Mashing Up Search Services

Mashup languages offer new graphic interfaces for service composition. Normally, composition is limited to simple services, such as RSS or Atom feeds, but users can potentially use visual mashup languages for complex service compositions, with typed parameters and well-defined I/O interfaces. Composing search services introduces new issues, however, such as determining the optimal sequence of search invocations and separately composing ranked entries into a globally ranked result. Enabling end users to mash up services through suitable abstractions and tools is a viable option for improving service-based computations.

Daniele Braga, Stefano Ceri, and Davide Martinenghi
Politecnico di Milano, Italy

Florian Daniel
University of Trento, Italy

The past few years have witnessed end users' increasing involvement in the content creation process of modern Web applications (as with Wikipedia) and the emergence of social Web applications (such as YouTube or MySpace). Although these applications benefit from collective user-generated content, a growing user community is also trying to profit from existing content and services by developing its own Web applications. The phenomenon, commonly known as *Web mashups*, is driven mostly by skilled users because reusing third-party content and services is a nontrivial task. This is especially true for Web services, which are based on the notion of programmable interface; for instance, integrating Amazon's search service with currency conversion or shipment services requires programming skills. *Service mashups*, or user-driven Web service integration approaches, gain momentum by also targeting less skilled users. They effectively alleviate the burden of service composition, but users still need to manage the flow of service calls and their I/O interaction.

In this article, we propose an enhanced service mashup language for graphically composing and automatically executing queries over online data-sharing services — that is, services that let users query remote data sources. We distinguish between exact and search services. *Exact services* provide answers that consist of unranked data items that aren't associated with any assessment of their confidence or quality with respect to the input data. Due to a potentially large number of items, *search services*, on the other hand, require ranking composition and properly managing result sets. When composing answers from multiple services, we must produce the output in a global ranking that appropriately combines the various partial rankings. This explicit distinction between service types distinguishes our approach from other mashup languages, such as Yahoo Pipes (http://pipes.yahoo.com/pipes/) or Damia (http://services.alphaworks.ibm.com/damia/).[1]

Our proposed approach operates without individual business protocols regulating possible long-lasting interactions between a client and the service. Our services are stateless because subsequent invocations of the same service are independent, and there's no need to pass correlation data from one call to another. Service composition scenarios that preserve the state of computations require more sophisticated instruments — such as the Business Process Execution Language (BPEL; www.oasis-open.org/committees/wsbpel/) — and are beyond ordinary users' reach; therefore, they are out of the scope of our formalism. Nonetheless, our proposed approach is expressive enough to cover a wide range of complex queries over the Web.

Designing Service Mashups

Let's consider a complex query that an academic user belonging to the local database community might issue: "Find all database conferences held within the next six months in locations with an average temperature at the time of the event above 28°C, reachable by a low-cost flight from Milan, and offering affordable accommodation in a five-star hotel."

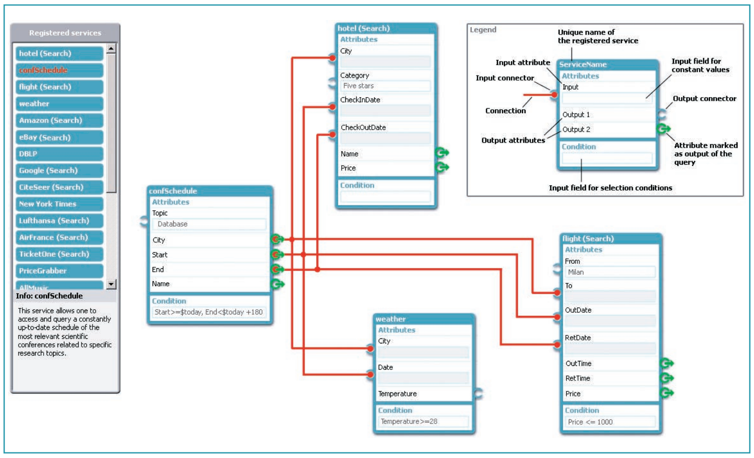
Presently, no general-purpose search engine, Google or Yahoo, can satisfactorily answer such a multidomain query. Although a search service might cover a single domain, the user must orchestrate their interaction. In fact, without appropriate support for search-service composition, the only feasible way to deal with such a query is to separately invoke dedicated services and then feed one search's results as inputs to another, or to compare search results manually. We propose a mashup language that can express this query as well as a system architecture that can accept, optimize, and process the query and then produce the result.

In our visual language, we divert the focus away from Web services' technical aspects (such as standards and description languages) and simply regard each service as a signature with a name and a list of attributes. Each attribute is associated with a domain. Instead of concrete domains, such as String, we use abstract domains, which have an underlying concrete domain and let us distinguish strings representing, for example, city names from strings representing conference topics. We organize the abstract domains into a hierarchical taxonomy that represents the shared knowledge necessary to support service interoperability. We further classify attributes as either input or output attributes. This characterizes the data flow corresponding to a service invocation: the caller provides values for all input attributes and, in response, receives values for all output attributes. Technically speaking, the language lets us build directed acyclic graphs with nodes that are service invocations and arcs that are connections between services representing parameter passing (from a service's output to another service's input).

Services are made available for use in the framework or platform via explicit registration. For each service, the framework designer (an expert in service integration who knows the services' semantic domains) specifies its signature and declares the abstract domain of each input and output parameter. The framework designer is responsible for choosing an existing domain or creating a new one to comply with the semantics of the service being registered. Once registration occurs, the framework can then check compatibility between parameters and even suggest join paths. Indeed, registration is crucial for the actual joinability of parameters from different services.

A huge amount of work has been done (and still needs to be done) on semantic issues to support and automate data integration and schema matching. (See related work for an interesting survey on the topic.[2]) In this article, we avoid data-integration issues and rely on the framework designer's knowledge and skills to resolve interoperability issues. Of course, not all services will lend themselves to seamless registration, despite the framework designer's best effort.

Figure 1 shows the user interface's two regions. The *canvas region* (the blank space at the right-hand side in the figure, not including the legend) contains the user's query and covers the whole screen, except for the left margin, which



This service allows one to access and query a constantly up-to-date schedule of the most relevant scientific conferences related to specific research topics.

Figure 1. Graphical environment for user-driven composition of service mashups. A user can drag registered services from the registered services region onto the canvas at the right, where the selected services are mashed up (composed) graphically by drawing connectors. Each added box represents a service call, and the connectors represent parameter passing.

contains the *registered services region*. The latter lists the names of all services (all in rounded boxes, with search services explicitly labeled) that are available to users to be mashed up to compose their query. Users can drag and drop the rounded boxes onto the canvas. Once on the canvas, a service is displayed in larger rounded boxes with two areas: attributes and condition.

The attributes area lists all of the service's attributes, each of which is equipped with a connector; the inputs have the connector on the left-hand side, the outputs on the right-hand side. Input attributes are further provided with input fields — that is, text boxes in which the user can type a constant value for that attribute.

The condition area contains an input field that helps specify a filtering condition, such as a comparison between an attribute and a constant value, or a threshold to restrict how many of the answers returned by the service are to be considered. Typically, conditions for exact services are Boolean selections on the service's attributes, whereas conditions for search services limit the size of the result set to a fixed number.

Placing a service on the canvas corresponds to adding an invocation of that service in the user's mashup. Out of the answers to the invocation, only those satisfying the selection condition contribute to the result. To enable the service invocation, users must bind a service's input attributes to an input value, which they can specify by entering the value in the corresponding input field or linking the input connector to another service's output connector. Linking thus expresses a precedence relation between service invocations. Of course, typing a constant into an input field disables the ability to attach a connector to that field and, symmetrically, attaching a connector disables the text box. Users can place a service on the canvas several times; in this case, the graphical interface adds a number as a subscript to the service name to distinguish the occurrences.

To conclude this simple syntax of graphical operations (summarized in the legend in Figure 1), the user can select some of the services' output attributes as result attributes for the query. In such cases, the corresponding connector is marked in green and has an outgoing arrow. In practice, all search services should provide at least an output value because the actual ranking provided by a search service must be subject to users' validation and judgment.

Figure 1 shows four available services:

- *confSchedule*, an exact service requiring an input value for the topic attribute and returning city, start date, end date, and name of conferences covering that topic;
- *weather*, an exact service returning the temperature of a given city at a given date;
- *hotel*, a search service returning accommodation offers (hotel name and price), sorted by price, corresponding to a given city, category, and check-in and check-out dates; and
- *flight*, a search service returning price, departure time, and return time (sorted by price) for low-cost travel solutions corresponding to the given origin, destination, and dates.

Users form the first part of the query by filling in the constant value "Database" in the input field corresponding to `confSchedule`'s topic attribute and imposing as a condition in the respective panel that the conference's start and end date are between the current date (captured by the environmental variable `$today`) and 180 days after that.

The `City`, `Start`, and `End` output attributes feed the input arguments of the other three services on the canvas. `City` is also used as input for weather and hotel and as a flight destination. Because we know the abstract domains of all the registered services' attributes, it's easy to draw connections between services. When a user clicks on an output attribute, the system highlights all attributes (on other services) with a compatible domain and, as soon as the user selects one, draws the corresponding connection. For example, when `City` in `confSchedule` is selected, the system highlights `City` in hotel, weather, and from and to flight. The behavior is similar when the user clicks on an input attribute, except that it isn't possible to draw a connection between two input attributes.

The rest of the query is self-explanatory: some input values are hard-coded in the query in the corresponding input fields ("Five stars" as hotel category and "Milan" as departure city), the relevant conditions (city and flight cost) are indicated in the respective condition panels, and the other input attributes that aren't fed with a constant value are matched with the corresponding outputs from `confSchedule`. Before executing a query, the system checks its executability. Namely, each service on the canvas must be callable — that is, either each of its inputs is filled with a constant or, inductively, if it is not filled with a constant then it is connected to an output of a callable service.

With this approach, the user query is specified declaratively: the semantics of the service graph in the canvas corresponds (using database terminology) to the Cartesian product of the involved services. The three classical relational query primitives (selection, join, and projection) are expressed as follows: selections are expressed in the various condition panels, the connections express equality between the values of two attributes (equi-joins, in formal database terminology), and projections on the output attributes are expressed by the green arrows. Each selection is addressed to a single service, and each join is an equi-join. Enriching the expressive power would require introducing less intuitive constructs, such as join and selection nodes. In our proposed formalism, instead, all nodes are services. The query's structure doesn't yet dictate the exact order of service calls, and several degrees of optimization exist concerning, for example, the execution sequence for partially ordered service calls and the degree of parallelism versus pipelining. The execution engine builds an access plan compatible with the implicit ordering determined by the connections that express the mapping of outputs to inputs (which imposes, for example, to invoke `confSchedule` before weather) and decides how to interact with services.

Currently available interactive feed aggregators, such as Yahoo Pipes and Damia, instead use a fully procedural approach: the user graphically assembles data feeds by fully specifying the data flows among the services as well as the operators used to combine them, and no mediation occurs between the language description and the underlying language implementation. Previous work[3] proposed an alternative procedural approach that uses a service-mashup-specific programming language for the Swashup platform; data mediators, service APIs, protocols, choreographies, and UIs are first-class language concepts immediately available to the developer who programs the mashup. In principle, Yahoo Pipes, Damia, and Swashup could express mashups over search services, but due to the lack of an explicit notion of search service, they can't handle such cases properly. In particular, these approaches don't support paging large result sets or combining ranked results from different services.

Executing Service Mashups

Our work on integrated search services is part of a more general research project, named Next Generation Search (NGS), funded by the Ministry of University of Italy (MIUR).[4] The project aims to integrate techniques for query answering over Web data sources[5] with strategies for joining results from different search engines.[6] Executing a mashup query requires instantiating several degrees of freedom, not fixed in the declarative user query, such as generating the actual schedule of service invocations, orchestrating and synchronizing such invocations, or joining the results from different services into a ranked list of outputs with global ordering that must be consistent with each of the partial orderings induced by the results of each search service.

We address such issues in part by translating the user query into a physical service access plan (at compile time) and in part during a plan's actual execution (at runtime). In particular, orchestrating the query's execution means scheduling and timing the invocations so as to ideally maximize the degree of parallelism, minimize the size of intermediate results, and promptly provide the user with the best results first. With this, we aim to generate the most promising physical access plan by leveraging several aspects:

- *Chunking.* Several Web services return results in chunks (or pages). The most popular example (at the user interface level) is the way in which Google returns paged results, with 10 references per page. For such services, the system that orchestrates the query execution must perform multiple fetches to retrieve their complete result set. In such cases, successive service calls represent one logical invocation because the result production after a given number of chunk requests continues with the next chunk. Chunking is an important optimization opportunity because it prevents the query engine from waiting for complete service responses and enables query processing over partial response sets. Also, it lets us pipeline the query execution.
- *Parallelism versus pipelining.* Depending on the precedence constraints that the input and output bindings enforce, it might still be possible to decide whether to invoke two (or more) services in a series or in parallel. As the invocation globally returns a single sequence of results, in both cases we say that the two services are joined. The execution engine exploits parallelism when it invokes independent services in parallel and processes the results as they're retrieved. Even when service calls have precedence dependencies, determined by the input and output bindings, it isn't necessary to wait for the first service to completely execute (in a blocking style) because its results can be fed as input to the second service as soon as they're available, in essence pipelining the join execution.
- *Merge-scan versus nested-loop join strategies.* The execution engine performs parallel joins by comparing items from two possibly ranked lists returned by two services. If we think of two Cartesian axes representing such lists, the points in the plan the axes define represent candidate results to be tested by the join condition, and the plan represents the search space for finding the best results — that is, those with higher ranking that also satisfy the join condition. Different join strategies correspond to different explorations of the search space.[6] Among the various strategies, we chose nested loop (NL) and merge scan (MS) because these classical join strategies (with their roots in traditional database optimization) provide two alternative ways to join services by taking into account their characteristics. The generation process of the physical access plan uses NL when one service dominates the other's invocation, either because it has a lower execution cost or because it produces all the good results with few fetches. In this case, the query engine explores the result space by executing all the relevant fetches for the dominating service and then a variable, user-controlled number of fetches for the other service. The generation process uses MS when there's no a priori distinction between the services to be joined. In this case, the query engine executes fetches in parallel for both services and produces results in output by traversing the search space diagonally.

These aspects show that, for a given query, multiple physical access plans might be able to produce a requested output.

Optimization Opportunities

Depending on the way in which the aforementioned aspects are utilized, there might be different optimization strategies for generating the access plan and orchestrating its execution. For instance, we might have a strategy that maximizes the parallelism among all services and another that minimizes the number of calls to search services by choosing an appropriate join strategy. The search for the best access plan is based on a branch-and-bound method, leveraging a set of strategy-specific heuristics based on information available from registration and service profiling.

Figure 2 gives the notation we use for modeling and representing physical access plans. An access plan has a unique start node (representing the user's input values) and a unique end node (representing the final result in output). As in an earlier work,[7] we define *selectivity* as the average number of tuples a service outputs in response to an invocation. Accordingly, a service is *selective* if its selectivity is at most 1, *proliferative* otherwise. Selective, exact services are represented as simple boxes; proliferative, exact services are represented as boxes labeled with a *. Search services are represented as boxes shaded by a gray trapezium icon, representing, from left to right, the decrease in results ranking as we move from the initial results to the final ones.

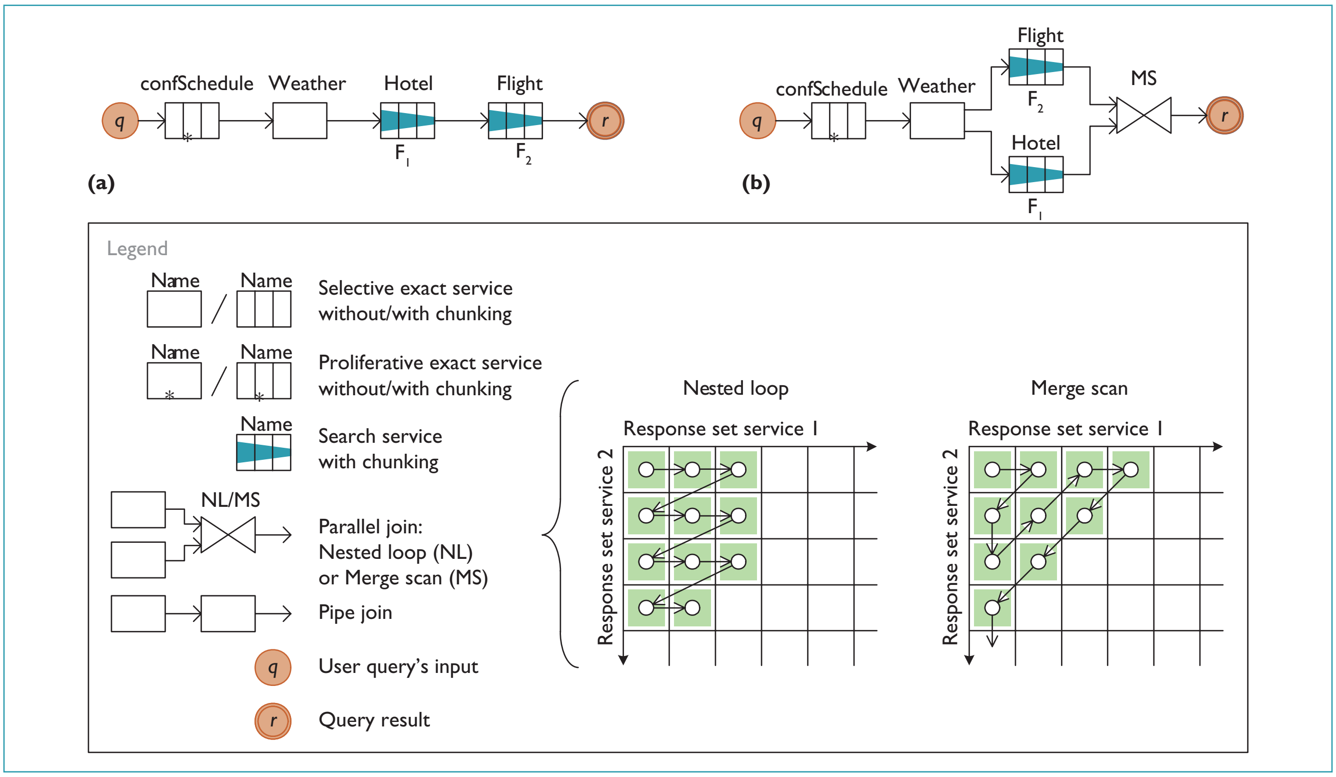

confSchedule
Weather
Hotel
Flight
F1
F2
q
r
(a)
(b)
MS
Legend
Name
Selective exact service without/with chunking
Proliferative exact service without/with chunking
Search service with chunking
NL/MS
Parallel join: Nested loop (NL) or Merge scan (MS)
Pipe join
User query's input
Query result
Nested loop
Merge scan
Response set service 1
Response set service 2

Figure 2. Graphical models of (a) and (b) two possible plans for the running example. The legend introduces the adopted modeling notation and explains the nested-loop and merge-scan join strategies in more detail.

In the physical access plan, it's important to specify whether each service is invoked via chunking or not so we can estimate the service's cost. Services called with output chunking have vertical lines splitting the box into smaller boxes. Search services that support chunking are always invoked in this way, whereas exact services are always invoked without chunking (in many cases, they return a single result tuple).

The notation also distinguishes between types of joins. A *parallel join* is represented by a join symbol with an NL or MS label expressing the chosen join strategy. The *pipe join* is denoted by an arc connecting two boxes, indicating that the join is computed by feeding the reached box with the other box's output.

Figure 2 shows alternative plans for our running example. In Figure 2a, the query engine invokes all services sequentially; it passes the locations of the first chunk of conferences to the weather service, and then passes those with the desired temperature to the search service for hotels (together with the binding on the dates instantiated by the first service's invocation). Finally, the engine invokes the flight service for those destinations with the cheapest five-star hotels available. In Figure 2b, the first returned conferences bind the locations and dates passed first to the weather service and then in parallel to the remaining services. Flights and hotels are joined by fixing a flight and trying to combine it with several hotels before moving to the next flight.

From a system perspective, there's a fundamental difference in how we cache partial results and in the order of invocations, but this is transparent to the end user. The most remarkable difference from the user's perspective is the order in which the query engine returns results because chunks are processed in a different order. If the user imposes a strict limit on the final output's size, the results produced by the strategies will be different, but still compatible with all single-service rankings. As an extension to this, we intend to provide optimization-aware users with interfaces for fine-tuning the optimization logic.

Cost Models

The access plan generation process compares candidate physical access plans according to a suitable cost metric so as to associate a cost estimation to each and find the optimal plan. Of course, we can define optimality with respect

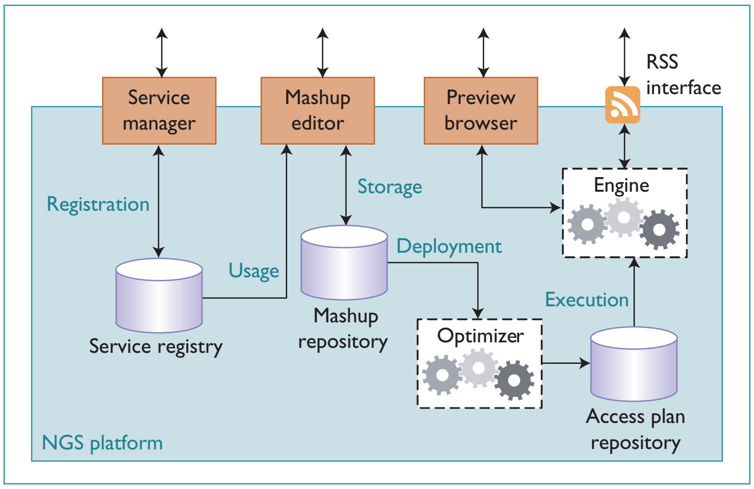


Figure 3. Functional platform architecture. The user mashes up registered services via the mashup editor and stores the result in the mashup repository; the optimizer translates the mashed up query into an executable plan; the engine executes the plan by invoking the respective services; and the user previews the result or consumes it via the RSS interface.

to different criteria. We can apply the following metrics (or their combination) to our scenario:

- The *bottleneck metric* considers the slowest service in the query (that is, the most demanding in terms of computation and response time) as the dominant one.
- The *request-response metric* is based on the overall number of service invocations and accounts for settings in which the transfer of data over the network is the dominating cost factor.
- The *time-to-screen metric* is based on the time required to present the user with the first output chunk, accounting for settings in which the user expects a prompt interaction.

Applying a cost metric thus lets us choose the plan that finally can be executed by the query engine in our service mashup platform.

Service Mashup Platform

Figure 3 shows our service mashup platform's functional architecture, which enables the hosted definition and execution of service mashups. The platform supports the following:

- *Service registration.* As anticipated, users can only compose services that have been previously registered with the platform. Registering a new service requires specifying its name and input and output attributes, each associated to its respective domain. This information is needed to assist the user in the interactive mashup definition. Also, the service must be classified as either an exact or search service. If a Web service exposes multiple operations, each operation requires its own service registration, which is performed via the service manager Web interface. Registered services are stored in a dedicated service registry. The service manager provides cooperative service-management features, such as browsing, searching, viewing, and so on, and is open to the public.
- *Service mashup.* The Ajax-based mashup editor implements the service mashup interface and allows for the server-side storage of completed mashups on the platform, in a "mashup repository," for execution and later editing. Mashup designs must be deployed by invoking the optimizer, which parses the mashup query and outputs the best access plan according to the techniques. An access plan repository collects deployed mashups' plans and enables the final execution.
- *Preview and test.* To assess the designed service mashups and the quality of the results, the platform provides a preview browser, which gives the user an immediate graphical response. The preview function builds on the internal engine, which is in charge of the physical access plan's execution and, hence, automatically invokes the necessary services. Thus, the query designer can easily debug, modify, and redeploy existing mashups. The final output is a flat list of all parameters in the service mashup environment.
- *RSS consumption.* Deployed mashups are also immediately online and ready for consumption via a dedicated RSS interface. The platform assigns each mashup a unique URI representing an RSS resource on the platform (created during the mashup's deployment), which, when accessed, enables the query's on-demand execution and the easy integration of the mashup into other applications.

Orthogonally to these features, the platform provides suitable identification and authentication mechanisms to enable users to decide which mashups are publicly accessible and which are private. In the current version, private mashups' final RSS mashup endpoints adopt a basic HTTP access authentication based on a username and password that users provide when accessing the RSS resource. We're currently finalizing the implementation of the Web front ends, and the platform's internals (repositories, optimizer, and engine) are in their test phase.

Service mashup systems are still highly unstructured, and applications often rely on hacked Web pages or services. In this respect, mashing up Web services poses several interesting challenges:

- The inherent complexity of service composition must be adequately abstracted (for example, using graphical modeling formalisms) to give end users easy-to-use development environments.
- Abstract formalisms must be equipped with suitable runtime environments capable of deriving executable service invocation strategies.
- Users must be able to master more complex dependencies among services (such as business protocols or choreographies[8]), override the optimization choices, and handle complex data formats.
- Advanced issues such as long-lasting service compositions, reliability, and transaction support must be provided to establish service mashups in business contexts.

This article addresses the first two issues. We've shown an intuitive visual modeling paradigm that lets users declaratively compose services in a drag-and-drop fashion and hides low-level implementation details. Our framework handles advanced issues, such as selecting an optimal invocation order of services and composing results extracted from multiple services. In the future, we expect to extend the framework to offer suitable interfaces to its internals and to provide support for long-lasting, reliable, and transactional services, thus covering the third and fourth issues.

References

1. M. Altinel et al., "Damia: A Data Mashup Fabric for Intranet Applications," *Proc. 33rd Int'l Conf. Very Large Databases* (VLDB), 2007, ACM Press, pp. 1370–1373.
2. E. Rahm and P.A. Bernstein, "A Survey of Approaches to Automatic Schema Matching," *The Very Large Databases J.*, vol. 10, no. 4, 2001, pp. 334–350.
3. E.M. Maximilien et al., "A Domain-Specific Language for Web APIs and Services Mashups," *Proc. Int'l Conf. Service Oriented Computing* (ICSOC), Springer, 2007, pp. 13–26.
4. D. Braga et al., "NGS: A Framework for Multi-Domain Query Answering," *Proc. Workshop Information Integration Methods, Architectures, and Systems* (IIMAS), IEEE CS Press, 2008, pp. 254–261.
5. A. Calì and D. Martinenghi, "Querying Data under Access Limitations," *Proc. Int'l Conf. Data Eng.* (ICDE), IEEE CS Press, 2008, pp. 50–59.
6. D. Braga et al., "Joining the Results of Heterogeneous Search Engines," to be published in *Information Systems*.
7. U. Srivastava et al., "Query Optimization over Web Services," *Proc. 32nd Int'l Conf. Very Large Databases* (VLDB), 2006, ACM Press, pp. 355–366.
8. F. Daniel and B. Pernici, "Insights into Web Service Orchestration and Choreography," *Int'l J. E-Business Research*, vol. 2, no. 1, 2006, pp. 58–77.

Daniele Braga is an assistant professor at the Dipartimento di Elettronica e Informazione at the Politecnico di Milano. His research interests include XML data management to schema mapping, schema integration, and Web service integration. Braga has a PhD in computer science from the Politecnico di Milano. Contact him at braga@elet.polimi.it.

Stefano Ceri is a professor of database systems at the Dipartimento di Elettronica e Informazione at the Politecnico di Milano. His research interests focus on extending database technology to incorporate distribution, rules, and XML, on design methods for data-intensive Web sites, on integration of search services, and on reasoning with data streams. Contact him at ceri@elet.polimi.it.

Florian Daniel is a postdoctoral researcher at the University of Trento, Italy. His main research interests include Web application and service mashups, adaptivity and context-awareness in Web applications, quality, and privacy in business intelligence applications. Daniel has a PhD in information technology from Politecnico di Milano. Contact him at daniel@disi.unitn.it.

Davide Martinenghi is a postdoctoral researcher at the Dipartimento di Elettronica e Informazione at the Politecnico di Milano. His research interests are in data integrity maintenance, data integration, Web data access, service mashups, and Web search. Martinenghi has a PhD in design and management of information technology from Roskilde University, Denmark. Contact him at martinen@elet.polimi.it.