MatchUp: Autocompletion for Mashups

Serge Abiteboul [1] Ohad Greenshpan [2] Tova Milo [2] Neoklis Polyzotis [3]

[1] *INRIA Futurs & University of Paris XI*
[2] *Tel-Aviv University*
[3] *University of California, Santa Cruz*

***Abstract*— A *mashup* is a Web application that integrates data, computation and GUI provided by several systems into a unique tool. The concept originated from the understanding that the number of applications available on the Web and the need for combining them to meet user requirements, are growing very rapidly. This demo presents *MatchUp*, a system that supports rapid, on-demand, intuitive development of *mashups*, based on a novel *autocompletion* mechanism. The key observation guiding the development of *MatchUp* is that mashups developed by different users typically share common characteristics; they use similar classes of mashup components and glue them together in a similar manner. *MatchUp* exploits these similarities to predict, given a user's partial mashup specification, what are the most likely potential *completions* (missing components and connection between them) for the specification. Using a novel ranking algorithm, users are then offered top-k completions from which they choose and refine according to their needs.**

I. Introduction

A (music) mashup is a composition created from the combination of music from different songs. *Web mashups*, in a similar spirit, stem from the reuse of existing data sources or Web applications, with an emphasis on GUI and programming-less specification. As described in [1], the concept of mashups originated from the understanding that the number of applications available on the Web is growing very rapidly, and so is the need to combine them to meet user requirements. Such applications are typically complex, access large and heterogeneous data, and have varied functionalities and built-in GUIs. As a result, it often becomes an impossible task for IT departments to build them in-house as rapidly as they are requested to. The role of mashups is to facilitate this rapid, on-demand, software development task.

A mashup consists of several smaller components, namely *mashlets*, implementing specific functionalities. For instance, a mashlet may model a data source, e.g., a news RSS feed. It may implement some visual functionality, e.g., draw a map, or it may realize a specific operator, e.g., extract location information from an RSS feed input. It may also contain logic that "glues" together other mashlets, in which case we refer to it as a *glue pattern* (GP for short). As an example, a GP may combine the aforementioned three mashlets in order to present a map with the locations of recent news feeds.

Following the previous model, a user builds a mashup by selecting specific mashlets and specifying the GPs that link them. Given the large number of available mashlets, however, selecting the right components and the appropriate connections between them can be a daunting (and error-prone) task for inexperienced users. To address this issue, we draw inspiration from integrated development tools and propose the use of autocompletion. The idea is simple and intuitive: The user selects some initial mashlets that are indicative of the mashup that he/she aims to build, and the system proposes possible completions with GPs and perhaps other mashlets. The user can then select one or more of the possible completions, perform some refinements, and continue building the mashup in this iterative fashion.

Our goal is to demonstrate a system that implements the aforementioned autocompletion functionality. The system generates possible completions from a large database of real mashlets and GPs available on the Web. The important point is that it employs an intelligent recommendation engine that takes into account the incomplete specification of the user, the interactions among mashlets in the database, and also the "collective wisdom" of previous users that have successfully built mashups. Thus, even if there is no GP that links directly the mashlets selected by the user, the system will identify GPs that seem relevant to the incomplete specification, and are also favored by the user community in the creation of other mashups.

The demonstration will illustrate the proposed approach through the interactive design of an Extended Patient Health Record (xPHR) mashup. A screenshot of the (completed) mashup is shown in Figure 3. The mashup itself is a rich application that involves several mashlets and GPs. Among other things, the mashlets access various clinical data of a patient, compare them against survey data, allow the user to search the Web for doctor services and related information. The demonstration will illustrate both the user interaction with the system, showing how autocompletion greatly simplifies the mashup development task, as well as the system internals, showing the operation of the algorithms that support this useful autocompletion paradigm.

Autocompletion is a classical problem found in various domains, e.g., phrase prediction [2], email fields [3], file locations [3]. However, we are not aware of any work on autocompletion for mashups. Some related work in the context of Web services has studied how to substitute a Web service for another or how to fulfill a particular goal by composing Web services [4]. Our contribution is different: it recommends the best possible GPs or other mashlets to gradually improve a current mashup. Finally, we note that there exist several tools for the creation of mashups, e.g., MashMaker [5] and DAMIA [6]. However, none of the developed tools supports the autocompletion mechanism that we propose in this paper.

The paper is organized as follows. Section II describes the data model we use for the specification of mashlets (including GPs) and introduces the notion of mashlet *inheritance*, an important ingredient of our approach. Section III introduces the problem of mashup autocompletion and describes our solution. Section IV describes the demo scenario, and the high-level system architecture of *MatchUp*.

II. Model Ingredients

We first propose a formal model of mashups and their composition. We then discuss an important aspect of the model, inheritance. Due to space limitations, the presentation will be very brief.

A. Mashlets and Glue Patterns

The formal model has been designed to facilitate (dynamic) modular mashlets composition, interaction, inheritance and reuse. The basic components of the model are *atomic mashlets*. An atomic mashlet is a module that implements a specific functionality, and supports an interface of variables and methods that are visible to other mashlets. More concretely, an atomic mashlet has the following components:

1) Input and Output Variables: they define the input and output fields respectively of a mashlet. This constitutes the external interface of the mashlet that is manipulated by other mashlets or users in the system.
2) Mashlet data: they define local data of the mashlet. They can be specified as visible or not outside the mashlet.
3) Rules: they specify the logic implemented by a mashlet. This logic describes how the output variables are set based on the values of the input variables and the local data. One possibility is to encode this logic using datalog-style active rules, which enables taking advantage of advanced existing technology, notably query optimization. It is also possible to implement the logic using a high-level programming language such as Java or C++. In that case, the mashlet behaves like a "black box". We do not discuss further this aspect of our model, as it is not relevant to the proposed demonstration.
4) Inheritance relationship: We elaborate on this in the next section.

The left column of Figure 1 shows two example atomic mashlets named "Map" and "Yahoo! Map". The "Map" mashlet may contain input coordinate variables, such as "longitude", "latitude", and "zoom", that control the location displayed on the map. The "Yahoo! Map" mashlet may in addition contain a "view" input variable that controls whether the map displays a satellite view or a normal view.

A compound mashlet is typically composed of other (atomic or not) mashlets. Thus, in addition to the above mentioned components, a mashlet may include *imported* mashlets, as well as rules to specify how its imported mashlets interact with each other (e.g. how the output of one mashlet is transformed into the input of another). Since the main contribution of such mashlets reside in the "glue" they provide between the mashlets they use, we call them *Glue Patterns* (GPs for short).

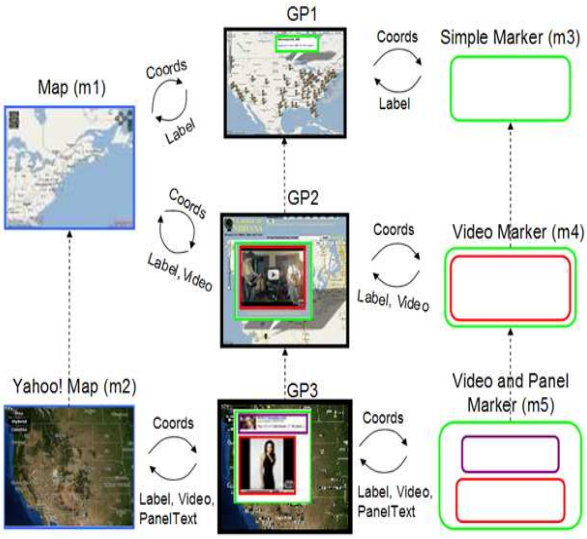

Fig. 1. Inheritance of Mashlets and Glue Patterns

Figure 1 shows three GP examples. For instance, $GP1$ combines the basic "Map" mashlet with a "Simple Marker" mashlet to display a list of locations on a map using simpler markers. $GP2$ performs the same task except that it uses the "Video Marker" mashlet for the markers. In both cases, the GP passes information from one mashlet to the other using the corresponding external interfaces.

B. Inheritance of Mashlets and their Glue Patterns

Similar to software components, mashlets may share properties with other mashlets and comply with the inheritance paradigm. As an example, observe that the "Map" and "Yahoo! Map" mashlets implement very similar functionality, and it may be actually possible to use a "Yahoo! Map" in any GP that uses a "Map" as one of its components. Based on this intuition, we analyzed in detail Programmableweb.com [7], currently the most extensive collection of mashups on the Web. This lead us to the understanding that a large number of mashups are similar to each other, in their components and in the logic they offer to users. For example, at the time of our study, 1669 mashups (39% of all mashups) included maps provided by various vendors (Google, Yahoo!, etc.). Since their characteristics are often standard, it is easy to reuse the composition logic defined for one, for another one. Even if some of the functionalities may not be enabled, the core logic should be reusable.

Motivated by the previous observation, we introduce in our model an inheritance relationship among mashlets. More specifically, mashlet m_2 inherits from mashlet m_1 if the interface of m_2 (input/output variables) is a superset of the interface of m_1. This definition of inheritance implies that mashlet m_2 can be used in any composition that employs an instance of mashlet m_1. We note that inheritance can be achieved using explicit language means, e.g., by importing the code of a mashlet and refining it in subclasses. It can also be realized by simply "cloning" the interface of a mashlet.

Similarly, our model supports inheritance among GPs. In this case, the inheritance relationship is defined based on the

mashlets linked by a GP. Informally, GP g_2 inherits from GP g_1 if it connects mashlets that inherit from those of g_1 plus possibly some additional new mashlets. As an example, $GP2$ in Figure 1 inherits from $GP1$, in the sense that $GP2$ can also link a "Map" to a "Simple Marker", and thus it can be used in any composition that uses $GP1$.

We henceforth represent a set of mashlets and GPs as a directed graph. Mashlets and GPs are represented as nodes, and GPs are connected to the mashlets that they glue together. Moreover, the graph contains inheritance edges among mashlets and GPs.

III. The Mashup Autocompletion Problem

At an abstract level, the mashup autocompletion problem can be defined as follows: Given a database of mashlets and glue patterns, and a set of mashlets selected by the user, identify and rank Glue Patterns that link a subset of the selected mashlets. Clearly, the generation of autocompletions involves two interrelated tasks:

Identification of GPs that match the selected mashlets: Intuitively, a good GP would glue all the mashlets selected by the user without introducing additional mashlets in the mashup. Such a GP, however, may not exist in the database, in which case the system should try to generate relaxations of this ideal solution. For instance, a GP may link a proper subset of the selected mashlets, or introduce additional mashlets. Another option is to use a GP that does not link the mashlets directly, but instead links mashlets they inherit from. As an example, assume that the user selects "Yahoo! Map" and "Video Marker" as the starting mashlets. As shown in Figure 1, there exists no GP that links the two mashlets directly, but it is possible to use $GP2$ since "Yahoo! Map" inherits from "Map". The downside, of course, is that $GP2$ does not take full advantage of the map's capabilities.

Ranking of Candidate GPs: By ranking candidate GPs, the system can propose to the user a meaningful short list of completions. The rank of a candidate GP intuitively depends on its "tightness", i.e., its coverage of the selected mashlets. Hence, the omission of mashlets or the introduction of additional mashlets penalize the quality of a candidate. At the same time, it is important to take into account the generality of the GP with respect to inheritance relationships. Going back to our previous example, $GP2$ should be ranked higher than $GP1$, since the latter links generalizations of both "Yahoo! Maps" and "Video Marker", whereas $GP2$ can take advantage of the capabilities of the video markers. Finally, it is important to take into account the "collective wisdom" of the user community when presenting choices to the user. For instance, $GP1$ might be more frequently used and rated as more stable by users compared to $GP2$, in which case it might have to be ranked higher even if it is a little less specific. We refer to this concept as the *static importance* of a mashlet.

The following sections describe our solution to the autocompletion problem. We first describe an algorithm to compute the top-k candidates, assuming that we are given a function Imp that reflects the static importance of a mashlet. Next, we discuss possible choices for computing Imp.

A. Identifying and Ranking Completions Efficiently

As a first step, we define a rank metric that quantifies the quality of a candidate GP relative to a set of user-selected mashlets. Our approach is to map each GP in the database to a point in a multi-dimensional space that captures the inheritance relationships in the database relative to the selected mashlets. The "ideal" GP that links just the selected mashlets is also mapped to a point in this space. The distance between this point and a GP point is used as the rank value for the GP.

Our approach is best illustrated with an example. For simplicity, we assume that all importance values are in the range $[0, 1]$. Suppose again that the user selects mashlets $m2$ and $m4$ ("Yahoo! Maps" and "Video Marker" respectively). We consider the three-dimensional unit cube, where the dimensions correspond to (1) $m2$, (2) $m4$, and (3) the glue pattern that would link the two mashlets. The ideal candidate is represented as the point $(1, 1, 1)$, meaning that it links precisely the two mashlets. The candidate $GP2$ is mapped to the point $(1-1/Imp(m1), 1, Imp(GP2))$, which is interpreted as follows. $GP2$ links $m1$ that is a generalization of $m2$, but the penalty of generalization, as measured by the deviation from the ideal coordinate value 1, depends on the importance of $m1$. Hence, the penalty is low if $m1$ is an important mashlet, as judged by the community. The second coordinate is 1 since $GP2$ takes full advantage of $m4$. Finally, the third coordinate is equal to the importance of $GP2$. The distance between $(1, 1, 1)$ and $(1-1/Imp(m1), 1, Imp(GP2))$, e.g., measured by cosine similarity or simple Euclidean distance, provides the rank of $GP2$. Hence, a candidate gets a good rank if it covers precisely all the selected patterns and the corresponding GP has a high static importance.

We can extend the previous example to GPs that omit selected mashlets by setting the respective coordinates to 0. Also, it is possible to model the introduction of mashlets, by adding a distinct dimension for each added mashlet and setting the coordinate of the ideal candidate to 0. (This transformation preserves the computation of distances to other candidates.) Longer inheritance paths can also be handled directly, by increasing the penalty of generalization with each super mashlet. Finally, it is possible to scale the dimensions so that they reflect the relative importance of mashlets, e.g., so that the omission of an important mashlet increases the penalty of the candidate. We do not discuss these technical details further, because of space limitations. Overall, the proposed metric is intuitive and has the nice property that it takes into account both the interactions of mashlets through inheritance, and also the static importance of mashlets.

We have developed an algorithm that computes efficiently the top candidates given the metric defined above. The efficient computation of the top completions is important in our setting, in order to maintain short interaction times with the user. The algorithm operates on an index of the database that is built off-line, and it is based on the general idea of threshold-based

top-k algorithms [8]. The interesting aspect of our algorithm is that it uses a non-monotonic ranking function, yet we are able to prove strong theoretical guarantees on its performance. While we omit details due to space constraints, we note that part of the demonstration will be focusing on the algorithm.

B. Computing Importance

Up to this point, we have assumed the existence of an Imp function that measures the static importance of a mashlet or GP, i.e., its quality as measured by its usage in mashups created by the user community. One obvious idea is to use the download count of a mashlet as a value for Imp, based on the intuition that importance follows the frequency of use. Another idea is to maintain an explicit rating system, where users are asked to rate mashlets based on different criteria.

We utilize a different approach that is based on the mashlet graph. The intuition is that a mashlet is important if it is referenced by important GPs, and a GP is important if it is referenced by important mashlets. This is essentially the PageRank [9] idea applied to the mashlet graph. We thus assign an initial importance to each mashlet, e.g., using the download count or an explicit rating, and then use a set of recursive equations to transfer importance along the edges of the mashlet graph. Note that the graph is much less connected than the Web graph, so one has to be a bit careful in the PageRank computation. We bias it at each stage with these initial importance values. An interesting point is that importance may flow through inheritance edges as well, i.e., a mashlet that inherits from an important mashlet may get a boost in its importance. We regulate this type of flow with a weight in the recursive equations, which allows the metric to be more or less conservative with respect to inheritance relationships.

IV. Demo Scenario

Our demo shows the *MatchUp* system that enables incremental composition of mashups, based on the autocompletion mechanism presented above.

The system architecture is shown in Figure 2. It includes a database of real mashlets (that includes the mashups of ProgrammableWeb), and a recommendation engine that tracks the user's actions and proposes GPs as possible completions. The front-end of our system is implemented using Adobe ® Flex ™2, Rich Internet Application development tool set.

The demonstration is structured as follows. First, we will show and analyze a sample of the imported mashlets using the model presented in Section II. The goal is to show the applicability of our model to real-world mashlets, and in particular to demonstrate the characteristics of GPs and atomic mashlets. Moreover, this analysis will illustrate the validity of the inheritance model which is central in our approach.

Second, we will demonstrate the development of an Extended Patient Health Record (xPHR) mashup using our tool. The development will be done incrementally, meaning that the user will place some mashlets on the screen, obtain possible completions, proceed with adding more mashlets, and so on, until the application is complete. The goal is to demonstrate the interaction between the system and the user, showing how autocompletion greatly simplifies mashup development.

The third and final part will demonstrate the workings of the underlying completion algorithm, including the multi-dimensional model for mashlet ranking and the top-k algorithm. Essentially, we will perform a fast forward replay of the previous part, showing at specific points how completions are computed and ranked. We will also demonstrate the effect of our model's parameters on the generation of completions, by comparing the completions for different settings of the parameters (e.g., with and without inheritance, or using different functions for Imp).

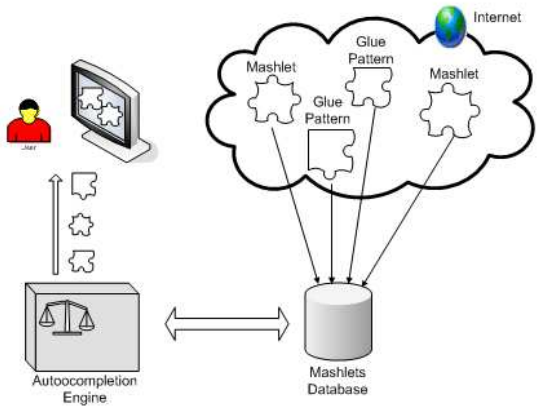

Fig. 2. MatchUp System Architecture

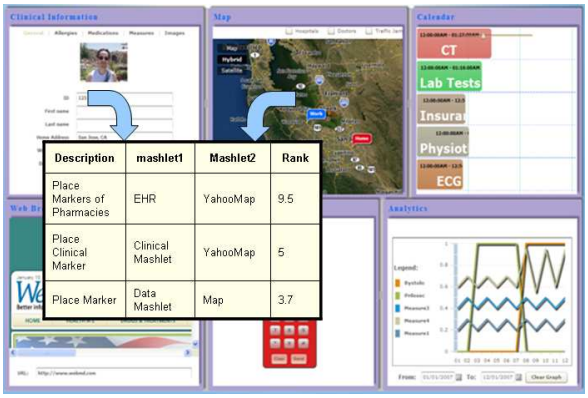

Fig. 3. Example of an autocompletion

References

[1] J. Anant, "Enterprise information mashups: Integrating information, simply," in *VLDB*, 2006.

[2] A. Nandi and H. V. Jagadish, "Effective phrase prediction," in *VLDB*, 2007.

[3] B. Myers, S. E. Hudson, and R. Pausch, "Past, present, and future of user interface software tools," *ACM Trans. CHI*, vol. 7, no. 1, 2000.

[4] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang, "Similarity search for web services," in *VLDB*, 2004.

[5] R. Ennals and M. Garofalakis, "Mashmaker: mashups for the masses." in *SIGMOD*, 2007.

[6] M. Altinel et al., "Damia - a data mashup fabric for intranet applications," in *VLDB*, 2007.

[7] "Programmableweb," http://www.programmable.com/.

[8] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *PODS*, 2001.

[9] S. Brin, R. Motwani, L. Page, and T. Winograd, "What can you do with a web in your pocket?" *Data Eng. Bulletin*, vol. 21, no. 2, 1998.