Wishful Search: Interactive Composition of Data Mashups

Anton V. Riabov, Eric Bouillet, Mark D. Feblowitz, Zhen Liu and Anand Ranganathan
IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532, USA
{ riabov, ericbou, mfeb, zhenl, arangana } @ us.ibm.com

ABSTRACT

With the emergence of Yahoo Pipes and several similar services, data mashup tools have started to gain interest of business users. Making these tools simple and accessible to users with no or little programming experience has become a pressing issue. In this paper we introduce MARIO (Mashup Automation with Runtime Orchestration and Invocation), a new tool that radically simplifies data mashup composition. We have developed an intelligent automatic composition engine in MARIO together with a simple user interface using an intuitive "wishful search" abstraction. It thus allows users to explore the space of potentially composable data mashups and preview composition results as they iteratively refine their "wishes", i.e. mashup composition goals. It also lets users discover and make use of system capabilities without having to understand the capabilities of individual components, and instantly reflects changes made to the components by presenting an aggregate view of changed capabilities of the entire system. We describe our experience with using MARIO to compose flows of Yahoo Pipes components.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous

General Terms

Algorithms, Design, Experimentation

Keywords

Composition, Programmable Web, Tag Cloud

1. INTRODUCTION

Configurable applications for automated processing of syndication feeds (i.e. Atom and RSS) are gaining increasing interest and attention on the Web. As of writing this paper there are over 30,000 customized feed processing flows (referred to as "pipes") published on Yahoo Pipes [15], the most popular service of this kind. Yahoo Pipes offers hosted feed processing and provides a rich set of user-configurable processing modules, which extends beyond the typical syndication tools and includes advanced text analytics such as language translation and keyword extraction. Yahoo Pipes' service also comes with a visual editor for flows of services and feeds. In the example of Figure 1, the feeds are Yahoo Answers and Yahoo News, which can be parameterized, and truncation, union and sort are services. There exist similar frameworks that are provided as a hosted service (e.g., DAMIA [5]) or as a downloadable server-side software (e.g., /n software's RSSBus [7], IBM Mashup Starter Kit [5] and IBM Project Zero [4]).

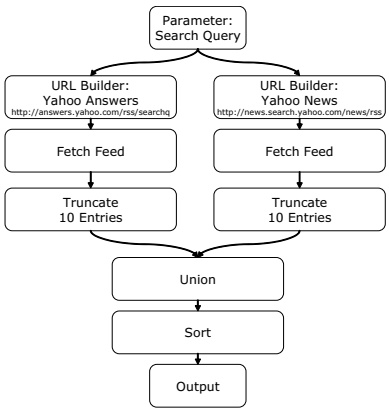

Figure 1: Example of a Flow of Feeds and Services.

This new breed of flow composition tools and execution runtimes helps lower the entry barrier for user-configurable data processing, also referred to as *data mashup*. However, as friendly as the visual programming environment of Yahoo Pipes is, it still requires careful manual assembly of the processing flows by the end user. As noted by technology reviewers, while this new technology makes data mashup more approachable for non-programmers, it still is not as easy as drag and drop [8].

With the goal of drastically enhancing the consumability of data mashup for the end users, we have developed MARIO (Mashup Automation with Runtime Orchestration and Invocation) tool for automatic composition of flows of services and feeds. This tool can be adapted to compose and deploy flows in a variety of flow execution runtimes, including Yahoo Pipes. As is well known, the easiest way for new users to start creating new data mashups is by cloning an existing pipe that roughly matches the needs of the user. In MARIO we take this practice to another level by building a tool that not only helps the user to find an existing flow that matches the processing needs, but also generates new flows on demand to match the user's request. It helps users discover the capabilities of the system by guiding them through the process of expressing composition requirements

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

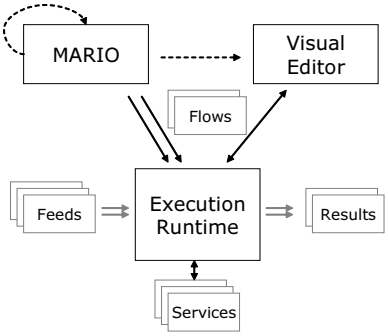

Figure 2: Interaction of MARIO with other systems.

that can be supported by the system. The tool makes automatic composition approachable by maintaining an abstraction of taxonomy-extended tag-based search in the space of existing and generated flows.

Using such a search abstraction, the users express their composition requirements as tag queries describing the desired flow output. For each request the users are immediately presented with an automatically generated and deployed flow, along with a number of alternative flows that users can visualize and deploy. The tool helps users discover and refine flows by providing context-dependent instructions.

The main contributions we present in this paper are:

- A new tool for rapid generation of new data mashups that uses a tag-based search abstraction for automatic goal-driven composition of flows.
- A simplified metadata model for describing the semantics of services, feeds and flows that enables automatic composition.
- An efficient planning algorithm that enables the search abstraction for automatic flow composition and discovery of system capabilities.

The paper is organized in the following way. Section 2 presents an overview of MARIO. Section 3 describes the user interface of MARIO, with the focus on search goal specification and refinement. Section 4 introduces the component model and component specification. Section 5 illustrates some application examples of MARIO. Section 6 explains the automatic composition engine. Section 7 discusses related work, and finally, Section 8 concludes the presentations.

2. MARIO OVERVIEW

Figure 2 shows a high-level overview of interactions between systems that can be triggered through the user interface. The end user interacts with MARIO to create a flow. The flow is deployed to the execution runtime. The runtime executes the flow by calling services to process feeds and produce results. In practice, the processing can be activated by a Web service request sent to the runtime, and the runtime can respond with results represented as a feed, similarly to Yahoo Pipes. The visual editor, if one is available, can be invoked to edit the composed flow. To open the flow composed by MARIO, the editor can retrieve flow definition directly from the runtime, or obtain it from MARIO.

We have connected MARIO to our own testing runtime that simulates Yahoo Pipes functionality, but does not include a visual editor. We evaluate the approach by using MARIO to compose a set of operators that are equivalent to Yahoo Pipes modules, but implemented as services in our runtime. This experiment demonstrates that the expressivity of semantic descriptions in MARIO is sufficient to describe modules defined within an external system. We also study the scalability by adding a large number of feeds to the existing configuration.

To support iterative refinement of queries, MARIO extends the tag-based search with the use of tag taxonomies. Searching for a tag denoting a high-level category in a taxonomy returns results tagged with any sub-category tags.

MARIO does not require the taxonomies to be specified explicitly in a top-down fashion. Instead, taxonomies can emerge implicitly in a bottom-up process that is typical of folksonomies. Nevertheless, our search mechanism allows an explicitly defined taxonomy of tags, or a set of such taxonomies, to be added at any time to enhance the search. The use of tag taxonomies together with context-dependent query refinement interface in MARIO support an intuitive iterative goal specification process, where the goals are expressed as general categories at first, and are subsequently refined to more specific tags as necessary. On the other hand, it does not prevent the users from jumping to specific tags directly, effectively shortcutting the iterations.

Recognizing the difficulties associated with obtaining detailed unambiguous descriptions of service semantics, we have taken the approach that relies on light-weight semantic metadata annotations by making use of tags, folksonomies and simple taxonomies to describe the semantics of services, feeds and flows. Given these simple annotations, MARIO uses a small set of rules to compute a set of tags that describes each potentially composable flow.

The use of tag-based descriptions greatly simplifies this task compared to heavier ontology-based approaches proposed in prior work. Descriptions of feeds can be obtained, for example, from social bookmarking web sites like Syndic8.com [1]. The descriptions of services, however, may require slightly more careful and consistent design. As we show in the paper, this is not an obstacle in practice, especially in applications where the set of services is small compared to the set of feeds.

One of the main benefits of automatic composition is instant adaptation to changes. The interface dynamically adapts to the changing set of feeds and services, providing instant feedback to the user. Just like the information presented in the goal specification interface, the feedback about the new or changed capabilities is provided in user-understandable terms, explaining the effect of the change in terms of the results that can be produced by new data mashups. With a sufficiently evolved taxonomy that includes a good set of abstract concepts, the end users of MARIO are shielded from having to understand the low-level details of specific feeds or services. The use of automatic composition together with abstractly specified goals also enables instant automatic adaptation of composed flows to changes in the environment without user's involvement.

3. USER INTERFACE

This section describes the user interface for composing flows by specifying processing goals, which is the operation that is most often performed by the end users of MARIO. There is another group of users that must interact with our

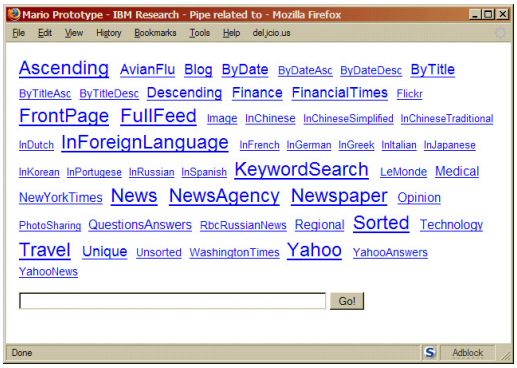

Figure 3: MARIO interface for an empty goal.

system for another purpose, namely to input the tag-based descriptions of feeds and services. We have not developed any graphical tools for that group of users, and in our implementation the required descriptions are provided simply via dynamically loaded configuration files. This will be described in Section 4.

The user interface described in this section is not specific to automatic composition per se in many respects. It was our intent to develop an interface that can simplify navigation and search in a large set of tagged and ranked objects. Those objects do not have to be feeds that are generated on the fly – they could also be taken from an external catalog.

It is the efficient planning algorithm that enables the use of this user interface for flow composition. It shields the user from the associated complexity, and makes dynamic composition appear as search over a static catalog. The details of the algorithm will be explained in Section 6.

3.1 Initial Goal Specification

The end user interacts with MARIO via a web browser. The first screen presented to the user contains a single tag cloud (see Figure 3). This tag cloud contains tags that are relevant to the application domain. The user can select one or more tags from the tag cloud to describe the desired results. Tags shown using large font sizes generally correspond to high-level categories. For example, the tag Newspaper appears in a larger font than WashingtonTimes. The larger font size indicates that the selection will constitute a broad goal that will likely need to be refined by adding other tags.

Clicking on a tag in the tag cloud adds that tag to the current goal, which is initially empty. The tag cloud shows only those tags that can be added to the goal such that the new goal can be planned, i.e., at least one flow can be composed to satisfy the goal. In further sections we will discuss how this is achieved. Practically it means that the tag cloud reflects the current capabilities of the system.

Due to screen space constraints the tags that would otherwise appear in the smallest font may be completely removed from the screen. To accommodate advanced users who want to enter these tags directly, MARIO interface includes a search string where tags can be typed in. The search string also lets users add more than one tag to the goal.

3.2 Goal Refinement Interface

Goal refinement is the main mode of interaction between MARIO and its users. When one or more tags are specified as the goal, new control elements appear in the user interface, as shown on Figure 4. New elements display the flow that matches the goal and a preview of the output produced by that flow. The interface also includes a number of other elements that help user understand current system capabilities and refine the goal. All user interface elements shown on Figure 4 appear simultaneously on one screen, and user input committed to any element changes the contents of all elements.

3.2.1 *Current Goal*

The "Current Goal" element displays the set of tags that constitute the goal. These tags, shown in black font, are referred to in what follows as the *current goal*. On Figure 4, the current goal is the set {Sorted, Yahoo Answers, Yahoo News}. The user can click on each of these tags to remove that tag from the goal.

Each time the user changes the current goal, the composer generates and ranks possible alternative flows for that goal, computing sets of tags describing these flows. It also chooses one flow with the best rank among the alternatives and submits it to the execution environment. We will refer to this flow as the *selected flow*.

The description of each of the alternatives must include all tags of the current goal, but may also contain other tags. These additional tags for the selected flow are shown in gray font in the "Current Goal" element. This gives user an indication of how MARIO interpreted the goal. Being able to see this set of guessed tags is especially helpful when the goal is ambiguous, for example is based on a general concept. On Figure 4, Sorted was specified as part of the goal, and MARIO selected the flow described by ByTitleAsc (among other tags) to satisfy the goal.

3.2.2 *Parameters*

The selected flow can have one or more parameters. The selected flow on Figure 4 has one parameter, "Destination". The parameters are automatically initialized with default values, but the users can change parameter values using the edit controls inside the "Parameters" element. Depending on the runtime environment, it can be possible to change the values of the parameters without redeploying the flow. In Yahoo Pipes, for example, the parameters can be specified in the URL corresponding to the deployed flow.

3.2.3 *Composed Flow*

The "Composed Flow" element shows a graphical representation of the selected flow and its configuration parameters. This is especially useful for the advanced users who have a good understanding of individual service modules, and use MARIO to quickly create a flow for their needs. The users who are less familiar with the individual services may be able to get better understanding of the selected flow functionality from the guessed tags in the "Current Goal" and the contents of "Flow Output".

3.2.4 *Flow Output*

The "Flow Output" element shows the results produced by the selected flow. In our implementation it shows the feed produced by the selected flow. Note that this is the only user interface element cannot be populated until the selected flow is deployed and produces results. Depending on the runtime, in some cases it may take longer than the

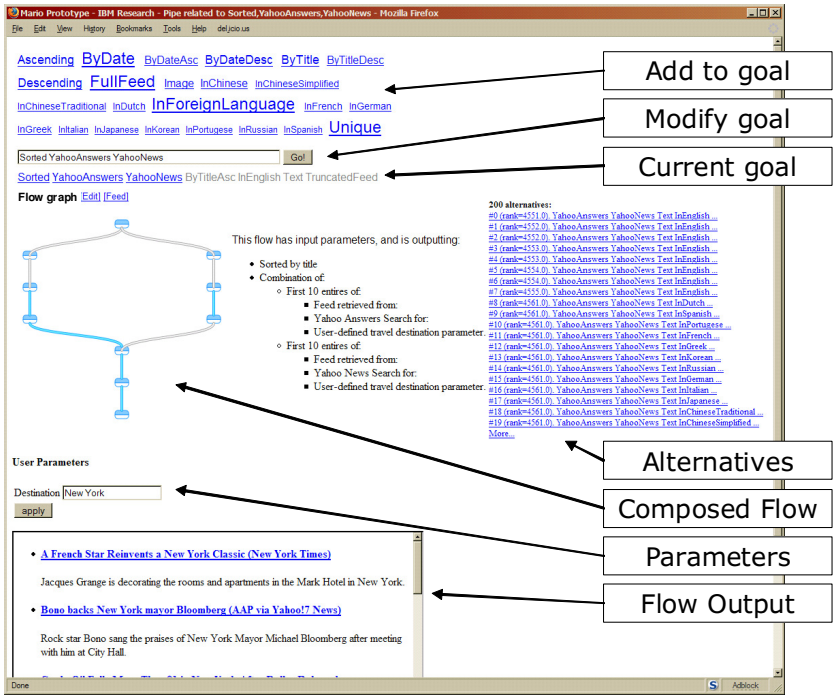

Figure 4: MARIO interface for goal {Sorted, YahooAnswers, YahooNews}.

user is willing to wait. In those cases the element can display a preview of the results obtained by other means.

3.2.5 *Add to Goal and Modify Goal*

The "Add To Goal" element shows a tag cloud similar to the one shown in Figure 3, but computed in the context of the current goal. In other words, the tag cloud in Figure 4 only shows the tags that can be combined with the current goal, such that there will exist at least one flow satisfying the new goal. Both the set of tags in the tag cloud and the size of the fonts used to display the tags in "Add To Goal" element may change depending on the current goal.

Clicking on a tag in the tag cloud adds the tag to the current goal, which results in composition and deployment of a new selected flow and changes the contents of all user interface elements.

The "Modify Goal" element allows specifying a new goal using a search string. This can be especially useful for experienced users who may find it tedious to click tags in "Add To Goal" or "Current Goal" elements to add or remove tags one by one. When the set of tags is large, the tag cloud may not show all of those tags, and several steps of goal refinement may be necessary to add a tag that is initially hidden. On the other hand, it can be the only way for the end user to discover that the tag exists and is supported by the system. The users who already know about the tag can type it directly into "Modify Goal" without intermediate refinement steps. After clicking on "Go" button, the set of tags entered in the search string becomes the new goal.

3.2.6 *Alternatives*

The list of alternative flows, their rankings and tag annotations are provided as reference to power users. Selecting and clicking one flow in the list of alternatives replaces the selected flow with the selected flow without changing the current goal. Some of the alternative flows can be suboptimal, i.e. there may exist better ranking flows with exactly the same description. Hence, the preferred way of selecting another flow is to modify the goal using other user interface elements, which guarantees that the selected flow is optimal for the selected goal.

3.2.7 *Commands*

The user interface provides access to a set of commands that operate on the selected flow (the commands are located above the flow graph in Figure 4). The set of commands depends on the functionality supported by the runtime. For example, "Edit" command can be used to open the Visual Editor, and "Publish" command can be used to make the flow public, i.e. accessible by others.

4. COMPOSITIONAL SEMANTICS

This section deals with the formal definition of the compositional semantics of a flow. We address this issue by defining a model for deriving the semantic description of a flow based on the descriptions of its individual components. A key characteristic of our model is that it captures not only the semantics of inputs and outputs, but also the functional dependency between the outputs and the inputs. This model can also be expressed using SPPL formalism (Stream Processing Planning Language, [10]) for describing planning tasks, which allows us to use an efficient planning algorithm for flows composition.

4.1 Composition Elements

4.1.1 *Objects, Tags and Taxonomies*

A taxonomy $T = \{t\}$ is a set of tags (i.e. keywords) t. An object o is described by a set of tags $d(o) \subseteq T$ selected from

the taxonomy T. An object can be, for example, a resource bookmark, as in del.icio.us [12], or a feed, as in Syndic8 [1].

In the simplest case, for example if T is formed as a folksonomy, by people specifying one or more tags to describe certain objects, the tags in T are unrelated and T is completely unstructured. Introducing a taxonomy structure in T, however, enhances query expressivity, as we explain below, and also helps keep tag-based descriptions succinct.

The structure of the taxonomy is described by specifying sub-tag relationship between tags. The following definition is the standard definition of a taxonomy sub-tag relation applied to tagging.

Definition 1. A tag $t_1 \in T$ is a sub-tag of $t_2 \in T$, denoted $t_1 :: t_2$, if all objects described by t_1 can also be described by t_2. The sub-tag relation is transitive, i.e. if $t_1 :: t_2$ and $t_2 :: t_3$ implies $t_1 :: t_3$ for $\forall t_1, t_2, t_3 \in T$.

For example, NewYorkTimes :: Newspaper. For notational convenience we will further assume that each tag is a sub-tag of itself, i.e.

$$\forall t \in T,\ t :: t.$$

If two tags $t_1, t_2 \in T$ are such that $t_1 :: t_2$ and $t_2 :: t_1$, these tags are synonyms, since by definition they describe the same set of objects. We will denote this as $t_1 \equiv t_2$.

4.1.2 Queries

Queries are used to describe the desired results produced by a composition (i.e., composition goals), or to specify the input conditions of an operator.

Definition 2. A tag query $q \subseteq T$ selects a subset $Q_q(O)$ of an object set $O = \{o\}$ such that each object in the selected subset is described by all tags in q, taking into account sub-tag relationships between tags. Formally,

$$Q_q(O) = \{o \in O | \forall t \in q\ \exists t' \in d(o) \text{ such that } t' :: t\}.$$

Note that this definition of a query remains equally effective in taxonomies with explicitly stated sub-tag relationships, as well as in configurations with implicit taxonomies, where the sub-tag relationships are not explicitly stated, but can be inferred from joint appearance of tags.

For example, consider a set of objects O_1 and a taxonomy T_1 where NewYorkTimes :: Newspaper, and some objects in O_1 are annotated with NewYorkTimes. Assume that O_2 is created from O_1 by annotating every object in the set $\{o \in O_1 | \{\text{NewYorkTimes}\} \subseteq d(o)\}$ with Newspaper tag, and taxonomy T_2 is the same as T_1 but with the sub-tag relationship between Newspaper and NewYorkTimes removed (thus defining an implicit taxonomy). As a result, for $q = \{\text{Newspaper}\}$ the selected subset will be the same in both sets of objects.

This is a very important property of the proposed approach. It allows mixing implicit taxonomies, typical of folksonomy-like bottom-up modeling approaches, with much more structured and elaborate top-down modeling, which is typical of taxonomies and ontologies. By effectively enabling an easy gradual transition from implicitly formed to explicitly stated sub-tag relationships between tags, as the model evolves, it greatly reduces the effort required for creating a first working set of descriptions compared to the top-down ontology-based modeling approaches, where the significant cost of defining taxonomies must be paid upfront.

4.1.3 Operators

An operator is a basic unit in the composition. Generally, it creates one or more new objects from a subset of existing objects. An operator can require no inputs. When one or more inputs are required, an input condition is specified for each input. The input condition is specified as a tag query, which must be satisfied by the corresponding object provided as input. The outputs are described by specifying tags that are added to and removed from the description of the new objects produced by the output. For example, consider a service that truncates an RSS feed to specified number of items. This service can be modeled by an operator that includes FullFeed in its input condition, and removes it from the output object description, adding ShortFeed.

The descriptions of the new objects functionally depend on descriptions of input objects. There are two methods of propagating information from the input to the output. The first, explicit, method involves using a typed tag variable that can be bound to one of the tags describing the input object, and then using this variable to describe one or more of the outputs. The type of the variable is a tag, and the variable can be bound to any single sub-tag of its type. In certain cases, however, operators must propagate sets of tags unrelated to the operator. For example, the truncation operator needs to propagate any tags describing feed origin, such as Newspaper. To enable the second method of propagation, a special "sticky" tag Ω is defined to serve as a label for automatically propagating tags. If any sub-tag of Ω appears in at least one input object description, it will be automatically added to the description of all output objects.

The following definition captures all of the properties of an operator explained above.

Let

- $p(f) \geq 0$ be the number of operator variables for operator f;
- $\vec{t}(f) = \{t_k(f) | t_k(f) \in T\}_{k=1}^{p(f)}$ be an array of tags representing the types of operator variables $\vec{v}$ for operator f;
- $n(f) \geq 0$ be the number of inputs of operator f;
- $\vec{q}(f, \vec{v}) = \{q_i(f, \vec{v}) | q_i(f, \vec{v}) \subseteq T\}_{i=1}^{n(f)}$ be an array of tag queries that define input conditions of operator f;
- $m(f) \geq 1$ be the number of outputs of operator f;
- $\vec{a}(f, \vec{v}) = \{a_j(f, \vec{v}) | a_j(f, \vec{v}) \subseteq T\}_{j=1}^{m(f)}$ be an array of sets of added tags for outputs of operator f;
- $\vec{r}(f, \vec{v}) = \{r_j(f, \vec{v}) | r_j(f, \vec{v}) \subseteq T\}_{j=1}^{m(f)}$ be an array of sets of removed tags for outputs of operator f.

Given the above parameters of an operator, and

- an object set O;
- an array of tags $\vec{v} = \{v_k\}_{k=1}^{p(f)}$ assigned to operator variables, such that $v_k \in T$ and $v_k :: t_k(f)$;
- an array of input objects $\vec{o} \subseteq O$ satisfying the input conditions parameterized with $\vec{v}$, i.e., such that $\vec{o} = \{o_i\}_{i=1}^{n(f)}$ and $o_i \in Q_{q_i(f,\vec{v})}(O)$

we define the operator as follows.

Definition 3. Operator $f = \langle p, \vec{t}, n, \vec{q}, m, \vec{a}, \vec{r} \rangle$ is a function on the object set, defined as $f(O, \vec{v}, \vec{o}) = O \cup O'$, where $O' = \{o'_j | o \notin O\}_{j=1}^{m(f)}$ is the set of new objects produced by the operator, and where

$$d(o'_j) = \left[\bigcup_{i=1}^{n(f)} \{t' \in d(o_i) | t' :: \Omega\} \right] \cup a_j(f, \vec{v}) \setminus r_j(f, \vec{v}).$$

The definition above provides a formula for computing descriptions of new objects produced by the operator: the description of each object is the union of automatically propagated tags derived from Ω and operator-output-specific added tags, minus the set of operator-output-specific removed tags.

4.2 Composition

4.2.1 Composition Semantics

A composition of operators is defined simply as the result of applying one operator to the object set produced by another operator.

Definition 4. The composition of l operator instances formed by operators $f_1, f_2, \ldots f_l$ applied to object subsets $\vec{o_1}, \vec{o_2}, \ldots, \vec{o_l}$ and parameterized with tags $\vec{v_1}, \vec{v_2}, \ldots, \vec{v_l}$ correspondingly is the composite operator $f = \circ f_j,\ j = 1..l$ defined as

$$f(O) = f_l(\ldots(f_2(f_1(O, \vec{v_1}, \vec{o_1}), \vec{v_2}, \vec{o_2})), \vec{v_l}, \vec{o_l}).$$

Notice that $f(O) = O \cup O'_1 \cup O'_2 \ldots \cup O'_l$, where O'_i is the set of new objects produced by operator f_i. Also note that input objects for each subsequent operator can be selected from the object set produced by the preceding operator, i.e.

$$\vec{o_1} \subseteq O_0 \equiv O$$
$$\vec{o_2} \subseteq O_1 \equiv O \cup O'_1$$
$$\ldots$$
$$\vec{o_l} \subseteq O_{l-1} \equiv O \cup O'_1 \cup O'_2 \cup \ldots \cup O'_{l-1}$$

Definition 5. The composition is valid when the input conditions of each operator instance f_j are satisfied by the object array $\vec{o}_j$, i.e. $\forall i, j\ o_{j_i} \in Q_{q_{j_i}(f_j, \vec{v_j})}(O_{j-1})$.

Subsequent instances of operators may use objects produced by preceding operators as inputs, i.e. there could exist i and j, $i < j$ such that $\vec{o_j} \cap O'_i \neq \emptyset$. In other words, there is a data dependency between $\vec{o_j}$ and $\vec{o_i}$. Data dependencies between operator instances within a composition can be represented using a data dependency graph where arcs connect operator outputs to inputs of other operators, similarly to the flow graph in Figure 1. Note that under this model the directed data dependence graphs will always be acyclic.

4.2.2 Goal-Driven Composition

The problem of goal-driven composition can now be simply defined as the problem of finding a composition of operators that produces an object satisfying a given query. As an additional simplifying assumption, we assume that the composition is applied to an empty object set. This assumption is not significantly constraining, since the initial objects can always be produced by operators that do not require any input objects. On the other hand, the assumption allows uniform modeling of both feeds and services as operators.

Given a composition problem $\mathcal{P}(T, \mathcal{F}, g)$, where:

- T is a tag taxonomy,
- $\mathcal{F} = \{f\}$ is a set of operators,
- g is a composition goal specified as a tag query, $g \subseteq T$,

the solution set is defined as follows.

Definition 6. The set of solutions $\mathcal{S}(T, \mathcal{F}, g)$ to the goal-driven composition problem $\mathcal{P}(T, \mathcal{F}, g)$ is the set of all valid compositions F of operators in $\mathcal{F}$ such that

- $Q_g(F(\emptyset)) \neq \emptyset$;
- for all operator instances in F, at least one object produced by this instance serves as input to another operator instance, or satisfies the goal query.

The second condition in the definition above helps eliminate from consideration inefficient compositions that have dead-end operator instances producing unused objects.

4.2.3 Composition Ranking

Before the set of compositions $\mathcal{S}(T, \mathcal{F}, g)$ can be presented to the user, the compositions must be ranked, with those most likely to satisfy user's intent appearing first in the list. The ranking is based on a heuristic metric reflecting composition quality. Each operator $f \in \mathcal{F}$ is assigned a fixed cost $c(f)$. Cost of an operator instance in a composition is equal to the cost of the corresponding operator.

Definition 7. Rank $rank(\hat{f})$ of the composition

$$\hat{f}(O) = f_n(\ldots(f_2(f_1(O))\ldots)$$

is the sum of the costs of operator instances, i.e.

$$rank(\hat{f}) = \sum_{i=1}^{n} c(f_i).$$

By default for all operators $c(f) = 1$. Hence, the best compositions are the shortest ones. During configuration of the system, the $c(f)$ can be changed for some operators to reflect feed or service quality.

4.2.4 Goal Refinement Tag Cloud

The refinement tag cloud, as shown in "Add to Goal" area of user interface in Figure 4, provides valuable help to the user in refining the goal. The tag cloud is simply a popularity-weighted set of tags computed over the descriptions of outputs of all compositions in a solution set $\mathcal{S}(T, \mathcal{F}, g)$. In theory, if the goal g is empty, the tag cloud is computed over all valid compositions. Although the set of all compositions may indeed be very large, the set of compositions with differently described outputs is much smaller. The planner that we describe later in this paper can compute the tag cloud without constructing all compositions.

Note that the queries in our model behave as though the super-tags from the taxonomy are always included in object description with the corresponding sub-tags. The same approach should be used during tag cloud computation. Even if the super-tags are not included in object description explicitly, they are added to the description automatically for the purposes of computing the weights in the tag cloud. This ensures that even if certain tags do not accumulate enough weight to appear in the visible portion of the tag cloud, they add weight to their super-tags, and will still be accessible through those super-tags.

5. APPLICATION EXAMPLE

In this section we describe how the concepts introduced at an abstract level in the previous section can be applied in practice, using the set of Yahoo Pipes modules as an example of a set of feed processing services.

5.1 Execution Runtime

The Yahoo Pipes modules for processing feeds are only available through the visual editor. Therefore, MARIO can

```
<flow>
<flowInput name="SearchQuery"/>
<call name="yAnswers" class="com.example.URLBuilder">
  <input name="prefix"
   value="http://answers.yahoo.com/rss/searchq"/>
  <input name="suffix" link="SearchQuery"/> </call>
<call name="yNews" class="com.example.URLBuilder">
  <input name="prefix"
   value="http://news.search.yahoo.com/news/rss"/>
  <input name="suffix" link="SearchQuery"/> </call>
<call name="fetchNews" class="com.example.FetchFeed">
  <input name="url" link="yNews"/> </call>
<call name="fetchAnswers" class="com.example.FetchFeed">
  <input name="url" link="yAnswers"/> </call>
<call name="truncNews" class="com.example.Truncate">
  <input name="feed" link="fetchNews"/> </call>
<call name="truncAnswers" class="com.example.Truncate">
  <input name="feed" link="fetchAnswers"/> </call>
<call name="union" class="com.example.Union">
  <input name="feed1" link="truncAnswers"/>
  <input name="feed2" link="truncNews"/> </call>
<call name="sort" class="com.example.Sort">
  <input name="feed" link="union"/> </call>
<flowOutput link="sort"/>
</flow>
```

Figure 5: Example flow description.

compose a flow of Yahoo Pipes modules, but cannot deploy it as a pipe. Deploying a flow is necessary, however, to show the preview of flow output in MARIO interface (Figure 4).

To overcome this difficulty, as part of our implementation we have built a simple Java-based runtime that plays the same role. Each service in this runtime implements interface `Service` with a single public method named `process` that receives and returns a hashmap containing input and output object values:

```
interface Service {
  Map<String,Object> process(Map<String,Object> inputs);
}
```

The set of hashmap keys used to identify input and output objects in the input and output hashmaps is specific to each service. A separate description is provided to specify the hashmap keys recognized by the service, as well as tag-based annotations on inputs and outputs. This description is then used to construct a description of an operator. Service implementation can invoke web services for advanced processing, such as language translation, when necessary.

A simple XML format is used to define a flow and deploy it in the runtime. Once deployed, the flow can be called with user-defined values of parameters, and will produce results. Figure 5 presents a sample description corresponding to the flow shown on Figure 1.

Flow definition consists of flow inputs (i.e., external parameters), calls (i.e., operator instances) and a flow output. The call elements instruct runtime about the Java classes to be used to process data, and the input objects to be included in the input map. The objects can be specified as string values by specifying `value` attribute, or linked to outputs of other calls by specifying a `link`. In the example above, each output map contains just one element, so specifying the name of the call is sufficient to describe a link. Otherwise, for operators that produce more than one object, "callName.elementName" notation is used.

5.2 Descriptions

MARIO requires descriptions of services, feeds, parameters, and taxonomies. These descriptions are translated into operators and other elements of the model described in Section 4, which is then used by the planner to generate flows. All descriptions can be specified in one file or broken into multiple files, which are then automatically combined into one logical file before processing.

```
tag {_URL - _Format}
tag {_Feed - _Format}
tag {_Source - _StickyTag}
tag {FrontPage - _Source}
tag {Opinion - _Source}
tag {Travel - _Source}
tag {News - _Source}
tag {Newspaper - News}
tag {Blog - _Source}
tag {NewYorkTimes - Newspaper}
tag {NYTFrontPage - NewYorkTimes FrontPage}
tag {Yahoo - _Source}
tag {TruncatedFeed - _FeedLength}
tag {FullFeed - _FeedLength}
tag {InForeignLanguage -_Language}
tag {InEnglish - _Language}
tag {InFrench - InForeignLanguage}
tag {Sorted - _SortOrder}
tag {_NotSorted - _SortOrder}
tag {NaturalOrder - _NotSorted}
tag {Unsorted - _NotSorted}
```

Figure 6: Fragment of a tag taxonomy.

5.2.1 Tag Taxonomies

Taxonomies are described by specifying sub-tag relationships between tags. A tag does not need to be explicitly declared before it is used, but a `tag{}` statement is necessary to declare parents of a tag, which follow after '-', for example: `tag {NYTFrontPage - NewYorkTimes FrontPage}`.

Tag names beginning with underscore "_" are hidden tags that are never displayed in user interface, but otherwise behave as normal tags. Hidden tags can be used to express composition constraints that are internal to the system, for example, type constraints. The special tag Ω is represented as `_StickyTag`. Figure 6 shows a fragment of tag taxonomy used in our experiments.

5.2.2 Feed Descriptions

In the example of feed description below the output annotation uses tags to describe the content of the feed, as well as its language.

```
feed NYTFrontPage {
  output{ NYTFrontPage InEnglish _URL }
  url {http://www.nytimes.com/services/
 xml/rss/nyt/HomePage.xml} }
```

Such descriptions can be generated automatically, for example using Syndic8 tags and default values for language. The description is translated into an operator that has no inputs, and produces a single output object tagged with all tags used in output annotation. If this operator is included in a flow composed by the planner, during flow execution the runtime will bind the corresponding operator instance to a built-in service that returns the URL string as a single entry in the hashmap of output objects.

5.2.3 Service Descriptions

Each service can have a number of inputs and outputs. Service description is directly translated into an operator that requires and produces the corresponding number of objects. For example, the following describes a FetchFeed service.

```
service FetchFeed {
  java {com.example.FetchFeed}
  var {?lang - _Language}
  input[url]{ ?lang _URL }
  output{?lang FullFeed NaturalOrder _Feed Text} }
```

This description uses a variable `?lang` of type `_Language`, and declares an input and an output. The output list enumerates tags added by the operator. Tags that are preceded with ~ are interpreted as removed tags.

Note that sub-tags of `_Language` are not sticky (i.e. are not derived from the special tag Ω represented as `_StickyTag`), and therefore must be propagated explicitly from input to output using a variable. However, if `FetchFeed` operator is applied to the output of the feed operator in the example above, `NYTFrontPage` tag will be propagated to the output of `FetchFeed` as well, since that tag is sticky according to the taxonomy in Figure 6.

Each input and output in the description can have a port name specified in square brackets. In this example only the input has a port name "url". The port name is the name of the entry in the hashmap that is used to carry the corresponding input or output object. Since there is only one output port, the runtime does not need to know the name of the output object. Finally, `java` description element specifies the name of the Java class that implements the service.

5.2.4 Flow Parameters and Constants

Flows that take external parameters can also be composed using the same framework. When two or more services within a flow are parametric, the planner can decide whether to expose the service parameters as one input parameter of the flow, or as several separate parameters. This is achieved by using tags to describe service input parameters (as inputs to services), and representing parameter values similarly to feeds, i.e. as operators that produce a single object described by tags. The following is an example of service description that has an external parameter.

```
param Destination {
  default{London}
  output{_SearchQuery Travel} }

service YNewsSearchURL {
  java {com.example.URLBuilder}
  input[prefix]{"http://news.search.yahoo.com/news/rss"}
  input[suffix]{_SearchQuery}
  output{_URL YahooNews InEnglish} }
```

Service `YNewsSearchURL` has two inputs, but the corresponding operator will have only one input. The constant string in quotes is used to initialize the `prefix` parameter to a constant. In the plan `suffix` parameter will be connected to the object produced by the operator corresponding to `Destination` service. Note that including constants into the description makes it possible to specify different semantic descriptions for different configurations of the same service.

5.2.5 More Service Description Examples

The following examples from the sample application further illustrate different services that can be described in this model.

```
service Truncate10 {
  java {com.example.Truncate}
  var {?lang - _Language}
  var {?sort - _SortOrder}
  input[feed]{_Feed  ?lang FullFeed ?sort}
  input[length]{"10"}
  output{_Feed ?lang ShortFeed ?sort} }

service TranslateEnFr {
  java {com.example.Translate}
  var {?len - _FeedLength}
  input[feed]{_Feed  InEnglish ?len NaturalOrder}
  input[fromLanguage]{"en"}
  input[toLanguage]{"fr"}
  output{_Feed InFrench ?len NaturalOrder}  }

service Union2 {
  java {com.example.UnionOfTwoFeeds}
  var {?lang - _Language}
  var {?len - _FeedLength}
  input[feed1]{_Feed  ?lang NaturalOrder ?len}
  input[feed2]{_Feed  ?lang NaturalOrder ?len}
  output{_Feed ?lang ?len Unsorted}  }
```

These descriptions were used in the application shown in Figure 4. In addition to the goal shown in that figure, the application supports a set of interesting goals, such as NewYorkTimes InFrench, Technology News ByDate, NewYorkTimes Flickr Image, etc. We will continue using this application for illustration in the next section, where it is used as a benchmark to evaluate planner performance.

6. PLANNING ALGORITHM

To compose flows according to the semantic model described in Section 4, we have developed an improved version of SPPL planner [11] by proceeding with both plannability exploration and analysis of related goals. We also added functionality necessary for generating tag clouds. The semantic model naturally maps to SPPL formalism, which describes the planning domain as a set of actions that can be composed by the planner. The set of actions is created based on the set of operators. Action preconditions, described by predicates, are created based on operator input conditions. Tags are represented as types in SPPL, and preconditions are specified using a variable of the corresponding type. Action effects also are mapped to operator outputs. SPPL predicate propagation mechanism is used for propagation of sticky and regular tags.

6.1 Presolve Optimizations

During the presolve phase the planner performs action grounding and obtains results of preliminary problem structure analysis that are later used for optimizing the search.

Intelligent Grounding. Grounding of actions during presolve may lead to a combinatorial explosion in the number of actions. To avoid the explosion, the planner performs grounding intelligently. First, it analyzes the set of ground actions in the first tier, i.e. those actions that can be applied directly in the initial state. Next, it creates the set of possible groundings for actions that can be applied to the results of the actions in the first tier. This procedure is repeated, until a steady state is reached, and no new ground actions are created.

Source Grouping. Source actions are actions that do not have inputs. Such actions correspond to operators without inputs, and most often correspond to feeds. In certain cases it is possible to combine multiple sources into one action to reduce the total number of actions considered by the planner. In particular this is possible when the output description of two sources cannot be distinguished using any of the input conditions of other actions, i.e. all input conditions are either satisfied or not satisfied by both sources. Source grouping procedure is carried out after grounding,

and results in significant performance improvements in configurations with large number of sources.

Construction of an Action Graph. The planner analyzes compatibility of inputs and outputs of ground actions by comparing the effect and precondition predicates. This procedure helps the planner to eliminate connections between incompatible actions during search. The result of this domain preprocessing is an *action graph* where directed edges represent potential connections between an effect of one ground action and precondition of another ground action. Action graphs can contain cycles. An important property of the action graph is that for any connection between action instances in any valid plan there is a directed edge between the corresponding ground actions of the action graph.

In addition to speeding up the search, an action graph can be used for reachability analysis. If the planning task has a solution there must be a path from all goals in $\mathcal{G}$ to at least one of the initial state object. During presolve, the planner builds shortest path trees from the goals to all nodes of the action graph. Actions that are not on any of the paths from goal to initial state are labeled as disconnected and are not considered during plan search. The shortest distances to the goal in action graph are used to direct plan search.

6.2 Plan Search

The planner searches for plans using a forward search strategy. It creates new objects by applying actions that have satisfied preconditions. At the initialization of the algorithm the current set of objects S is empty. A set of actions $\bar{L}$ is also created, containing the ground actions for which all preconditions can be satisfied by objects contained in S. The planner applies actions from $\bar{L}$ one by one. The ground action to be applied next is an action from $\bar{L}$ that has the least distance to the goal based on reachability analysis. The action is skipped if the cost of the associated subplan is higher than the cost bound $\vec{B}$. When applied, a new action instance produces a set of new objects, one object for each effect. The predicates on the new objects are computed using predicate propagation rules, and the objects are added to S. New candidate action instances are determined using the action graph and added to $\bar{L}$.

Algorithm SPPL_Planner(Π)

Presolve

1. Action set $\bar{A} \leftarrow$ Intelligent_Grounding(Π).
2. $\bar{A} \leftarrow$ Source_Grouping($\bar{A}$);
3. $G(\bar{A}) \leftarrow$ Action_Graph($\bar{A}$);

Forward Search

4. $S \leftarrow \mathcal{I}$; $\bar{L} \leftarrow \{\bar{a} \in \bar{A} \,|\text{prec}(\bar{a}) \subseteq S\}$;
5. **for each** $\bar{a} \in \bar{L}$
6. **for each** $\hat{a} \in$ instances$(\bar{a}, S, G(\bar{A}))$
7. **if** $\vec{\text{cost}}(\text{subplan}(\hat{a})) \leq \vec{B}$ **and** $[\exists o \in \text{effect}(\hat{a}) : o \notin S$ **or** $\nexists p \in \text{subplans\_producing}_S(\text{effect}(\hat{a}))$ such that $[p \subseteq \text{subplan}(\hat{a})$ **and** $\vec{\text{cost}}(\text{subplan}(\hat{a})) \geq \vec{\text{cost}}(p)]]$
8. $S \leftarrow S \cup \{\text{effect}(\hat{a})\}$;
9. $\bar{L} \leftarrow \bar{L} \cup \left\{\bar{a}' \in \bar{A} | \text{prec}(\bar{a}') \subseteq_{G(\bar{A})} \text{effect}(\hat{a})\right\}$;
10. **if** $\exists \bar{g} \in \mathcal{G} :$ effect$(\hat{a}) \subseteq_{G(\bar{A})} \bar{g}$ **and** $\mathcal{G} \subseteq S$
11. **add_plan_candidate**((subplan$(\hat{a})$));
12. $\bar{L} \leftarrow \bar{L} \setminus \{\bar{a}\}$;

The same object can be produced by many different subplans, and each time the same object is produced, a subplan (i.e. a set of action instances and their connections) for producing the object is registered. The subplan has an associated cost vector. Once a subplan is registered, the action graph is used to create a list of candidate new actions that can be applied next. The search algorithm terminates when no actions can be applied and no new objects can be produced.

An important optimization technique used during plan search is a verification step (line 7) that prevents registration of a new subplan, if the same object can be produced by a subset of that subplan with lower or equal cost. For many actions the predicates of the output satisfy the input preconditions, and without this verification step the search would cycle producing new identical instances of the same object. Hence, the number of action instances created by the planner is significantly reduced in those cases. The original backward search based SPPL planner [10] does not perform the comparison step, which is difficult to implement in backward search, and our forward search planner is therefore significantly more efficient on this type of planning tasks.

The result of the search is a set of candidate plans that satisfy the goals. The candidate plans are then sorted by cost, and only the best plans are returned. Note that the search can be terminated early, producing only a subset of possible candidate plans. Hence, it is possible to obtain a number of suboptimal plans before the search finishes, which can be important if planning time is limited.

The same algorithm can be enhanced to generate tag clouds while searching for plans. The tag weight computation for the tag cloud uses tag membership statistics that are collected from descriptions of outputs of candidate plans.

6.3 Complexity Challenges

As is shown in [10], SPPL planner performs much better in typical flow composition problems than AI planners. Problems that are easily solved by SPPL planner can at the same time be very difficult for the best general AI planners. By making objects a part of the domain model, SPPL planner avoids unnecessary grounding and symmetries, and the search space is reduced by an exponential factor as a result.

However, the problem of finding optimal plans remains a difficult one. In general, there do not exist optimal or constant-factor-approximation SPPL planners that can guarantee termination in polynomial time on all tasks unless $P{=}NP$. In particular, Bylander has shown that STRIPS planning, which is a special case of general SPPL planning, is PSPACE-complete [3]. Nevertheless, we have shown empirically that our planner can find plans for large tasks in very short time [11].

6.4 Performance Evaluation

We have measured the response time of the flow composer by measuring the time it takes to generate a flow and a tag cloud for a given goal. In MARIO user interface this is a measure of the delay between user changing the current goal using one of the controls, and rendering of the next screen. Since the performance of the runtime can be variable, we did not include the time to deploy the flow, run it and render results. The measurements were taken on a PC with a dual-core Intel CPU at 1.86GHz and 1GB RAM.

As a first experiment we have created descriptions of several feeds and Yahoo services modules (including translation, sorting, truncation, union and keyword extraction) together with descriptions of external input parameters resulting in 73 SPPL actions, 24 of which corresponded to feeds. With

this set of descriptions, planning any goal reachable through the user interface took 5 seconds or less.

To experiment with larger sets of descriptions, we have imported additional descriptions of feeds from Syndic8.com. Planning times (in seconds) for several sample goals in this setting are presented in the table below. The columns correspond to the number of described feeds.

	100	150	200
first tag cloud ("any feed" goal)	0.41	0.51	0.61
{Sorted, Travel, YAnswers, YNews}	0.09	0.10	0.11
{NewYorkTimes, InFrench}	0.72	0.91	1.03
{NewYorkTimes, Image}	0.73	0.92	1.05

As these results show, MARIO can provide instantaneous response to user requests while analyzing large sets of feeds and creating complex flows and tag clouds, and quickly generate user interface screens, such as Figure 4.

7. RELATED WORK

Work has been done on automatic goal-driven composition in the past, especially within the AI planning and semantic web communities. Some of this work uses ontologies and associated standards such as OWL-S to describe components used in composition [13]. Other work uses process models or transition systems [9]. In this work we developed a radically simplified tag-based component description approach to reduce the knowledge engineering work required upfront in order to start using an automatic composer.

Other differences of our approach are the innovative "wishful search" abstraction and the underlying formalism of planning using tag taxonomies and actions that create new objects. In prior work, earlier versions of this formalism called SPPL were used for stream processing and ontology-based semantic web service composition [10, 11, 6]. In the current work we have significantly extended the SPPL planner to add support for tag taxonomies, tag-based operator descriptions, added tag cloud computation functionality, and developed an innovative user interface based on tag clouds.

A formalism that is similar in expressivity to SPPL, and in certain aspects exceeds it, has been proposed in Semantic Streams system [14], which allows users to pose queries based on the semantics of sensor data. We believe that SPPL planners are a better choice MARIO, since they can be extended to compute tag clouds and have been shown to be highly scalable with the number of operators [10, 11].

Interesting results have also been published on intelligent and incremental interaction with users, that are in the same spirit as the interactive goal refinement in MARIO. ACE [2] helps users incrementally formalize text statements by making formalization suggestions based on ontologies. RIA [16] can interpret multimodal inputs, including typed-in text, and dynamically finds information that is tailored to user's interaction context, including interaction history. However, we are not aware of any prior work that proposed taxonomy-supported iterative refinement of composition goals and making use of planner output generated for partial goals.

8. CONCLUSIONS

We have developed MARIO, a tool for automatic data mashup composition that implements a new "wishful search" pattern of interaction with the user. In a wishful-search-enabled catalog of feeds and services, users state their wishes by selecting tags that describe a feed that they would like to subscribe to. The composer then matches their request to an existing feed, as in regular search, or, if such a feed does not yet exist, it "grants the wish" by producing a new feed using an automatically composed flow. It also helps users understand what they can wish for, by providing context-dependent controls for refining the requests. In this paper we show that all of this can be done very efficiently and in real time, which opens many application possibilities.

We believe that wishful search is a general user interaction pattern that can be implemented in a wide variety of systems. To become useful within a system, it only requires the system to have documents and components that process documents. One important application is Web Service composition. In that scenario, the pattern also helps addresses the problem of dynamic web service discovery, by reflecting changes to service descriptions in the results of wishful search. Other potential application areas include Stream Processing and Grid, or any other component-based system where the end users will benefit from interactive goal-driven composition provided by wishful search.

9. REFERENCES

[1] J. Barr and B. Kearney. http://www.syndic8.com/, 2001.

[2] J. Blythe and Y. Gil. Incremental formalization of document annotations through ontology-based paraphrasing. In *WWW'04*, 2004.

[3] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[4] IBM Corp. http://www.projectzero.org/, 2007.

[5] IBM Corp. Damia. http://services.alphaworks.ibm.com/damia/, 2007.

[6] Z. Liu, A. Ranganathan, and A. Riabov. A planning approach for message-oriented semantic web service composition. In *AAAI'08*, 2008.

[7] /n software inc. RSSBus. http://www.rssbus.com/, 2007.

[8] T. O'Reilly. Pipes and filters for the Internet. http://radar.oreilly.com/archives/2007/02/pipes_and_filte.html, February 2007.

[9] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite BPEL4WS web service. In *ICWS*, 2005.

[10] A. Riabov and Z. Liu. Planning for stream processing systems. In *AAAI'05*, July 2005.

[11] A. Riabov and Z. Liu. Scalable planning for distributed stream processing systems. In *ICAPS'06*, 2006.

[12] J. Schachter. http://del.icio.us/, 2003.

[13] E. Sirin and B. Parsia. Planning for Semantic Web Services. In *Semantic Web Services Workshop at 3rd ISWC*, 2004.

[14] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *EWSN'06*, 2006.

[15] Yahoo, Inc. http://pipes.yahoo.com/, 2007.

[16] M. X. Zhou, K. Houck, S. Pan, J. Shaw, V. Aggarwal, and Z. Wen. Enabling context-sensitive information seeking. In *IUI '06*, 2006.