Navigating in Complex Mashed-Up Applications*

Daniel Deutch
Tel-Aviv University

Ohad Greenshpan
Tel-Aviv University &
IBM Research Labs

Tova Milo
Tel-Aviv University

ABSTRACT

Mashups integrate a set of Web-services and data sources, often referred to as *mashlets*. We study in this paper a common scenario where these mashlets are components of larger Web-Applications. In this case, integration of mashlets yields a set of inter-connected applications, referred to as *Mashed-up Applications* (abbr. *MashAPP*). While interactions between the mashlets enrich the individual applications, they also render navigation within them more intricate for the user, as actions in one application may affect others. To assist users in their navigation through *MashAPPs* we provide a solution based on a simple, generic model for *MashAPPs* and navigation flows within them. Queries over the model allow users to describe navigation flows of interest, and an effective query evaluation algorithm provides users with recommendations on how to navigate within the *MashAPP*. The model and algorithms serve as a basis for the *COMPASS* system, built on top of the Mashup Server.

1. INTRODUCTION

Mashups integrate a set of Web services and data sources, often referred to as *mashlets*. For instance, a health-care related mashup may integrate the following components (among others): (1) a patient's personal list of prescribed drugs, (2) a pharmacies directory and (3) a map service. A mashup may glue these mashlets by feeding the drugs list to the (search facility of the) pharmacies directory, obtaining all pharmacies offering these drugs, and feeding addresses of retrieved pharmacies to the map service, to present their location. Such connections are called *Glue Patterns* (GPs)[11].

Previous works on mashups typically considered mashlets as isolated atomic services [11, 17, 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 interconnected *Mashed-up Applications* (abbr. *MashAPP*). The 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 [15, 16].

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* [16], along with the difficulty of users in optimally exploiting such *MashAPPs* [22], calls for a solution that assists users in their navigation. The present paper depicts the development of such system, based on a generic model that describes the *MashAPP* structure, the interaction between participating applications, and the possible navigation flows within the *MashAPP*. The model allows to weigh the possible flows based upon user-specified criteria such as number of clicks, required manual user input, popularity etc. We then introduce a simple query language allowing to specify navigation flows of a *MashAPP* that are of interest to the user. Last, we develop an effi-

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

Proceedings of the VLDB Endowment, Vol. 3, No. 1

cient algorithm that, given such query and the current user location within the *MashAPP*, finds the k best weighted continuations of the user navigation. These possible continuations are presented to the user, assisting in her navigation. Our experiments indicate the run-time efficiency of our algorithms as well as their contribution to users.

The model and algorithms described here served as the basis for the development of *COMPASS*, a system that assists users in their navigation through *MashAPPs*. Users of *COMPASS* are presented with an abstract graphical representation (similar to a site-map) of the *MashAPP*. By clicking on this site-map, users may easily form queries. The system then computes top-k navigation flows satisfying the query. The flows are ranked based on a user-chosen metric (currently, choices popularity and number of incurred clicks are suggested as metrics). Then, the recommendations are displayed along-side the *MashAPP* itself, as a list of instructions for navigation (e.g. press button X, follow link Y, etc.). Users may either follow the recommendation as is, or may follow other paths than those proposed, in which case new recommendations, consistent with the actual choices made by the user, are automatically computed. *COMPASS* is integrated with the Mashup Server and its operation was demonstrated in ICDE '10 on a real-life IBM patient portal [10]. The demonstration [2] only provides a high-level description of the system, while the current paper presents the underlying model and algorithms.

Note. We focus in this paper on the structure of the *MashAPP*: which mashlets are connected and in what way this connection affects the navigation flow within the *MashAPP*. Such connections can be thought of as a generalization of a *site-map*, typically used for stand-alone Web-sites, to *MashAPPs*. Naturally, the *semantics* of the individual mashlets (e.g. their implementation, the data they manipulate etc.) also need to be considered. However, previous works have shown that one quickly meets undecidability or very high complexity [18, 6] when considering such semantical features. For a navigation assistance system to be effective, it must be interactive and respond in split-seconds (as oppose to e.g. a static analysis tool [18]). Consequently, our algorithms focus on analyzing *MashAPPs* structure, but allow for plug-ins that implement application-specific, semantics-aware analysis, where an efficient such analysis is possible.

Related Work. We conclude the Introduction with a brief overview of related work, highlighting our contribution.

Recent work has led to the development of many platforms which assist mashup *designers* [17, 11, 7]. In contrast, our work is the first (to our knowledge) to assist mashup *users* in their navigation.

Various works [4, 6, 5] 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 [5, 6], querying the combination of execution flows and data they manipulate, and the interplay thereof, incurs very high complexity. The work in [4] took an approach similar to ours of querying application structure and suggested a PTIME evaluation algorithm for the restricted setting of a single isolated application. In contrast, we show that there is an inherent added hardness in the *MashAPP* settings, as if $P \neq NP$ such PTIME algorithm does not exist here. This added hardness calls for the dedicated algorithms and optimization techniques that

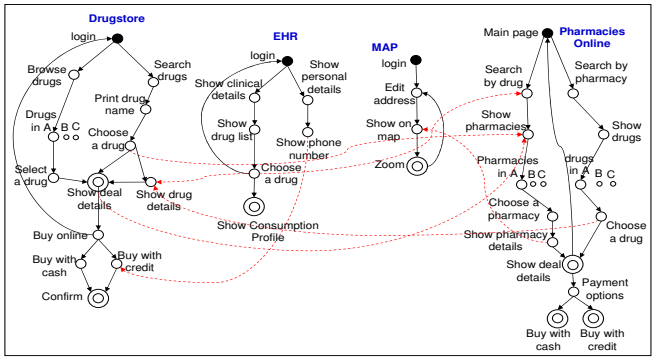

Figure 1: Real-Life *MashAPP*

we present here. We also note that while our model resembles that of Petri Nets (PNs) [19], it bears a weaker (yet sufficient for capturing the structure of real-life *MashAPPs*) expressive power allowing for efficient query evaluation

Top-k queries were studied extensively in the context of relational and XML data [9, 14]. In graph theory, the problem of k-shortest paths was studied extensively [8, 21]. We show that the multiple-application settings of a *MashAPP* renders top-k query evaluation harder than for a single graph, but, as shown in Section 3, we exploit such k-shortest paths algorithms as tools within our evaluation algorithms.

Paper Organization. The paper is organized as follows. Section 2 describes our data and query model. Section 3 analyzes the complexity of the top-k problem and provides a practically efficient query evaluation algorithm. Section 4 describes the implementation of the *COMPASS* system and our experiments. We conclude in Section 5.

2. PRELIMINARIES

We start by presenting our *MashAPPs* model, along with intuitive examples.

Mashlets, Glue Patterns, and Applications. A *mashlet* is a module implementing some functionality and supporting an interface of input and output variables. Taking a syntactic viewpoint, we first consider only the mashlet interface. We will consider the incorporation of semantic analysis, that also takes into account the mashlets logic, later on.

We assume a domain $\mathcal{L}$ of mashlet names; following [11] 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 graph, corresponding to a *site-map*. To formally define such applications, we assume a domain $\mathcal{N}$ of nodes, each bearing a unique identifier. Nodes are labeled by mashlet names from $\mathcal{L}$; the same mashlet name may annotate multiple nodes.

DEFINITION 2.1. *A* Web-Application *is a node-labeled directed graph $A = (V, E, \lambda)$, where $V \subseteq \mathcal{N}$ is the set of nodes, $E \subseteq V \times V$ is the set of edges, and $\lambda : V \longmapsto \mathcal{L}$ is a labeling function over V. A has a distinguished single node $start(A)$ standing for its starting point, and a non-empty set of nodes $end(A)$ standing as its possible end points. We use $Nodes(A)$ $(Edges(A))$ to denote V (resp. E).*

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 2.2. *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.

The reason for this last requirement will become apparent when we discuss below the role of GPs in navigation flows.

EXAMPLE 2.3. *Fig. 1 depicts a graphical abstract view of a part of a real-life MashAPP, namely a patients portal [10]. This partial MashAPP consists of four applications (Drugstore, Electronic Health Record, a Map application and Pharmacies Online). Each application has a unique start node (marked as a full circle), and possibly multiple end nodes (marked with doubly bounded circles), representing the completion of some task (e.g. the purchase of a drug from Pharmacies Online or Drugstore). Regular edges detail the logical flow within each application (e.g. in the Map application,* Show on map *follows* Edit address *and may receive its input address from it). Dashed edges stand for GPs (e.g. the same mashlet may also receive input from* Show pharmacy details *in Pharmacies online). Note that a single mashlet may be the source of several GPs whose targets reside in distinct applications. For instance, "Choose a drug" in EHR is the source of one GP whose target is "Show drug details" in Drugstore, and also of another GP whose target is "Search by drug" in Pharmacies Online.*

2.1 Navigation Flows

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) the application along its flow. As for a *MashAPP* that consists of several such applications, the navigation point is modeled via a *Program Counter* (PC) signifying the current locations (nodes) of the flow in all applications.

A navigation flow in a *MashAPP* induces a 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 an application A may be activated even if the PC node of A has not reached it, but rather the *combination* of the output of mashlets (in other applications) that are glued to it, along with the output of its current mashlet in A, feeds its required input. In this case, the PC node of A may "jump" to m, continuing the flow from that point. We next formalize these notions.

DEFINITION 2.4. *Given a MashAPP* $M = (Apps, GPs)$ *with applications* $Apps = \{App_1, ...App_m\}$, *a* Program Counter *is a tuple* $PC = [n_1, ..., n_m]$ *where* n_i *is a node of* App_i.

We say that $[n_1, ..., n_m] \rightarrow_{(n_i, n'_i)} [n'_1, ..., n'_m]$ *if:*

1. $(n_i, n'_i) \in Edges(App_i)$, *and*
2. *for each* j *such that there exists a gp* $(n'_i, target) \in GPs$, *with* $target \in App_j$, *and where condition (*) (given below) holds,* $n'_j = target(gp)$, *and*
3. $n'_j = n_j$ *for all other indices* j.

Condition (*) states that for every n'_j, $in(n'_j) \subseteq out(n_j) \bigcup out(n'_i) \bigcup \{out(n_k) \mid (n_k, n'_j) \in GPs\}$.

Intuitively, for two PCs PC, PC', we say that $PC \rightarrow_{(n_i, n'_i)} PC'$ if PC' is obtained from PC by the user advancing in application App_i, from n_i to n'_i, consequently (possibly) activating GPs that cause PC nodes of the corresponding applications to "jump". (n_i, n'_i) is referred to as the *user choice*.

A *navigation flow* is then a sequence of program counters and user choices $F = PC_0 \rightarrow_{e_1} PC_1 ... \rightarrow_{e_t} PC_t$. We say that F is *successful* if each node in PC_t is an end-node of the corresponding application. A successful flow is *full* if $PC_0 = [start(App_1), ..., start(App_m)]$ and is otherwise referred to as a *continuation*.

EXAMPLE 2.5. *Consider a simple navigation flow in the MashAPP of Fig. 1. We use in the sequel the notation AppName.MashletName to refer to the node annotated by "MashletName" and appears in the "AppName" application. Assume that the user navigates in parallel in the Drugstore and EHR applications, reaching the "Drugstore.searchDrug" and the "EHR.Show Drug List" nodes; in particular, this means that the user has already logged into Drugstore. Now assume that the user continues navigating in EHR, reaching the "EHR.Choose a Drug" node and making a choice of a drug. Now, note that there is a GP connecting "EHR.Choose a drug" to the "DrugStore.Show drug details", supplying to the latter the chosen drug name. Focusing on the latter mashlet, we assume that its required input consists of user login information (as only registered users can see drug details using this service), as well as a requested drug name. Since the PC at Drugstore already passed the "Drugstore.login" mashlet, user login information was also provided (and we assume that it is passed from one mashlet to the next), thus the PC of Drugstore "jumps" to "Show drug details". The user may then proceed and make navigation choices in any of the mashed applications.*

In this example the user input caused a "jump" in a single application, but in general it may cause such jumps in multiple applications, taking place in parallel.

We note here that, except for the GPs, the order in which one advances in the different applications (i.e. whether we first perform a given step in one application and then in the other, or vice versa) is not significant, as long as for each individual application the sequence of traversed nodes (and its ordering) stays intact. Two navigation flows that defer only in the above sense are considered to be equivalent (a formal definition of this equivalence relation is given in the Appendix).

2.2 Top-K Queries

So far we have discussed the notion of navigation flows and distinguished between successful and non-successful navigations based on whether or not some end-nodes defined by the application are reached. In many cases, there is also a set of nodes that the user wishes to traverse in a specific order. To that end, we next introduce the notion of *queries* that

describe the navigation flows that are of interest. We define a query Q over a *MashAPP* as an acyclic graph, defining nodes that must appear in a qualifying flow and a required partial order over their traversal. An edge of Q may be marked as transitive, and then its end-nodes do not have to appear consecutively in the navigation sequence.

DEFINITION 2.6. *A query Q over a MashAPP $M = (Apps, GPs)$ is a pair (G, T), s.t. $G = (V, E)$, where V is a subset of the nodes appearing in Apps, $E \subseteq V \times V$ is an acyclic edge relation over V and T is a subset of the edges in E, that are marked as* transitive.
A successful *navigation flow $PC_0 \rightarrow_{e_1} \ldots \rightarrow_{e_t} PC_t$ of a MashAPP M satisfies a query (G, T) if for each node n of G, there exists t' s.t. $n \in PC_{t'}$, and furthermore for each edge (n, m) in G ($(n, m) \in T$), there exist $t'' = t' + 1$ ($t'' > t'$, resp.) such that $m \in PC_{t''}$.*

Note that as the application graphs may be cyclic, the same node may appear multiple times in the navigation flow. Definition 2.6 requires that each constraint is satisfied by at least one of these occurrences. This requirement follows in spirit the notion of bisimulation [18], commonly used for process analysis.

In general, there may still be many qualifying flows to a given query, and users typically prefer some flows over others. To this end, we introduce a weighted model for *MashAPPs* and for navigation flows within them, as follows.

Weighted Model. We assume a domain $\mathcal{W}$ of weights and use a weight function that assigns a weight $w_e \in \mathcal{W}$ to each *edge* e (possible user choice) in each application in *Apps*. The edge weight intuitively reflects its cost in terms of user input, its likelihood of being chosen in a random navigation flow etc. The individual edge weights occurring along a navigation flow are aggregated to obtain the flow weight. The aggregation function $aggr : \mathcal{W} \times \mathcal{W} \rightarrow \mathcal{W}$ receives two weights as inputs; the first intuitively corresponds to the aggregated weight computed so far, and the second is the new edge weight to be aggregated with the previous value. For instance, when computing total number of clicks, $aggr = +$ and $\mathcal{W} = [0, \infty)$; choices popularity may be modeled by a *likelihood function*, and then $aggr = *$ and $\mathcal{W} = [0, 1]$.

We consider here aggregation functions that satisfy the following intuitive constraints:

1. $aggr$ is associative and commutative, namely for each $x, y, z \in \mathcal{W}$, $aggr(aggr(x, y), z) = aggr(x, aggr(y, z))$, and $aggr(x, y) = aggr(y, x)$.
2. $aggr$ has a neutral value, denoted 1_{aggr}. Namely for each $x \in \mathcal{W}$, $aggr(x, 1_{aggr}) = aggr(1_{aggr}, x) = x$.
3. $aggr$ is monotonically increasing or decreasing over $\mathcal{W}$. Namely, either for each $s, x, y \in \mathcal{W}$ $x \geq y \implies aggr(s, x) \geq aggr(s, y)$ and $aggr(s, x) \geq s$, or the same for $\leq$.

Given a navigation flow $F = PC_0 \rightarrow_{e_1} PC_1 \rightarrow_{e_2} \ldots \rightarrow_{e_t} PC_t$ we define W_F, the weight of F, in a recursive manner as follows: $W_{[PC_0]} = 1_{aggr}$, and $W_{[PC_0 \rightarrow_{e_1} \ldots \rightarrow_{e_i} PC_i]} = aggr(W_{[PC_0 \rightarrow_{e_1} \ldots PC_{i-1}]}, W_{e_i})$. A *weighted MashAPP* is then a *MashAPP* along with such weight functions.

Note. The model introduced here assumes that the weight of each individual choice does not depend on previous choices. This is sometimes the case in practice (e.g. for weight functions reflecting number of clicks), but sometimes is not (e.g. for popularity of choices). For simplicity of presentation, we will assume below independence between choices, but stress that our results go through to an extended model allowing dependency. We explain the adaptations required for supporting this model in the Appendix.

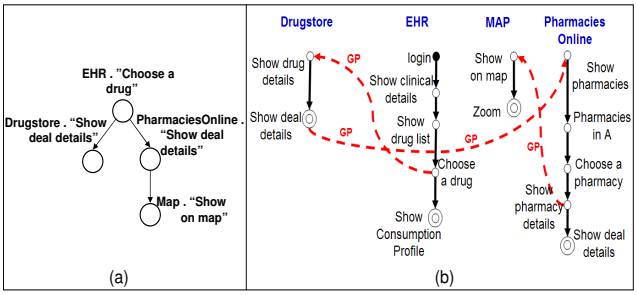

Figure 2: Query and Answer

Top-K Answers. Observe that when $aggr$ is monotonically increasing (resp. decreasing), so is the weight of execution flows. Generally, when $aggr$ is monotonically increasing (as, e.g., for total number of clicks), users are interested in the k flows having the lowest weight (e.g. requiring the least overall number of clicks). When $fWeight$ is monotonically decreasing (as, e.g., for popularity of flows), we will be interested in k flows having the highest weight (e.g. the most likely ones). We refer to both problems uniformly as *top-k*. The *top-k* results of a query Q with respect to a weighted *MashAPP* M and the user *current* Program Counter (navigation point) PC, denoted $top - k(Q, M, PC)$ are thus defined as a set of k-best (highest for *decreasing* functions, lowest for *increasing* functions) weighted successful navigation flows of M satisfying Q, starting at PC (if there are multiple such sets, we pick one arbitrarily). Due to the fact that two navigation flows may be equivalent (in the sense described after Example 2.5), in order to allow diversity of recommendations to the user, we output at most one navigation flow from every equivalence class.

EXAMPLE 2.7. *Re-consider the MashAPP M depicted in Fig. 1 and assume a simple weight function that assigns equal weights to all edges, with $aggr = +$. This function counts the number of mashlets being traversed during navigation, and assume that we wish to to minimize it. Further consider a user that wishes to shop for a drug that appears in her Electronic Health Record (EHR). The user wants to check the drug price in an online drug store, and in pharmacies close by to her place of residence. She may issue a query Q, as depicted in Fig. 2(a). The query requires flows to traverse the "Choose a drug" mashlet in EHR to choose the prescribed drug, then go to "Show deal details" in Drugstore to find the price for the chosen drug. The choice of drug is also followed by "Show deal details" in Pharmacies Online, to show details of deals in pharmacies offering this drug, and the latter is in turn followed by "Show on map" in the Map application, to show the pharmacy location.*

Fig. 2(b) depicts the nodes and edges traversed in the top-1 navigation flow of M satisfying Q. The flow starts by navigating to "Choose a drug" in the EHR, then utilizing a GP that causes a "jump" in DrugStore to "Show drug details", showing the chosen drug details. The flow continues by navigating in Drugstore up to "Show deal details", which is connected by a GP to the pharmacies site and causes a jump in Pharmacies Online to "Show pharmacies". Then, it continues navigating in Pharmacies Online to find the pharmacy of interest, and finally uses a GP connecting the pharmacies site to the map, to show the location of the chosen pharmacy.

3. TOP-K QUERY EVALUATION

We study in this section the evaluation of top-k queries over *MashAPPs*. Unfortunately, as we next show, no PTIME (data complexity) algorithm for computing top-k query answers is possible, unless $P = NP$. Let `BEST-FLOW` be the problem of deciding, given a weighted *MashAPP* M, a query Q over M, and a bound B, whether there exists a successful navigation flow of M that satisfies Q and further has a weight higher than B.

THEOREM 3.1. `BEST-FLOW` *is NP-complete in the size of the input MashAPP* M*. Furthermore, this holds even if all applications in* M *have the form of a chain.*

The proof (deferred to the appendix for lack of space) is based on a reduction from 3-SAT.

A naive EXPTIME algorithm can be obtained by looking at the configuration graph G of the *MashAPP*, namely the nodes of G are the *MashAPP* Program Counters, and its edges correspond to the $\rightarrow_e$ relation over such PCs. Note that there are many algorithms (e.g. [8, 21]) in the literature that find top-k paths in a single graph, with polynomial complexity in the graph size. The naive algorithm would first choose an order of traversal over the query nodes, then choose for each query node a corresponding node in the configuration graph (namely a node whose PC at the appropriate application is at the query node), and repeatedly apply a top-k paths algorithm over G, to find top-k subpaths in-between the chosen nodes, according to the chosen order of their corresponding query nodes. By considering all such choices, the top-k flows of the given *MashAPP* can be found. However, the number of nodes in G, and thus the algorithm complexity, is exponential in the number of applications, with the exponent base being the size of each individual application. If the applications graphs contain thousands of nodes (as is often the case in practice [13]), the complexity of this naive algorithm becomes infeasible.

Fortunately, a more careful analysis of the NP-hardness source (and the consequent complexity of query evaluation) shows that in fact it depends on the number of GPs interconnecting applications in the *MashAPP* and on the query size, and both are relatively small in practical cases. Indeed, we show below that we can obtain a practically efficient algorithm with complexity where the exponent depends on this quantity, and the basis of its exponent is a small constant.

Simplifying Assumptions. We make below three assumptions, for ease of presentation: we assume that (1) each application has at most one end node, (2) end nodes of applications are not sources of GPs, and (3) each GP source can provide, by itself, sufficient input for the activation of its target (independently of the current PC location in the target application, or in other applications). We stress that these assumptions are made solely for ease of presentation, and we explain in the Appendix the necessary changes to our algorithms when these assumptions do not hold.

3.1 Observations

Before depicting our top-k algorithm, we present key observations utilized in its development. Consider a *MashAPP* $M = (Apps, GPs)$, a query $Q = (G, T)$, s.t. $G = (N_q, E_q)$, and an arbitrary navigation flow F that satisfies Q. F traverses all nodes of N_q, in some order conforming to E and T (we call such order *compatible* with the query). It also possibly traverses some nodes that are sources of GPs, and terminates with all PCs residing at accepting states. Let us denote by $SourcesSet$ the GP source nodes traversed by F. Now, consider all possible navigation flows that traverse nodes of $N = N_q \bigcup SourcesSet$ in the same order that F does. Denote this class of navigation flows by $\Gamma(F)$. A key observation is that, to find the top-k flows of $\Gamma(F)$, we may compute *independently* the top-k sub-flows leading from $UserPC$, the current PC of the user, to a PC_1 containing the first node of N, then compute the top-k sub-flows leading from PC_1 to a PC_2 containing the second node of N and so forth up until the last node of N. The overall top-k flows are the k best-weighed concatenations of these sub-flows.

These top-k sub-flows computations may be done independently due to the fact that the order chosen over N determines in particular an order over the GP sources. We thus know that a valid sub-path starting from one node of N and ending at the next node, does not traverse any GP sources along the way. This observation is important in two ways: (1) it allows to compute the next PC from which the next step of computation will start, independently of the choice of actual path at the current step: this next PC is the same as the current PC for all applications except for that of the next node n of N and for the target applications of GPs that n is their source, and (2) it allows the computation to be restricted to top-k paths in the *single* application where the next node of N resides. As we shall show below, this allows to plug-in an application-specific top-k analysis.

So far we considered top-k flows corresponding to a single $\Gamma(F)$ set. More generally, the top-k flows satisfying q can be found by repeating this procedure for all possible choices of $SourcesSet$ and of an order over $N_q \bigcup SourcesSet$ that is compatible with the query.

3.2 `TOP-K` Algorithm

We start by depicting a black-box plug-in used by our algorithm, then depict the algorithm itself.

TopKPaths. We use in our algorithm an external plug-in called $TopKPaths$, that given a single application graph G and two end nodes n_1 and n_2 in G, computes top-k paths from n_1 to n_2. $TopKPaths$ may be implemented in a purely graph-theoretic way, as a shortest (weighted) paths algorithm [8]. However, we stress that this is the point where a semantic analysis, when can be performed efficiently, may take place. As our algorithm applies $TopKPaths$ on a *single* application graph at a time, $TopKPaths$ may implement an application-specific, *semantic-aware* analysis [4, 5] that is incorporated in a transparent way within our algorithm.

Algorithm. We next depict an efficient algorithm for top-k query evaluation. 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 all *compatible* (as defined above) orders over the query nodes and a subset of the GP sources. For each such order O, the algorithm computes the top-k navigation flows conforming to O, by computing the top-k partial navigation flows in-between each two consecutive nodes in O, then combine the results to obtain top-k full navigation flows for each such order. For efficiency, the algorithm advances *in parallel* in the computation done for the different orders; this parallel

computation is done in a branch-and-bound style.

The details of Algorithm TOP-K are given in Algorithm 1. It initializes the *Out* structure (Line 1), then uses (Line 2) a priority queue *Frontier* of possible orders, with priorities set by the top-1 (partial) navigation flow computed so far for each order. Initially, all orders have the same priority. At each iteration TOP-K pops (Line 4) the order O having highest priority out of *Frontier* and first checks for our stop condition (Line 5) - namely whether k full flows that are better (weight-wise) than the top-1 computed sub-flow of O, were already found. If so, we break the loop (Line 6) and terminate. Otherwise, we look at $O.index$, signaling the index of the last node reached in computation for O, and continue the computation from this point by invoking $HandleSubNav$. As shown below, at each iteration $HandleSubNav$ computes (and stores in a data structure called $O.TopK$) the top-k sub-flows from UserPC to the $(O.index)$'th node of O. The algorithm then raises the index of O by 1 to indicate the point where the next computation for O will start from (Line 11). If this value reaches the size of O, then we have finished the computation for O and the top-k flows found for it (stored in $O.TopK$) are merged with the overall top-k (Line 13). Consequently (Line 14), the priority of O in *Frontier* is updated. Recall our notion of equivalent EX-flows; only EX-flows that are not equivalent to already existing ones are added to the overall top-k. Finally, we output the top-k results (Line 16).

Algorithm 1: TOP-K

Data: *MashAPP* $M = (Apps, GPs)$, a query $q = (G, T)$; a Program Counter $UserPC$ of M, number of requested results k
Result: $top - k(q, M, UserPC)$

1 Init *Out* as empty array of size k of navigation flows ;
2 Initialize Frontier as a priority queue of all *Orders* ;
3 **while** *not done* **do**
4 $\quad O \leftarrow pop(Frontier)$;
5 $\quad$ **if** $TopWeight(O.TopK) \geq BotWeight(Out)$ **then**
6 $\quad\quad$ break ;
7 $\quad$ **end**
8 $\quad i \leftarrow O.index$;
9 $\quad CurrPC \leftarrow UserPC$;
10 $\quad HandleSubNav(CurrPC, O, i)$;
11 $\quad O.index \leftarrow O.index + 1$;
12 $\quad$ **if** $O.index = |O|$ **then**
13 $\quad\quad Out \leftarrow MergeSort(Out, O.TopK)$;
14 $\quad\quad update(Frontier, O)$
15 $\quad$ **end**
16 **end**
17 Output *Out* ;

HandleSubNav **Method.** The subroutine $HandleSubNav$ is given as input $CurrPC$, O, and an index i in O. O consists of (1) an order over the nodes that should be traversed and (2) a set of top-k sub-flows from $UserPC$ to the $(i-1)$'th node of O (stored in $O.TopK$). $CurrPC$ is the PC obtained after traversing one of these top-k sub-flows (as explained above, this PC is unique among choices of sub-flows). Then, $HandleSubNav$ extends these sub-flows to be the top-k sub-flows from $UserPC$ to the i'th node of O ($O[i]$), and updates $CurrPC$ appropriately. It does that by first computing top-k sub-flows from $CurrPC$ to a PC containing $O[i]$, then finding top-k concatenations of these sub-flows to those stored in $O.TopK$.

To find the top-k sub-flows from the $CurrPC$ to $O[i]$, $HandleSubNav$ sets $O[i]$ as its *target* node and its source node (*src*) to be the node of $CurrPC$ corresponding to the same application A that *target* resides in (Lines 1-2). We then perform a top-k analysis (Line 3) for paths in-between *src* and *target*. This analysis is performed, using $TopKPaths$ mentioned above, over a *single* graph G obtained from the application graph of A, by omitting all GP source nodes other than *src* and *target*. The top-k analysis results are now concatenated (Line 4) with the top-k sub-flows found so far for O; the top-k such concatenations, forming top-k sub-flows from $UserPC$ to $O[i]$, are kept. This is done using the $ConcatSort$ method that computes the top-k concatenations incrementally (each time choosing either the next sub-flow or concatenation, according to the weights order). Finally (Lines 5-7), $CurrPC$ is updated according to the *target* location. Furthermore, this may affect PCs in other applications, due to GPs (recall that due to assumption (2), a GP is activated whenever its source is reached). We thus find all GPs whose source node is *target* (Line 6). $CurrPC$ is updated correspondingly (Line 7), also performing all "jumps" incurred by these GPs.

Algorithm 2: HandleSubNav

Data: A Program Counter $CurrPC$, order O, location i in O
Result: Computes Top-k sub-navigations from $CurrPC$ to $O[i]$, and updates the top-k computed prefixes for O, along with $CurrPC$

1 $target \leftarrow O[i]$;
2 $src \leftarrow CurrPC$ in app of *target* ;
3 $PartialTopK \leftarrow TopKPaths(src, target)$;
4 $O.TopK \leftarrow ConcatSort(O.TopK, PartialTopK)$;
5 Update $CurrPC$ with *target* ;
6 **foreach** $gp \in GPs$ *s.t.* $source(gp) = target$ **do**
7 $\quad$ Update $CurrPC$ with $target(gp)$;
8 **end**

Complexity. We recall that an algorithm A is *Fixed Parameter Tractable* FPT [12] if for every input of size x and parameter p, A operates in time $O(f(p) * Poly(x))$ for some function f. Where the parameter p is typically relatively small, intuitively an FPT algorithm is "almost polynomial". In our case, we can use $p = \mid N \mid = \mid sources(GPs) \mid + \mid N_Q \mid$, namely the number of distinct sources of GPs in M, plus the number of query nodes. The following theorem holds.

THEOREM 3.2. *The worst case time complexity of* TOP-K *is* $O(p! * poly(\mid M \mid, k))$*, with* p *as above. Consequently,* TOP-K *is FPT, when taking* p *as a parameter.*

As we shall see in Section 4, the value of p is typically rather small, and the algorithm performance is very good in practical cases. Note that TOP-K incurs space that is exponential in p, but this exponent is typically small enough to allow the required space to fit in main memory. Where this is not the case, one may "group" together subsets of orders, where each such subset is small enough to fit in memory, and perform TOP-K over a single subset at each iteration, then take the top-k overall results.

4. EXPERIMENTS

The model and algorithms described in this paper served as the basis for the development of *COMPASS*, a system

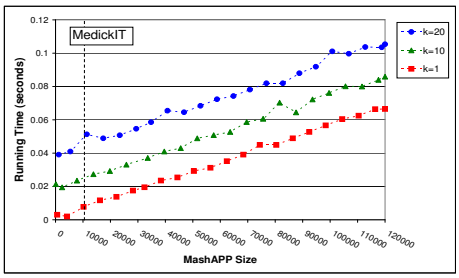

Figure 3: Dependency on MashAPP Size

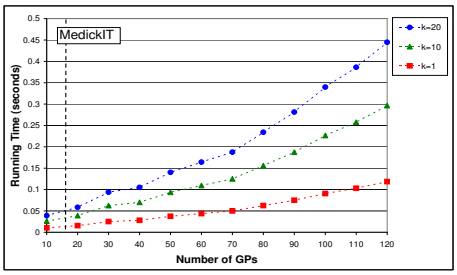

Figure 4: Dependency on # of Glue Patterns

that assists users in their navigation through *MashAPPs* (due to lack of space, the detailed architecture of *COMPASS* is given in the Appendix). We next describe our experimental study of the algorithm performance (on both real-life and synthetic data) as well as the system contribution to users. The system is implemented as plug-in to the Mashup Server, and the experiments were performed on a Pentium 1.65GHz Dual-Core and 2GB RAM. Each performance experiment was run 10 times and the graphs show the average results. In all cases, the standard deviation did not exceed 5%.

4.1 Synthetic Data

The synthetic data was generated with varying characteristics, such as the number of nodes in an application, the number of GPs inter-connecting them, the number of targets that a single GP is connected to, etc. To gain intuition on typical values of these parameters, we (1) examined ProgrammableWeb.com, the most extensive *MashAPP* repository available on the web, with over 4500 *MashAPPs*, (2) referred to the statistics available in [13] on Web Application sizes and (3) examined *MashAPPs* developed at IBM over the Mashup Server.

Typical Parameter Values. [13] states that the size of a typical, large-scale, Web-Application is approx. 2000 nodes. Our analysis of ProgrammableWeb shows that 99% of the *MashAPPs* in this repository bear 4-6 applications, rendering the total size of a typical *MashAPP* to be around 10000 nodes. The average number of GPs in the examined *MashAPPs* is approx. 15, with 5 distinct GP sources.

While these numbers seem relatively small, the obtained *MashAPPs* are still rather complicated for the users. First, there are many possible navigation paths even within a single application. The combination of 5 such applications increases the number of options by a power of 5, and the existence of GPs introduces intricate connections in-between the applications and further complicate the navigation process. Furthermore, to show scalability of our algorithm, we ran our experiments with parameter values ranging from 1 up to 10 times the typical values found for real-life *MashAPPs* (and up to approx. 5 times the maximal values found).

We also considered various *structures* for the applications participating in the *MashAPPs*, to examine their effect on the algorithm performance. We varied the number of GPs from 1 to 120 (with number of distinct GP sources ranging from 1 to 12). The locations of GP sources and targets were chosen from a uniform distribution over the application nodes (while guaranteeing that multiple targets of GPs reside in different applications). For weight functions, we used multiplication for aggregation and examined, for the individual edge weights, various distributions over $[0, 1]$ (uniform, Gaussian, Zipfian, equal weights for all edges). We observed no significant change in performance, and thus show here only the results obtained for the uniform distribution. We varied the number of *query* nodes from 1 to 20, and k (number of requested flows) from 1 to 20; we report the the results for 3 representative values $(1, 10, 20)$ of k.

Experimental Results. The first experiment examines the effect of the size of a given *MashAPP* on the performance of the `TOP-K` algorithm. We fixed the number of GPs to 15, the query size to 10, and varied the *MashAPP* size from 1 to 120000 nodes. Fig. 3 depicts our algorithm execution time as a function of number nodes composing the *MashAPP*. We observe that for such *MashAPPs*, the execution time of `TOP-K` does not exceed 0.11 seconds, for $k = 20$. Furthermore, it is apparent that the execution time of our algorithm grows very moderately with respect to the *MashAPP* size. Even for a *MashAPP* with 100000 nodes (approx. 10 times the size of a typical *MashAPP*), query evaluation time is about 0.1 seconds.

The second experiment examines the effect of the number of GPs on the algorithm performance. We varied here the number of GPs from 1 to 120, while fixing the number of applications to 6, the *MashAPP* size to 120000, and the query size to 10. Fig. 4 shows that the execution time of the `TOP-K` algorithm is again below 0.1 seconds even for 120 GPs (nearly 10 times the typical value). We further examined the effect of query size; not surprisingly (see Thm. 3.2), the execution time behaves similarly to its behavior with respect to the number of GPs. We thus omit the obtained performance graph.

We also note on the memory consumption of `TOP-K` as observed in our experiments. While we studied parameter values that are up to approx. 10 (5) times the typical (maximal) values observed in real-life *MashAPPs*, the memory consumption of `TOP-K` was low enough to fit in main memory. When this is not the case, we may employ a "grouping" mechanism (see Section 3) to parallelize the work over subsets of the possible orders, that do fit in main memory. The execution time of the refined algorithm then grows approx. linearly with the number of such subsets (which is dependent on the ratio between required and available memory).

4.2 Real-life Data and Effectiveness

To assess the algorithms performance on real-life mashApps, as well as the effectiveness of our system assistance to users, we have considered a real-life *MashAPP*, namely `MedickIT` [10], a patient personal portal consisting of five applications (Drugstore, Electronic Health Record, a map service, Pharmacies Online and an SMS messages service). The size of the *MashAPP* is approx. 10000 nodes, and it contains 15 GPs inter-connecting its applications.

We next explain the necessary steps for applying *COMPASS* over a real-life *MashAPP*, using `MedickIT` as an example for such *MashAPP*. We then describe our experimental results.

Setup. Recall that our algorithm requires an abstract model consisting of (1) a graph representation (site-map) of each application participating in the *MashAPP*, (2) a description of the Glue Patterns (GPs) inter-connecting these applications, and (3) a weight function over application graph

edges, and over GPs. For `MedickIT`, the involved applications structure was given since they were built in-house. We note however that many Web-Applications are specified in declarative languages e.g. BPEL [1], allowing automated extraction of their site-map. The GPs connecting the mashlets were automatically retrieved from the Mashup Server. As for weight functions, two such functions were employed: the first weighs flows by the total number of clicks required, and the second captures popularity of choices. Weights accounting for navigation steps were obtained from the applications site-map; the popularity weights were obtained as follows: for flow edges connecting services of a Web-Application, information about transition popularity was obtained by statistically analyzing user logs. For GPs, popularity ranks were obtained from the *MatchUp* system [11] that computes such ranks for connections in-between mashlets.

Once the abstract model is in place, *COMPASS* allows users to issue *queries* over the model. To that end, the system has a UI that enables the user to (1) view a graphical representation of the *site-map* of the loaded *MashAPP*, (2) compose a query, by consecutively clicking on nodes of the site-map in some particular order, (3) be presented with *COMPASS top-k* navigation recommendations and (4) examine various recommendations by clicking on them and viewing their highlighted image on the *MashAPP* flow-map. The UI of *COMPASS* is presented along-side that of the original *MashAPP*, allowing the user to continue her navigation, while viewing the recommendations. While experimenting with the system, we have witnessed that the interface was easily used by most users to design queries; however, a small subset of users, that lack experience in working with computers, had some difficulties with designing queries. Therefore, we enhanced the system with a repository of stored, commonly used, queries available to users.

Experimental Results. We sum up the section by reporting the system performance and the experience of its users, in the context of assistance for `MedickIT`. The performance results comply with our reported results for synthetic data, and are depicted in Fig. 3 (the vertical line stands for MedickIT). The top-k recommendations were computed, on average, in 0.01 seconds for k=1, 0.03 seconds for k=10, and 0.05 seconds for k=20, and with variance of less than 5%.

We have then performed effectiveness experiments with a group of 80 users, half already familiar with MedickIT and half not, and measured the time it took the users to navigate within the MashAPP, with and without the assistance of *COMPASS*. We split both the experienced and inexperienced users groups into two sub-groups: users of the first sub-group navigate by themselves while being able to observe the site-map of MedickIT, and those of the second subgroup are assisted by *COMPASS* during navigation, benefiting from the query-based recommendations it provides. All users were given three tasks, as follows: (1) Find the details of the physician that is closest to your work address, (2) find the details of the physician that is closest to your work address and send to your cellular phone an SMS message containing the physician's details, and (3) view the personal graph of drug consumption of a given patient.

The average time it took inexperienced users that were not aided by *COMPASS* to complete each of these relatively simple tasks was 8:06 minutes; for experienced users, this average time was only 3:04 minutes. In both groups, users performed significantly better in their tasks, when assisted by *COMPASS* (an overall average of 71% improvement, with variance of less than 4%): Inexperienced users now performed the tasks in 1:47 minutes on average, constituting a 78% improvement for them; experienced users now achieved an average time of 1:08 minutes, better by 63% compared to their performance without the system assistance. We see significant contribution of *COMPASS* for both users classes.

5. CONCLUSION

We consider in this paper the difficulties encountered by users while navigating in complex *MashAPPs*. We presented a solution that assists users in their navigation through Mashed-Up applications. Our solution is based on a simple, generic model for *MashAPPs* and navigation flows within them, and an intuitive query language that allows users to define navigation flows of interest. We studied the complexity of top-k query evaluation in this context, provided an efficient algorithm that assists users in navigation, and explained how the algorithm is utilized for developing *COMPASS*. Future research includes the incorporation of knowledge on user context and preferences, and on mashlets semantics, and automatic inference of the MashAPP model.

6. REFERENCES

[1] Business Process Execution Language for Web Services. http://www.ibm.com/developerworks/library/ws-bpel/.

[2] D. Deutch, O. Greenshpan, and T. Milo. Navigating through mashed-up applications with compass. In *ICDE '10*.

[3] D. Deutch and T. Milo. Evaluating top-k projection queries over probabilistic business processes. In *ICDE '09*.

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

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

[6] A. Deutsch, M. Marcus, L. Sui, V. Vianu, and D. Zhou. A verifier for interactive, data-driven web applications. In *SIGMOD '05*.

[7] R. J. Ennals and M. N. Garofalakis. Mashmaker: mashups for the masses. In *SIGMOD '07*.

[8] D. Eppstein. Finding the k shortest paths. In *FOCS*, 1994.

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

[10] O. Greenshpan, K. Kveler, B. Carmeli, and P. Vortman. Towards Health 2.0: Mashups To The Rescue. In *NGITS '09*.

[11] O. Greenshpan, T. Milo, and N. Polyzotis. Autocompletion for mashups. In *VLDB '09*.

[12] M. Grohe. Parameterized complexity for the database theorist. *SIGMOD Rec.*, 31(4), 2002.

[13] T. C. Jones. *Estimating Software Costs*. McGraw-Hill, 2007.

[14] B. Kimelfeld and Y. Sagiv. Matching twigs in probabilistic xml. In *VLDB '07*.

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

[16] C.-J. Lee, S.-M. Tang, C.-C. Tsai, and Y.-C. Chen. Toward a new paradigm: Mashup patterns in web 2.0. *WSEAS Trans. Info. Sci. and App.*, 6(10):1675–1686, 2009.

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

[18] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, 1992.

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

[20] P. L. T. Pirolli and J. E. Pitkow. Distributions of surfers paths through the world wide web: Empirical characterizations. *World Wide Web*, 2(1-2), 1999.

[21] E. Ruppert. Finding the k shortest paths in parallel. *Algorithmica*, 28(2), 2000.

[22] J. Wong and J. I. Hong. Making mashups with marmite: towards end-user programming for the web. In *CHI '07*.

[23] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A framework for rapid integration of presentation components. In *WWW '07*.

APPENDIX

A. EQUIVALENT NAVIGATION FLOWS

Recall that we have informally introduced, in Section 3, the notion of equivalent navigation flows, that is important to allow diversity of the results that are shown to the user. Intuitively, two different flows are equivalent if they induce the same flow in every application in separate, and differ only in the order in which the advancements in different applications interleave (e.g. whether the flow has first advanced in some application A and then in application B, or vice versa). Our `TOP-K` algorithm then outputs at most one out of each equivalence class.

To define this formally, we define the restriction of a flow to an application, signifying the relevant part of the flow to this application. Thus, two flows are equivalent if they have same restrictions for all involved applications. Furthermore, note that only parts of flows restrictions in which an advance has been made are "interesting" for studying equivalence. Thus, consecutive PCs in which the state stays the same are ignored. This will be captured by the notion of canonic restriction defined next.

DEFINITION A.1. *(equivalent navigation flows) Given a Program Counter PC and an application App, denote by $PC|_{App}$ the node of App appearing in the PC. For a navigation flow $F = PC_0 \rightarrow_{e_1} PC_1 \rightarrow_{e_2} ... \rightarrow_{e_t} PC_t$ we then define $F|_{App}$, the restriction of F to App, as the nodes sequence $[n_1 = PC_0|_{App}, n_2 = PC_1|_{App}, ..., n_t = PC_t|_{App}]$. Furthermore, we define the* canonic *restriction of F to App obtained from $F|_{App}$ by repeatedly omitting from it every node n_i such that $n_i = n_{i-1}$, until no such n_i exists. Finally, we say that two navigation flows F_1, F_2 of a MashAPP M are equivalent if for each application App in M, the canonic restriction of F_1 to App is the same as the canonic restriction of F_2 to App.*

B. PROOF OF THEOREM 3.1

PROOF. We show NP-hardness by reduction from 3-SAT. We first show a reduction where some of the graphs do not have a chain form, then refine the proof. Given a 3-CNF formula $C_1 \bigwedge C_2 \bigwedge ...C_m$ over variables $x_1, ...x_n$, we generate a *MashAPP M* as follows. M consists of $n + m$ applications, $A_{x_1}, ...A_{x_n}$, $A_{C_1}, ..., A_{C_m}$. $A_{C_1}, ..., A_{C_m}$ are simply very long chains, with length $L = 4 * n$, whose final nodes are end-nodes. Each A_{x_i} consists of a single root, with two children corresponding to assignments of $true$ or $false$ to x_i. Their corresponding true / false children are connected, by GP, to all end-nodes of paths corresponding to clauses in which the literal appears positively / negatively. The weight function W assigns 1 to every edge, $aggr = ' + '$ and we set $B = 3 * n$.

A slightly different construction uses only chains: instead of A_{x_i} we have two applications, $A_{x_i=t}$ and $A_{x_i=f}$. Each such application is a chain of three nodes. The second node of $A_{x_i=t}$ ($A_{x_i=f}$) is connected by a GP to the third node in the chains of all clauses containing x_i ($\neg x_i$), as well as that of $A_{x_i=f}$ ($A_{x_i=t}$).

As for inclusion in NP, we note that Algorithm `TOP-K` may be easily transformed into an NP algorithm for `BEST-FLOWS`, as follows: the NP algorithm "guesses" an a subset of the GP sources, then guesses an order O over the nodes N_q and the chosen subset, and verifies that this order is compatible with the query edges E_q; if so, then it checks for the existence of a flow that conform to O and has a better weight than the given bound B, as done by Algorithm `TOP-K`. □

B.1 Withdrawing Assumptions

Throughout the paper, we have made several assumptions to ease the presentation of our model and results. We next withdraw these assumptions, and explain the necessary changes to the model / algorithm.

Model. We have assumed above (Definition 1) that the weight of every edge along the flow is *independent* of previous flow activities. In some practical cases such independency does not hold: for instance, consider a weight function that measures popularity of choices. Users that use the "Search drugs" mashlet in Drugstore are possibly more likely to finalize a reservation without resetting (once getting to the "Buy online" mashlet), than users that navigate to the "Browse drugs" mashlet. This is because users that manually click their drug name of choice are typically certain about the drug they wish to purchase.

To account for such dependencies, we first extend our weighted model such that the weight function W is no longer a function of a single edge, but rather is a function of an edge along with the sub-flow that preceded it (referred to as flow "history"). We then follow lines similar to that of the algorithm in [3], to generate a new *MashAPP* M' where we encode, within the node labels, a sufficient amount of information about the flow history that preceded it. We may than construct a weight function W' that is independent of the flow history, and simulates W. All of our algorithms then apply to (M', W').

The size of M' is greater than that of the original *MashAPP* by a factor that depends only on "how much" of the history affects the weight, namely how many previous choices need to be recorded to decide the weight of a given edge. Studies of Web-Applications [20] show this number to be very small (approx. 4) in real-life cases, allowing our algorithms to remain highly effective even in presence of dependencies.

Algorithms. To simplify the presentation of our algorithms we have assumed that (1) each application in the MashAPP bears a single end-node, (2) the end nodes of applications are not sources of any GPs, and (3) every GP may be activated when its source is reached, independently of the current PC location.

First, to relax assumption (1), if multiple end-nodes exist for a single application then we simply insert them to the orders O studied, after the query nodes and GP sources, in any order in-between them. Assumption (2) may also easily be relaxed, as follows: we first create a new *MashAPP* M' by adding a single child n' to each end-node n, then marking n' as end-node, instead of n, then we apply our algorithms over M'. To relax assumption (3), we extend our algorithms as follows. For every order O, and for each node s in O that is the source of some GP, we generate new orders $O_1, ..., O_m$. Each such order is obtained from O by choosing some subset of nodes that allow the GP activation and some order over the chosen subset, then inserting these new nodes, in order, to O, right before the appearance of s. These new orders are then traversed by Algorithm `TOP-K` as depicted above.

C. SYSTEM ARCHITECTURE

We next present a brief overview of the system architecture. *COMPASS* is developed on top of the Mashup Server developed in IBM Haifa Research Labs. The server allows to compose mashups (and thereby *MashAPPs*) and use them. The architecture of *COMPASS* is depicted in Fig. 5. We describe below the main system components and explain how they work together.

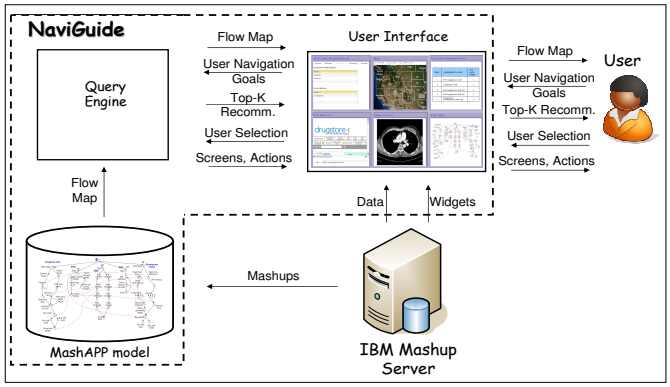

Figure 5: System Architecture

COMPASS UI. This component is the user interface for navigating through *MashAPPs*. To construct the UI of *COMPASS*, we used as a basis the user interface of the Mashup Server, and enhanced it with plug-ins enabling the user to (1) view the flow-map of a given *MashAPP*, (2) set navigation goals; (3) obtain from *COMPASS top-k* navigation recommendations and (4) examine various recommendation by clicking on them and viewing their (highlighted) image on the *MashAPP* flow-map.

MashAPP abstract Model. The *MashAPP* model, stored in the *COMPASS* database, includes the flow-map of each participating application, as well as the GPs connecting them. The first component, namely the flow-map for each application, was manually configured for all applications participating in the demonstration, following their logical structure. We note however that many Web applications are specified in declarative languages such as BPEL [1], allowing automated extraction of their flow-map. The GPs connecting the applications mashlets were retrieved from Mashup Server, along with input and output specifications of each mashlet. Last, we consider the construction of a weight function over flows. We demonstrate here two weight functions: the first weigh flows by the total number of navigation steps (user clicks and input) taken, and the second captures the total popularity of user choices. Weights accounting for navigation steps are easily obtained from the applications flow-map; the popularity weights were obtained as follows: for flow edges connecting services of a Web application, we obtain information about transition popularity by an experimental study, collecting and statistically analyzing user logs. For GPs, we employ the *MatchUp* system [11] that computes popularity ranks for connections in-between two mashlets.

MashAPP Adaptive Query Engine. This last component computes and provides users with recommendations. This engine, querying the abstract model Database, is incorporated as a plug-in to the *Mashup Server*. Queries are received through the *COMPASS* UI, directed to the Query Engine that processes them and returns the result back to the *COMPASS* UI for presentation. Furthermore, the Query Engine remains active throughout the navigation process, to receive from the UI a report on each user navigation step. Then, new recommendations consistent with the user choices are computed and sent to the UI, and so forth.