

Mashing Up Oil and Water

Combining Heterogeneous Services for Diverse Users

Connecting heterogeneous services is a complex endeavor that requires support at both the middleware and user-interface levels. Offering users a varied palette of mashup development environments and service interfaces lets users choose elements appropriate to their skill levels and their tasks. The authors discuss their experiences using various development paradigms, such as spreadsheets and high-level scripting, to mash up diverse services. They describe their middleware and service adapters, which abstract the difference between service interfaces, and compare several mashup interfaces aimed at different user groups.

Željko Obrenović
Technical University Eindhoven

Dragan Gašević
Athabasca University

One of the most basic rules that students learn in science classes is that oil and water don't mix. To some extent, the same applies when you combine Web services with other software services and components. In our projects, for example, we've combined highly diverse software services, mixing the world of Web services with that of low-level devices.¹ The former use high-level XML data structures with a relatively slow response, whereas the latter employ low-level data structures with high performance. Creating applications that mash up both worlds requires solutions that bridge many semantic and temporal gaps among the services' interfaces and data structures. In addition, different users

have different expertise and requirements and need mashup environments that suit them. Ideally, mashups should be easy for novices to pick up but also provide the ambitious functionality that experts need. However, most existing mashup solutions – such as Yahoo Pipes (<http://pipes.yahoo.com>) and iGoogle (www.google.com/ig) – require underlying services uniformity, don't support the development of highly interactive applications that include local services and devices, and provide limited interface variability.

Here, we describe our experiences in mashing up interactive applications using heterogeneous software services. Our solutions let users choose various environments, such as spreadsheets or

scripting, to mash up various remote and local services, such as connecting a Google search service with a local text-to-speech (TTS) service. Our work's central challenge is managing the complexity of connecting heterogeneous services and suiting diverse user needs; the goal isn't to reduce this diversity, but rather to support it both at the middleware and UI levels. Although diversity increases the solutions' complexity, it also exposes many new possibilities, such as easier integration of existing software and services and a lower learning curve for users because the environments are familiar.

Our discussion addresses two main issues. First, we describe the challenge of interconnecting heterogeneous services, briefly highlighting our middleware and service adapters that abstract the difference between service interfaces. Our middleware opens up possibilities for mashups that aren't purely Web-oriented, letting us compose local services with those available on the Web. Second, we explore how diverse end users and developers mash up these heterogeneous services, describing and comparing several mashup development interfaces – including spreadsheets, Web browser extensions, and scripting and programming languages – that we've built on top of our middleware to suit different user groups.

Mashing Up Heterogeneous Services

In our project and academic experiences, we've dealt with hugely diverse software services and developer backgrounds. Our initial work shows the benefits of providing a uniform view on diverse services and rapidly prototyping those services using higher-level XML languages.¹ The solutions we discuss here result from our attempts to make our products accessible to a broader end-user audience, such as interaction designers, who usually have highly diverse backgrounds but little knowledge about advanced programming and markup languages (such as XML). Although our work is still in its early stages, our solutions have been successfully used by a range of people – from first-year industrial design undergraduates with no programming experience to experienced developers – each of whom chose and combined different components of our framework.

As Figure 1 shows, our approach's basic idea is to abstract the differences between diverse service interfaces and enable the use of

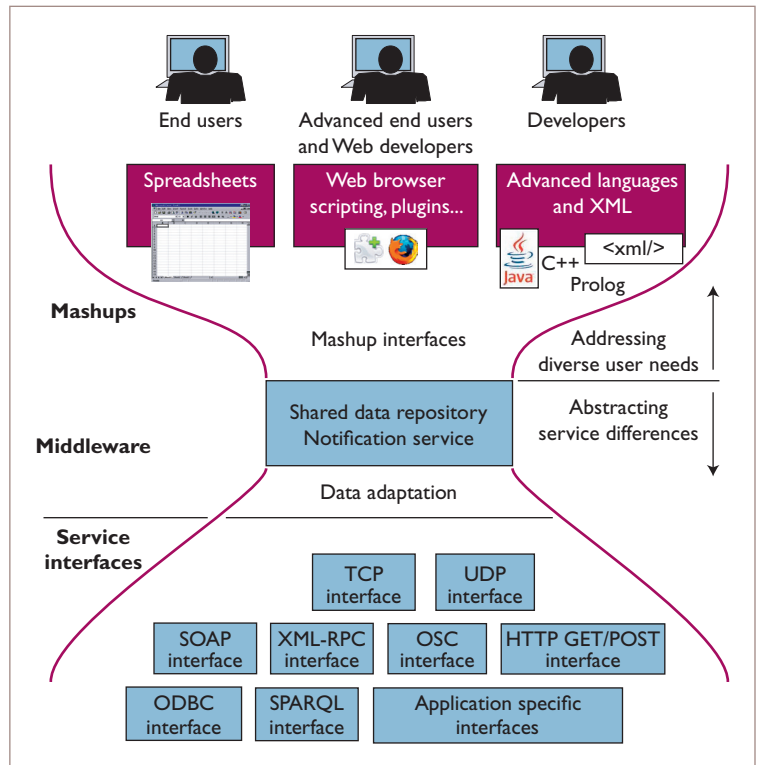


Figure 1. Our basic approach to mashup creation. By abstracting the differences between diverse service interfaces, we can create different mashup interfaces on top of the abstraction to support end-user service composition.

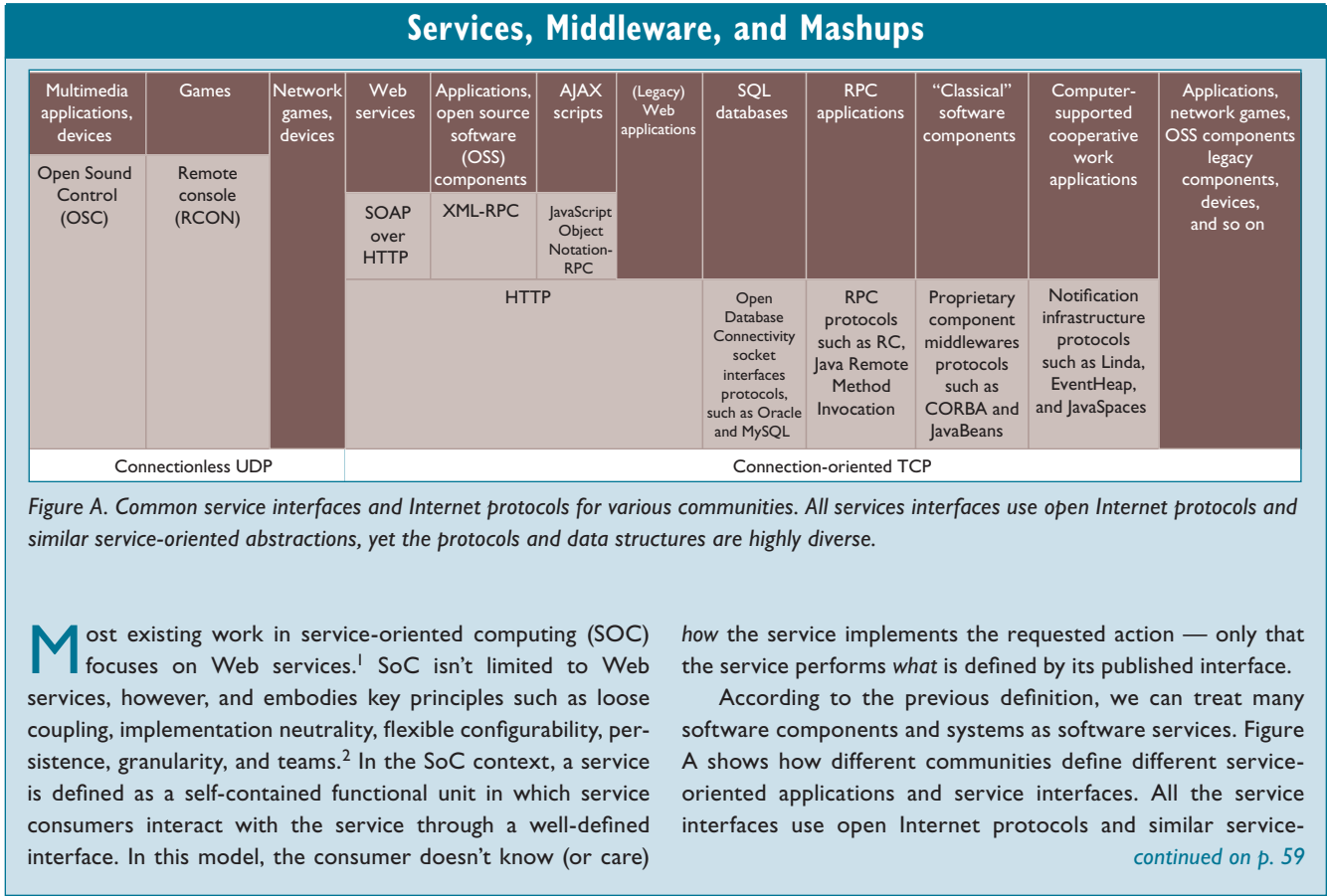
different mashup development environments on top of this abstraction. Doing so supports services composition by users with different technical skill levels. (The sidebar, “Services, Middleware, and Mashups,” discusses related work in this area.)

Rather than introducing one integrated environment, we provide a palette of development environments and service interfaces and let users and developers choose the elements most appropriate to their skills and tasks. Our framework, therefore, introduces two basic elements:

- middleware and a set of service adapters that abstract the service differences and provide a common data integration space, and
- multiple mashup development environments – including spreadsheets, scripting, and advanced programming languages – built over this abstraction to address diverse user needs.

Middleware and Service Adapters

Our Adaptable Multi-Interface Communicator (Amico) middleware is a message-oriented sys-



tem based on the ideas from loosely coupled notification infrastructures and coordination languages.^{2,3} Our original aim with Amico was to support rapid prototyping with diverse open-source software components and thereby support interactive applications development in domains such as interactive television.¹ We've extended the middleware to enable interaction with Web and many other services.

Amico's core is a shared data model with named untyped data slots, or *variables*. In essence, this data model is a simple variables list, implemented as a hashtable with all variables placed in the same address space. We decided to use basic untyped data objects because they're simple and flexible.⁴ We also wanted to make this data structure manageable by end users, so that they could directly explore basic services' functionality without programming. Having variables in a single-address space is already familiar to many users through system variables and properties tables, and our initial experiences with less-experienced users and students have shown that the concept is easy for them to under-

stand. Mapping simple data structures is also straightforward for end-user environments such as spreadsheets and scripting environments, as well as in declarative and dynamically typed languages.

The Amico middleware supports several functions common for message-oriented systems: UPDATE a variable, GET the variable's value, REGISTER for notifications about variable updates, and DELETE a variable. To simplify implementation of service and mashup adapters, we implemented the functions in a fault-tolerant way: updates of a nonexistent variable create a new variable, and requests for a nonexistent variable simply return an empty string value rather than an error. We therefore don't introduce explicit actions for creating variables; Amico creates them by first update, similar to how they're created with dynamically typed languages.

One of our middleware's main innovations is that we've opened its basic functionality (update, get, register, and delete) through numerous service interfaces that we've used to connect services and development environ-

Services, Middleware, and Mashups, cont.

oriented abstractions, but there's a huge diversity of protocols and data structures. Each interface has a significant supporting community, and efforts toward unification are limited.

If you want to combine services that use different service interfaces, existing solutions don't provide many possibilities. Most middleware solutions focus on one of the higher-level protocols — such as SOAP, common object request broker (Corba), or JavaBeans — and require services to adapt to one common interface.³ Researchers are also attempting to connect different service interfaces, building bridges between, for example, SOAP and Corba, and Java remote method invocation (RMI) and Microsoft.net Internet Inter-ORB Protocol (IIOP; <http://j-integra.intrinsyc.com>). However, given the number of service interfaces, there are many possible combinations, and maintaining the links is rather complex. The Service-Oriented Device Architecture (SODA) is an attempt to increase user services' diversity by modeling devices as services embedded on an enterprise service bus.⁴ SODA makes device access and control available to a range of enterprise applications. Although it's an interesting conceptual proposal, however, there are still no existing middleware solutions that support this idea.

Mashups let users create Web sites and applications that combine content from several sources into a single integrated experience.^{5,6} ProgrammableWeb.com illustrates the current state of mashups on the Internet. Although existing service mashup inter-

faces let users combine numerous services, they usually require uniformity of service interfaces — often focusing on Web services — and thus exclude many existing software services and components. Also, most mashup environment interfaces primarily suit less-experienced developers and offer a server-based pipeline service composition. Further, mashup infrastructures such as iGoogle and Yahoo Pipes aren't open source and introduce the "lock in" problem — that is, customers not only become dependent on a particular vendor for products and services, but switching to another vendor entails a substantial cost.

References

1. F. Curbera et al., "Unraveling the Web Services Web," *IEEE Internet Computing*, vol. 6, no. 2, 2002, pp. 86–93.
2. M.N. Huhns and M.P. Singh, "Service-Oriented Computing: Key Concepts and Principles," *IEEE Internet Computing*, vol. 9, no. 1, 2005, pp. 75–81.
3. I. Gorton, A. Liu, and P. Brebner, "Rigorous Evaluation of COTS Middleware Technology," *Computer*, vol. 36, no. 3, 2003, pp. 50–55.
4. S. de Deugd et al., "SODA: Service-Oriented Device Architecture," *IEEE Pervasive Computing*, vol. 5, no. 3, 2006, pp. 94–96.
5. S. Murugesan, "Understanding Web 2.0," *IT Professional*, vol. 9, no. 4, 2007, pp. 34–41.
6. A. Jhingran, "Enterprise Information Mashups: Integrating Information, Simply," *Proc. 32nd Int'l Conf. Very Large Databases*, U. Dayal et al., eds., ACM Press, 2006, pp. 3–4.

ments. We support the low-level TCP and User Datagram Protocol (UDP) interfaces — which let applications update or read variables by exchanging simple string messages — as well as many higher-level interfaces such as HTTP Get/Post, Extensible Markup Language Remote Procedure Call (XML-RPC), Open Sound Protocol (OSC), or SOAP. For each of these interfaces, we developed adapters that map variable updates to service calls and services results to variable updates. Our platform is extendable and lets users add new service interfaces. This enables much easier reuse of existing services and components because users don't have to adapt them to a common interface — that is, they can adapt them using the component's technology and the interface they're most familiar with.² We provide various data adapters for each service interface. For example, for handling SOAP-based XML data structures, we let users define XML adapters (see Figure 2). For XML-RPC, OSC, and other service interfaces, we define mapping between variables and the string equivalent of the interfaces' basic data types (such as integer or char).

Multiple Mashup Development Interfaces

We've built several mashup development interfaces on top of our middleware. Each mashup environment provides similar capabilities (update, get, register for, and delete variables) but is based on different standards and appropriate for users with different expertise levels. The motivation behind our interface design is similar to a multilayered UI philosophy⁵ — it lets users start with a basic mashup development interface (such as a spreadsheet) and switch to more advanced development interfaces as their expertise develops or they need more complex integration functionality. For example, in our courses on intelligent UI design and interactive systems sketching, students used spreadsheets in the beginning to quickly sketch, discuss, and evaluate interactive systems prototypes and then switched to advanced scripting languages or Web browser extensions to create more complex solutions. Various tools also follow this design philosophy: many video games have dozens of layers, most search engines (including Google and Yahoo) have novice and advanced layers, and many art and video tools (such as

```

<?xml version="1.0" encoding="UTF-8"?>
<soap-adapter endpoint-url="api.google.com"
  endpoint-service-name="/search/beta2">
  . . .
  <method name="doSpellingSuggestion"
    trigger="spelling"
    state-variable="google-spelling-state">
    <parameter name="phrase" type="xsd:string"
      type-qualifier="xsi:type">
      &lt;%=spelling%&gt;
    </parameter>
    <result update-variable="spelling-suggestion"/>
  </method>
  . . .
</soap-adapter>

```

Figure 2. Mapping Amico simple data structures (variables) to the service parameters. An XML fragment of the Amico SOAP adapter configuration file that defines the mapping of Amico variables to the Google spelling checker Web service. When the Amico variable `spelling` is updated, the Amico SOAP interface calls the Web service method `doSpellingSuggestion`, sending the variable's content as a parameter. The method's result is stored in the Amico variable `spelling-suggestion`. The Amico SOAP adapter also updates the variable `google-spelling-state` with the method call's state (that is, working or finished).

Apple Final Cut Pro and Adobe Premiere) have three or more workspaces. Indeed, some tools have as many as eight layers to accommodate a wide range of expertise and ambition.

Using diverse mashup development approaches could enable more end-user developers to mash up new solutions. End-user development is a highly popular form of computer interaction; each day, millions of users create their own solutions using various environments. In the US alone, there were 55 million end-user developers in 2004 compared to 2.75 million professional software developers.⁶

Benefits and Limitations of Our Approach

Our solutions complement, rather than replace, existing Web-based mashup integration solutions, letting users combine the results of Web services integration with non-Web-based software services in a rapid-prototyping manner. Our solution's main benefits are that it opens up the possibilities for mashups that aren't purely Web-oriented – that is, users can combine local services with Web services, but also choose the development environment best suited to their knowledge and experience.

To abstract data structures, we simplify them; we don't support complex data integrations at the middleware level.⁷ Simple untyped variables are easy to map to most existing service interfaces and development environments, but they put the burden of data cleaning, mapping, and transformations on users and their environments. This also limits the scope of applications – that is, our approach might not be feasible for merging complex data structures that would require users to manage hundreds of variables. In our experience, the framework is most useful in early development phases when users are exploring the possibilities space and looking for novel and useful service compositions.

Our aim isn't to provide fully integrated environments such as the Web-based Yahoo Pipes or Marmite environments.⁸ Often, our mashup environments serve as the service connection “backplane,” and we implement the UI as a separate process – within a Web page, for example. In our Web examples, we use scripting and advanced programming languages to connect services within Amico middleware and then integrate the resulting functionality within the Web browser using plug-ins as the UI.

Mashup Development Environments

To let users mash up services using diverse development paradigms, we implemented a diverse set of development environment extensions, including

- spreadsheets, for users with little to no programming skills;
- scripting and advanced programming mashups – including support for mainstream programming languages and nine scripting languages – for more advanced developers; and
- Web browser mashups – such as Asynchronous JavaScript and XML (Ajax), plug-ins, and applets – for advanced users and Web developers.

Users can add a new language or tool to our framework in two ways. The first, loosely coupled approach lets users run a development environment as a separate external process. We use this approach with spreadsheets and Web browser extensions. These environments

run the code in a separate process, updating or receiving updates of Amico variables through any supported Amico interface (we usually used TCP and UDP interfaces, along with HTTP for Ajax). Alternatively, developers can write Java-based Amico middleware plug-ins to integrate scripting-language support. In this case, our middleware instantiates the plug-ins as internal objects and communicates with them through a more efficient internal interface.

Spreadsheet Mashups

Many people find it easier to perform calculations in spreadsheets than to write an equivalent sequential program. Spreadsheets use spatial relationships rather than time as the primary organizing principle in computational tasks. Because they exploit users' natural spatial perception and reasoning, spreadsheets are immensely successful and popular. References between spreadsheet cells can take advantage of spatial concepts such as cell relative and absolute positions, as well as named locations, to make the spreadsheet formulas easier to understand and manage.

To enable spreadsheets-based service composition, we've implemented add-ins for OpenOffice.org Calc and Microsoft Excel.⁹ From the user's viewpoint, the add-ons introduce only a few additional functions for use in spreadsheet formulas. These functions let users call and receive the results of any Amico-connected service.

For example, Figure 3 shows an OpenOffice.org Calc spreadsheet that connects various services. The Amico middleware runs service adapters and local services in separate processes that are connected to the spreadsheets through an add-in. When users enter text, the system calls several services: it then updates the spreadsheet cells with a translation of the text in other languages, spelling suggestions, phrase definitions, and links to Web pages related to the entered text; it also plays a recording of the original and translated text using TTS output.

Specifically, as Figure 3 shows, AMICO_WRITE("spelling";B10) is evaluated every time cell B10 is updated and calls the Google spelling-checker service (triggered when the Amico "spelling" variable is updated). AMICO_READ("spelling-suggestion") then reads the current value of "spelling-suggestion" that

Figure 3. A spreadsheet mashup example. This mashup combines OpenOffice.org's Calc spreadsheet with services that update the spreadsheet's cells with text translations, spelling suggestions, phrase definitions, and links to related Web pages. It also plays recordings of the original and translated texts using text-to-speech output.

the Google spelling-checker adapter updates and also registers notifications about future variable changes.

Scripting and Advanced Programming Mashups

Advanced users and developers who are skilled in scripting and mainstream programming languages and XML can access Amico and mashup services from their development environments.

Scripting languages support. Scripting languages are usually easy for end-user developers to learn because they use typeless approaches to achieve a higher level of programming, which enables more rapid application development than system programming languages. However, there's a huge diversity of scripting languages, and each has a significant supporting community. We therefore decided to support several popular scripting languages, including

- JavaScript, a dynamic, weakly typed, prototype-based language;
- Python, a high-level programming language that supports multiple programming paradigms (object-oriented, imperative, and functional);

JavaScript	Prolog
<pre>function variableUpdated(name, value, oldValue) { if (name == "spelling-suggestion") {if (value = "") { amico.update("tts-text", "No suggestions"); } else { amico.update("tts-text", value); } } }</pre>	<pre>variable("spelling-suggestion", "") :- amico_update("tts-text","No suggestions") variable("spelling-suggestion", V) :- amico_update("tts-text", V).</pre>

Figure 4. JavaScript and Prolog script examples. Both scripts send the result of the spelling-suggestion Web service to a local text-to-speech service when the "spelling-suggestion" variable is updated.

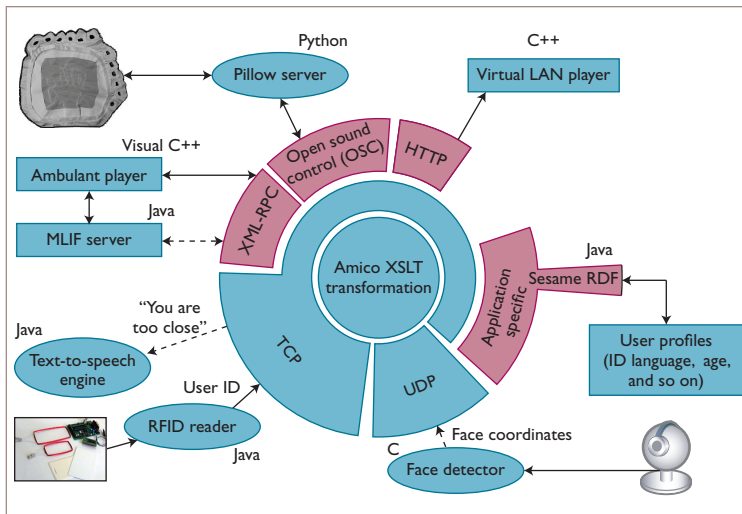


Figure 5. An advanced programming language mashup. In this example, taken from the Passepartout project, we created a prototype of multimodal interaction with multimedia players, combining several components written in different languages that are accessible through diverse service interfaces.

is based on the JLog project (<http://jlogic.sourceforge.net>). Our middleware runs the scripting engine for each language, extending each language with functions for updating and reading variables (the UPDATE and GET functions). To receive the notifications about variable updates (the REGISTER function), the script must contain a hook function, variableUpdated, which the middleware then calls when a given variable is updated. For Prolog, a nonprocedural scripting language, we introduce special predicates for updating, reading, and receiving notifications about variable updates (see Figure 4). We also support Extensible Stylesheet Language Transformation (XSLT) scripts, implemented using standard Java XML libraries. These scripting extensions let users and developers use different programming modes – such as declarative programming, object-oriented programming, or logic programming – or combine them to compose services.

- BeanShell and Groovy, two Java-based scripting languages;
- Ruby, a dynamic, reflective, general-purpose, object-oriented programming language;
- TCL, a popular tool command language;
- Sleep, a procedural scripting language inspired by Perl and Objective-C;
- Haskell, a standardized, purely functional programming language with nonstrict semantics; and
- Prolog, a logic programming language.

Our general scripting support implementation is based on the Java Scripting Project (<https://scripting.dev.java.net>); Prolog support

Mainstream language support. Because the interfaces Amico uses are widely supported, users can access Amico middleware using almost any standard programming language on almost any platform. Developers can access Amico services using libraries for interfaces, such as XML-RPC (www.xmlrpc.com/directory/1568/implementations) or OSC (<http://opensoundcontrol.org/implementations>), or by exploiting lower-level TCP or UDP interfaces using the socket library. We also offer libraries for Java and C++, which makes work with the socket library easier.

Figure 5 shows results from the Passepartout project, in which various partners use different programming languages and service

interfaces to access and combine Amico services and export their functionality to other components.¹⁰ In this example, we used several of the supported service interfaces and mashup environments.

Web Browser Mashups: Ajax, Plug-Ins, and Applets

Most service mashups compose services on the server side, presenting an aggregate interface to the UI-based client. For our solutions, we also required client-side mashups, so users could combine local services – such as a camera-based gesture recognizer or TTS output – with remote Web services. Clients can create such mashups in Web browser clients using Ajax, applets, and browser plug-ins.

Figure 6 shows two examples built using Amico Web browser mashups; you can find more elaborate descriptions of using the three extensions and their applications elsewhere.⁴

Browser plug-ins. We developed a generic Firefox/Mozilla browser extension using a Firefox/Mozilla extension mechanism based on the Massachusetts Institute of Technology's Semantic Interoperability of Metadata and Information in Unlike Environments (Simile) open source Java Firefox Extension (<http://simile.mit.edu/java-firefox-extension>). The extension gives users full access to browser functions and a Web page's content.

The browser plug-ins in our examples usually serve as a front end for service mashups defined using other approaches. In the Figure 6a example, users can select any text from the page, ask for a translation into a selected language, and hear the translation with the appropriate TTS engine.

Ajax. Figure 6b shows a screenshot of a Web page with Ajax and applets that let a face detector control playback on a movie player embedded within the Web page. Ajax and Web browser scripting functions can use the XMLHttpRequest object to access Amico through the Amico HTTP interface. A mashup of Amico services then occurs within scripting functions and event tags, and users can combine page and browser functionality with Amico services. We've also been experimenting with adding advanced graphical UIs – such as Flex and Scalable Vector Graphics – on top of Ajax to

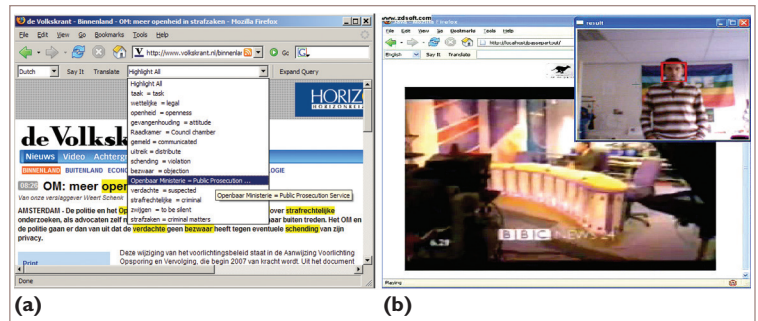


Figure 6. Web browser mashup examples. (a) The AMICO:WEB browser toolbar connects a Mozilla/Firefox browser with the Babelfish Web translation service and to local database and text-to-speech engines. (b) A Web page with Ajax and applets that lets a face detector control playback of an embedded movie player.

enable, for example, the use of drag-and-drop functionality to connect services.

This approach's advantage is that it lets developers use a well-established browser scripting environment without any extensions. The main limitation is that Ajax script functions can update or request Amico values only as a response to events within the page; they can't receive notifications from Amico.

Consequently, we can't use only Ajax to build interactive applications that require frequent updates, such as from a face-detection device.

Applets. To overcome Ajax limitations and enable fully bidirectional communication between Amico middleware and a Web browser, we combined scripting and a custom-built Java applet. The applet can update Amico variables and receive Amico service notifications, mapping them to calls of any embedded scripting functions.

The disadvantage of using applets is that it requires users to run a Java virtual machine, which can introduce significant browser overhead. Also, browsers can impose complex security limitations on applets.


The diversity of software services and user needs is often a problem, but – as our work shows – it can also open many new possibilities, including easier integration of existing software and lower learning curves. Our framework follows three basic design principles:

- **Simplicity.** Existing mashup interfaces usually work with complex XML schemas that

might be hard for users to understand. Our aim is to support end-user development and rapid prototyping for people who can't spend much time learning new environments and configuring the details of every service. To achieve this, we provide simple service abstractions and compositions that a range of users can understand.

- *Diversity and extensibility.* Most existing mashup solutions support one group of standards, such as Web service standards, and aren't open or easily extensible. However, there are many service standards, and although Web services standards are increasingly accepted, there will always be communities that use different standards, introduce new ones, and build solutions accordingly. Our mashup framework supports many existing standards but allows for the addition of new service interfaces and mashup development environments.
- *Reuse of existing development environments.* Existing mashup interfaces usually introduce novel functionality that requires additional learning, whereas building new development environments is a tedious and time-consuming task. In either case, it's hard to predict whether users will accept the results. Our framework adapts existing environments, such as spreadsheets, letting users build on previous experiences and learn how to compose services faster and more efficiently.

In our future work, we plan to introduce more graphical mashup development interfaces. We also plan to develop rich libraries of ready-to-use functionality for each of the mashup environments that our framework supports. One potentially interesting new idea is to add an abstraction layer that would permit better monitoring, analysis, and debugging of randomly composed services, as well as easier and better reusability of existing mashups in new contexts.

Our framework is freely available at <http://amico.sourceforge.net>. 

References

1. Ž. Obrenović and D. Gašević, "Open Source Software: All You Do Is Put It Together," *IEEE Software*, vol. 24, no. 5, 2007, pp. 86–95.
2. W. Keith Edwards, "Putting Computing in Context: An Infrastructure to Support Extensible Context-Enhanced Collaborative Applications," *ACM Trans. Computer-Human Interaction*, vol. 12, no. 4, ACM Press, 2005, pp. 446–474.
3. G.A. Papadopoulos and F. Arbab, "Coordination Models and Languages," M. Zelkowitz, ed., *Advances in Computers – The Engineering of Large Systems*, vol. 46, Academic Press, 1998, pp. 329–400.
4. Ž. Obrenović and J. van Ossenbruggen, "Web Browser Accessibility Using Open Source Software," *Proc. 2007 Int'l Cross-Disciplinary Conf. Web Accessibility (W4A 07)*, vol. 225, ACM Press, 2007, pp. 15–24.
5. B. Shneiderman, "Promoting Universal Usability with Multilayer Interface Design," *Proc. 2003 Conf. Universal Usability (CUU 03)*, ACM Press, pp. 1–8.
6. A. Sutcliffe and N. Mehendjiev, "End-User Development: Introduction," *Comm. ACM*, vol. 47, no. 9, 2004, pp. 31–32.
7. A. Thor, D. Aumüller, and E. Rahm, "Data Integration Support for Mashups," *Proc. 6th Int'l Workshop Information Integration on the Web, (IIWeb 07)*, AAAI; http://dbs.uni-leipzig.de/file/IIWeb2007_final.pdf.
8. J. Wong and J.I. Hong, "Making Mashups with Marmite: Towards End-User Programming for the Web," *Proc. SIGCHI Conf. Human Factors in Computing Systems (CHI 07)*, ACM Press, pp. 1435–1444.
9. Ž. Obrenović and D. Gašević, "End-User Service Computing: Spreadsheets as a Service Composition Tool," *IEEE Trans. Services Computing*, vol. 1, no. 4, 2008, pp. 229–242.
10. F. Nack et al., "Pillows as Adaptive Interfaces in Ambient Environments," *Proc. Int'l Workshop Human-Centered Multimedia (HCM 07)*, ACM Press, pp. 3–12.

Željko Obrenović is an assistant professor at the Industrial Design Department of the Technical University Eindhoven (TU/e), and he did part of the work reported here while working at CWI, Amsterdam. His research interests include human-computer interaction, interaction design, end-user programming, and software engineering. Obrenović has a PhD in computer science from the University of Belgrade. Contact him at z.obrenovic@tue.nl.

Dragan Gašević is a Canada Research Chair in semantic technologies and an assistant professor in the School of Computing and Information Systems at Athabasca University. His research interests include the Semantic Web, model-driven software engineering, service-oriented architectures, and technology-enhanced learning. Gašević has a PhD in computer science from the University of Belgrade. Contact him at dgasevic@acm.org.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.