Harnessing Data Management Technology for Web Mashups Development*

Ohad Greenshpan

Under the supervision of Prof. Tova Milo
Tel-Aviv University

ABSTRACT

Web mashups are Web applications that integrate heterogenous data sources, services and full applications, on the web. The new evolving *mashup* technology stems from the understanding that the number of assets being available on the web and the needs to combine them to meet user requirements, are growing rapidly. Mashups aim to enable *ad hoc* reuse of existing assets, the emphasis being on GUI and programmingless specification.

Our research focuses on two main dimensions of mashup construction: (1) *Mashup Design*: Help mashup designers plan, construct and develop their mashups; (2) *Mashup Usage*: Assist users to utilize mashups and exploit the potential and advantages mashups can offer. We also plan to enhance our focus in the future to *Mashup Optimization* aiming to find ways to improve mashups performance and usability.

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, services or Web applications into more complex assets, with the emphasis being on GUI and programmingless specification. The concept of mashups originated from the understanding that the number of applications available on the Web and the needs to combine them to meet user requirements, are growing very rapidly. However, these applications are often complex, provide access to large and heterogeneous data, varied functionalities and built-in GUIs, so that it becomes in many cases 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.

Mashups aim to enable *the user* ad hoc integration of a wide variety of full applications, live data sources and services, and rich navigation that involves them all, and therefore they have to be based an abstract generic model that support development of tools for mashup assembly and utilization. Since they aim to enable high customizability and adaptivity to the requirements of each and every user, they have to be modular and be based on high inference and recommendation techniques. Since they have high level of interaction with the end-user in real time, they have to be adaptive to changing needs. All these requirements call for advanced set of solutions that are based on sound theoretical foundations. Let us illustrate things with an example.

Example. A simple example of a mashup is shown in Figure 1. The mashup is composed of basic components that we call *mashlets*. A mashlet can be GUI-based (e.g., a widget) or not (e.g., Web Service or RSS stream). To allow for modular application development, a mashup can itself serve as a mashlet in some other (more complex) mashups. The screenshot here (see Figure 1) contains the following mashlets: an Electronic Health Record (EHR) mashlet (at the top left corner), a map, a weather status application, a calendar, a medical search engine, an SMS mashlet, and a medical data analyzer. Once connected, these mashlets could interact in many ways and therefore enrich the functionality and experience mashups users have. For instance, addresses of doctor appointments can be pulled from the *calendar* and then be fed into the *map* for display; The appointment times can be retrieved from the *calendar* and sent as *SMS* reminders to phone numbers taken from the patient *EHR*. This may be done automatically by the application (e.g. in response to some events), or on user demand (e.g. by drag and dropping a calendar entry on the SMS mashlet).

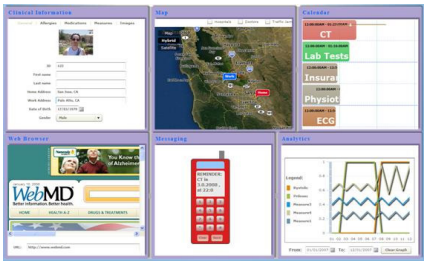

Figure 1: Example Mashup

We will now describe several key challenges that we identified in mashup development. These will be the main focus of our research.

• Mashup Design: This is the planning phase in which the mashup is composed. Since mashups combine already-given

*This research was partially supported by the Israeli Ministry of Science, the Israel Science Foundation, the US-Israel Binational Science Foundation, and IBM Research.

VLDB 2010 PhD Workshop, September 13, 2010, Singapore

applications, services and data sources, mashup designers first need to find the relevant components that suit best their goal, out of all available components on relevant repositories. Then, they need to learn their specification and decide how these should be glued to one another to form a mashup. This will be done by defining what data would be transferred and when. Since mashups are widely developed on the web for various purposes, it is desirable to provide users the ability to make use of work done by other designers who encountered similar problems. This described process is complicated since it involves both the requirements and constraints the mashup designer addresses, as well as experience of other designers encountering similar problems and inventing similar solutions. We focus on providing tools to facilitate this process for mashup designers.

• Mashup Usage: Mashups integrating existing web components, from different types, under a single application, enrich user experience and navigation, but at the same time might impose burden on their users. Users working with composed mashups need tools that will help them orient better, study their mashups' capabilities and understand how these can be used to accomplish various tasks. They need tools which would assist them navigate in the mashed-up assets, query relevant data and perform actions. We will focus on development of such tools to facilitate mashup usage.

• Mashup Optimization: Since mashups enable integration of large amount of remote assets at real-time, optimization is required to make composed mashups perform better, more effectively and with minimal use of resources. We will explore ways to enable optimization of the resources mashups consume and optimization of mashup usability to users.

The solutions that we proposed to some of the challenges described above were presented in [1, 2, 13, 6], some are still under development and some are planned for future work. In the following sections we will describe what has been achieved, and what is yet planned to be explored.

2. MASHUP MODEL

In order to facilitate development of solutions for the challenges mashups pose, a formal model that captures well the notion of mashup in its globality, needs to be devevloped. Since there was no appropriate model that covers all aspects of mashups, this was our first contribution of our research. Previous works have typically focused on *specific* aspects of mashups. Among those, one can list works dealing with service composition [21, 22], works that studied (semantic) data integration [3], and works considering interaction between mashup components and interaction with users [9].

Each such aspect is clearly interesting in itself. However, we believe that it is also essential to understand the notion of mashup in its globality, and in particular the interaction between the various facets previously mentioned.

We next briefly overview the main components of our Mashup model. For space constraints, we mostly focus on key components that will be used in the following sections. Full details can be found in [1].

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

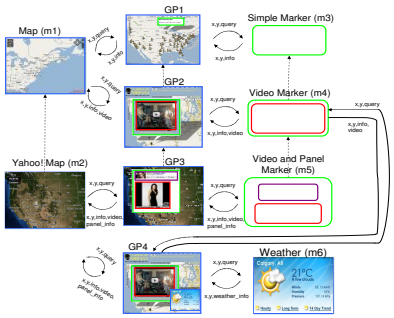

Figure 2: Inheritance of Mashlets and Glue Patterns

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:

• 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.

• Internal relations: they define local data of the mashlet. They can be specified as visible or not outside the mashlet.

• 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. In that case, the mashlet behaves as a black box.

The left column of Figure 2 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 2 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 but 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.

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,

the work presented in the following section focuses mostly on the syntactic aspects of the mashlet, i.e. its input and output interfaces, which can be identified easily from its specification. 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.

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, 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.

One can define syntactic or semantic notions of inheritance that express how mashlets/GPs relate to one another. We focus here on the syntactic notion that bases the inheritance relationship on their external interface, since these are usually exposed by developers following the mashup paradigm.

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 2 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 $GP2$.

3. MASHUP DESIGN

Following the described 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 [19, 14]. 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 developed MatchUp, a system that allows for rapid, on-demand, intuitive development of *mashups*, based on a novel *autocompletion* mechanism.

3.1 Our Approach

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 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 possible completions, refine them, and continue building the mashup in an iterative fashion. Given a database of mashlets and GPs with some inheritance relationships, and a set of mashlets selected by the user, the goal would be to identify and rank GPs that link a subset of the selected mashlets.

The mashup autocompletion problem needs to address two unique challenges. 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. 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 take less advantage of these mashlets specific capabilities. Finally, it is important to take into account the "collective wisdom" of the user community when presenting choices to the user.

3.2 Algorithm

We introduce a very high-level description of our algorithm that can solve the aforementioned problem efficiently. The detailed description, along with proof of correctness and optimality, can be seen in [13].

Setting and Definitions. As described above, we assume that we have a database of database of mashlets and GPs with some inheritance relationships, and a set of mashlets selected by the user.

We then model each GP in the database to a point in a

multi-dimensional space, having scores that reflect its popularity and relevance (either direct or the one that takes into account inheritance relationship) to the user 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.

We store these scores as follows: Each mashlet m in $\mathcal{M}$, the repository of all mashlets and GPs, is associated with a $GPset$, L_m, that records the GPs that link the mashlet m or its generalizations (as explained in the Inheritance section), along with the penalty of the generalization.

DEFINITION 1. *For a given mashlet m, the GPSet L_m is a set of (g, w) where g is a GP that links m with genelization penalty of w. Formally, $L_m = \{(g, w) | g \in \mathcal{M} \wedge \ g : m \mapsto m' \wedge \ w = Dist(m \mapsto m')\}$.*

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 the $ImportanceSet$, L_0, that contains all GPs and their importance metrics, and we assume a similar sorted access model.

DEFINITION 2. *The ImportanceSet, L_0, is the set of (g, w) that stores each GP g, along with its distance in importance from the GP with the highest importance. Formally, $L_0 = \{(g, w) | g \in \mathcal{M} \wedge$*

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

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.

Following these definitions, each GP g is a multi-dimensional point of which scores are held in the $ImportanceSet$, L_0, and the $GPSets$ of all mashlets in the repository $\mathcal{M}$.

Our algorithm is a version of top-k algorithms, a la TA [11], with some improvements that address the special characteristics of our problem that might appear in other similar problems in Web environments. We will discuss these characteristics after we provide a brief sketch of the algorithm.

Initial Approach. Our basic 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 contained in O. Hence, the algorithm terminates and returns the K completions in O.

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 [**?**]. Note also that typically, most 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.

Improved Approach. Our improved algorithm has the same definition as Algorithm 1 except that it employs the boxed lines. 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). In order for the algorithm to still provide correct results, a more careful computation of the threshold t is required. 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*. Therefore, we assign t with maximum penalty, in the lists that correspond to mashlets that are not part of *the user mashlets*. The proof of correctness, along with discussion on its optimality, is given in [13].

MatchUp. These algorithms served as a basis for the development of *MatchUp*, a system that facilitates mashup design. The system enables the user to place some initial mashlets in her mashup and ask the system for autocompletion, i.e. mashlets and GPs that would best fit. MatchUp was presented in [2, 13].

4. MASHUP USAGE

Up until now, we focused on techniques for mashup design. As described in section 1, another important aspect of mashup construction is *Mashup Usage*. Due to the complexity of mashup applications, users often find it difficult to orient in the involved mashed-up assets and navigate towards their offered functions. Mashup platforms will therefore require mechanisms to provide such guidance.

As shown earlier, mashups are web applications integrating a set of complementary Web services and data sources, referred to as *mashlets*. Previous works typically considered mashlets as isolated atomic services [2, 18, 23]. In real-life, however, mashlets are often incorporated as services within larger applications. For instance, the patient drugs list may be part of an Electronic Health Record application (EHR); the pharmacies directory may be part of a pharmaceutical Web site, etc. The gluing of mashlets, in this case, yields a set of inter-connected *Mashed-up Applications* (abbr. *MashAPP*). Development of such *MashAPPs* is a current trend, and their number, as well as the number of their users, are estimated to significantly grow in the near future [16, 17].

Users of *MashAPPs* may navigate, in parallel, in several, *interacting* applications. For instance, consider a patient wishing to find out where her prescribed drugs are sold. Without exploiting the interactions between applications, she could navigate *separately* within the EHR, pharmaceutical, and map applications: this may require her to login to her EHR account, retrieve the prescribed drugs, then login to the pharmaceutical application, manually searching for pharmacies that offer these drugs, then turn to the map and repeatedly search for the location of relevant pharmacies. Alternatively, she can exploit the *MashAPP* interconnections to complete her navigation much faster: she may still need to first login to her account at the EHR and

at the pharmaceutical application, due e.g. to security constraints imposed by the applications, but now a single click on each prescribed drug feeds the data to the pharmaceutical application, that retrieves pharmacies offering this drug, and the pharmacies locations then appear on the map.

The above example illustrates two connections within the *MashAPP*, but there are typically many others (e.g. information on pharmacies offering deals may be found in medical forums, payment may be done online, etc.). This implies that the number of possible relevant navigation flows in a *MashAPP* may be very large (even in a single application, the number of flows is large [4] and inter-connections between the applications further increase it). Some of these navigation options exploit the *MashAPP* structure in a much better way and are significantly better for the user than others (e.g. save work, induce less errors, etc.), but identifying them may be a significant challenge [22].

The growing popularity of *MashAPPs* [17], along with the difficulty of users in optimally exploiting such *MashAPPs* [22], calls for a solution that assists users in their navigation.

Model Enhancement. In order to provide such solution, our original model had to be enhanced, to support interaction between integrated applications, and the possible navigation flows within the *MashAPP*. The new model preserves the Mashlets and GPs, and adds the notion of *Web-Applications* which defines a flow that connect Mashlets.

For the definition of a *Web-Application*, we assume a domain $\mathcal{L}$ of mashlet names; following the definitions in section 2, each such mashlet name $l \in \mathcal{L}$ is associated with two sets of relations: a set of *input* relations, denoted by $in(l)$, that must all be fed so that the mashlet may operate, and a set of *output* relations, denoted by $out(l)$, that are the output of the mashlet operation.

A *Web-Application* is then modeled as a directed node-labeled graph, corresponding to an application *site-map*.

We then define the notion of a *MashAPP* as a collection of Web-Applications whose mashlets are inter-connected via Glue Patterns (abbr. GPs). Intuitively, a GP connects a source mashlet of one application to a target mashlet of another application, supplying some of its required input.

DEFINITION 3. *A MashAPP is a pair* $M = (Apps, GPs)$ *where Apps is a set of Web-Applications and GPs is a set of* Glue Patterns. *A Glue Pattern (GP) is a pair* $gp = (s, t)$ *where* s *and* t *are two nodes of distinct applications in Apps. We call* s (t) *the source (resp. target) of gp, and denote it by* $source(gp)$ $(target(gp))$. *We require that for every mashlet in Apps that is the source of multiple GPs, the target nodes of these GPs belong to distinct applications in Apps.*

A *Navigation Flow* is an actual instance of navigation within a *MashAPP*, consisting of a sequence of *navigation steps*. Intuitively, a flow of a single stand-alone application starts at the application starting point and follows its edge relation, with the input of each mashlet along the flow being fed either by the output of the previously activated mashlet, or by the user. The navigation point of each application is captured by the location (node) of the application along its flow. As for a *MashAPP* that combines several applications, the navigation point is modeled via a *Program Counter* (PC) signifying the current locations of the flow in all applications.

A navigation flow in a *MashAPP* induces parallel navigation in all participating applications and is captured by a sequence of PCs. In the absence of GPs, a navigation step in a *MashAPP* is a single step in one application, updating the corresponding PC node and keeping all others intact. However, GPs allow to bypass the standard application flow, and have mashlets be activated by other applications and receive their input from them. Thus, a mashlet m in application A may be activated even if the PC node of A has not reached it, but rather the *combination* of outputs of mashlets (in other applications) glued to it, along with the output of the current mashlet of A, feed its required input. In this case, the PC node of A may "jump" to m, continuing the flow from there. A formal definition of PCs is given in [5].

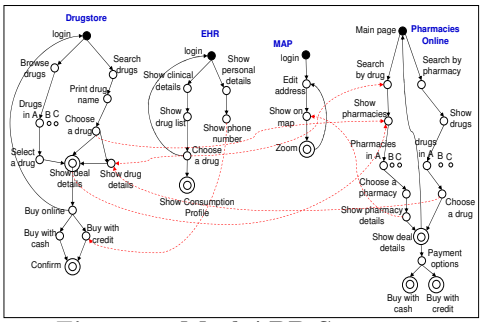

Figure 3: MashAPP Structure

Recommendations. As can be seen, for a given *MashAPP*, there is a large number of possible flows that users might choose to achieve a certain goal. Some of the flows are considered better than others, e.g. due to preferences and constraints involved with the user or with the goal to be achieved. In order for the user to define his goal, we developed a query language which is modeled as a graph pattern over the *MashAPP*, which aims to provide as a result a set of flows that may guide the user to achieve her goal. In order to rank the various flows, we developed a weight function that allows to weigh the possible flows based upon user-specified criteria (e.g. number of clicks, popularity). In order to enable the desired navigation, there is a need for an efficient algorithm that, given such query and the current user location within the *MashAPP*, finds the k best-weighted continuations, which should be presented to the user during navigation. As described in [6], the problem is $\sharp P$-hard (data complexity). However, further analysis shows that the complexity depends on the number of GPs connecting the applications within the *MashAPP*, and on the query size. We show [5] that both are relatively small in practical cases, and present a algorithm that returns such top-k paths efficiently. The algorithm receives as input a *MashAPP*, a query, a Program Counter indicating the current user position, and a number k of requested results, and outputs the top-k qualifying navigation flow continuations in a data structure referred to as Out. The algorithm considers in parallel all possible invocation orders for the GPs (compatible with the query constraints). For each order O, it computes the top-k navigation flows conforming to O, by computing the top-k partial navigation flows in-between each two consecutive GP sources in O, then combine the results to obtain top-k full navigation flows for each such order.

Compass. The model and algorithms described here served as the basis for the development of *Compass* [6], a system that assists users in their navigation through *MashAPPs*. Users of *COMPASS* are presented with an abstract graphical representation of the *MashAPP*, they easily form queries and are given back top-k navigation flows satisfying the query. The system was presented in [6].

5. RELATED WORK

We give in this section a review of related work, highlighting the relative contributions of our results. Due to space constraints we give here only a limited subset of the related results. See [1, 2, 13, 6] for further review.

Mashup Composition and Assembly. Several works provide complementary tools to assist in Mashup assembly. [21] proposes a tag-based navigation technique to compose data mashups. [9] analyzes the data currently viewed by the user to suggest widgets that might handle this data. However it does not provide the glue to connect the proposed widgets with the present mashup (in our terminology, the best fitting GPs). [8] recommends users a set of possible outputs for a specific mashup. Once such output is chosen, it computes an extension of the mashup which will achieve the selected output. Our solution however recommends GPs for a specific set of mashlets, and therefore these two can be complementary for mashup designers.

Navigation Assistance. Various works [4, 7] considered analysis of navigation flows within the context of a *single* application. Most of these works analyze also the *data* manipulated by the Web-Application; as noted in [7], querying the combination of execution flows and data they manipulate, and the interplay thereof, incurs very high complexity. [4] took an approach similar to ours of querying application structure and suggested a PTIME evaluation algorithm for the restricted setting of a single application. We show that there is an inherent added hardness in the *MashAPP* settings which calls for dedicated algorithms and optimization techniques that presented here. We also note that while our model resembles that of Petri Nets [20], it bears a weaker (yet sufficient for capturing real-life *MashAPPs* structure) expressive power allowing for efficient query evaluation.

Top-k Queries. *Top-k* queries were studied extensively in the context of relational and XML data [12, 15]. In graph theory, the problem of k-shortest paths was studied extensively (e.g. [10]). We present here novel versions of top-k query evaluation that suits the web environment with large volumes of information which needs to be narrowed, as shown in section 3, and helps in navigation in multiple applications in parallel, as shown in section 4.

6. FUTURE WORK

In previous sections we summarized the results obtained so far. We now review our main goals for further research.

Query Facility. Over the past few years, the number of mashups and mashlets that are available for developers is growing rapidly. Therefore, an advanced mechanism to search and query these assets is required to facilitate mashup construction. Such mechanism will be based on techniques to compare mashups, propose alternative compositions, and so on. The retrieved mashlets can then be provided as input to the recommendations mechanism presented in section 3.

Collaborative Mashup Design. Currently, mashups are developed by different users over the web, independently and asynchronously. We would like to facilitate collaborative design of mashups, by enabling users to recommend relevant components, define how data flow between them.

User Preference and Context. The work done until now is generic for every user and state. It is agnostic to the context of users and to their preferences. We would like to refine the recommendations we propose during the mashup design, by enabling the system to identify what users need and prefer at any time and place, and integrate these as factors in the proposed recommendations for assets to use and on how to use them. This requires the development of a generic framework for defining (and identifying) relevant contextual information, as well as for exploiting such information.

Mashup Optimization. Due to the complexity mashups impose, development of appropriate optimization techniques are required to improve their performance. They integrate various components that each may consume large number of sources, and they are used by large amount of users on the web in parallel. Therefore they are required to scale up well. We plan to develop various optimization techniques, such as cost-based optimization to maximize performance and minimize cost on system resources, data-centric optimization that focuses on the data being consumed, and others.

7. REFERENCES

[1] S. Abiteboul, O. Greenshpan, and T. Milo. Modeling the mashup space. In *WIDM'08*.

[2] S. Abiteboul, O. Greenshpan, T. Milo, and N. Polyzotis. Matchup: Autocompletion for mashups. *ICDE '09*.

[3] A. Ankolekar, M. Krotzsch, T. Tran, and D. Vrandecic. The two cultures: Mashing up web 2.0 and the semantic web. *Web Semant.*, 6(1):70–75, 2008.

[4] D. Deutch, T. Milo, and T. Yam. Goal Oriented Website Navigation for Online Shoppers. In *VLDB '09*.

[5] Daniel Deutch, Ohad Greenshpan, and Tova Milo. Navigating in complex mashedup applications. In *Submitted*.

[6] Daniel Deutch, Ohad Greenshpan, and Tova Milo. Navigating through mashed-up applications with compass. In *ICDE '10*.

[7] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *ICDT*, '09.

[8] H. Elmeleegy, A. Ivan, R. Akkiraju, and R. Goodwin. Mashup advisor: A recommendation tool for mashup development. In *ICWS '08*.

[9] RJ. Ennals and MN. Garofalakis. Mashmaker: mashups for the masses. In *SIGMOD '07*.

[10] D. Eppstein. Finding the k shortest paths. In *35th IEEE Symp. Foundations of Comp. Sci.*, 1994.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, pages 614–656, 2003.

[12] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 2003.

[13] O. Greenshpan, T. Milo, and N. Polyzotis. Autocompletion for mashups. *PVLDB '09*, 2(1):538–549, 2009.

[14] IBM Mashup center. http://www-01.ibm.com/software/info/mashup-center/.

[15] B. Kimelfeld and Y. Sagiv. Matching twigs in probabilistic xml. In *Proc. of VLDB*, 2007.

[16] Narayanan Kulathuramaiyer. Mashups: Emerging application development paradigm for a digital journal. *J. UCS*, 13(4):531–542, 2007.

[17] Cheng-Jung Lee et al. Toward a new paradigm: Mashup patterns in web 2.0. *WSEAS Trans. Info. Sci. and App.*, 6(10), 2009.

[18] B. Lu et al. sMash: semantic-based mashup navigation for data API network. In *WWW '09*.

[19] Microsoft Sharepoint. http://www.microsoft.com/SharePoint/.

[20] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of IEEE*, 77(4), 1989.

[21] A. Riabov et al. Wishful search: interactive composition of data mashups. In *WWW '08*.

[22] J. Wong and JI. Hong. In *CHI '07*, New York, NY, USA.

[23] J. Yu et al. A framework for rapid integration of presentation components. In *WWW '07*.