A Collaborative Programming Environment for Web Interoperability

Adam Cheyer and Joshua Levy

SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, USA
adam.cheyer@sri.com, levy@csl.sri.com

Abstract. We describe a new type of collaborative system that exhibits much of the simple, cooperative nature of a wiki, but allows dynamic sharing of functionality as well as of content. In contrast with traditional wikis, pages in this system are executable, and interoperate with each other by passing and returning data structures of known type, such as messages, URLs, or locations. This collaborative programming environment is well suited to retrieving and combining content available on the Web. Since code within pages can access any type of Web content, the environment provides a collaborative way to convert diverse, unstructured information into semantically annotated content that can be combined into new and useful services. We discuss how these ideas have been applied in WubHub, a prototype Web portal with a command-line interface.

1 Introduction

The rise of dynamic, massively multi-user Web applications has led to the rapid success of wikis, social bookmarking services, and similar tools, and begun to demonstrate the immense potential of large-scale collaborative software. Prominent recent successes include Wikipedia, Flickr, and del.icio.us [1–3], but current examples have become too numerous to list.

So far, most such systems have focused on the capture and retrieval of data that is mostly unstructured, like text (in wikis), or of a small number of predefined types, such as URLs (del.icio.us), images (Flickr), or tags (numerous sites). Increasingly, the key features of wiki-like tools – collaboration and ease of use – are being applied in new contexts, in systems that work with new types of data, and with data of increasingly rich variety.

This trend shows great promise in addressing the long-standing and difficult problem of semantic interoperability on the Web. A fundamental tenet of the Semantic Web effort is that increased semantic annotation increases opportunities for use of online services, including automated or semi-automated service discovery, composition, and invocation [4, 5]. At the same time, as the "Web 2.0" phenomenon has demonstrated, collaborative tools, even with very limited and informal semantic support, can dramatically increase the knowledge captured and accessible to users. The challenge – and the opportunity – is to find a way to accommodate greater semantic precision while simultaneously encouraging

and reaping the benefits of large-scale community collaboration. Recent efforts ranging from microformats [6] to semantic wikis [7–9] are identifying a variety of approaches to reconciling these apparently conflicting goals.

In this paper, we outline a somewhat different angle on the same problem. Most existing collaborative sites, including newer systems like semantic wikis, are content oriented. We describe a collaborative framework that is *service oriented*, permitting sharing of services as well as of content. It is a collaborative programming environment that can access and integrate existing online content and services. In particular, it can extract structured information from existing unstructured Web content and combine it to produce new functionality. The collaborative process of defining new services simultaneously enables semantic annotation of existing unstructured data. Although it is still immature, we believe this framework could lead to practical and effective approaches to solving some current problems in Web interoperability.

In the next section we give a brief example of collaboratively defined services. Section 3 addresses design considerations for the underlying collaborative programming environment. Section 4 describes our proof-of-concept implementation. Sections 5 and 6 hold some additional discussion and related work. Future work is described in Sect. 7.

2 A Collaborative Service Portal

We motivate our discussion by sketching some examples of user interaction with WubHub, a proof-of-concept Web portal we have built that demonstrates collaboratively programmable services.

2.1 Calling and Combining Commands

The user visits the Web portal. Its interface consists of an input field – a Web-based command line – and a frame that displays content. The user can type in commands that perform many different services, such as retrieving information, performing an action or computation, or converting data. Commands can accept input as arguments, and provide a return value, which is displayed in the content frame. Commands can perform many useful everyday tasks. Figure 1 shows some typical commands. Figure 2 shows the portal in use.

Commands in WubHub can be composed to produce novel functionality, either within a new command definition or directly on the command line. For example, a user might employ three independently contributed functions to display a map with friends' locations, by typing

```
map: geocode: friends()
```

2.2 Finding and Editing Pages

What is novel about WubHub is not the commands themselves, but that these commands are built collaboratively by users. Just as wiki pages may be created

- `distance(94702, 94025)`: Use an online map service to calculate directions between two US zip codes, and return the driving distance between those points.
- `dict(forgetful)`: Look up the word "forgetful" in an online dictionary, extract the definition from the page, and return it to the user.
- `geocode("San Francisco, CA")`: Return the latitude and longitude of the specified address.
- `slashsearch(linux)`: Get a list of links to recent Slashdot articles mentioning Linux.
- `azn(dickens)`: Redirect the user's browser to Amazon's search results for "Dickens."

Fig. 1. Some example WubHub commands.

by one user and improved upon by others, WubHub pages containing content, presentation logic, data conversion, or computational functions can be woven together in an iterative way by a distributed community. Commands that perform services are written as pages containing a scripting language that allows access to remote Web sites and flexible manipulation of structured and unstructured data. Like conventional subroutines, pages may call other WubHub pages. The portal also provides simple development support, allowing the user to try writing new scripts and to debug them.

As users create new content and commands, they can organize and search for them using collaborative tagging. Additional meta-data makes it easy to discover pages added by the community using commands and subscriptions such as "whatsnew" and "popular" (Fig. 3). Finally, users can view the source code to any WubHub page, including system commands, and either modify the page directly (if they have sufficient privileges) or copy it to a local space where they can make their own variants. Just as a "view source" capability was essential to the proliferation of the HTML-based Web, we believe it can accelerate innovation and adoption in a WubHub-like environment.

2.3 Collaborative Programming and Semantic Annotation

In short, WubHub is a collaborative platform for service development, where users can add new services to the system, and data can be passed from service to service easily. In addition, since the scripting layer has the ability to extract data with known semantics from unstructured, ad hoc Web content, it can simultaneously provide a way to add semantic annotation to unstructured data on the Web.

3 Collaborative Programming

The collaborative service portal we have sketched is an example of a general type of collaborative system: A lightweight, highly collaborative programming

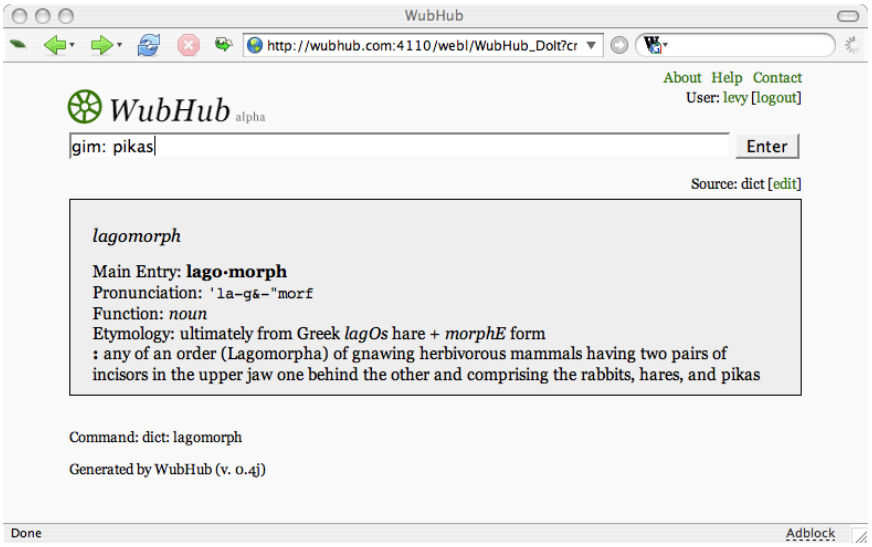

Fig. 2. A screen shot of WubHub, showing the result of executing a command to get a definition of a word. The user is now typing in a second command that will redirect to a Google image search for "pikas."

environment. This notion is quite new, and the proper design for such an environment, or even a complete set of requirements for one, is not yet fully apparent. However, our experience with WubHub has helped illuminate some basic requirements and an architecture, which we outline in this section.

3.1 Pages as Functions

The environment's main elements are

- Shared storage for *pages* holding pieces of content or code
- An execution environment, with a programming language and type system
- A user interface, with support for invoking commands and wiki-like editing of pages

A *page* in the environment is like a function in a programming language, with arguments and a return value, that is visible to, and modifiable by, many users. Essentially, the wiki-like editing capabilities are a front end for editing function definitions, and these functions may be executed, as if they were in a conventional (single-user) scripting environment. As in a traditional wiki, pages may actually contain only declarative data, such as text or HTML, which is really a special case where they accept no arguments, and provide their content as a return value. In effect, pages are variables within the programming environment, which may hold either functions or data.

- `ls`: List visible commands.
- `tags`: Browse pages by community tags.
- `whatsnew`: List recently created commands.
- `view(whatsnew)`: View the source code to the `whatsnew` command.
- `edit(whatsnew)`: Open a form for editing the source to the `whatsnew` command.
- `create`: Open a form for creating a new command.

Fig. 3. Selected commands for discovering, creating, and modifying pages.

3.2 Page Storage

Page storage is shared, so that all users have immediate access to pages from other users. Essentially, we desire a shared, transparent program and data persistence mechanism. Pages may be called from other pages directly, as if they were functions defined within the programming language.[1]

In addition to the page name and the body of a page, page records include signature information, a description, and other meta-data, such as tags added by users. For better organization, pages may be arranged into a hierarchy of modules. In general, the typical features of content repositories, including search, versioning, conflict detection, and access control, must be supported. A primary goal of the system is to encourage collaboration, so it is natural to make most content readable by all users.[2]

3.3 Programming Language

Pages must be written in a programming language. The requirements for such a "collaborative programming language" are a somewhat different than for conventional single-user programming languages. Some desired language features are

- *Interpreted (or dynamically compiled) execution*: We need the ability to compile or interpret new source code at all times.
- *Strong, dynamic typing*: Strong data types within the language are essential, so that it is possible for a routine to know the type of a value, and to ensure safe execution. Although static variable types could work in principle, in this environment the ease of use and flexibility of dynamic types may outweigh the benefits of compile-time type correctness.
- *A rich, extensible set of types*: The set of possible data types can be large, with a subtype relation and possibly other relations as well. It must be possible to add new types at runtime.

[1] The term "page" may be slightly misleading, since a page holds code or data, not a Web page. We use this term simply to distinguish pages from variables or functions in most programming languages, which are not necessarily persisted and do not hold the same meta-data.

[2] In principle, users could publish pages that are executable but not readable, but this "closed source" approach removes opportunities for collaborative bug fixing and individual customization.

- *Safe code execution*: The execution environment must provide security mechanisms that permit safe execution of untrusted code. It is necessary to run pages within a sandbox that minimizes the likelihood that scripts will compromise the host system's integrity (e.g., modifying system files), confidentiality (e.g., exposing sensitive local data), or availability (e.g., consuming excessive memory, processing, or bandwidth). The sandbox should also provide mechanisms that restrict abusive behavior that could adversely affect remote systems (such as posting comment spam to open websites).
- *Access control*: While not essential in all applications, access control support for data and code within the environment (such as user-based read, write, and execute permissions on pages) is highly useful.
- *Library support*: Support for rich data structures and parsing tools to convert ad hoc data formats to convenient internal representations, particularly for ubiquitous formats such as HTML and XML.
- *Ease of use*: For collaboration to work on a large scale, the language must be accessible to casual developers.

We know of no existing programming languages that fully meet all the above needs, but several popular languages, including Ruby, Python, JavaScript, and Lisp/Scheme, come close to meeting most of them. The least-supported language feature, and arguably the greatest technical obstacle to building the execution environment, is safe code execution. The Java platform is one of the few general-purpose programming platforms that provides a mature and full-featured sandbox model [10, 11], though this model still does not support resource restrictions, such as CPU or network bandwidth limits. Ruby provides some sandboxing support, but it is not as expressive or mature. Probably the most widely used execution sandbox is the JavaScript environment within most Web browsers.

3.4 Data Types

Types in a collaborative programming language form the glue that enables multiple functions to work together, extracting, aggregating, processing, and rendering information. Values may have numerous types, such as numbers, strings, lists, URLs, zip (US postal) codes, HTML Web pages, or street addresses. A type (e.g., "movie listing") has relationships with other data types, in particular subtype (e.g., "movie listing" is a kind of "event") and containment (e.g., a "movie listing" includes a time, duration, a movie, and a location of type "address").

As users develop new pages that utilize new data types, they are effectively adding to an ontology of possible data types. New data types can be created and used in response to needs for particular services. For instance, an HTML page with driving directions might contain a map, an address, an estimated distance, and other information. Various services might extract useful data from such a page. A service interested only in the map could create a data type for storing the map image, while a distance-computing service would use (or create) only a numeric type for distance.

3.5 Composing Services

Pages should be able to pass and return data values. Once content from the Web is brought into the system as a known data type, it can be passed as an argument to any other page that accepts that data type. New services can be built by composing the functionality offered by multiple pages. For example:

- One user writes a page that uses an online directions service to compute the distance and driving time between two zip codes. Another user writes a custom page that accepts a single address and determines the driving time from her house to that address.
- Several users write pages that query various online event-listing websites for upcoming events. Although the websites present the data in different formats, each page would extract information and return its result as an "event" data type holding a title, description, date, and URL. Another page is used to aggregate upcoming events at a particular location into a single array of objects, and a final page can be used to render them in a table.

3.6 Rendering and Data Conversion

The execution environment requires support for display of all kinds of data values. This is particularly helpful in keeping a clean separation between the underlying representation of data values, and the various presentations of them shown to the user. A simple technique is to build rendering functions that accept a value of a particular type and convert it to a presentable form (such as a fragment of HTML). These functions are simply pages themselves. For convenience, types have default rendering functions. That is, when a value is returned for display to the user, the system checks the value's type to determine an appropriate rendering page. The rendering page is called with that value as its argument, and the result of that call is now an immediately displayable value (for a Web interface, HTML).[3]

This type of rendering can actually be considered a special kind of data conversion: in effect, it is implicit type coercion in the programming environment. Other simple, automatic conversions can be convenient, such as converting an ISBN to a "book" object with author, title, and so on. Going further, it is possible to imagine an extensible registry of data conversions, specifying which types can be converted, the circumstances under which the conversion may occur, and what page implements the conversion.

3.7 User Interface

Users can interact with the system in two ways: To issue "commands" that dispatch existing services, and to build new services. The user interface for issuing

[3] If a user-defined type does not have a default rendering function defined for it, a generic object rendering function is chosen. Also, it can also be useful for a page explicitly to select the rendering method for its return value.

commands simply collects a page name and an optional set of arguments. A simple way to do this is to provide an actual command line, much like a shell prompt. More graphical user interface widgets can provide similar capabilities in a more polished form.

The programming interface is potentially more complex, and ideally is closer in spirit to an integrated development environment. The simplest possible editing environment is inspired by wiki editing, and consists of a set of pages that can be edited from within a Web browser. The benefits are simplicity and ease of use. However, additional features can facilitate development of new pages. One particularly useful feature is a read-eval-print loop for executing script expressions. For instance, when trying to extract content from a Web page, the Web page can be fetched and parsed, and then the user can interactively try evaluating various expressions to determine how to extract data of interest. Pages can also be invoked interactively during the development process. Of course, as with any development environment, more advanced graphical interfaces could offer shortcuts for composing existing functions or pages.

3.8 Web Protocol Support

The framework can interoperate with other Web services only if it has library support for relevant protocols. Structured formats such as RSS, SOAP, or Semantic Web service protocols are useful, but support for common, human-browsable (and in practice, non-standards compliant) HTML is also key, as this format is still the dominant form for free information on the Internet. In particular, there should be a parser and document representation (such as the W3C document object model [12]) that allows "Web scraping" and general manipulation of XML and HTML documents.

3.9 Architecture

We have mentioned all three key architectural elements of the framework: a storage mechanism, an execution environment, and a front end. The simplest deployment is to put all three parts on a single server. In this case, a Web-based user interface can be provided by a Web application server, which hosts the execution environment and the page storage. The thin clients are simply Web browsers.

Other architectural arrangements may offer greater benefits. The only element that must be shared, and hence is most easily kept on a centralized server, is the storage component. Page execution and the user interface could be deployed on a thicker client. This option may no longer offer a "zero install" client, but has the advantages of greater scalability. Another critical factor in this choice is risks of and incentives for abuse. Execution on the client removes incentives for abusing server resources, but increases opportunities for malicious client-side exploits.

```
// Fetch page, using localsearchmaps.com's free geocoder
var P = GetURL("http://www.localsearchmaps.com/geo/?loc=" + Url_Encode(addr));
// Parse results, extracting the numbers (lat, long)
var LatLong = Pat(P, `([-\d.]*)`);
if (Size(LatLong) > 2) then
   [. long=LatLong[1][1], lat=LatLong[2][1], label=addr .]
else
   nil
end;
```

Fig. 4. The source code for the `geocode` page, which accepts a parameter `addr`, passes it to a free geocoder service, and then extracts the coordinates from the results.

4 WubHub Implementation

4.1 Features

We have built a simple prototype of the environment, the WubHub portal, and, as shown in Sect. 2, it demonstrates most of the basic service editing capabilities we have described. Some of the more advanced features we have described, such as a rich type system or an extensible system for implicit data conversion, are not yet fully supported.

Our prototype is built with and uses the WebL programming language [13], a small scripting language for the Java platform that has built-in support for fetching, manipulating, and creating Web content. WubHub supports a variety of page types: WebL, HTML templates, plain text, URL redirects, and aliases. WebL pages are simply functions, written in the WebL, that accept parameters and return a value. (An example of a short WebL page is displayed in Fig. 4.) The HTML type provides a convenient way to create regular Web pages, like in a wiki, as well as a templating feature, to display results of WebL computations with HTML formatting.

When a user invokes a command, the corresponding page is executed, and the result is rendered and returned to the user. In the case of HTML templates, the page, with appropriate substitutions, is returned. In the case of URL redirects, an HTTP redirect, with appropriate substitutions, is sent to the browser. All commands have a corresponding invocation URL, so commands such as search results or data feeds can be bookmarked by users. For instance, if a command returns data in RSS format, a user could direct an RSS reader to subscribe to the URL of that command.

Pages are stored within a number of modules, including a global module, a system module, and individual modules for each user. All pages are completely public for viewing and execution, although a user may not modify pages in the system module or in the module of another user.

4.2 Implementation

The WubHub implementation is quite simple; in fact, WubHub is largely written using itself. Almost all basic features, such as page creation and page editing, are

themselves handled by pages. (To prevent accidental or malicious damage to site functionality, these important pages may be edited only by site administrators.) One advantage of choosing WebL is that it provides all needed parsing tools for real-world, ill-formed HTML. It provides XML and HTML parsing, selection, search, and creation capabilities as native parts of the language. Data is passed between pages via native WebL types, which can represent HTML fragments, strings, numbers, arrays, and objects. Objects are used to create new types, and hold a set of named fields.

The execution sandbox consists of two layers. Because WebL is a very small language, it is relatively straightforward to check for any commands that might be abused, such as file-system operations. A second layer of protection is via a standard Unix chroot() environment. We have not yet implemented any resource constraints within the sandbox, so it is still possible to consume excessive CPU or network bandwidth.[4]

4.3 Deployment and Use

WubHub is remarkably easy to use as a portal that provides simple services rapidly. In many cases, it is faster and more efficient to use a command that performs a Web query than to browse the original Web site. As a special case, redirects to a URL do not provide any data extraction capabilities, but behave more like a bookmarking or search service.

We have deployed the system for use by a small community of pilot users, who have contributed a variety of simple commands. Examples range from assembling search results from well-known news or search engines, to sending SMS messages, to building statistical charts.

The current version of WubHub is a proof of concept only, and requires a number of improvements for broad use. A key difficulty is ease of programming. WebL is a somewhat arcane language, and is completely unfamiliar to most users. Debugging support via the Web is fairly limited. The type system should also be enhanced. Objects are essentially structures with named fields, without inheritance, so implementing more complex types and relationships between types is cumbersome. Finally, scalability and security, including resource throttling, are further areas that will require more substantial development effort.

5 Discussion

5.1 Practical Issues

The use of hand-coded Web scraping tools instead of well-defined Web service protocols raises an immediate concern: Because there is no defined service API,

[4] One approach to enforcing such constraints is to build a supervisor thread that monitors running threads and terminates those that exceed certain limits. In the meantime, to discourage abuse, we have employed an invitation system, where members can join only via an invitation from an existing member.

the code for extracting information from an unstructured Web site could break whenever the Web site changes the way it presents its information. On the other hand, the community of users has an interest in correcting such problems, and we should take care not to underestimate the strength of numbers.

Another issue is that some content providers legally restrict non-interactive access to their content (for example, because they depend upon click-through advertising revenue). We believe, however, that future Web applications will trend toward "software-as-a-service" publishing of functionality, and that business models are likely to arise that will encourage providers to participate in a collaborative, Web-scale marketplace of services and content.

5.2 Type Systems and Ontologies

Exactly what sort of type system is best suited to this environment is not yet clear. At one end of a spectrum, types and type checking could be quite simple; for instance, the data types could consist only of primitive types, arrays, and named records, and type checking would involve raising runtime exceptions when invalid operations, such as accessing invalid fields, occur. At the other end, we can imagine rich types with a great deal of semantics, and possibly reasoning support as part of the type checking process. For instance, some services apply only to data values that satisfy certain conditions – say, geographical locations within North America – and these subtype conditions could be checked automatically, either at development time or at runtime.

On a recent collaborative research and development project,[5] the authors approached this issue from the semantic side of the spectrum, employing OWL, Jena, and a "semantic object" framework to model system data and services [14]. However, for the applications and community targeted by WubHub, we have found it is an advantage to follow the philosophy of collaborative systems like wikis, where there is strong emphasis on ease of use and a low barrier to users correcting problems. A more relaxed approach to data typing can lead to semantic imprecision and bugs, but this risk is counterbalanced by rapid, iterative modeling and greater numbers of users identifying and correcting problems. Loose typing practices such as "duck typing," which has become popular among Python and Ruby programmers [15, 16], are also useful, as they encourage aggressive reuse of code.

5.3 Three Approaches to Semantic Annotation

One way to put the collaborative programming approach in context is to consider some of the other approaches to semantic annotation of unstructured content. In the traditional Semantic Web vision, information from all sources is transformed from unstructured formats to standardized semantic representations. Unfortunately, this vision has been relatively slow to take shape. A key impediment is

[5] CALO (http://www.caloproject.sri.com) is a DARPA-funded project where SRI is leading 25 subcontractors to construct a personal assistant that learns.

that interoperability typically depends on data providers converting or adapting their data to a new standard before it can be consumed by others – an expensive process, and one for which there is often no strong incentive to be an early adopter. Moreover, a standardization process is needed before potential users gain enough confidence to adopt a new format. In particular, with this approach, benefits are not immediate or incremental.

A second approach is to leverage the power of a community to collaboratively annotate content. Collaborative systems can solicit relatively small contributions from many early adopters, and do incrementally increase in value as more contributions are added. Note, however, that users must still do the work of migrating content into a new format and repository before it is useful (such as with del.icio.us).

But can we go one step further, supporting collaboration and also allowing existing data to remain in its existing formats and locations? This may be possible by shifting our focus from collaboratively developed content to collaboratively developed services. If collaborative infrastructure becomes powerful enough that users can share annotation *mechanisms*, rather than just annotated data itself, we can benefit from community collaboration to extract semantic information from many existing, unmodified data sources. Because the construction of new services is possible immediately, the framework provides an incentive to contribute, which accelerates adoption.

6 Related Work

6.1 Programmable Websites

Executable code within a wiki is not a new idea; some existing wikis already support dynamic variables and other execution directives (e.g. ZWiki [17]).[6] More ambitious "programmable wikis" have been discussed [18], but none have become mature or popular, mainly because of the difficulties of sandboxing (particularly for legacy wiki systems, such as those based on PHP or Perl), and perhaps also because of a lack of compelling example applications.

The inspiration for WubHub's command-line interface comes from the collaborative website YubNub [19], which allows users to define and share commands that redirect to other Web sites, particularly Web searches. However, it has no general programming mechanism.

A few websites have recently begun to allow users to develop new Web applications through the Web. These include Ning, YouOS, and EyeOS [20–22]. However, most of these sites aim to replicate a traditional development style, with a single developer (or perhaps a single team) producing a standard Web or desktop-type application. As far as we know, our system may be the first to encourage users to compose services from each other's contributions, so that a single service is actually the result of a community effort.

[6] Of course, Web application tools such as JSP, PHP, or executable templates provide another example of dynamic pages, but these pages are themselves rarely edited collaboratively.

6.2 Widgets and Browser Tools

Several systems, such as Yahoo's Konfabulator [23] and Google Widgets [24], encourage a community of developers to create distributable functionality that can be inserted into the user's workspace. In contrast with WubHub pages, each widget provides a standalone block of functionality tightly connected to graphical output and cannot be composed to produce additional functionality.

Active browsing, such as implemented by the Greasemonkey [25] browser plug-in, allows users to program their Web browsers to manipulate Web content. The purpose of scripts is usually to make specific adjustments to pages as they are browsed, such as adding links to related content. These "user scripts" can be shared via centralized websites [26], but currently there is little support for fully extracting content from websites or for composing scripts to get new functionality.

6.3 Semantic Web Services

Semantic Web services [27, 5] are the standard approach to providing services on the Semantic Web, and some subsequent research has investigated ways to involve users in semi-automated composition of services [28, 29]. There have been a number of tool-building efforts related to the major Semantic Web services initiatives. For example, the Protege-based OWL-S Editor [30] has facilities for creating and composing OWL-S service descriptions, and provides a graphical editor that supports service composition and allows discovery of relevant services from local or remote service repositories. Maximilien et al. [31, 32] have discussed orienting Semantic Web services around human activities, and argue that keeping humans in the process of Semantic Web service automation is an important way to encourage widespread adoption. Aspects of this argument are very similar to the case we have made for involving users in the creation of new services.

6.4 Other Collaborative Environments

The Croquet system [33] is an effort at building a graphical collaboration system architecture. It shares some key features with the environment we have described, including the blending of user and development environments and a shared space where users can interact with content and scripts. It differs in that it is a larger, more ambitious system, with an emphasis on 3D visualizations, and it focuses more on users and user interactions within the system, rather than interoperable software and services and interaction with the Web.

While the setting is quite different, some similar collaborative programming ideas have previously arisen in the context of multi-user online "MUD" games. For instance, the MOO programming language [34] allows users to program the MOO game server in an object-oriented way, adding rooms or objects with scripted functionality. User-scripted functionality is also possible in some recent online games, including Second Life [35].

7 Conclusions and Future Work

We have described a collaborative programming environment where users compose and create new services that access nearly any type of existing online services. A key strength of the environment is that members of the user community can build upon each other's work, assembling new services that provide new or more customized features. As a part of this process, legacy content and services are reshaped into a community-extensible set of known data types, effectively adding semantic annotation to existing Web content.

The WubHub prototype has demonstrated that users can collaboratively write and compose services by using such a framework. It also presents numerous opportunities for integration with other applications, such as allowing WubHub-supplied data to be integrated into desktop widgets via remote procedure call. However, as we have discussed (Sect. 4.3), a variety of implementation issues must be addressed before the system is ready for broad use.

The requirements and design we have outlined for the collaborative programming environment leave much room for future work. As already mentioned (Sect. 3.3), popular programming languages do not provide all the language features desired for such environments, so there are opportunities for designing language enhancements that make collaborative programming easier, more powerful, and more practical. Key areas include enhanced security and permissions management, and greater flexibility and semantic precision for data types. The programming environment also needs more modular software development support, user interface enhancements, such as graphically specified compositions, and more powerful debugging tools, such as automated testing and perhaps type inference or static analysis. Such enhancements present both technical and social challenges: In addition to solving knowledge representation and software engineering issues, they must also encourage productive community effort. If both aspects are addressed, the result may be a significant step toward functional and ubiquitous online services.

8 Acknowledgments

We wish to thank Jonathan Cheyer for his help with deployment, David Martin and Natarajan Shankar for their comments, and the WubHub users who have contributed ideas and their own new services, including John Bear and Andreas von Hessling.

References

1. (Wikipedia) http://www.wikipedia.org.
2. (Flickr) http://www.flickr.com.
3. (del.icio.us) http://del.icio.us.
4. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American (5) (2001) 34–43

5. Martin, D., McIlraith, S.: Bringing semantics to web services. IEEE Intelligent Systems (2003) 90–93
6. (Microformats) http://www.microformats.org.
7. (Semantic MediaWiki) http://meta.wikimedia.org/wiki/Semantic_MediaWiki.
8. (Platypus Wiki) http://platypuswiki.sourceforge.net.
9. (IkeWiki) http://ikewiki.salzburgresearch.at.
10. Gong, L., Mueller, M., Prafullchandra, H., Schemers, R.: Going beyond the sandbox. In: USENIX Symposium on Internet Technologies and Systems, Monterey, CA (1997) 103–112
11. Gong, L., Schemers, R.: Implementing protection domains in the Java Development Kit 1.2. In: Internet Society Symposium on Network and Distributed System Security, San Diego, CA (1998) 125–134
12. W3C: Document Object Model (DOM) level 1 specification (1998) http://www.w3.org/TR/REC-DOM-Level-1/.
13. Marais, H.: Compaq's Web Language: A programming language for the Web (1999) http://www.hpl.hp.com/downloads/crl/webl/library.html.
14. Cheyer, A., Park, J., Giuli, R.: IRIS: Integrate. Relate. Infer. Share. In Decker, S., Park, J., Quan, D., Sauermann, L., eds.: Proc. of Semantic Desktop Workshop at the ISWC, Galway, Ireland. Volume 175. (2005)
15. van Rossum, G.: Python tutorial. (2005) http://docs.python.org/tut/tut.html.
16. Thomas, D., Fowler, C., Hunt, A.: Programming Ruby: The Pragmatic Programmer's Guide. Pragmatic Programmers (2004)
17. (ZWiki) http://www.zwiki.org.
18. (Meatball wiki's discussion of programmable wikis) http://usemod.com/cgi-bin/mb.pl?CommunityProgrammableWiki.
19. (YubNub) http://www.yubnub.org.
20. (Ning) http://www.ning.com.
21. (YouOS) http://www.youos.com.
22. (EyeOS) http://www.eyeos.com.
23. (Konfabulator) http://widgets.yahoo.com.
24. (Google Widgets) http://www.google.com/ig/directory.
25. (Greasemonkey) http://greasemonkey.mozdev.org.
26. (UserScripts) http://www.userscripts.org.
27. McIlraith, S., Son, T., Zeng, H.: Semantic web services. IEEE Intelligent Systems (Special Issue on the Semantic Web, March/April) (2001)
28. Sirin, E., Hendler, J., Parsia, B.: Semi-automatic composition of web services using semantic descriptions. In: Workshop on Web Services: Modeling, Architecture and Infrastructure, in conjunction with ICEIS2003. (2002)
29. Sirin, E., Parsia, B., Hendler, J.: Composition-driven filtering and selection of semantic web services (2004)
30. Elenius, D., Denker, G., Martin, D., Gilham, F., Khouri, J., Sadaati, S., Senanyake, R.: The OWL-S editor: A development tool for Semantic Web services. (2005) 78–92
31. Maximilien, E.M., Cozzi, A., Moran, T.P.: Semantic web services for activity-based computing. In: Proceedings of 3rd International Conference on Service-Oriented Computing (ICSOC 2005), Amsterdam, The Netherlands (2005)
32. Maximilien, E.M.: Semantic web services for human activities (2005) To appear.
33. Smith, D.A., Kay, A., Raab, A., Reed, D.P.: Croquet: A collaboration system architecture (2003) http://www.opencroquet.org/About_Croquet/whitepapers.html.
34. Curtis, P.: LambdaMOO programmer's manual (1997) ftp://ftp.research.att.com/dist/eostrom/MOO/html/ProgrammersManual_toc.html.
35. (Second Life) http://secondlife.com.