experts to specify and maintain the semantics of the modeling constructs (which they don't do).

Driven by a user study on how end users would like to be assisted during mashup development [2], we have developed ***Baya***, a plug-in for Yahoo! Pipes that provides *interactive, contextual recommendations of reusable composition knowledge*. The knowledge Baya recommends is re-usable *compo-*

sition patterns, i.e., model fragments that bear knowledge about how to compose mashups, such as the one in Figure 1. For instance, Baya may suggest a candidate next component or a whole chain of constructs. Upon selection of a recommendation, Baya weaves the respective pattern automatically into the current model in the modeling canvas[1]. Baya mines community composition knowledge from existing mashup models publicly available in the online Yahoo! Pipes repository and provides the respective patterns as a service to client-side modeling environments.

In this demo paper, we describe Baya, outline the concepts and architecture behind its simple user interface, and provide insight into its implementation and future evolution.

2. THE BAYA APPROACH

Baya aims to seamlessly extend existing mashup or composition instruments with advanced knowledge reuse capabilities. It targets both expert developers and beginners and aims to speed up the former and to enable the latter.

The ***design goals*** behind Baya can be summarized as follows: We didn't want to develop *yet another* mashup environment; so we opted for an extension of existing and working solutions (in this demo, we focus on Yahoo! Pipes; other tools will follow). We wanted to *reuse* composition knowledge that has proven successful in the past; mining modeling patterns from existing mashups allows us to identify exactly this, i.e., recurrent modeling practice. We wanted to support a variety of *different* mashup tools, not just one; as we will see, the sensible design of a so-called canonical mashup model serves exactly this purpose. Modelers should not be required to *ask* for help; we therefore pro-actively and interactively recommend contextual composition patterns. We did not want the *reuse* to be limited to simple copy/paste of patterns, but knowledge should be *actionable*, and therefore, Baya features the automated weaving of patterns.

2.1 Composition Knowledge

Considering the typical actions performed by a developer in a graphical modeling environment (e.g., filling input fields, connecting components, copying/pasting model fragments), Baya specifically supports the following set of ***pattern types***:

- ***Parameter value pattern.*** The parameter value pattern represents a set of recurrent value assignments for the input parameters of a component. This pattern helps filling input parameters of a component that require explicit user input.

- ***Connector pattern.*** The connector pattern represents a recurrent connector between a pair of components, along with the data mapping of the target component. The pattern helps connecting a newly placed component to the partial mashup model in the canvas.

- ***Connector co-occurrence pattern.*** The connector co-occurrence pattern captures which connectors occur together. The pattern also includes the associated data mappings. This pattern is particularly valuable in those cases where people, rather than developing their mashup model in an incremental but connected fashion, first select the desired functionalities (the components) and only then connect them.

- ***Component co-occurrence pattern.*** Similarly, the component co-occurrence pattern captures couples of components that occur together. It comes with the two associated components as well as with their connector, parameter values, and data mapping logic. The pattern helps developing mashups incrementally in a connected fashion.

- ***Component embedding pattern.*** The component embedding pattern captures which component is typically embedded into which other component, both being preceded by another component. The pattern helps, for instance, modeling *loops*, a task that is usually not trivial to non-experts.

- ***Multi-component pattern.*** The multi-component pattern represents recurrent model fragments that are composed of multiple components. It represents more complex patterns, such as the one in Figure 1, that are not yet captured by the other pattern types.

This list of pattern types is extensible and will evolve over time. However, this set of pattern types at the same time leverages on the interactive modeling paradigm of the mashup tools (the patterns represent modeling actions that could also be performed by the developer) and provides as much information as possible.

2.2 Discovery, Recommendation and Weaving

Figure 2 details the internals of the Baya architecture. The overall architecture is devided into two blocks, namely, the *recommendation server* and the *client-side extension* of the chosen mashup tool, i.e., Yahoo! Pipes.

The ***Baya recommendation server*** (at the left in Figure 2) is in charge of discovering and harvesting composition knowledge patterns from existing mashup compositions. The first step for discovering composition patterns consists in taking the *native models* of the target mashup tools from a repository of existing compositions and translating them into a *canonical mashup model*, a step that is performed by a dedicated *model adapter*. The canonical model is able to represent a variety of similar mashup languages and allows the development of more generic mining algorithms. The *pattern miner* runs a set of *pattern mining algorithms* on the data in the canonical model and discovers the above introduced patterns. Discovered patterns are stored back into a database of *canonical patterns*, transformed by the *data transformer*, and loaded into the *persistent knowledge base* (KB). The persistent KB consists in a database that is structured in such a way that patterns can be efficiently queried and retrieved by the *client-side browser extension* for interactive recommendation.

The ***Baya Firefox extension*** consists of two main components: a recommendation engine and a pattern weaver. In the client, we have the actual interactive modeling environment (Pipes), in which the developer can visually compose components by dragging and dropping them from a component tool bar and connecting them together in the canvas. The developer therefore performs composition *actions* (e.g., select, drag, drop, connect, delete, fill, map,...), where the

[1]This is also the capability that inspired the name of the tool: the *Baya weaver* is a so-called weaverbird that weaves its nest with long strips of leaves.

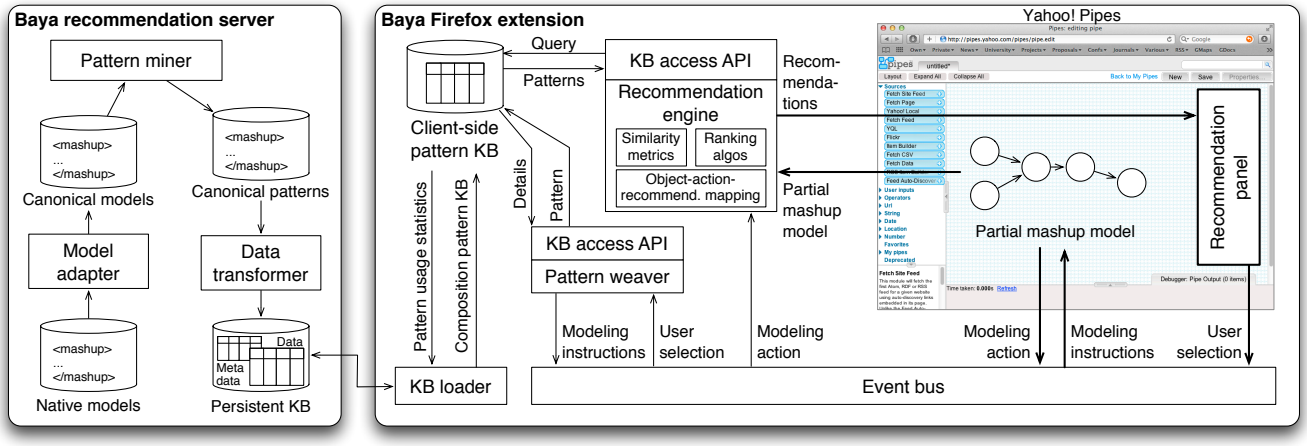

Figure 2: The internals of Baya: functional architecture for pattern discovery, recommendation and weaving

action is performed on a modeling construct in the modeling canvas; we call this construct the *object* of the action. For instance, we can *drop* a component onto the canvas, or we can *select* a parameter to fill it with a value, and so on. Upon each interaction, the *action* and its *object* are published on a browser-internal *event bus*, which forwards them to the ***recommendation engine***. Given a modeling *action*, the *object* it has been applied to, and the partial mashup model, the engine queries the *client-side pattern KB* via the *KB access API* for recommendations (pattern representations) and gets a list of candidate patterns. Baya uses both *exact* and *approximate* pattern matching algorithms [6] to determine the final candidate set of recommendations that also match the current composition context, ranks them in order of their similarity and popularity, and finally renders them in the *recommendation panel*.

Upon the selection of a pattern from the recommendation panel, the ***pattern weaver*** weaves it into the partial mashup model in the modeling canvas. For each supported pattern type, Baya retrieves a *basic weaving strategy* (a static set of modeling instructions; see `http://goo.gl/Xk7VF`), which is independent of the partial mashup model, and derives a *contextual weaving strategy*, which applies the basic strategy to the partial model at runtime. Applying the mashup operations in the basic strategy may require the resolution of possible *conflicts* among the constructs of the partial model and those of the pattern to be weaved. For instance, if we want to add a new component of type *ctype* but the mashup already contains an instance of type *ctype*, say *comp*, we are in the presence of a conflict: either we decide that we reuse *comp*, which is already there, or we decide to create a new instance of *ctype*. In order to choose how to proceed, Baya allows one to choose among different policies (see `http://goo.gl/9jJtK`). Given a final, contextual strategy, the pattern weaver applies the respective modeling actions to the partial mashup model.

Upon successful weaving of a recommended pattern into the partial composition, the *usage statistics* of the selected pattern in the client-side KB get updated, and simultaneously this information is sent to the server-side *persistent KB* via the *KB loader*. This updated metadata is used for future recommendation filtering and ranking. In the Baya client side, we also consider the option for saving patterns, in which users can select and store to the pattern KB new user-defined patterns from their current composition. This feature is part of our on-going development and will be available in future versions of Baya.

3. IMPLEMENTATION

Baya is implemented as Mozilla Firefox (`http://mozilla.com/firefox`) extension for Yahoo! Pipes, adding an interactive recommendation panel at the right of its modeling canvas. Baya implementation is based on JavaScript for the business logic (e.g., the algorithms) and XUL (XML User Interface Language, `https://developer.mozilla.org/En/XUL`) for UI development. The use of JavaScript in Firefox Extension development framework eases the interaction with the HTML DOM elements in the browser window and the implementation of dedicated listeners to intercept modeling events on elements in the DOM tree (e.g., model constructs in the Pipes modeling canvas). A screenshot of Baya in action is shown in Figure 3.

The server side is implemented in Java. This comprises the model adapter (cf. Figure 2), which is able to convert Yahoo! Pipes' internal JSON representation of mashups into our canonical mashup model as well as the necessary mining algorithms for the discovery of the patters (a description of the algorithms can be found at `http://goo.gl/Dis5V`). Parts of our mining algorithms make use of frequent itemset mining, for which we used the tool ARMiner (`http://www.cs.umb.edu/~laur/ARMiner/`).

Discovered patterns are transformed and stored in a knowledge base that is optimized for fast pattern retrieval at runtime. The implementation of the persistent pattern KB at server side, is based on MySQL (`http://www.mysql.com/`). Via a dedicated Java RESTful API, at startup of the recommendation panel the KB loader synchronizes the server-side KB with the client-side KB, which instead is based on SQLite (`http://www.sqlite.org`). The pattern matching and retrieval algorithms are implemented in JavaScript and triggered by events generated by the event listeners monitoring the DOM modifications related to the mashup model.

The weaving algorithms are also implemented in JavaScript. Upon the selection of a recommendation from the panel, they derive the contextual weaving strategy that is necessary to weave the respective pattern into the partial

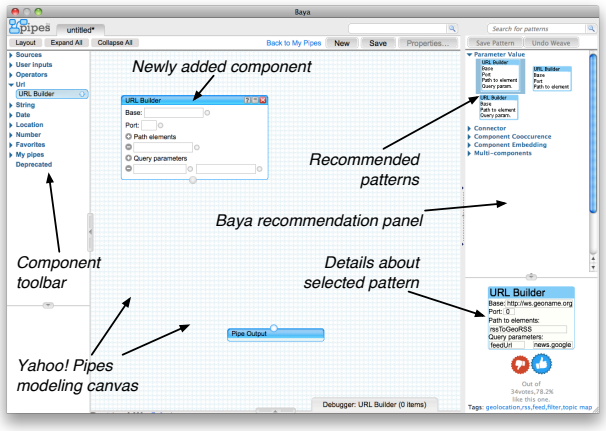

Figure 3: Screenshot of Baya in action.

mashup model. Each of the instructions in the weaving strategy refers to a modeling action, where modeling actions are implemented as JavaScript manipulations of the mashup model's DOM elements. Both the weaving strategies (basic and contextual) are encoded as JSON arrays, which enable us to use the native `eval()` command for fast and easy parsing of the weaving logic.

For our experiments we extracted 303 pipes definitions from the repository of Pipes. The average numbers of components, connectors and input parameters were 12.7, 13.2 and 3.1, respectively, indicating fairly complex mashups. We were able to identify patterns of all the types described above. For example, the minimum/maximum support for the *connector patterns* was 0.0759/0.3234, while the one for the *component co-occurrence patterns* was 0.0769/0.2308. We used these patterns to populate our KB and generated additional synthetic patterns to test the performance of the recommendation engine (the sizes of the KBs ranged from 10, 30, 100, 300, 1000 multi-component patterns) [6]. The complexity of the patterns ranged from $3-9$ components per pattern, and we used queries with $1-7$ components. In the worst case scenario (KB of 1000 patterns, approximate similarity matching of patterns), the recommendation engine could retrieve relevant patterns within 608 millisecond – everything entirely inside the client browser.

4. DEMONSTRATION STORYBOARD

During the live demonstration, we will showcase Baya at work and take our audience through the theoretical as well as the usage aspects of the tool, using a mix of slides and hands-on examples. In particular, we intend to organize the demonstration as follows:

1. ***Introduction***: A short intro to the goals and key concepts of Baya.
2. ***Example***: A simple example developed by us with the use of the interactive recommendations.
3. ***Non-assisted development by audience***: A similar modeling exercise for a member of the audience, however without the help of the interactive recommender.
4. ***Assisted development by audience***: The same modeling scenario as in 3, this time however with the help of the interactive recommender.
5. ***Patterns and discovery***: An explanation of the pattern types supported by Baya, along with the mining approach underlying the pattern knowledge base.
6. ***Architecture and internals***: Explanation of the internal architecture of Baya and of the recommendation and weaving algorithms working behind the scenes.
7. ***Conclusion***: Lessons learned and outline of future works and the evolution of Baya.

This process will allow us to introduce the audience to Baya and help us evaluate the efficacy and usability of the tool. We hope we will get valuable feedback from the audience, in order to further fine-tune Baya's UI and algorithms.

An introduction to and a screencast of Baya is available at `http://www.youtube.com/watch?v=RNRAsX1CXtE`.

5. STATUS AND LESSONS LEARNED

Baya was born in the context of the EU research project OMELETTE, in order to assist mashup development inside the project's own mashup editors. Soon, however, we recognized that the kind of knowledge discovery algorithms we were working on and the conceptual approach to pattern recommendation and weaving are generic enough to be applied in the context of many other modeling or mashup tools. As a proof of concept, we therefore developed Baya, an apparently simple, yet effective tool. The idea of composition knowledge as a service makes it unique among other assisted development approaches, and a-priori definition of pattern structures allows us to extract meaningful knowledge also from single mashup models.

Next, we will extend the mining algorithms to other composition paradigms and develop dedicated clients for different composition tools. The idea is to make Baya publicly available and to study how effectively pattern recommendation and weaving can help users to develop own mashups.

Acknowledgment. This work was supported by the European Commission (project OMELETTE, contract 257635).

6. REFERENCES

[1] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[2] A. De Angeli, A. Battocchi, S. Roy Chowdhury, C. Rodríguez, F. Daniel, and F. Casati. End-User Requirements for Wisdom-Aware EUD. In *IS-EUD'11*, pages 245–250.

[3] O. Greenshpan, T. Milo, and N. Polyzotis. Autocompletion for mashups. *VLDB'09*, 2:538–549.

[4] M. Henneberger, B. Heinrich, F. Lautenbacher, and B. Bauer. Semantic-Based Planning of Process Models. In *Multikonferenz Wirtschaftsinformatik'08*, 2008.

[5] A. Ngu, M. Carlson, Q. Sheng, and H. young Paik. Semantic-based mashup of composite applications. *IEEE TSC*, 3(1):2 –15, 2010.

[6] S. Roy Chowdhury, F. Daniel, and F. Casati. Efficient, Interactive Recommendation of Mashup Composition Knowledge. In *ICSOC'11*, pages 374–388, 2011.

[7] J. Wong and J. I. Hong. Making mashups with marmite: towards end-user programming for the web. In *CHI'07*, pages 1435–1444.