Understanding Mashup Development

Web mashups are Web applications developed using contents and services available online. Despite rapidly increasing interest in mashups over the past two years, comprehensive development tools and frameworks are lacking, and in most cases mashing up a new application implies a significant manual programming effort. This article overviews current tools, frameworks, and trends that aim to facilitate mashup development. The authors use a set of characteristic dimensions to highlight the strengths and weaknesses of some representative approaches.

Jin Yu
and Boualem Benatallah
University of New South Wales

Fabio Casati
and Florian Daniel
University of Trento

Web mashups[1] are Web applications generated by combining content, presentation, or application functionality from disparate Web sources. They aim to combine these sources to create useful new applications or services. Content and presentation elements typically come in the form of RSS or Atom feeds, various XML formats, or as HTML, ShockWave Flash (SWF), or other graphical elements. Publicly available APIs (in JavaScript, for example) typically provide application functionality. Content, functionality, and presentation are then glued together in disparate ways: via JavaScript in the browser, server-side scripting languages such as Hypertext Preprocessor (PHP) or Ruby, or traditional languages such as Java or C#.

"Mashup" has become one of the hottest buzzwords in the Web applications area, and many companies and institutions are rushing to provide mashup solutions (or to relabel existing integration solutions as mashup tools). Amidst this frenzy, it's difficult to distinguish between mashups and traditional integration efforts. This article aims to provide some clarity in regard to

- what a mashup is (and isn't);
- how mashups resemble or differ from traditional forms of integration, such as application, data, and presentation integration;
- what fundamental characteristics and dimensions mashup approaches share; and

1089-7801/08/$25.00 © 2008 IEEE

- how current tools compare with respect to these characteristics and dimensions.

Specifically, we overview some of the popular mashup tools and show how they facilitate the development of rich Internet applications. Our aim isn't to identify the kinds of available support in terms of mashup development but rather to understand and identify emerging characteristics and dimensions under which we can compare and analyze the tools and approaches.

Mashup Development Approaches

Mashup development differs from traditional component-based application development mainly in that mashups typically serve a specific situational (short-lived) need and are composed of the latest, easy-to-use Web technologies (such as Representational State Transfer [RESTful] Web services or RSS/Atom feeds). As such, the Web is their natural environment.

The HousingMaps (www.housingmaps.com) application in Figure 1 is an example of a successful mashup. It combines property listings from Craigslist with map data from Google Maps to assist people moving from one city to another and searching for housing. Typically, when people are browsing through a list of properties, a property's address doesn't give them enough information if they aren't yet familiar with the new city. HousingMaps gives users a list of properties and plots the respective locations and property information on the map upon selection (using the popup cloud visible in Figure 1).

We could manually develop such a mashup application using conventional Web programming technologies. However, dedicated mashup tools could benefit such development, eventually letting even end users compose their own mashups.

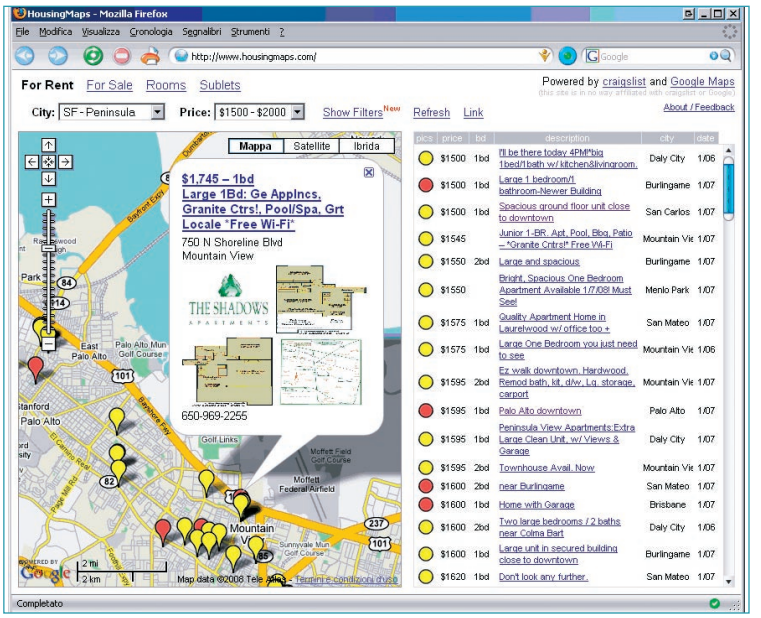

Figure 1. The HousingMaps application. HousingMaps integrates Craigslist housing offers with Google Maps into a homogeneous user interface.

Manual Mashup Development

Generally, integrating enterprise data and applications into a coherent and value-adding application requires programming skills and intimate knowledge about the schemes and semantics of data sources or the business protocol conventions for message exchange. Fortunately, new technologies, such as Ajax and RESTful services, and microformats, such as RSS and Atom, have simplified mashup development. In addition, intelligent source components largely assist the integration of contents, application logic, and user interfaces. Nevertheless, manual mashup development is still a prerogative of skilled developers.

Assuming you have no specific development tools, what's involved in manually developing an application like HousingMaps? First, you must become familiar with the two source applications (Craigslist and Google Maps) and identify how you will reuse or extract data from the two sites. Whereas Google Maps offers a publicly available JavaScript API that you can leverage, Craigslist provides its listings via RSS. Therefore, to extract property and address data, you must parse and interpret the RSS feed from Craigslist. To configure the clickable markers that will display the property information in a popup cloud window upon a click, you must interact with the Google Maps JavaScript API. Enabling the automatic popup of this cloud requires a specific JavaScript function that listens for the property selection and reacts by invoking the Google Maps API to select the respective marker. Although Google Maps has its own user interface, letting users select properties wrapped from Craigslist requires that you fill and appropriately format a suitable table. Finally, you must lay out the two components properly to form the composite application's user interface. Such intricate and time-consuming tasks prevent average users from programming their own mashups.

Tool-Assisted Mashup Development

To speed the overall mashup development process, but also to enable even inexperienced end users to mash up their own Web applications, numerous mashup-specific development tools and frameworks have recently emerged. These instruments typically come with a variety of features and a mixture of composition approaches. A close look at them lets us identify the open issues and research challenges characterizing the mashup phenomenon.

For presentation purposes, we selected the most popular or representative approaches of end-user mashup tools and show how they can support the HousingMaps application's development. We discuss a few alternative or complimentary approaches in the "Related Work in Reusable Components" sidebar.

Yahoo Pipes. Yahoo Pipes (http://pipes.yahoo.com) lets you mix popular data feeds to create data mashups via a visual editor. A pipe is a data-processing pipeline consisting of one or more data sources (for example, RSS/Atom feeds or XML sources) and a set of interconnecting operators, each of which performs a specific task. It includes operators for manipulating data feeds (for example, sorting or filtering) and operators for features such as looping, regular expressions, or counting. It also supports more advanced features, such as location extraction (for example, geocoordinates identified and converted from location information found in text fragments) or term extraction (for example, keywords). Yahoo Pipes aims to let users design data-processing pipelines that filter, transform, enrich, and combine data feeds and are again exposed as RSS feeds.

Consider how Yahoo Pipes could aid the development of the HousingMaps example. Because Pipes doesn't provide user interfaces — that is, it outputs an RSS feed — we can't implement the user interface shown in Figure 1. Instead, we could use Pipes to process the Craigslist feed and identify location information (geocodes) by leveraging the pipes' location extractor. We could use the identified location information to augment the Craigslist feed with a link that lets users display the property's address on the map by passing the geocodes to Google Maps.

Google Mashup Editor. GME (http://editor.googlemashups.com) provides a template-based environment for mashup development. It offers a set of standard modules that lets users encapsulate and lay out external data. For example, the *list* module represents an RSS/Atom feed as a list, whereas the *item* module represents a single item in a feed. Modules can fire predefined events, which other modules can capture and act on accordingly. Creating mashups involves developing user interface templates that contain a mixture of XML control tags and HTML/CSS layout elements with embedded JavaScript code. At runtime, GME fills the user interface templates and presents them as Web pages.

For the HousingMaps application, we could integrate the Craigslist feed using a list module and use the item module to show a particular property's details. GME's map module natively supports Google Maps. When the user clicks on a property in the Craigslist module, the module emits a "select" event, which the map module can capture to pop up the cloud window on top of the marker and display information about the selected property. We must embed the Craigslist module and Google Maps into the user interface template that specifies the actual mashup application's layout.

Microsoft Popfly. Popfly (www.popfly.ms) offers a component-based, visual environment for developing mashups. In Popfly, reusable components, or *blocks*, can act as middlemen between externally provisioned services, such as Web services,[2] or implement a useful function (in JavaScript) — for example, a function that calculates a circle's area given a radius. Blocks have operations with inputs and outputs, which are specified in a dedicated XML descriptor. A block might also act as a display surface — that is, a piece of user interface that takes data from other blocks and displays them, letting the user interact with them and enabling the mashup developer to lay out the mashup application.

To build the HousingMaps application, we need three blocks:

- an RSS feed block for the Craigslist feed,
- a map block, and
- a table block.

If we use Virtual Earth (http://microsoft.com/virtualearth) instead of Google Maps, the three blocks are already available. We must therefore

drag the blocks onto the mashup design surface and then connect the output of the RSS block's getItems operation to the two display blocks for the RSS and Virtual Earth. Correctly configuring the initial set of markers might require extending the RSS block with a suitable JavaScript operation.

Intel Mash Maker. Mash Maker (http://mashmaker.intel.com) provides an environment for integrating data from annotated source Web pages based on a powerful, dedicated browser plug-in. Rather than taking input from structured data sources such as RSS or Atom, Mash Maker lets users annotate Web pages' structure while browsing and use such annotations to scrap contents from annotated pages. Advanced users can leverage the integrated *structure editor* to input XPath expressions using FireBug's DOM Inspector (a plug-in for the Firefox Web browser). Composing mashups with Mash Maker occurs via a copy-and-paste paradigm, based on two modes of merging contents:

- *whole page* merging, in which the user inserts one page's content as a header into another page; and
- *item-wise* merging, in which the user combines contents from two pages at row level, based on additional user annotations.

You can use the two techniques to merge more than two pages.

For the HousingMaps example, we first annotate the appropriate Craigslist page's structure because Mash Maker operates on regular HTML content rather than on RSS. Next, we merge the Craigslist page with the Google Maps page using the copy-and-paste mechanism. Specifically, we adopt item-wise merging because we plot each item from the Craigslist page as an individual marker on the map.

Quick and Easily Done Wiki. QedWiki (http://services.alphaworks.ibm.com/qedwiki) is IBM's proposal for a wiki-based "mashup maker," fully running inside the client browser and allowing access to IBM's Mashup Hub (http://services.alphaworks.ibm.com/mashuphub). The Hub supports the creation of data feeds and user interface widgets and incorporates Data Mashup Fabric for Intranet Applications (Damia)[3] for data assembly and manipulation. As a wiki environment, it lets users edit, immediately view, and easily share mashups. Mashups are assembled from JavaScript- or PHP-based widgets, whose wiring determines the mashup's behavior. Widgets represent application components and might or might not have their own user interface. To assemble a mashup, a user selects a page layout (an HTML template) and then drags and drops widgets onto the page grid and interactively configures them.

To develop the HousingMaps application with QedWiki, we first create a new wiki page and select a grid layout. In our case, we opt for a layout that lets us place Google Maps and the housing offers side by side (in two columns). We then search for the GoogleMap widget in the widget palette, drag it over the grid layout, and drop it over the left column. We use the LoadFeed widget to access the Craigslist RSS feed and populate a ShowData widget with the housing offers (by telling the ShowData widget that it should source data from the LoadFeed widget). To locate properties on the map, we can now simply drag addresses from the ShowData widget at runtime and drop them onto the GoogleMap widget.

Characterizing Mashup Approaches

As you'll have noticed, the tools we've described differ in two complementary aspects:

- the mashup paradigm at the basis of the approach and
- the software instrument that implements the chosen paradigm.

Much like in data and application integration, we characterize the mashup paradigm by looking at the objects of integration (the components) and how such objects are glued together (the composition logic). As for the software instrument, it's important to separately look at the design-time support (the development environment) and the runtime support (the runtime environment) provided.

Component Model

The component model determines the nature of components and influences how they can be glued together — that is, how they can be composed. A well-defined component interface, for instance, facilitates reusability, whereas a flexible component interface ensures extensibil-

ity. We characterize a component model using three properties.

The first property is *type*. A component can be a data (DA), application logic (AL), or user interface (UI) type, depending on whether it acts as a pure data source, a component providing access to application logic, or a component that also provides a GUI to users.

Second, we look at the model's *interface*. A component might expose a create-read-update-delete (CRUD) interface, APIs in specific programming languages or in IDL/WSDL, XML/HTML markup, or it might only expose GUI elements to the end users. A component might also expose a combination of these elements.

Finally, the *extensibility* property explains whether the user can create new components or extend the component model to accommodate specific application requirements, such as new operations.

Yahoo Pipes supports DA and AL components through operators that provide access to RSS/Atom feeds and external Web services. DA components have a read-only interface, and external Web services have a RESTful interface based on JavaScript Object Notation (JSON) or RSS. Yahoo Pipes component models are fixed.

GME supports DA, AL, and UI components. DA components are typically interfaced via markup, AL components via JavaScript, and UI components via both markup and JavaScript. GME component models are flexible.

Popfly also supports DA, AL, and UI components. In Popfly, all components are interfaced using JavaScript, and component models are extensible.

Intel Mash Maker supports DA components extracted from annotated Web pages (for example, table and map). Their interface can be interpreted as XML markup, and the component models are fixed.

QedWiki focuses mainly on UI components (the Mashup Hub supports DA and AL components). Components are equipped with a JavaScript interface and can be extended.

In traditional integration, extensible application-level components with application-specific APIs would characterize Web services. Data integration[4] applications are instead characterized by data-driven components that are often fixed or have limited extensibility. For example, extract, transform, load (ETL) applications have a large set of built-in modules that perform join or lookup operations on relational database or XML documents. Custom behavior is typically supported through a generic SQL component.

Composition Model

The composition model determines how components are integrated to form the mashup, assuming components are readily available. To facilitate end-user compositions, the composition model should be as simple as possible. A composition model has several distinct characteristics.

First, we distinguish the model's *output type*. As with components in input, composition output can be of type DA, AL, or UI, depending on whether the composition provides data, programmable APIs, or applications with a user interface.

The second characteristic is *orchestration style*. Orchestrating components implies specifying how you'll define and synchronize the components' execution. Three main approaches exist:

- *Flow-based* styles define orchestration as sequencing or partial order among tasks or components and are expressed through flow chart-like formalisms.
- *Event-based* approaches use publish–subscribe models. They're particularly powerful for maintaining synchronized behavior among components.
- In the *layout-based* style, components (with or without user interfaces) are arranged in the composite application's common layout. Each component's behavior is specified individually by accounting for the other components' reactions to user interactions.

Third, we look at the model's *data-passing style*. We define two data-passing approaches:

- a *dataflow* approach, in which data flows from component to component; and
- a *blackboard* approach, in which data is written to variables, which serve as the source and target of operation invocation on components, much like in programming languages.

In addition, a composition can be *instance-based* or *continuous*. An instance-based model is the traditional service composition model, in which a certain kind of message's arrival activates a new instance of the composition, and the system executes the instance within the

same main thread and context (much like a program run).

Conceptually, the continuous model has one instance per component in the composition model. Each component works as a thread, processing the input data feed and transforming or filtering it to generate the output.

Another property relates to *exceptions* and *transactions*. A composition model might or might not support exception and transaction handling. If supported, exception handling can follow the throw-and-catch approach (Java style) or can be rule based (using event-condition-action [ECA] rules coupled to the composition). Transactions, if supported, always follow some variation of the Saga model.[5]

Yahoo Pipes is probably the best representative of DA output (pipes are RSS feeds). Its graphical modeling language is flow-based; accordingly, data is also passed via data flows. Pipes is instance-based; it doesn't provide exception handling or support transactions.

GME produces UI output. GME is event-based and achieves data passing through event parameters in a dataflow fashion. In addition, UI components are continuous. Finally, GME doesn't support exceptions and transactions.

Popfly produces UI output. It uses an event-based orchestration style and a dataflow approach for data passing. Components are continuous. Like GME, Popfly doesn't support exceptions and transactions.

Intel Mash Maker focuses on UI output. In Mash Maker, contents are glued together in a layout-based style (whole page merging) or in a flow-like style (item-wise merging). It uses data extracted from annotated Web pages in a blackboard style. Mash Maker's instance model is most similar to the instance-based one, and it doesn't support exceptions and transactions.

Like GME, Popfly, and Mash Maker, QedWiki produces UI output. It proposes a layout-based orchestration style and components pass data in a blackboard fashion. Widgets are continuous. QedWiki doesn't support exceptions or transactions.

Traditional integration is typically flow-based (think of the Business Process Execution Language [BPEL] and most ETL processes), with an XML-based data model following a blackboard approach for service-oriented architectures (SOA) and a relational data model with data flow for ETL. Both use an instance-based instantiation model. However, in SOA, an instance is created with the arrival of a certain message (such as a purchase order), whereas in ETL, an instance is created periodically (for each data extraction). Traditional integration produces DA or AL output, whereas mashups typically include some form of integration at the UI level. In terms of exception and transaction, traditional integration offers Java-like exception handling and Saga-like transaction support, with predefined but extensible exception types (for example, SOAP faults in BPEL or DB errors in ETL).

Development Environment

The characteristics of the mashup tools' development environments affect mashup development efficiency and determine the tools' success. Mashup tools vary greatly in the level of support they provide to their users. Some tools are strictly for developers, whereas others are more oriented toward end users. Several properties characterize mashup development environments.

Mashups are about simplicity, usability, and ease of access. This simplicity has the upper hand over feature completeness or full extensibility.

The first property is the environment's *interface paradigm* and *target users*. Mashup tools can support design via different interface/modeling paradigms, such as visual drag-and-drop features, textual editors, or a combination of the two. The interface can target average Web users, advanced (tech-savvy) users, or programmers. The interface's ease of use is the key factor in bringing mashup capability to average and advanced Internet users.

A development environment is also characterized by *system requirements*. The mashup tool's execution might require specific additional modules, plug-ins, or browser features, whose absence might prevent the instrument's use.

Yahoo Pipes provides a pure visual drag-and-drop Ajax editor targeted at users with basic programming skills. The editor is executed in a standard Web browser with support for the XMLHttpRequest JavaScript object.

GME's browser-based textual Ajax editor with syntax highlighting and automatic tag completion is targeted at programmers. It can be fully executed in a standard Web browser.

Microsoft Popfly offers a graphical and textual editor with drag-and-drop support for Web users. Popfly is based on Microsoft's Silverlight (www.microsoft.com/silverlight) technology, a mandatory browser plug-in.

Similarly, Intel Mash Maker supports a point-and-click user interface that lets advanced users and programmers annotate pages and nonexpert Web users extract and merge data via copy-and-paste. Mash Maker requires a dedicated plug-in that extends the browser with mashup features.

QedWiki comes with an easy-to-use drag-and-drop interface for advanced users. In this interface, components are immediately visualized. QedWiki runs in a standard Web browser and doesn't require any plug-ins.

Unlike these mashup tools, traditional integration technologies typically offer desktop development applications rather than browser-based ones. In addition, they require a steeper learning curve because they're more sophisticated and feature-rich. Traditional approaches, which always target programmers, offer neither end-user data integration nor application integration.

Runtime Environment

Typically, each mashup tool also provides a separate runtime environment that enables the execution of the tool's mashups and determines how it will deliver the mashups to its users. Possible system requirements imposed by the runtime environment might affect the adoption of mashups developed with the respective tools. We distinguish four properties in the runtime environment.

The first property is the *deployment style*. As with conventional Web applications, you can deploy a mashup application in a standalone fashion on any Web server managed by the mashup developer, or through a third-party Web server (typically belonging to the mashup development environment provider).

Another property is the *runtime location*. You can assemble mashups at the server side (for example, via PHP or Ruby), the client side (for example, inside a Web browser via JavaScript), or both. If the integration occurs at the server side, the browser merely displays the resulting composite application. Server-side approaches can use an engine-based or Webapp-based implementation style. The engine-based approach implies that a mechanism analogous to a process engine executes the mashup (for example, collects and processes the feeds). In the Webapp-based approach, the mashup is implemented as a Web application, so the Web and application servers execute the mashup.

The third property of a runtime environment is *system requirements*. Similar to the development environments, a mashup's execution can depend on the availability of additional browser plug-ins or extensions.

Finally, we look at the environment's *scalability*. We can consider scalability from three perspectives:

- the number of data sources,
- the number of models (compositions), or
- the number of users.

In general, client-side approaches don't suffer from scalability problems. The mashup is executed on the client, so no bottleneck exists (except from the overload on the data sources themselves, but this is outside the mashup's control). Here, the scalability problems relate to the number of instances and, hence, the number of users and the mashup's complexity (which is related to the number of sources and the related data processing). In all cases, client-side approaches use the same scalability techniques as do traditional integration or Web applications, relying on workflow scalability techniques for engine-based runtimes and on Web application scalability for Web-application-based runtimes.

Yahoo Pipes compositions are hosted on a Yahoo server. The system computes and assembles pipes at the server side (apparently engine-based), so executing a pipe doesn't pose any particular system requirements on the client. However, the server-side engine that executes the pipes might suffer if many pipes are run, if numerous users access the same pipe, or if the pipe consists of hundreds of sources.

GME mashups are hosted on a Google server. Mashups are executed at the server side and have no particular system requirements. The system compiles mashups into conventional Web applications.

Popfly mashups are hosted on a Microsoft server. Execution of a Popfly application, however, occurs at the client side and typically re-

Related Work in Reusable Components

The tools we describe in the main text effectively let end users easily compose data and application logic starting from reusable components. E. Michael Maximilien and his colleagues propose a programming language that's specific to mashups of Web services (for example, Representational State Transfer [REST], SOAP, RSS, or Atom services);[1] however, the language is more oriented toward developers. Nonetheless, the approach is in line with the "approachable programming model" that characterizes mashups in general.[2]

Portals focus on the integration of components with their own user interface (portlets[3]). They represent an affirmed solution in the development of large-scale Web applications but generally offer weak support for intercomponent communication and end-user-oriented development.[4] Integrating portlets sourced from the Web is still hard, but work is ongoing.[5] Tools such as Dapper (www.dapper.net) and Openkapow (http://openkapow.com), instead, are popular for developing components. They provide powerful support for data or user interface extraction (wrapping) from existing Web sources. The OpenAjax Alliance (www.openajax.org) aims to develop a standardized client-side hub for a publish–subscribe-based event communication among Ajax components, which are increasingly becoming the natural environment for mashup components. Component development, however, is out of this article's scope.

References

1. E.M. Maximilien et al., "A Domain-Specific Language for Web APIs and Services Mashups," *Service-Oriented Computing* (ICSOC 07), LNCS 4749, Springer, 2007, pp. 13–26.
2. S. Watt, "Mashups — The Evolution of the SOA, Part 2: Situational Applications and the Mashup Ecosystem," *IBM DeveloperWorks*, 2007; www.ibm.com/developerworks/webservices/library/ws-soa-mashups2.
3. A. Abdelnur and S. Hepper, *Java Portlet Specification*, 2003; http://jcp.org/en/jsr/detail?id=168.
4. F. Daniel et al., "Understanding User Interface Integration: A Survey of Problems, Technologies, and Opportunities," *IEEE Internet Computing*, vol. 11, no. 3, 2007, pp. 59–66.
5. OASIS Web Services for Remote Portlets (WSRP) technical committee, www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrp.

quires the availability of the Silverlight plug-in. The client-side execution facilitates scalability because the integration of multiple sources occurs mostly at the client side.

Intel Mash Maker mashups are stored on the client PC and are executed inside the Web browser using the Mash Maker plug-in. Although it seems unlikely that large numbers of sources will be mashed up, Mash Maker should be able to scale adequately.

QedWiki pages are hosted on an IBM server. Mashups are executed mostly on the client side, and a standard Web browser can execute QedWiki pages. The wiki engine might encounter difficulties if it must integrate numerous sources.

SOA or ETL deployments typically have a centralized engine server that runs the process definition by invoking services or data storages. Distributing the workload over multiple engines guarantees scalability.

Going Forward

Many of the differences between mashups and traditional forms of integration descend from the basic observation that mashups focus mainly on opportunistic integration occurring on the Web for an end user's personal use and for non-business-critical applications. Traditional composition (for example, BPEL-like), on the other hand, focuses on systematic and repeatable enterprise processes. Enterprise processes also have a wide set of nonfunctional requirements, such as security and reliability, that few mashups share and that make languages, tools, and the overall development fairly complex. Also, unlike conventional Web applications, many of today's mashups still have a limited audience (such as individuals or small user groups) so scalability isn't a big issue. This might become a problem if and when a "killer mashup" appears.

In general, mashups are about simplicity, usability, and ease of access. This simplicity has the upper hand over feature completeness or full extensibility (as in SOA or BPEL). With improved tool support (such as a better user interface) and the abundance of components or modules, end users will be able to compose their own mashups. In this context, we also see a need for end-user-oriented integration paradigms for allowing easy and simple exploration, organization, search, and integration of mashups. This will help move mashup development from manual and time-consuming scripting to a set of easy-to-find and extensible parameterized patterns that characterize most of the heterogeneities among mashup services. Mashup component search is also likely to improve over time, not only because of Google-like search mechanisms but also because of an emerging trend toward online communities of mashup

taggers and bloggers, following the style of the social Web.

Therefore, mashups can learn useful lessons from traditional integration. Specifically, to simplify mashup development, we need a user interface component model, so mashup developers can abstract and reuse a user interface as in traditional services. A user interface component model will likely have aspects similar to traditional components in addition to user interface-specific items. It should also be fundamentally simpler, consistent with the Web's philosophy.

In addition, we need middleware for user interface integration. Today's middleware is essentially the Web, which offers no mashup-specific support. Perhaps the Web is sufficient, but middleware paradigms, such as publish–subscribe, which have been extremely successful in EAI, are also well-suited to mashups. This suitability is due to the nature of mashups, which are strongly event-based (they're essentially reactive applications sensitive to events from sources such as news feed content or user interactions).

Finally, we must bridge user interface integration with traditional forms of integration. Mashups are evolving toward components that are a mix of user interface aspects and traditional application logic. The challenge here is identifying component and composition models that can cater to the needs of both kinds of integration. One is more event-driven and user-oriented, the other is more orchestrational and enterprise-oriented.

Without these elements, mashup development will largely be an ad hoc effort requiring programming skills that typical Web users don't possess, or it will be restricted to specific technologies or domains.

We've begun to investigate these issues in a framework called Mixup.[6,7] However, we've just scratched the surface of these research problems. Mixup is an instance of a trend that brings together the different forms of integration (user interface, application, and data) while ensuring ease of development and maintenance with minimum learning curves.

References

1. D. Merrill, "Mashups: The New Breed of Web Application," *IBM DeveloperWorks*, 2006, www-128.ibm.com/developerworks/library/x-mashups.html?ca=dgr-lnxw16MashupChallenges.
2. G. Alonso et al., *Web Services: Concepts, Architectures, and Applications*, Springer, 2004.
3. M. Altinel et al., "Damia: A Data Mashup Fabric for Intranet Applications," *Proc. Very Large Databases Conf.* (VLDB 07), VLDB Endowment, 2007, pp. 1370–1373.
4. M. Lenzerini, "Data Integration: A Theoretical Perspective," *Proc. Symp. Principles of Database Systems* (PODS 02), ACM Press, 2002, pp. 233–246.
5. H. Garcia-Molina and K. Salem, "Sagas," *Proc. ACM Special Interest Group on Management of Data 1987 Ann. Conf.* (SIGMOD 87), ACM Press, 1987, pp. 249–259.
6. J. Yu et al., "A Framework for Rapid Integration of Presentation Components," *Proc. Int'l World Wide Web Conf.* (WWW 07), ACM Press, 2007, pp. 923–932.
7. J. Yu et al., "Mixup: A Development and Runtime Environment for Integration at the Presentation Layer," *Proc. Web Eng.* (ICWE 07), LNCS 4607, Springer, 2007, pp. 479–484.

Jin Yu is a PhD candidate at the School of Computer Science and Engineering, University of New South Wales, Sydney. He also serves as the VP of Engineering at Martsoft Corporation. His research focuses on rich Internet applications and user interface integration. He is a member of the IEEE and the ACM. Contact him at jyu@cse.unsw.edu.au.

Boualem Benatallah is a professor in the School of Computer Science and Engineering at the University of New South Wales, Sydney. His research interests lie in the areas of Web services, business processes, and data integration. Benatallah has a PhD in computer science from Grenoble University, France. He is member of the IEEE and the ACM. Contact him at boualem@cse.unsw.edu.au.

Fabio Casati is a professor of computer science at the University of Trento. He has a PhD in computer science from Politecnico di Milano, Italy. Casati is coauthor of a book on Web services, a member of the editorial board of *ACM Transactions on the Web*, and a member of the steering committee of the international conferences on service-oriented computing and business process management. Contact him at casati@dit.unitn.it.

Florian Daniel is a postdoctoral researcher at the University of Trento. His research interests include Web engineering, mashups, and business intelligence applications. He has a PhD in information technology from Politecnico di Milano. He is a member of the organizing committee of the Adaptation and Evolution in Web Systems Engineering Workshop and the WebML research group. Contact him at daniel@disi.unitn.it.