Autocompletion for Mashups*

Ohad Greenshpan
Tel Aviv University & IBM Research Labs
ohadg@il.ibm.com

Tova Milo
Tel Aviv University
milo@tau.ac.il

Neoklis Polyzotis
UC Santa Cruz
alkis@ucsc.edu

ABSTRACT

A *mashup* is a Web application that integrates data, computation and UI elements provided by several components into a single tool. The concept originated from the understanding that there is an increasing number of applications available on the Web and a growing need to combine them in order to meet user requirements. This paper 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 recommend useful completions (missing components and connections between them) for a user's partial mashup specification. The user is presented with a ranking of the recommendations from which she can choose and refine according to her needs. This paper presents the data model and ranking metric underlying our novel autocompletion mechanism. It introduces an efficient top-k ranking algorithm that is at the core of the MatchUp system and that is formally proved to be optimal in some natural sense. We also experimentally demonstrate the efficiency of our algorithm and the effectiveness of our proposal for rapid mashup construction.

1. 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 [4], 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, or 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 programmer builds a mashup by selecting specific mashlets and specifying the GPs that link them. Several mashup editors are available on the market to perform this task [12, 13, 29]. However, building a mashup-based application, as done nowadays in many IT departments (e.g. for health-care or government-support), is still a fairly complex (and error-prone) task. It involves not only finding the most suitable domain-dependent mashlet components, but more importantly, gluing them together in an effective way. This gluing is non-trivial as the names of the mashlets input/output variables are not always meaningful/uniform, they include state variables that one is not always aware of, types are inconsistent, etc. To address this problem we present MatchUp, a system that allows for 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, in similar contexts, typically share common characteristics, i.e., they use similar classes of mashup components and glue them together in a similar manner. It can thus be very helpful to use the "wisdom" of other users in order to determine the wirings among the components of a new mashup. However, a given mashlet might have been used/glued (in different contexts) in thousands of different mashups; browsing through all to identify common and suitable wirings is too time consuming. This is precisely where our system comes into play—it instantly retrieves those GPs that are potentially most relevant to the user's current needs.

We draw our inspiration from integrated development tools and propose the use of *autocompletion*. The idea is simple

*This work was partly supported by the EU project Mancoosi and by the Israel Science Foundation.

VLDB '09, August 24-28, 2009, Lyon, France

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 possibly 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.

The characteristics of mashup composition introduce two unique challenges to the problem of autocompletion that are not found in other environments. The first concerns the *identification of potentially relevant GPs*. 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 relax the requirements. 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 exact mashlets, but instead links mashlets that are similar to them. As an example, assume that the user selects "Yahoo! NY Map" and "New York Restaurants" as the starting mashlets. Even if there is no GP that links these exact two mashlets, a GP that links a generic "Yahoo! Map" mashlet with a "Restaurants" one, may still be useful here, e.g. to show the user how restaurants and maps are normally linked.

The second important challenge is the *ranking of candidate GPs* so that 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., the omission of mashlets or the introduction of additional mashlets should penalize the quality of a candidate. At the same time, it is important to also take into account the tightness of the GP with respect to inheritance relationships. Intuitively, GPs that link mashlets that are more general than those specified by the user (e.g. the generic "Yahoo! Map") take less advantage of the specific capabilities of the given mashlets (e.g "Yahoo! NY Map"). Finally, it is important to take into account the "collective wisdom" of the user community when presenting choices to the user. For instance, GPs that use the general "Yahoo! Map" mashlet might be more frequently used and rated as more stable by users compared to other GPs that use "Yahoo! NY Map", in which case the system may choose to rank the former higher even if they are a little less tight.

Of course, mashlets' semantics still need to be taken into consideration by the programmer when adopting a suggested GP. MatchUp does not try to reason about semantics in order to generate the recommendations; instead, it leverages the efforts of programmers who already spent time on understanding these semantics, in order to assist other programmers engaged in similar efforts. This takes advantage of the recent new phenomenon: massive volume of developers sharing experience.

We note that the paradigm of autocompletion has been used successfully in many domains, e.g., phase prediction, file name completion, and in integrated development environments, but these previous efforts do not address the important issues mentioned above.

Our Contributions. In this paper we present the MatchUp system that supports an autocompletion functionality for mashup composition. MatchUp uses 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, in order to generate possible completions from a large database of real mashlets and GPs available on the Web. We describe the principles of our autocompletion mechanism and its underlying algorithmic foundation. We stress that our goal is not to invent yet another mashup editor/platform, but rather to develop a generic novel technique that can be plugged into any such editor/platform [9, 13, 21, 29] to facilitate faster, intuitive, and more efficient mashup development.

The technical contributions of this paper can be summarized as follows:

- We introduce a simple generic model for describing mashlets and GPs. A key ingredient of this model is an inheritance relationship between mashlets/GPs, which in turn enables notions of generalization and relaxation as described above. The model also enables an intuitive importance metric for mashlets and GPs that captures their interaction with other mashlets/GPs and their rating according to the user community. Based on this model, we develop a formal definition of the autocompletion problem and introduce a ranking function for ranking possible completions.

- We describe an efficient algorithm to generate the top completions given a partial mashup specification. We describe the physical structures used by the algorithm and analyze its performance theoretically. One 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.

- We briefly discuss the implementation of the MatchUp system that incorporates the aforementioned model and algorithm. (A first prototype of MatchUp was demonstrated in [2].) MatchUp operates within an existing system (IBM Mashup Center platform [13]) for mashups design, which demonstrates the feasibility of our approach in practice. Note that while our implementation uses the IBM Mashup Center as the platform for mashup design, the autocompletion mechanism that we propose is generic and can similarly be used to enhance other existing mashup systems.

- We present an empirical study on the efficiency and effectiveness of the proposed framework. Our experiments demonstrate that MatchUp generates meaningful recommendations sufficiently fast to maintain an interactive user experience, even when it operates on a mashlet database that is 10× larger than the largest mashlet database available today on the Web.

The paper is organized as follows. Section 2 describes our data model and Section 3 defines the autocompletion problem. Section 4 explains how candidate completions are found and ranked, assuming some knowledge about the relative importance of mashlets and GPs. Section 5 then shows how such importance is measured. The system implementation and our experiments are described in Section 6. Finally, we consider related work in Section 7 and conclude.

2. THE MODEL

This section details a model for describing mashlets and GPs that forms the basis of the auto-completion problem defined later. The model extends the formal mashup model in [1]. We first briefly (and informally) recall the main ingredients of the model (for full description see [1]), then explain

how it is extended to capture mashlets/GPs inheritance and reuse.

Mashlets and Glue Patterns. The basic components of the model are *atomic mashlets*. A mashlet is a module that implements a specific functionality and supports an interface of variables and methods visible from other mashlets. For mathematical simplicity, the model is based on relations as in relational database systems: the state of a mashlet is maintained and represented by a set of relations, and the logic of the mashlet (which includes its interaction with the external world) is represented by a set of Datalog-like (active) rules. A distinction, however, is that the standard relational model assumes first normal form, i.e., the component of a relation is a tuple of atomic values. This restriction is relaxed here and the model is not strictly first-order, but is more in the spirit of nested relations. In particular, to model complex mashlet data, tuples in mashlet relations may contain other relations, and even entire mashlets. More concretely, an atomic mashlet has the following components:

1. Input and Output Relations: they capture 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. Internal relations: 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 relations are populated based on the values of the input relations and the local data. In the model, this logic may be encoded using Datalog-style active rules, which enables taking advantage of advanced existing technology, notably query optimization. The logic may alternatively be provided in a high-level programming language such as Java or C++. In that case, the mashlet behaves as a black box.

The left column of Figure 1 shows two example atomic mashlets named "Map" and "Yahoo! Map". The "Map" mashlet contains a *coordinate* input relation with attributes such as *longitude*, *latitude*, and *zoom*, that control the location displayed on the map. "Yahoo! Map" may contain and additional *view* input attribute, controlling 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 compound 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).

Figure 1 shows four GP examples, labeled $GP1$ - $GP4$. 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.

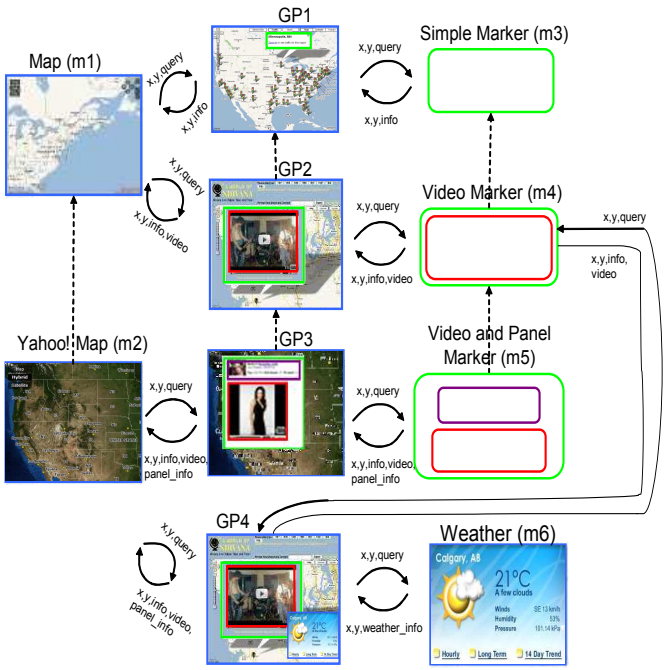


Figure 1: Inheritance of Mashlets and Glue Patterns

The model presented above includes both syntactic and semantic features for mashlets and GPs. Reasoning about semantics is clearly challenging, given also that the logic of mashlets may be implemented in different languages. Thus, our work focuses solely on the syntactic features which can be identified easily from the specification of the mashlet. Hence, in the remainder of the paper, a mashlet is modeled as its set of input and output relations, i.e., its public external interface. Similarly, a GP is modeled as the graph of connections between the input and output relations of the mashlets that it links.

Inheritance. Mashlet inheritance plays a central role in the design of mashlets and in the autocompletion mechanism we introduce in the paper. We next extend the model of [1] with such a mechanism in a rather standard manner.

The high-level observation is that, 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 [20], currently the most extensive collection of mashups on the Web. This led 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 mashlet on another similar mashlet. Even if some of the functionalities may not be enabled, the core logic should be reusable.

Following our modeling of mashlets and GPs according to syntactic features, we adopt a model of syntactic inheri-

tance that is defined with respect to the external interface of the mashlets/GPs. More specifically, a mashlet m_2 inherits from mashlet m_1 if the interface of m_2 (its input/output relations) is a superset of the interface of m_1, and for each input (resp. output) relation of m_1, the attribute-set in the corresponding relation in m_2 is a superset of that in m_1. This definition of syntactic inheritance implies that mashlet m_2 can substitute m_1 in any composition that uses an instance of m_1.

Similarly, we can define syntactic inheritance among GPs. In this case, the inheritance relationship is defined based on the mashlets linked by a GP. Formally, a GP g_2 inherits from GP g_1 if there is mapping h from the mashlets linked by g_1 to the mashlets linked by g_2 such that if $h(m) = m'$ then m' inherits from m. As an example, $GP2$ in Figure 1 inherits from $GP1$, in the sense that $GP1$ can also link a "Map" to a "Video Marker", and thus it can be used in any composition that uses $GP2$.

Clearly, the inheritance relationship can be computed in a straightforward manner based on the external interfaces of mashlets and GPs. Note also that these interfaces are already provided in most mashlet repositories, and they can be extracted automatically from the definitions of mashlets. In the following sections, we assume that the inheritance relationship among mashlets/GP is already computed. We henceforth view 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.

We note that our work does not rely on the specific type of inheritance—on the contrary, our techniques can be applied unchanged to any inheritance relationship, provided that it can be pre-computed offline. For instance, one possibility is to employ *semantic inheritance*, which is a stricter, more amorphic notion, that refers also to the goal that a mashlet/GP serves. A mashlet (resp. GP) m_2 is required to provide all the services that m_1 provides and possibly more. An example is a "Yahoo! Map" that provides all that a generic "Map" does, plus the possibility to choose whether the map displays a satellite view or a normal view. We choose to work with syntactic inheritance because it can be readily computed based on existing information in mashlet repositories, and our experiments show that it works well in practice. We intend to investigate different types of inheritance as part of our future work.

3. THE PROBLEM

We next define the autocompletion problem. We first develop some intuitions based on the informal presentation already given in introduction, and then formalize the problem.

Basic Ideas. At an abstract level, the mashup autocompletion problem can be defined as follows: Given a database of mashlets and GPs with some inheritance relationships, and a set of mashlets selected by the user, identify and rank GPs that link a subset of the selected mashlets. As explained in Section 1, the generation of recommended completions involves two interrelated tasks: identifying the GPs relevant to the users mashlets, and ranking the candidate GPs.

Given a set of user mashlets, an "ideal" GP would glue together exactly those mashlets and not more. Relaxations of this ideal solution include GPs that link some more general variants of the given mashlets, link only a subset of the selected mashlets (or their more general variants), or introduce some additional mashlets. 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 exactly these two mashlets only. 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. Alternatively, we can use $G4$, adding $m6$ to our mashup.

The rank of a candidate GP reflects the "tightness" of its coverage of the selected mashlets relative to the ideal GP, that is, how many user mashlets are omitted, how many new mashlets are introduced, and how far are the GP mashlets (w.r.t the inheritance hierarchy) from the user mashlets. 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. This "collective wisdom" can also help us choose between two, otherwise incomparable, relaxations such as $GP2$ and $GP4$.

The previous discussion indicates that the final ranking function has to take into account several factors. Indeed, the problem definition that we provide later follows this approach. More specifically, we first assume the existence of a function $Imp(m) \in [0, 1]$ that reflects the static importance of a mashlet/GP m, and which captures the collective wisdom mentioned above. Second, we assume the existence of a function $Dist(m \mapsto m') \in [0, 1]$ that quantifies the penalty of substituting m' for m based on the inheritance relationship. The penalty is low if most of the interface (input/output attributes) of m is covered by m' and hence the coverage is good, otherwise the penalty is high. (We discuss the definition of *Dist* and *Imp* later.) The final ranking function is defined in terms of Imp and $Dist$ and also captures how well a GP links the mashlets selected by the user. The idea is to map each GP in the database to a point in a multi-dimensional space based on the inheritance relationships 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 the basis to rank the GPs.

Our approach is best illustrated with an example. 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) the importance of the glue pattern that would link the two mashlets, (2) the mashlet $m2$, and (3) the mashlet $m4$. The candidate $GP2$ is mapped to the point $(1 - Imp(GP2), Dist(m2 \mapsto m1), 0)$, that is interpreted as follows. The value of the first coordinate is 1 minus the importance of $GP2$, hence is close to 0 if $GP2$ is of high importance. Let us look now at the second coordinate. Instead of using $m2$, $GP2$ links $m1$ that is a generalization of $m2$. The penalty of generalization is captured by a "distance" metric, denoted as $Dist(m \mapsto m')$, that quantifies

the penalty of using m' instead of m in a GP. *Dist* is a value between 0 and 1; the penalty is low if most of the interface (input/output attributes) of m is covered by m'. Hence, the coverage is good; the more features are missed, the higher the penalty. The third coordinate is 0 since $GP2$ takes full advantage of $m4$. The ideal GP is represented as the point $(0, 0, 0)$, meaning that it has the highest importance value and links precisely the two mashlets. The distance between $(0, 0, 0)$ and $(1 - Imp(GP2), Dist(m2 \mapsto m1), 0)$, e.g., measured by the L_2 norm, will provide the rank of $GP2$. Hence, a candidate GP gets a good rank if it covers precisely all the selected mashlets and has a high static importance.

Formal definition. We now provide a formal definition of the auto-completion problem, following the approach outlined above. We use $\mathcal{M}$ to denote the collection of mashlets and GPs in the database. In what follows, we identify mashlets and GPs in $\mathcal{M}$ using unique integer ids from the set $\{1, \ldots, |\mathcal{M}|\}$. We use $m \in \{1, \ldots, |\mathcal{M}|\}$ to denote a mashlet, and we use g to distinguish a GP. The set of mashlets linked by a GP g is denoted as $Components(g)$. Each mashlet m in $\mathcal{M}$ is associated with a value $Imp(m)$ that measures the importance of m in the mashlet Web, and a similar value $Imp(g)$ is associated with a GP g. The computation of $Imp()$ is discussed in the following section.

Let m and m' be two mashlets such that m' inherits (directly or indirectly) from m. As mentioned above, we assume the existence of a "distance" metric, denoted as $Dist(m \mapsto m')$, that quantifies the penalty of using m' instead of m in a GP. This is computed as the fraction of the attributes of the input/output relations of m' that are not present in m. The definition of inheritance guarantees that $Dist(m \mapsto m') \in [0, 1)$, with 0 denoting that m' has exactly the same public interface as m.

We assume that the user specifies a partial mashup M consisting of n mashlets $m_1, \ldots, m_n$ from $\mathcal{M}$. A GP g is a candidate *completion* for M if it can link a non-empty subset of $m_1, \ldots, m_n$. We note that g can introduce more mashlets, or it may need to generalize some of $m_1, \ldots, m_n$ in order to link them. In the latter case, we write $g(m_i)$, $i = 1 \ldots n$, to denote the mashlet $m \in Components(g)$ that is a generalization of m_i and is closest to it, i.e. $Dist(m_i \mapsto m)$ is minimal. (We allow $g(m_i) = m_i$.) If several such generalizations exist, we pick one randomly; if none exist, $g(m_i)$ is undefined, denoted by $g(m_i) = \perp$. We use $Compl(M)$ to denote the set of completions for the partial mashup M.

Each completion g in $Compl(M)$ is associated with a score $\mathcal{S}(g)$ defined as follows. Let $D = |\mathcal{M}| + 1$. We build a conceptual D-dimensional space, where dimension 0 corresponds to the set of available GPs, and each dimension d, $1 \leq d \leq |\mathcal{M}|$, corresponds to the distinct mashlet from $\mathcal{M}$ with the same id. Without loss of generality, we will assume that the user-specified mashlets correspond to dimensions $1, \ldots, n$. A candidate g is mapped to a point p_g in this space with the following coordinates:

$$p_g[0] = \frac{\max\{Imp(g') | g' \in \mathcal{M}\} - Imp(g)}{\max\{Imp(g') | g' \in \mathcal{M}\} - \min\{Imp(g') | g' \in \mathcal{M}\}}$$

$$p_g[m] = \begin{cases} Dist(m \mapsto m') & 1 \leq m \leq n \wedge\ g(m) = m' \\ 1 & 1 \leq m \leq n \wedge\ g(m) = \perp \\ 1 & n < m \leq |\mathcal{M}| \wedge\ m \in Components(g) \\ & \quad \wedge \forall\ 1 \leq m' \leq n, g(m') \neq m \\ 0 & \text{otherwise} \end{cases}$$

Similarly, we create a point p^* that represents the "ideal" candidate as follows: $p^*[m] = 0$, $0 \leq m \leq D$. The score $\mathcal{S}(g)$ of the candidate g is then measured in reverse to the distance of p_g from the ideal p^*. (Closer points get higher score). Hence, based on the definition of p_g, g is penalized for a mashlet m ($p_g[m] > 0$) either if m is a user-selected mashlet that is not linked by g, or if m is linked by g but it is not selected by the user, or if g links a generalization of m' such that $Dist(m \mapsto m') > 0$. GP g is also penalized if it has a low importance value with respect to the other GPs in the database (i.e., $p_g[0]$ approaches 1).

One possible option for function $\mathcal{S}$ is to use the L_2 norm of the vector corresponding to p_g. Regardless of the choice, it is natural to require that the scoring function $\mathcal{S}$ obeys the following property:

DEFINITION 3.1. ***[Monotonicity]*** *Function $\mathcal{S}$ is monotonic if and only if the following implication holds for any two GPs g and g': $\forall m \in \{0, \ldots, D\}(p_g[m] \leq p_{g'}[m]) \Rightarrow \mathcal{S}(g) \geq \mathcal{S}(g')$.*

The autocompletion problem can now be formulated as follows:

DEFINITION 3.2. ***[Mashup Autocompletion Problem]*** *Given a partial mashup M, a monotonic scoring function $\mathcal{S}$, and an integer $K \leq |Compl(M)|$, generate K GPs from $Compl(M)$ that have the highest scores.*

We note that certain completions may have equal scores, which implies that there are several valid solutions to the problem. However, all valid solutions correspond to the same ordered sequence of top-K scores. The last score of this sequence is of particular interest and is termed the *termination score*. Finally, it is possible to modify the problem definition to take into account preferences denoted by the user on the importance of mashlets. This is achieved in a straightforward manner by scaling the dimensions of the space of candidates according to the specified preferences. We do not consider this extension further, as it requires a straightforward extension of our techniques.

4. THE ALGORITHM

We introduce next an algorithm that can solve the aforementioned problem efficiently, provided that certain query-independent information has been materialized off-line.

We define first the information used by the proposed algorithm. Each mashlet m in $\mathcal{M}$ is associated with a set $L_m = \{(g, w) | g \in \mathcal{M} \wedge\ g : m \mapsto m' \wedge\ w = Dist(m \mapsto m')\}$. Essentially, L_m records the GPs that link m or its generalizations, along with the penalty of the generalization. We assume that the elements of L_m can be accessed in increasing order of the weight w, and we use $L_m[i]$ to denote the i-th element of L_m in that order. We also define a set

$$L_0 = \{(g, w) | g \in \mathcal{M} \wedge\ w = \frac{\max\{Imp(g') | g' \in \mathcal{M}\} - Imp(g)}{\max\{Imp(g') | g' \in \mathcal{M}\} - \min\{Imp(g') | g' \in \mathcal{M}\}}\}$$

that contains all the GPs and their importance metrics, and we assume a similar sorted access model. Clearly, these sets can be built off-line by examining $\mathcal{M}$. The sorted access can be provided by indices (primary or secondary) over the sets.

We also assume that it is possible to retrieve efficiently the set $Components(g)$ for a specific GP g. Given that a GP is likely to connect few mashlets, we can consider inlining

the definition of the GP along with its id in each list L_i, $0 \leq i \leq |\mathcal{M}|$. Otherwise, we need an additional index that can also be built off-line.

We are now ready to define the algorithm. We start with a simple variant that is correct but inefficient, and then present a refined version that overcomes this drawback.

First attempt. The initial simple autocompletion algorithm, depicted in Algorithm 1, follows the standard lines of top-k algorithms, a la TA [10]. (For the time being, we ignore the lines framed in boxes.) The algorithm accesses sequentially the lists $L_1, \ldots, L_{\mathcal{M}}$ that correspond to the database mashlets, plus list L_0. The access is round-robin. For each accessed element (g, w), the algorithm finds the definition of g, computes $\mathcal{S}(g)$, and places g to an output queue O that holds the best K GPs that have been identified thus far. Let $O[1], \ldots, O[K]$ denote the completions identified thus far, in decreasing order of their scores. The algorithm maintains a per-mashlet threshold t_i that is always set to the w component of the last accessed element (g, w). The algorithm also maintains a threshold t on the best potential score of unexamined completions. The threshold is computed as the score of a conceptual candidate g' that corresponds to a point in the multi-dimensional space, having the value t_i for the i^{th} coordinate, $i = 0 \ldots \mathcal{M}$. When $\mathcal{S}(O[K]) \geq t$, it is not possible to generate a better completion than the ones already contained in O. Hence, the algorithm terminates and returns the K completions in O.

Algorithm 1: The autocompletion algorithm

Input: A partial mashup M with mashlet ids $\{1, \ldots, n\}$;
Monotonic scoring function $\mathcal{S}$;
Number of results K.

Output: K elements of $Compl(M)$ with highest scores.

1 Initialize a priority queue O that holds at most K completions;
2 $t_m \leftarrow \infty$ for $0 \leq m \leq \mathcal{M}$;
2 $\boxed{t_m \leftarrow \infty \text{ for } 0 \leq m \leq n}$;
3 $t \leftarrow \infty$;
4 **while** $|O| < K \vee \ \mathcal{S}(O[K]) < t$ **do**
5 $\quad i \leftarrow$ next index in round-robin from $\{0, \ldots, \mathcal{M}\}$;
5 $\quad \boxed{i \leftarrow \text{next index in round-robin from } \{0, \ldots, n\}}$;
6 $\quad (g, w) \leftarrow$ next element from L_i;
7 $\quad$ Retrieve definition of g and compute $\mathcal{S}(g)$;
8 $\quad$ Insert g in O;
9 $\quad t_i \leftarrow w$;
10 $\quad$ Let g' be a conceptual GP such that $p_{g'}[m] = t_m$ for $0 \leq m \leq \mathcal{M}$;
$\quad \boxed{\text{Let } g' \text{ be a conceptual GP such that } p_{g'}[m] = t_m \text{ for } 0 \leq m \leq n \text{ and } p_{g'}[m] = 0 \text{ for } n < m \leq |\mathcal{M}|}$;
11 $\quad t \leftarrow \mathcal{S}(g')$;
12 **end**

Given the monotonicity of the score function $\mathcal{S}$ and the fact that the algorithm follows essentially the same principles as the standard TA-style algorithms, it is easy to prove that the algorithm is correct. Nevertheless, it has one significant drawback. Observe that the number of lists that the algorithm manages is very large; it is equal to the number of mashlets in the database and a typical database may contain thousands of mashlets [20]. Note also that typically, most of the database mashlets are totally unrelated to the user mashlets. It is clearly desirable to ignore those and focus on the much smaller set of mashlets directly reflecting the user's interest. This is precisely what our refined algorithm, presented next, does.

The refined autocompletion algorithm. The refined algorithm has the same definition as Algorithm 1 except that it employs the boxed lines.

As shown, it accesses only the lists $L_1, \ldots, L_n$ that correspond to *the user mashlets* (plus, as before, the list L_0 describing the GPs importance). So in lines 2 and 5, m now ranges only from 0 to n, instead of 0 to $\mathcal{M}$, as in the previous version. Although it may seem that this simple fix would meet our needs, it actually leads to an incorrect algorithm. The key observation is that even though the score function $\mathcal{S}$ is monotonic, this property does not necessarily hold when only dimensions $0, \ldots, n$ are considered. We can formalize this observation as follows: There may be two GPs g and g' s.t. $p_g[m] \leq p_{g'}[m]$ for all $m \in \{0, \ldots, n\}$, but $\mathcal{S}(g) < \mathcal{S}(g')$. This may happen because g is closer (w.r.t the inheritance relationship) to the user mashlets but also introduces many additional mashlets, whereas g' is farther (w.r.t the inheritance relationship) from the user mashlets but introduces no other mashlets.

To account for this non monotonicity, a more careful computation of the threshold t is required. This is achieved by the updated line 10 in the algorithm. Namely, the threshold is computed as the score of a conceptual candidate g' that has importance t_0, links each user mashlet m through a generalization with penalty t_m, for $1 \leq m \leq n$, and does not link *any other mashlets*. As we show next, the monotonicity of the original scoring function $\mathcal{S}$ and the sorted access model guarantee that t is still a correct threshold.

We now provide a formal proof of correctness. In what follows, we refer to the algorithm as AC^* (for AutoCompletion). We first prove a useful property of the algorithm, and then present the main correctness theorem.

LEMMA 4.1. *Let t be the threshold at the end of one iteration. Let g be a candidate GP that has not been yet examined by AC^*, i.e., it has not been encountered in any of the accessed elements. Then, $\mathcal{S}(g) \leq t$.*

PROOF. Since g has not been examined, it must appear in the unread parts of lists $L_0, \ldots, L_n$. The sorted access model implies that $p_g[m] \geq t_m$ for $m \in [0, n]$. Moreover, for any other dimension $m \in (n, |\mathcal{M}|]$, it must hold that $p_g[m] \geq 0$. Overall, it holds that $p_g[m] \geq p_{g'}[m]$ for all $m \in [0, |\mathcal{M}|]$, and thus it follows by the monotonicity property that $\mathcal{S}(g) \leq \mathcal{S}(g') \equiv t$. □

THEOREM 4.1. *Algorithm AC^* returns a correct solution.*

PROOF. Let g_K be the last completion in the output of the algorithm. It suffices to show that there exists no candidate g such that g is not in O and $\mathcal{S}(g) > \mathcal{S}(g_K)$.

Let us assume that such a candidate g exists. Clearly, g is a candidate that the algorithm has not examined yet, otherwise it would have been inserted in the output queue O. Hence, g must be an unseen candidate. Let t be the threshold when the algorithm terminates. Lemma 4.1 implies that $\mathcal{S}(g) \leq t$. Moreover, we know that $\mathcal{S}(g_K) \geq t$ due to the termination criterion of the algorithm. It follows that $\mathcal{S}(g_K) \geq \mathcal{S}(g)$, which is a contradiction. □

Optimality. Next we consider the optimality properties of the algorithm. We examine the class $\mathcal{C}$ of deterministic algorithms that operate under the same access model as AC^*. Thus, a correct algorithm A in $\mathcal{C}$ receives as input the lists $L_0, \ldots, L_n$ corresponding to mashup M, a monotonic scoring function $\mathcal{S}$, and the number K of desired results, and returns the top-K completions for M. Algorithm A can use any order (i.e., not specifically round-robin) to access the lists, and it can also use any thresholding scheme as the termination criterion. The only restriction is that the algorithm must rely on information inferred solely by the accessed elements in the lists. As a counter example, A cannot use information on the size of the lists $L_0, \ldots, L_n$, as such information is not part of the input. It might be possible to apply optimizations if such or similar statistics are available, but we do not consider this option in this paper.

Given an algorithm A in $\mathcal{C}$ and an input instance I, we define $Depth(A, I, m)$ as the number of elements that A accesses from list L_m when solving instance I. The total cost of the algorithm is defined as $Cost(A, I) = \sum_{m=0}^{n} Depth(A, I, m)$. It is interesting to note that this cost metric focuses solely on the access of lists $L_0, \ldots, L_n$ and does not take into account the cost to recover the definition of a GP g. The assumption is that the latter is expected to be small, since it involves a single index look-up, or the definition of the GP may be inlined with each list entry.

To measure the efficiency of our algorithm, compared to other algorithms in the class $\mathcal{C}$, we use the standard notion of *instance optimality*, originally introduced in [10]: We say that an algorithm $A \in \mathcal{C}$ is instance optimal within class $\mathcal{C}$ if there are constants c and c_0 such that for every input instance I, $cost(A, I) \leq c \times cost(A', I) + c_0$ where $A' \in \mathcal{C}$ is the algorithm that solves optimally I. We refer to c as the *optimality ratio* of A.

We show that AC^* is indeed instance optimal within class $\mathcal{C}$. Our analysis works in two steps. We first show that the maximum depth of AC^* must be matched by any other algorithm, and then prove the main theorem.

LEMMA 4.2. *Let A be an algorithm in $\mathcal{C}$ and let I be an input instance. Then*
$\max\{Depth(A, I, i) | 0 \leq i \leq n\} \geq \max\{Depth(AC^*, I, i) | 0 \leq i \leq n\}$.

PROOF. For convenience of notation, we define $p_j \equiv Depth(A, I, j)$ and $p_j^* \equiv Depth(AC^*, I, j)$, $0 \leq j \leq n$. Let $p_{\max} \equiv \max\{Depth(A, I, i) | 0 \leq i \leq n\}$ and $p_{\max}^* \equiv \max\{Depth(AC^*, I, i) | 0 \leq i \leq n\}$. Let also g_K denote the last GP identified in the solution of A.

The proof works by contradiction. Assume that $p_{\max} < p_{\max}^*$, i.e., AC^* does not halt after reading $p_{\max}$ rounds. Let t_j^* be the per-input thresholds at the end of round $p_{\max}$, and t^* be the overall threshold. At that point, AC^* has examined a superset of the patterns that A has examined, and thus $\mathcal{S}(O[K]) \geq \mathcal{S}(g_K)$. Since AC^* continues pulling it holds that $t^* > \mathcal{S}(O[K])$, and hence it follows that $t^* > \mathcal{S}(g_K)$.

Based on this last statement, we construct an input instance I' that leads to an error with algorithm A . Instance I' is identical to I up to depths $p_0, \ldots, p_n$. We set $L_0[p_0 + 1]$ equal to (g, t_0^*), where g is a new GP that links exactly the mashlets $m_1, \ldots, m_n$. Accordingly, we set $L_i[p_i + 1]$ equal to (g, t_i^*), $1 \leq i \leq n$. Clearly, $\mathcal{S}(g) = t^* > \mathcal{S}(g_K)$. It is straightforward to verify that I' is a valid input instance. When algorithm A executes on instance I', it will halt again at the same depths $p_0, \ldots, p_n$ since it is deterministic, but it will have missed the solution g. This is a contradiction, since the algorithm is assumed to be correct. □

THEOREM 4.2. *Algorithm AC^* is instance optimal within class $\mathcal{C}$ with an instance optimality ratio of $(n + 1)$.*

PROOF. Consider an input instance I and the corresponding optimal algorithm Opt. Define $p_{\max}^{Opt} \equiv \max\{Depth(Opt, I, j) \ | 0 \leq j \leq n\}$ and $p_{\max}^*$ analogously for AC^*. Using the previous lemma, it holds that $p_{\max}^{Opt} \geq p_{\max}^*$. We can express the cost of algorithm AC^* as follows:

$$\begin{aligned} Cost(AC^*, I) &= \sum_{0 \leq i \leq n} Depth(AC^*, I, i) \\ &\leq (n + 1) p_{\max}^* \\ &\leq (n + 1) p_{\max}^{Opt} \\ &\leq (n + 1) Cost(Opt, I) \end{aligned}$$

Thus, for the specific instance I, it holds that $Cost(AC^*, I) \leq c \times Cost(Opt, I) + c_0$, for constants $c = n+1$ and $c_0 = 0$ that do not depend on the instance. Given that the instance was chosen arbitrarily, it follows that AC^* is instance optimal with optimality ratio $n + 1$. □

Discussion. Our autocompletion algorithm accesses only a small set of lists that correspond to the user selected mashlets, and we have seen above that it is instance optimal w.r.t the class of algorithms that also access only these lists. A natural question, however, is whether AC^* maintains this property when competing against the class $\mathcal{C}'$ of algorithms that may access *all* the mashlet lists, similar to Algorithm 1.

Clearly, an algorithm in $\mathcal{C}'$ may be "lucky" and access a top-K GP that lies at the beginning of the additional lists $L_{n+1}, \ldots, L_{|\mathcal{M}|}$, but appears only further down in the lists $L_1, \ldots, L_n$ accessed by AC^*. Consequently, AC^* may perform arbitrarily worse than this lucky algorithm, which in turn implies that AC^* is not instance-optimal within $\mathcal{C}'$. This may seem negative at first, but, as the following theorem shows, it may not carry any practical significance.

THEOREM 4.3. *There is no algorithm A that is instance optimal within class $\mathcal{C}'$ with an instance optimality ratio smaller than $|\mathcal{M}|$, the number of mashlets in the database.*

PROOF. The proof is similar to the proof of Theorem 9.5 in [10]. □

In other words, even if AC^* were instance optimal within $\mathcal{C}'$, its optimality ratio for real-world mashlet databases would be so high that it would not carry any practical benefits. Moreover, our experiments, described in Section 6, demonstrate the efficiency of our algorithm in a real life setting.

5. COMPUTING IMPORTANCE

We assumed so far the existence of an Imp function measuring the static importance of mashlets/GPs. We next consider its computation.

An importance measure could be based on the number of downloads or an explicit rating system by users. Our system allows using such "base" measures but also computes importance in the style of PageRank [7]. A mashlet acquires importance from its use in important GPs; GPs similarly acquire importance from using important mashlets. An interesting aspect of our computation is that importance flows

through inheritance edges as well, i.e., a mashlet/GP that inherits from an important mashlet/GP gets a boost in its importance.

More precisely, we assume a base importance value for each mashlet m and GP g, denoted *base(m)* and *base(g)* respectively, with $\Sigma\, base(m) = \Sigma\, base(g) = 1$. We use $Isa(m)$ (resp. $Isa(g)$) to denote the set consisting of m (respectively, g) and all its direct specializations. We assume that (i) each GP includes at least one mashlet (which is the case in practice), and that (ii) one GP has all the known mashlets as components (for this we introduce a dummy GP with importance 0). We will see further where these conditions are used. We use parameters, α, β, γ, with $\alpha + \beta + \gamma = 1$, to weigh the impacts of the three facets: base measure, usage of components, and inheritance. The following two recursive equations compute the importance of a mashlet m and a GP g, respectively:

$$Imp(m) = \alpha[\sum_{\{g|m\in Components(g)\}} \frac{Imp(g)}{|\{m'|m'\in Components(g)\}|}] + \beta[\sum_{\{\hat{m}|m\in Isa(\hat{m})\}} \frac{Imp(\hat{m})}{|\{m'|m'\in Isa(\hat{m})\}|}] + \gamma[base(m)]$$

$$Imp(g) = \alpha[\sum_{m\in Components(g)} \frac{Imp(m)}{|\{g'|m\in Components(g')\}|}] + \beta[\sum_{\{\hat{g}|g\in Isa(\hat{g})\}} \frac{Imp(\hat{g})}{|\{g'|g'\in Isa(\hat{g})\}|}] + \gamma[base(g)]$$

Consider the first equation. The first summand, weighted by α, transfers importance from each GP to its components. The second summand, weighted by β, transfers importance from each mashlet to its specializations in the inheritance hierarchy. The third summand, weighted by γ, corresponds to a bias we introduce according to *base*. It can be interpreted as a transfer of importance from each mashlet to all the mashlets, according to their base importance. The second equation computes the importance of GPs simultaneously in a similar manner. We initialize $Imp_0(m) = base(m)$ and $Imp_0(g) = base(g)$ for each m and g, and then iterate computing Imp_i from Imp_{i-1} using the two equations, $i > 0$.

We next consider convergence. As this is rather standard, our presentation will be brief. This is a PageRank computation on the graph consisting of the mashlets and GPs. There are edges from m to g and conversely, if m is a component of g. There are edges from components to their specializations. The edges are weighted according to the equations above. Observe that the importance of a mashlet is transferred conservatively to GPs (via the first summand of the 2nd equation), and to other mashlets (via the last two summands of the 1st equation); and symmetrically for GPs. By Perron-Frobenius [11], such a computation converges if the graph is aperiodic and strongly connected. Aperiodicity is not an issue and strong connectivity is guaranteed by assumptions (i) and (ii) above.

In MatchUp, the values of the weights were set to be equal (1/3), following our experiments with the system (reported in Section 6.3), and the fixpoint was reached fairly rapidly.

6. IMPLEMENTATION AND EXPERIMENTS

We present an extensive experimental study of a prototype implementation of MatchUp. In what follows, we summarize the main features of our implementation, and highlight the main experimental results.

6.1 Implementation

We completed a prototype implementation of the MatchUp framework described in this paper inside the IBM Mashup Center platform [13]. The latter implements the necessary functionality for mashup design, whereas MatchUp is used to enhance the system and provide on-demand completions. The implementation of MatchUp in an actual mashup platform validates the applicability of the framework in practice.

Before describing the system architecture, let us briefly review the main features of the IBM Mashup Center platform. The platform, based on the WebSphere Application Server, is composed of two layers. The first layer, called InfoSphere Mashup Hub allows users to create XML data feeds from a variety of information sources/Web services and mix them together to create new feeds called feed mashups. Once the feeds are defined, they can be assembled into a (visual) Mashup using the second layer, called Lotus Mashups. As a proof of concept, we have augmented this second layer with our novel autocompletion mechanism, showing how it can be used to speed up the visual Mashup design. Similar principles can be applied to the first layer as well.

An information source (InfoSphere data feed/Web service) is visually represented in Lotus Mashups by a Widget (a UI component). These correspond to mashlets in our model. Widgets communicate via events. They can define *published events* (which they may send to other Widgets) and *handled events* (the events they can receive as input). These correspond, respectively, in our mashup model, to the input and output relations of the Widget mashlets. The Lotus Mashups system provides a *wiring* function to specify which published events (from a source widget) are fed into which handled events (of a target widget). This correspond in our model to the GP that glues the corresponding mashlets.

The data about the widgets (mashlets) and wirings (GPs) is stored, in an XML representation, in the platform database. The current format does not include inheritance information, and we extended it to include this data. To speed up processing, we also keep a (materialized) relational view of the data in a DB2 database, where only the mashlet/GP details that are relevant to the autocompletion computation are stored. This view is queried by the autocompletion engine, to create the mashlets lists and compute the GPs score. Our autocompletion engine is written in Java, wrapped as a web-service and deployed on the InfoSphere Mashup Hub. It is accessed by Lotus Mashups as a web-service-based feed, gets as input the mashlets selected by the user and returns as output the computed autocompletion suggestions.

The system architecture and (part of) its execution flow are depicted in Figure 2. The user selects some initial mashlets that are indicative of the mashup that he/she aims to build (1). These, along with the configurable parameter k, are sent to the autocompletion Widget as input events (2). The application server queries the mashlets database (3) to get the relevant data (4). The top-k completions are computed and displayed to the user on the screen (5). Before the user chooses to use one of the suggestions, she can ask to preview its potential effect on the current mashlets. In this case, the system highlights the "wirings", showing which data items will be output/fed into which Widget. When the GP involves mashlets that are ancestors (in the inheritance hierarchy) of the user mashlets, the system also marks the specific input/output parameters of the user mashlets that

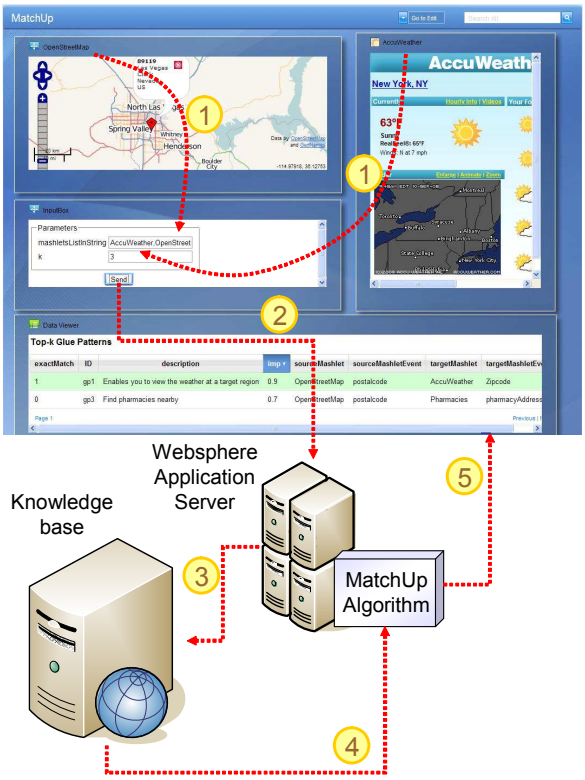


Figure 2: Architecture and flow

are not covered by the ancestor mashlets. Finally, if the autocompletion suggestion includes new mashlets, those are also automatically brought to screen. The user can then choose to adopt the suggestion as is, or refine it (e.g. by adding new wirings for the parameters not captured by the ancestor mashlets), in which case the system marks the new created GP as inheriting from the original GP.

We ran two series of experiments to validate our approach. First we tested the performance and scalability of our autocompletion algorithm. We measured its execution time on a variety of data sets, varying the main parameters that may influence the performance and quantifying their affect on the results. Second, we assessed the quality of the completion suggestions and their usefulness for mashup development. Here we asked a set of users to use the system for a particular mashup design and evaluated their satisfaction with the suggested GPs. These two series of experiments are presented next. All the experiments were performed on an IBM T60p laptop, with Pentium 1.65GHz Dual-Core and 2GB RAM. Each experiment was run ten times and the graphs show the average. But in all cases, the maximal deviation from the average was not more than 5%.

6.2 Performance

To evaluate the performance of our algorithm in a real life environment, we examined ProgrammableWeb.com, one of the main Web mashup directories, and built our data sets following its main characteristics. ProgrammableWeb.com currently contains around 1000 mashlets (called there APIs) and 3500 GPs (called there Mashups). Hence the ratio between the number of mashlets and GPs is 1:3.5, reflecting the fact that a typical mashlet is used by several GPs. The number of mashlets connected by a given GP ranges from 2 to 5, with fairly uniform distribution. Mashlets are split into categories (e.g., travel, search, maps, photos, politics) and there are currently 60 such categories of approximately the same size (around 20). Inheritance information is currently not recorded in the repository. However, an analysis of the mashlets belonging to a given category shows that they provide similar functionality and have similar interface, and essentially form inheritance hierarchies of depth ranging from 1 to 5.

We generated data sets with similar characteristics, varying the following parameters:

Number of mashlets We varied the number of mashlets in the database from 1 to 40,000 with a ratio of 1:3.5 between the number of atomic mashlets and the generated GPs.

GP structure The maximal number of mashlets connected by a GP in the database (which we refer in the following as the database *GP complexity*) was varied from 2 to 10. For each GP complexity c, the exact number of mashlets connected by a given GP in the database (a number in the range 2 to c) and the identity of the mashlets it connects, were drawn randomly with uniform distribution.

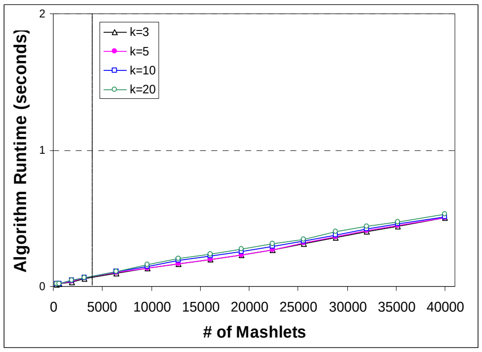
Inheritance depth Mashlets are split into sets, each consisting of 20 mashlets. A set corresponds to the notion of a category in ProgrammableWeb (e.g., travel, search, maps, etc.). We varied the maximal depth of the inheritance hierarchy within the categories from 1 to 20. For a given inheritance depth bound d, the specific depth of the hierarchy tree in a given category (a number in the range 1 to d) was drawn randomly with uniform distribution, and a corresponding tree was randomly generated.

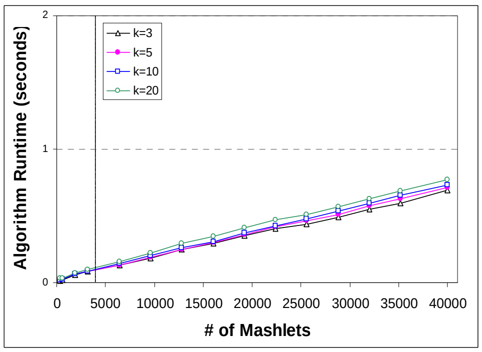
Mashlet importance To compute the importance of mashlets, we need the base importance vector and the weights α, β, γ. (See Section 5). For the base, we considered uniform and Zipfian distributions. For the weights, we experimented with a variety of values. The performance was practically insensitive to these changes. Of course, the top-k GPs depended on these choices. In the performance experiments presented below, the results were obtained with uniform distribution for the base and equal weights (i.e., 1/3 for each).

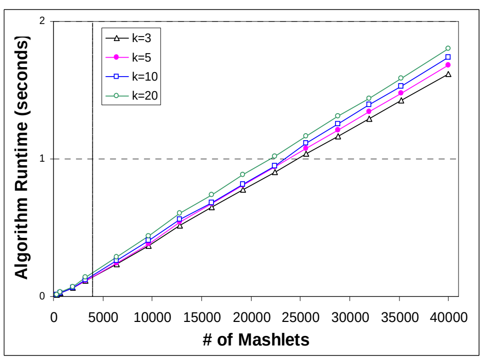
User input Finally, we varied the number of mashlets that the user places on screen from 2 to 10, and varied the number k or requested autocompletion suggestions from 3 to 20.

Let us first consider how the number of mashlets affects response time. Results for a particular set of experiments are shown in Figure 3. Consider first Figure 3(b) which depicts the running time of our top-k algorithm (in seconds) for a database with a growing number of mashlets and k values varying from 2 to 20. The maximal inheritance depth is 5 and the GP complexity is 5 too (the characteristics of ProgramableWeb.com). The number of input user mashlets is also 5. (We will consider the effect of changing these parameters further.) The vertical line in the left part corresponds to a database similar in size to that of ProgrammableWeb.com. So, clearly, the algorithm scales to a much larger number of mashlets. We can observe a moderate linear increase of the running time as the number of mashlets increases, and as k grows. But even when the data is 10 times the size of ProgrammableWeb.com, and $k = 20$, the response time is below 0.8 seconds which is more than adequate to maintain an interactive user experience.

The effect of GP complexity is demonstrated in Figures 3(a) and 3(c). In Figure 3(a) the GP complexity is 2, (namely,







(c) GP complexity 10

Figure 3: Runtime vs. #mashlets & k

mashlets are connected in pairs, as often done in IBM's Lotus Mashups) and in Figure 3(c) the GP complexity is 10. We can see that higher GP complexity implies longer computation time. This is because with high GP complexity mashlets participate in more GPs, which causes the mashlets lists manipulated by the algorithm to contain more candidate GPs. Indeed, measuring the lists length we observed that the average lengths for GP complexity 2, 5 and 10, were 35, 53, and 102, resp. But even for GP complexity 10 (double than the maximal GP complexity in ProgrammableWeb.com) the response time is below 1.8 seconds.

We next examined how the number of user mashlets affects the performance. We varied the number of mashlets that the user places on the screen from 2 to 10 and re-ran the above experiments for each case. The results were fairly uniform, showing stable performance, with only very marginal increase in computation time. Figure 4 depicts the results

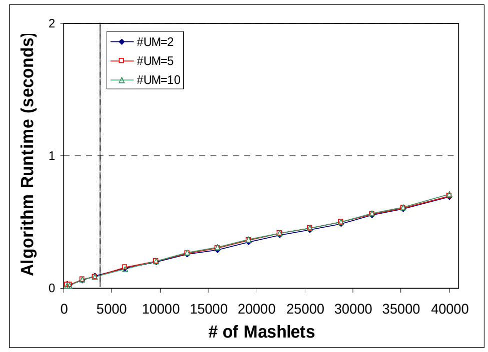


Figure 4: Runtime vs. #mashlets & User Mashlets

of a sample experiment, conducted with GP complexity 5, inheritance depth 5 and $k = 3$. The number of user mashlets (abbr. in the figure by #UM) here are $2, 5$ and 10, and we can see that the corresponding lines are practically indistinguishable.

Finally, we tested how the inheritance depth affect the performance. We varied the inheritance depth from 1 to 10 and re-ran the above experiments for each case. Again, no effect on the performance was observed since the length of the mashlets list is mostly affected by the overall number of mashlets and the connectivity of the GPs.

6.3 Quality

To assess the quality of the proposed autocompletions, we ran a second set of experiments with real users who were asked to build a travel-related mashup. We constructed a mashlets database in IBM Mashup Center, containing relevant mashlets and GPs in the spirit of what we have seen on ProgrammableWeb (e.g., including hotel and restaurant directories, weather forecasts, maps, news, and relevant GPs) as well as unrelated mashlets and GPs. For lack of available information on download rates, we set the initial importance of mashlets/GPs to reflect their relative Google PageRank. We asked the users to build a mashup incrementally, starting from a couple of mashlets that they place on screen, and followed their interaction with the system.

At each experiment we presented to the user the top-10 autocompletions computed by MatchUp. First we checked whether or not the users chose to adopt any of the suggested autocompletions for the mashup construction, and if so which ones. Second, we asked the users to browse through the database using the standard IBM Mashup Center tools and see whether they can find GPs that are more helpful/relevant for their needs than the ones suggested by MatchUp. We then asked each user to rank these GPs, together with the set proposed by MatchUp, to form their own top-10 list of relevant GPs, and measured the correlation between the ranking generated by users and the ranking generated by MatchUp.

Ten users participated in the trial. For each, we ran several experiments, varying the weights assigned to the different components of the importance formulas, and analyzing the effect on the grade. For all users except one, the best grades were obtained when the three ingredients of the importance formulas had equal weights. (For this user too the difference was only marginal.) Thus this is what we chose for our implementation.

In all the experiments (with these parameters), the users indeed chose to adopt one of the suggested autocompletions

for the mashup construction. Furthermore, the adopted autocompletions were among the top-5 proposals. Clearly, the results demonstrate the usefulness of the generated autocompletions.

The users' search for other more relevant GPs were generally unsuccessful. Only in two experiments the users identified a single additional GP, but ranked it relatively low in the list (replacing the 8th and 9th proposed GPs, resp.) Generally there was an agreement with the subsets of GPs proposed in slots 1–5 and 6–10, with the only difference being the order within these subsets. To measure the correlation between the rankings, we used Spearman's rank correlation coefficient [23], defined as $\rho = 1 - \frac{6\sum_{1,..n} d_i^2}{n(n^2-1)}$, n being the number of ranked elements and d_i the difference between the element's ranking. In the case where the users and MatchUp lists contained exactly the same set of GPs, $n = 10$ is the number of GPs in the set and d_i the difference between the rank of GP_i in the two lists. For the case where the users replaced one of the GPs, $n = 11$ with the omitted GP given the rank of 11. The grades thus range from 1 (the best case when the user's ranking is the same as MatchUp's) to -1 (reversed ranking).

Overall the grades ranged from 0.84 to 0.5, with the average being 0.81. An interesting conclusion that we drew from the analysis of the results was that the difference in ranking was typically a reflection of the user's "taste": When offered two GPs of the same importance, where the first omits some user mashlet while the second includes all mashlets but also adds some redundant one, some users consistently preferred the first option whereas others the second. Recall that MatchUp's score function views these two deviations from the user's request symmetrically, giving them the same score. In view of this analysis we consider including in the next version of MatchUp an adaptive version of the score function, where the dimensions of the multidimensional space that correspond to added/missing mashlets may be scaled to reflect this personal preference.

It is interesting to note that the gluing of mashelts in IBM's Mashup Center typically requires a fairly long sequence of steps(screens) where the relevant input and output events are individually selected and wired (six steps for the definition of a single "wiring", with a typical GP consisting of 5-6 such wirings). All this is replaced in MatchUp by a single "adopt this completion" click. So even in the case where a programmer knows in advance precisely which mashlets she wants to wire and how, adopting (even part of) the suggested autocompletion saves time. Another issue pointed out by the users was the increased level of confidence they felt for their wiring choices when these were indicated by MatchUpto be common choices, whereas they were more careful otherwise.

7. RELATED WORK

Autocompletion is a classical problem found in various domains, e.g., phrase prediction [17], email fields [16], file locations [16]. However, we are not aware of any work on autocompletion for mashups. Probably closest to our problem is that of phrase-prediction [17]. In our mashup autocompletion problem, GPs consisting of a few mashlets autocomplete the user initial mashlets; In phrase-prediction, phrases consisting of a few words autocomplete the user initial text. The two problems however have distinct properties that require different algorithmic solutions. One aspect that makes phrase prediction harder than mashup autocompletion is the lack of well defined phrase boundaries (in contrast to the well defined scope of a GP). On the other hand, an aspect that makes mashup autocompletion harder is the multi-dimensionality of the problem, where candidate GPs may not only add additional mashlets but may also alter the user input, omitting mashlets or replacing them by more general ones.

Recommendations systems have worked successfully for Web users in a variety of domains like music selection and shopping [3, 18]. A key contribution of this paper is in introducing such a recommendations mechanism for programmers, in order to simplify Web software development.

Related work in the context of Web services has studied how to substitute a Web service for another and how to compose Web services to fulfill a particular goal. The service composition in [19] gets as an input a set of abstract BPEL4WS [5] descriptions of component Web services, and a composition requirement, and automatically generates an equivalent executable BPEL4WS process that satisfies the requirement. The work in [25] performs similar processing, starting with web services described in OWL-S. In [6], a model for web-service composition is proposed, containing elements from the above mentioned works. The decidability of the composition and choreography synthesis problems is studied, providing double exponential complexity upper bounds under certain assumptions. An optimized, customized algorithm, is proposed in [15]. It uses high-level procedures and constraints to reduce the search space, but has the same worst case complexity bounds. While certainly a desirable goal, semantic composition of web services is a complex and computationally heavy task. It requires a declarative description of the services, which is often unavailable for Web mashlets, since they are coded individually by arbitrary Web users. In contrast, our lightweight autocompletion mechanisms relies solely on the syntactic signature (interface) of the mashlets/GPs and their importance, precomputed based solely on the mashlets graph and the users interest.

Of course the semantics of mashlets still need to be taken into consideration when building the mashup. Our approach is to build on the expertise of the very large body of expert programmers who have already spent time on understanding these semantics. The goal is to do efficiently the main part of the selection process. One could then imagine in a second phase using more sophisticated tools.

Our system is built on top of the IBM Mashup Center platform [13]. MatchUp uses the Mashup Center platform for Mashups design. Our new autocompletion algorithm is used to enhance the system and facilitate fast and intuitive mashup development. There exist several other tools for the creation of mashups. Damia [22] is a data integration infrastructure platform, known today as the InfoSphere Mashup Hub, which is one of the layers of IBM Mashup Center platform. Marmite [28] provides an end-user programming tool which lets end-users create mashups that repurpose and combine existing web content and services. Google Mashup Editor [12]is an Ajax-based system that enables users to assemble mashups. None of these tools provide the autocompletion mechanism that we propose in this paper. While our implementation uses the IBM Mashup center as platform for mashup design, our autocompletion mechanism is

generic and can similarly be used to enhance other existing mashup editors/systems.

Several works provide complementary tools to assist in Mashup assembly. [21] proposes a tag-based navigation technique to compose data mashups. [24] considers the extraction of valuable information from integrated data mashups. It exploits the sources semantics and thus shares similar advantages and disadvantages with the above mentioned works on Web services composition. Closest to our work is MashMaker [9] which analyzes the data currently viewed by the user to suggest widgets that might assist in handling this data. For example, the tool might suggest adding "map locations" or "distance" widgets if the user currently views a list of addresses. However it does not provide the glue to connect the proposed widgets with the present mashup (in our terminology, the best fitting GPs). Interleaving the two techniques a user would be able to browse the web, be presented with some relevant mashlets by MashMaker, glue them with our autocompletion algorithm, possibly introducing new additional mashlets or generalizing existing ones.

We have used ProgrammableWeb.com as a source for real life mashlets and GPs. Some additional available mashup directories include MashupFeed [14], which is based on the ProgrammableWeb data set and links to its mashups and APIs, the WebAPI.org website [26] that contains a relatively small collection of APIs and mashups, and Webmashup.com [27] that contains a larger data set with approximately 600 APIs and mashups, yet smaller than ProgrammableWeb.com.

8. CONCLUSION

This paper presents MatchUp, a system that enables the rapid, intuitive development of *mashups*, based on a novel autocompletion mechanism. MatchUp exploits similarities between the ways users glue together mashup components. Given a user's partial mashup specification, it recommends possible completions (missing components and connection between them) that are in some sense the best candidates for this specification. We presented the data model and ranking metrics underlying MatchUp, as well as our efficient top-k autocompletion algorithm. An 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. We also experimentally demonstrated the efficiency and effectiveness of our algorithm.

In our development, we assumed that the inheritance relationship among mashlets/GPs is given and used it to generalize the user input. While a simple approach may rely on syntactic inheritance, when the logic of mashlets/GP is given declaratively, e.g. as Datalog rules, semantic inheritance may be inferred automatically and we intend to study such automatic inference in the future. Another interesting direction for future research is distribution. In the Web context, information about mashlets/GPs may be distributed over the Web and stored in several mashlet directories. Efficient identification of relevant completions in such a distributed setting is challenging. Finally, incorporating semantics, user preferences and the context in which the mashup is developed is another interesting direction.

Acknowledgments. The authors wish to thank Serge Abiteboul for his comments and suggestions on earlier drafts of this paper.

9. REFERENCES

[1] S. Abiteboul, O. Greenshpan, and T. Milo. Modeling the mashup space. In *WIDM'08*, pages 87–94, 2008.
[2] S. Abiteboul, O. Greenshpan, T. Milo, and N. Polyzotis. Matchup - autocompletion for mashups (demo). In *ICDE'09*, 2009.
[3] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE TKDE.*, 17(6), 2005.
[4] J. Anant. Enterprise information mashups: Integrating information, simply. In *VLDB '06*, 2006.
[5] T. Andrews et al. Business process execution language for web services version 1.1. *Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems*, 2003.
[6] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *VLDB '05*, pages 613–624, 2005.
[7] S. Brin, R. Motwani, L. Page, and T. Winograd. What can you do with a web in your pocket? *Data Eng. Bulletin*, 21(2), 1998.
[8] O. de Moor, D. Sereni, P. Avgustinov, and M. Verbaere. Type inference for datalog and its application to query optimisation. In *PODS '08*, pages 291–300, 2008.
[9] R. J. Ennals and M. N. Garofalakis. Mashmaker: mashups for the masses. In *SIGMOD '07*, pages 1116–1118, 2007.
[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, pages 614–656, 2003.
[11] F. R. Gantmacher. *Applications of the theory of matrices.* Interscience Publishers, 1959.
[12] Google Mashup Editor. http://code.google.com/gme/.
[13] IBM Mashup center. www.ibm.com/software/info/mashup-center/.
[14] Mashup Feed website. http://www.mashupfeed.com/.
[15] S. Mcilraith and T. Son. Adapting golog for composition of semantic web services. In *Int. Symp. on Logical Formalizations of Commonsense Reasoning*, 2001.
[16] B. Myers, S. E. Hudson, and R. Pausch. Past, present, and future of user interface software tools. *ACM Trans. CHI*, 7(1), 2000.
[17] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB '07*, pages 219–230, 2007.
[18] netflix. http://www.netflix.com/.
[19] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite BPEL4WS web services. In *ICWS '05*, pages 293–301, 2005.
[20] Programmableweb. http://www.programmable.com/.
[21] A. V. Riabov, E. Boillet, M. D. Feblowitz, Z. Liu, and A. Ranganathan. Wishful search: interactive composition of data mashups. In *WWW '08*, pages 775–784, 2008.
[22] D. E. Simmen, M. Altinel, V. Markl, S. P., and A. Singh. Damia: data mashups for intranet applications. In *SIGMOD '08*, pages 1171–1182, 2008.
[23] C. Spearman. The proof and measurement of association between two things. *Amer. J. Psychol.*, 15:72–101, 1904.
[24] J. Tatemura, A. Sawires, O. Po, S. Chen, K. S. Candan, D. Agrawal, and M. Goveas. Mashup feeds: continuous queries over web services. In *SIGMOD '07*, pages 1128–1130, 2007.
[25] P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *Proc. of ISWC*, 2004.
[26] WebAPI.org website. http://www.webapi.org/.
[27] Webmashup.com website. http://www.webmashup.com/.
[28] J. Wong and J. I. Hong. Making mashups with marmite: towards end-user programming for the web. In *CHI '07*, pages 1435–1444, 2007.
[29] Yahoo Pipes. http://pipes.yahoo.com/.