Lightweight Solutions for User Interfaces Over the WWW

Sunil Mishra and **Andres C. Rodriguez** and **Michael A. Eriksen** and
Vinay K. Chaudhri and **John D. Lowrance** and
Kenneth S. Murray and **Jerome F. Thomere**
Artificial Intelligence Center, SRI International
333 Ravenswood Ave, Menlo Park, CA 94025
Email: {smishra,acr,eriksen,chaudhri,lowrance,murray,thomere}@ai.sri.com

Abstract

Lisp-based HTTP servers provide a starting point for building Web-based applications written in Common Lisp. They do not provide a complete solution. An application needs additional facilities for generating content and managing dialogs. We present some lightweight tools that have been essential in producing applications with complex interactive interfaces. Expressing computed responses is greatly simplified by scripting a template HTML page with some Lisp code. This is managed through our Active Lisp Pages (ALP). Conducting a dialog with a user requires managing an evolving state, for which we have implemented a simple interaction manager. We integrate interactive interfaces developed by third parties in our application, using a simple API based on XML and XSLT. Finally, SOAP messaging provides applications with a standard means of exchanging messages with custom interfaces. These tools have allowed us to assemble consistent and usable user interfaces for our applications.

Introduction

The last decade has seen HTTP (Fielding *et al.* 1999), URI (Berners-Lee, Fielding, & Masinter 1998), HTML (Raggett, Le Hors, & Jacobs 1999), XML (Bray *et al.* 2000) and other standards become commonplace in computing. At the Artificial Intelligence Center (AIC) at SRI International, we have built several applications that utilize these standards. Initial efforts involved retrofitting existing CLIM applications for displaying output to the WWW (Paley & Karp 1995). Now, many of our applications are written with the user interface being delivered over the Web. They have been written using Common Lisp Web servers, in particular AllegroServe (Foderaro & Dancy 2000) and CL-HTTP (Mallery 1994). The Web servers manage the receipt of HTTP requests and the return of computed responses. In addition, CL-HTTP offers an array of tools, such as parsing and generating XML and HTML, that can be very useful in application development. However, the servers do not offer assistance in writing a complete interactive application. For this, we must develop a layer of tools and policies. The specific problems we consider, and the tools we have built to address them, are

1. Generating computed HTML responses, using Active Lisp Pages (ALP).
2. Allowing the programmer to approach a user dialog as writing a set of function calls, using a lightweight interaction manager.
3. Integrating user interfaces developed independently from the application, using XML and XSLT.
4. Exchanging structured data with special purpose interfaces, using an implementation of the Simple Object Access Protocol (SOAP).

We present an overview of the approaches we have developed and used. Our tools and techniques are lightweight: they are simple and straightforward to implement and use, making tradeoffs as necessary to maintain this simplicity. They are, however, employed in building Web-based applications with user interfaces rivaling desktop applications.

These tools have been used in a number of applications at SRI, of which we list only three here. SEAS (Lowrance, Harrison, & Rodriguez 2000) is a collaborative document management system. It is a mature application. Shaken (Thomere *et al.* 2002) enables subject matter experts (as opposed to knowledge engineers) to author concepts in a knowledge base. It has been under development for about two years, and has end users. LAW, an application only a few months into development, enables the authoring of patterns for information extraction. Each of these systems displays its primary user interface over the Web, through HTML, JavaScript, Java, XML, and other standards now commonplace.

Lightweight tools

We characterize a lightweight tool through the following properties:

- **Ease of implementation.** The tool should be simple to implement, maintain and adapt. This is essential if it is to be used across multiple applications, because an immature tool invariably requires some change when being applied to a new application.
- **Simplicity of API.** Writing a tool that performs a number of tasks often requires that its API extend to

cover all of its functionality. Such an API can quickly become difficult to understand and use, degrading the tool's utility.

- **Transparency of execution.** A programmer should, with relatively little effort, be able to observe the functioning of the tool. This is crucial for being able to debug and extend the tool.
- **Focus on essential functionality.** The tool should not aim to handle every possible situation, if it involves a significant loss in the ease of implementation, simplicity, or transparency in the implementation of the essential functionality.
- **Unenforced use policy.** A tool should not attempt to enforce a usage policy. Such a tool is generally more complex and relatively inflexible. Instead, it should rely on an appropriately informed programmer.

The remainder of the paper describes some of the tools we have developed. Each tool identifies a particular problem and implements a solution. Collectively they provide a set of building blocks for interactive applications with Web-based interfaces.

Generating HTML with ALP

An interactive Web-based application accepts a parameterized request through HTTP, and usually produces a response in HTML that specifically satisfies that response. A number of solutions have appeared to solve the problem of computing the HTML response: Java Server Pages (JSP), Active Server Pages (ASP), and PHP. However, they are not written in Common Lisp, and do not allow Lisp scripting. They are therefore difficult to use with Lisp applications that use an HTTP server written in Lisp. We have therefore developed a tool called Active Lisp Pages (ALP) (Rodriguez 2002).

Writing Lisp programs that generate HTML is at first glance not difficult. However, the HTML output required from an interactive application tends to be fairly complex, and maintaining such Lisp code has proven to be difficult in practice. In addition, this HTML often includes JavaScript [1] and potentially other embedded programs. Tracking the overall design of the generated page becomes very difficult.

An ALP page is an HTML page with some Lisp code embedded in it. It thus has the essential property of looking like the desired output. The ALP package takes such a scripted page as input and compiles it to a Lisp function. When called, this Lisp function computes an HTML page as response. Thus, ALP provides a convenient mechanism for writing Lisp programs that generate HTML. Figure 1 shows a short ALP page. (Rodriguez 2002) gives a more detailed account of ALP scripting.

Developing ALP is an important step in building applications with the Web as the primary interface. When developing an ALP page, a programmer can directly leverage knowledge of HTML and other Web related technologies such as JavaScript and CSS. ALP pages are more maintainable than their program counterparts. ALP integrates seamlessly with the Lisp environment, allowing for a clean separation of content, logic and presentation: most of the logic can be programmed in separate Lisp source files.

[1] For more information about JavaScript, see `http://devedge.netscape.com/central/javascript/`.

```
<html>
  <body>
    <% (let ((a (read-from-string
                  (get-parameter "a")))
             (b (read-from-string
                  (get-parameter "b")))) %>
    <p>
      <%= a %> + <%= b %> =
                    <b><%= (+ a b) %></b>
    </p>
    <% ) %>
  </body>
</html>
```

(a) The ALP script

```
<html>
  <body>
    <p>
      3 + 4 = <b>7</b>
    </p>
  </body>
</html>
```

(b) The HTML result

Figure 1: A sample ALP page that can handle a request of the form `http://server/alp/add.alp?a=3&b=4`. The page would be compiled to a Lisp function that, when invoked, would produce HTML.

Managing Web-based dialogs

Although ALP helps us with HTML generation, it does not offer any guidance in writing interactive dialogs. A series of HTML pages involved in a dialog constitute an evolving state, where the input from one step must be carried over to the next. The general method of addressing this problem is for each HTTP request handler in the application to explicitly access the parameters in the request, and manage the output of the server's response. In such an application, the details of managing the output parameters often obscures any transitions that the application makes between the steps in the dialog. Inventing and re-inventing ad hoc mechanisms for transitioning between these steps within a single application often leads us to a situation where the application becomes difficult to maintain. In Shaken, we elected to implement a simple interaction manager, and a usage policy for that interaction manager, to make these steps and transitions explicit.

The interaction manager casts the process of stepping from one dialog step to the next as calling the

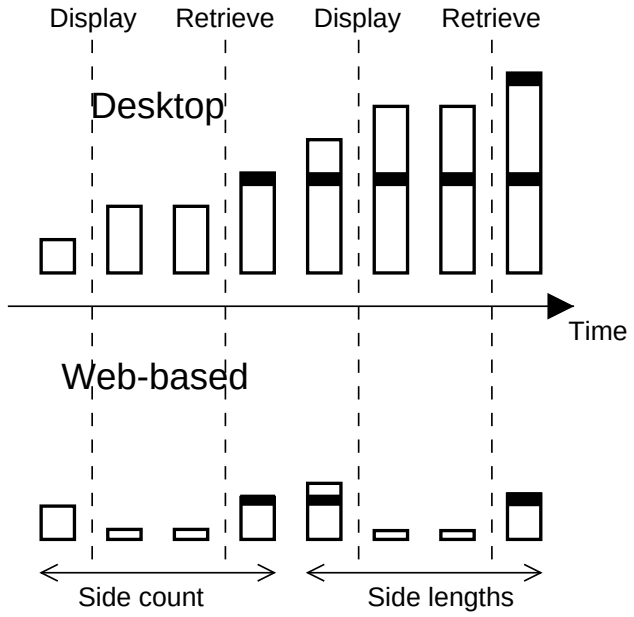

Figure 2: A Web-based application loses state as a dialog progresses.

next function from the present one. State is transitioned between one dialog step and the next by passing parameters between functions. The dialog itself thus begins to resemble a series of function calls. Such an approach based on function calls is easier for a programmer to comprehend. The interaction manager handles translations between the function arguments and the request parameters in the background.

State loss in Web applications

We organize a dialog into a series of steps, where a step consists of

1. Displaying some content where the user might enter data
2. Awaiting user input
3. Gathering and validating user input
4. Proceeding to the next step in the dialog

So, if we were asking the user about the dimensions of a polygon, one step in the dialog might be to obtain the number of sides in the polygon, and the next step to gather the length of the different sides. The execution thread in a desktop application first sets up a display with which the user may interact, perhaps a text entry field where the user may enter the number of sides. Then the thread would wait for user input, and when that input is available the program would retrieve it, and proceed to the next step. Figure 2 shows how the stack for the thread handling the dialog would look like. Each piece of information gathered from the user is present in the stack, represented by the darker stack frames.

Now consider how this same step for gathering the number of sides would operate on a Web application. The step would begin with the application receiving a request from the user, indicating that she wants to describe the dimensions of a polygon. The application would generate an HTML page with a form that allows the user to enter the number of sides. Then, instead of waiting, the execution thread in the server would discard the state generated in handling the user's request. This is necessary, for once the HTML page is sent as response, it is highly impractical to track the details of the user's actions at the granularity of a desktop application. When the user responds with the number of sides in the polygon, the execution thread that receives the request would have no context of what it is to do with this request. Unlike the desktop application, the Web application would have to be reminded of what it was doing before it would be able to make sense of the incoming request. Only then will it know how to process the data in the request, and what the next step in the dialog would be.

Tracking dialog state

The problem of state loss is one that is faced by every Web-based application. All solutions to this problem are basically similar in that they include in each HTML page some information about the current state of the computation, so that when the user returns a request to the server, the server can restore the previous state and continue with its computation.

In Shaken, we implement an interaction manager to manage stepping the Web application through a dialog. With this interaction manager Shaken is able to formalize and automate some of its dialog management:

- The means of indicating the current dialog state is formalized so that the thread handling an incoming request is automatically put into the right state before the request is handled.
- Each incoming request is translated to a function call on the server, so the programmer only has to write a function that handles the request.
- The error-handling policy is better standardized. The user repeats any step on which an error occurs, with the response showing a meaningful error message.

We begin with an analysis of how dialog steps must be organized to enable such an interaction manager. We have already described a stepwise organization of a dialog. Now we will further refine the step structure. There are two points of particular interest in managing a dialog:

- The point where we transition from one step to the next. This must be tracked so that we know how to begin a new step, and to return to a step if an error occurs.
- The point in the step where the user interacts with the step. Before this point, an HTML page must be

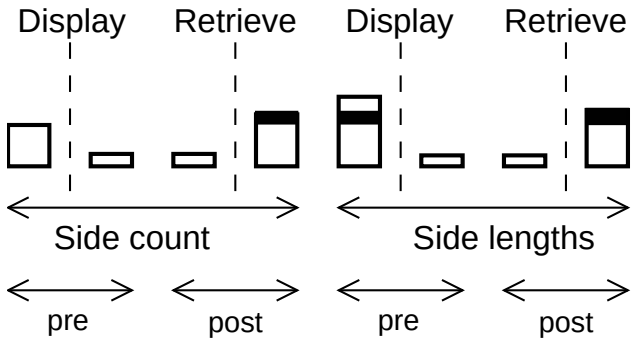

Figure 3: Step organization in Shaken's interaction manager.

generated for the user. After this point, the data the user provides must be processed by the server.

Figure 3 shows the step organization in Shaken's interaction manager. Each step is divided into a *pre-step* phase and a *post-step* phase. The tasks handled in these two phases may be correlated with the step structure we had proposed at the start of the section.

The pre-step phase is responsible for handling the display of an interface for the step, the first substep in a dialog step. The user then interacts with the Web client. This corresponds to the second substep within a step, where the application waits for user input. The thread of execution loses its state during this period between the pre-step and the post-step. The user interaction will typically result in the continuation of the dialog, with the user submitting a new request to the server. The post-step phase thus begins. Shaken's interaction manager then determines the step in progress, restores state for that step, and continues the dialog. In the process, it also retrieves all relevant parameters from the dialog request, thus allowing the Shaken programmer to concentrate on how the parameters should be processed rather than how they are to be retrieved. During the post-step, the programmer is expected to verify the arguments given as input. Thus, the post-step corresponds to the third substep in a dialog step. At the end of the post-step, the program makes a decision as to what step should be executed next, the fourth substep.

If an error occurs between the start of the post-phase of the current step or the end of the pre-phase of the next step (that is, between the server receiving a request and responding to it), the interaction manager resumes from the pre-phase of the current step. An error message is included with the parameters for generating the HTML page. For example, let us consider what happens if the user enters a non-numeric value for the number of sides of the polygon. This error will not be detected until the post-phase of the "Side count" step. A Lisp error is signaled here, which is caught and handled by the interaction manager. The interaction manager obtains a displayable string for the error, and reinvokes the pre-step of the "Side count" step with the error message. The net result is that the user is prompted again for the number of sides of the polygon, along with an error message explaining why the page has been redisplayed.

The implementation of Shaken's interaction manager is simple. Each step is implemented using a pair of functions, one corresponding to the pre-step and the second to the post-step. An incoming request indicates the step currently underway. The interaction manager resumes with the post-step function of this step. It obtains the argument list of the function using Lisp's introspective capabilities. Then, each argument is matched with the corresponding parameter name in the request. The post-step function is invoked with the argument list thus obtained. When the post-step function has finished its computations, including type checking and type conversions, it decides the next step to execute. The interaction manager calls the pre-step function for that step.

The pre-step functions and post-step functions are called within the context of an error handler. Thus, any error that occurs can be easily caught and handled within the interaction manager.

One final issue we have thus far glossed over is how the transfer takes place between the pre-step and the post-step. This is handled through policy, rather than implementation. The programmer is responsible for including each of the parameters that the post-step requires in the request that the HTML page will produce. This includes the step identifier to be given as input to the interaction manager.

Note that this is just one possible implementation of the interaction manager. Many other variations are possible in its design. We have chosen this one for the simplicity of its implementation. The interaction manager is completely independent of the Web server interface or the HTML generation engine that is employed. This greatly increases the transparency of how state is maintained across the dialog.

Integrating user interfaces

A significant portion of the Shaken server's development involves integrating components developed by individuals and institutions outside the AIC. Some of these components are exclusively for the server's use, and are not directly exposed to the user through a Web interface. An example is KM (Clark & Porter 1998), which handles all the knowledge representation and reasoning in Shaken. Other components, such as KANAL (Kim & Gil 2001) and the Structure Mapping Engine (Falkenheiner, Forbus, & Gentner 1989), have facilities with user interfaces directly available to the user. Shaken integrates these components not only at the server, but also at their user interfaces to be presented as part of Shaken's interface.

In a component's user interface we have the usual distinction between content and presentation. In addition,

we have the distinction between Shaken and component. We thus have four possible combinations: Shaken content, Shaken presentation, component content, and component presentation.

Presenting the user with an entirely different interface for a component would probably be confusing. Shaken's integration API must therefore extend from content generation, through the presentation of that content, to equipping the interface with appropriate interactivity. Shaken provides presentation for not only the header and footer of each page, but also for the various entities in Shaken's knowledge base (KB) that the component might access, such as classes, instances, and assertions. The API must allow the component to maintain these elements in a straightforward manner.

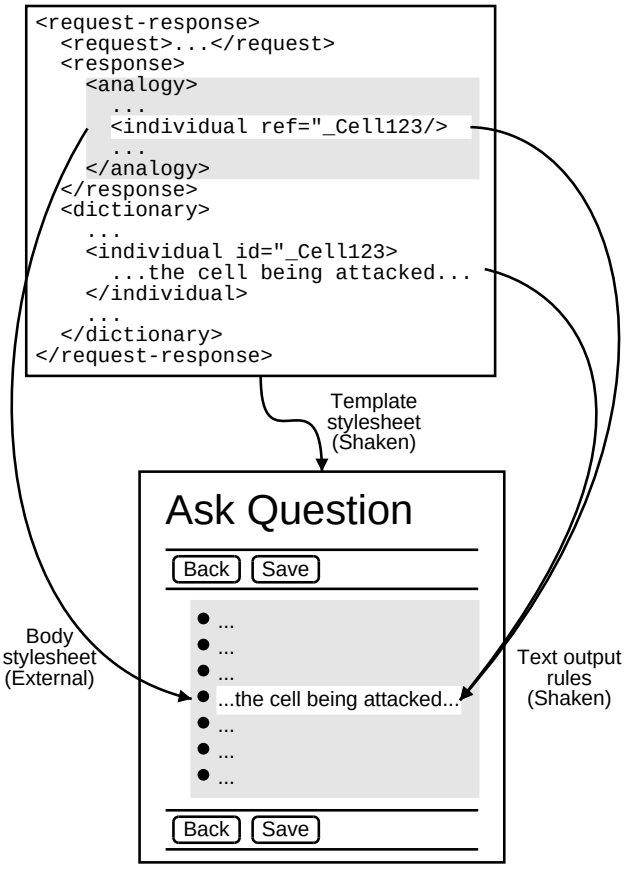

Figure 4: The structure of an integrated XML document, and the stylesheets and stylesheet rules that act on it to produce the HTML interface. The gray box represents the XML generated by the integrated component, and the HTML into which the XML is transformed. The XML for KB entities, and their transformation to XML, is handled by routines provided by Shaken, using the data captured in the dictionary.

Shaken's integration API is built around XML and XSLT (Clark 1999). XML allows the component to express what must be shown to the user, and XSLT provides rules for syntactically transforming the XML to HTML. XML and XSLT thus give us a natural means of separating content and presentation.

This integration has three phases: the generation of the XML, the transformation of the XML to HTML using XSLT, and the interaction of the user with the resulting HTML interface. The first two phases are depicted in Figure 4. Shaken provides an API to help the components through each of these phases. This API includes a set of Lisp functions and macros that a component can use to ensure the XML output conforms to the standard format that Shaken requires. The elements of this standard format are transformed using a template XSLT stylesheet at the WWW browser. The template XSLT stylesheet references a component XSLT stylesheet that transforms the response body of the generated XML into HTML. Finally, Shaken provides JavaScript functions that can be used in this HTML. These functions can send requests to the Shaken server for performing a number of standard operations. We describe each of these three phases in more detail below.

Generating XML

The XML we generate has four elements: `<request>`, `<response>`, `<dictionary>`, and an optional `<error>`. The `<request>`, `<dictionary>` and `<error>` elements are Shaken's responsibility, while `<response>` contains the main body for which the component's interface must be generated.

The `<request>` element contains a description of the input parameter in the request, for which the XML document is being generated as a response. These parameters are often required in producing the desired interface. More important, they are required in carrying state from one step to the next, as has been described above.

The `<response>` and `<dictionary>` elements are intimately linked. Shaken, as we recall, is an application designed to allow subject matter experts, rather than knowledge engineers, to enter knowledge into a KB. The entire contents of the KB cannot possibly be provided for use at the client end for transforming the XML to an HTML interface. Instead, Shaken records precisely what data is referenced from the KB when the XML response is generated. This subset of the KB is stored in `<dictionary>`, along with associated information, such as what text must be generated for each dictionary entry. Thus, we ensure that just enough content is available at the client end to be able to produce a complete presentation.

Finally, the optional `<error>` contains an error message suitable for display to the end user.

Transforming XML through stylesheets

Each HTML page that Shaken produces has a standard presentation style. The header is consists of a title, a standard toolbar, and an optional error message. Shaken similarly also specifies the footer content,

and a set of JavaScript libraries for various interactive elements on the page.

Shaken ensures that each component interface also follows this standard presentation. Shaken provides a template stylesheet for each component. It produces HTML for everything but the `<response>` element's content. This template stylesheet outputs the header and the footer, and the optional error message. It also produces references to the standard JavaScript libraries. Thus, the template stylesheet produces an environment identical to the one that Shaken uses for its own interfaces for the component's interface.

The transformation of the `<response>` is the responsibility of the component. The template stylesheet references the component stylesheet for this purpose. Thus, we effectively separate the transformations for which Shaken is responsible from those that the component must handle.

The component stylesheet in turn relies on Shaken to provide transformation rules for the KB entities. Shaken specifies how these KB objects must be rendered in the interface. The English descriptions for the KB objects are derived using the dictionary content, with XSLT rules provided by Shaken. We thus get a standard presentation of the KB's content across the entire Shaken interface, along with standard actions associated with this presentation.

Communicating structured data

Although much of the interaction with the server can be managed through HTML pages and JavaScript, some operations require a custom solution. Both the Shaken and LAW projects require the ability to edit graphs. Writing a graph editor in HTML would be quite difficult for the programmer, and awkward for the end user.

User interactions that require the user to provide or modify unstructured data can be quite easily implemented through ordinary HTML pages. For example, consider the facility for searching a KB. A KB constitutes structured data, and a server can easily generate a static representation of the KB. Such a static picture can also be made interactive so long as the user's interactions with it are fairly simple, for example, selecting a class in the KB. The user selecting a class in the KB can be represented by the name of the class, so this operation is still unstructured. The need for structure becomes more apparent as the user's operations become more complex. Adding a class to the KB requires a set of existing classes to act as the parents, and a new class name. In addition, as the operations become more structured, there is an increasing need for a more sophisticated for the user to enter data about the operations.

We focus our attention on Shaken's graph editor, which is considerably more mature than LAW's. Shaken's graph editor is written in Java. It communicates with the server through HTTP. Initial versions of the graph editor implemented fairly simple operations, such as connecting two nodes or adding a new node to the graph. Such operations can be represented using a flat parameter list quite easily. We have now implemented operations that are much more difficult to represent using parameter lists, such as constraints on the connections of a node in the graph. We must therefore implement a mechanism for exchanging structured data between the graph editor and the Shaken server.

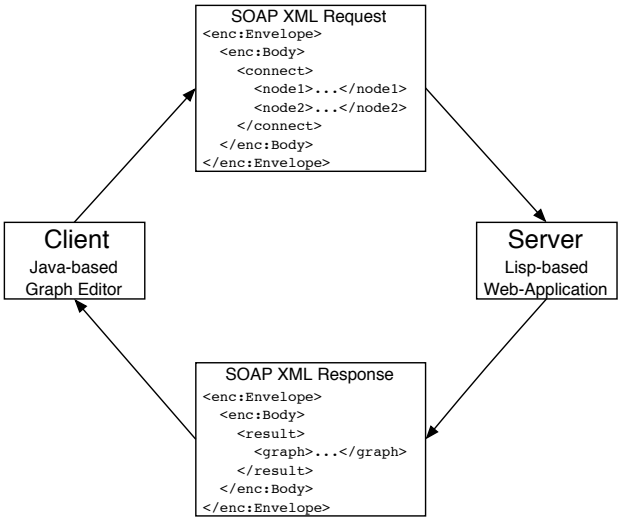

Figure 5: The use of SOAP messaging in Shaken's graph editor.

We have employed SOAP as the mechanism for exchanging structured data in Shaken. SOAP, or Simple Object Access Protocol (Box *et al.* 2000), is a simple RPC mechanism encoded that transmits requests and responses encoded in XML through HTTP. It has been developed with the intention of implementing application servers over the HTTP protocol, an application server being a server that handles requests for computations from applications rather than from users. Using SOAP allows us to continue working with a single communication protocol: HTTP. The XML messages that are exchanged are open to inspection, making the application easier to debug. In addition, we can leverage our existing knowledge of XML in writing the software we need. Therefore, SOAP can be incorporated into Shaken with relatively little effort.

Figure 5 shows how we use SOAP with Shaken's graph editor. When the user performs a graph editing operation in the applet, a SOAP message encoded in XML is sent to the server. The server hands this message over to the SOAP processor. The function name and arguments are deserialized, and the requested Shaken operation is invoked. The response is then sent back to the client, again serialized as a SOAP message. The graph editor finally deserializes the message and displays an appropriate response to the user.

Unlike Java, Lisp does not have any freely available SOAP processors. We have developed one for Shaken based on a pattern matcher. Each SOAP request con-

tains a function to call, and the arguments to be supplied to that function. Each such argument is potentially a structured object. Shaken registers a set of types and functions with the SOAP processor. When deserializing a SOAP request, the processor uses this information for instantiating the required Lisp objects, and invoking the Lisp function. The response returned is again serialized into an XML message using the same information.

The SOAP processor is designed following the ideas of lightweight tools. The implementation is through a set of pattern matching rules that operate on the LXML [2] representation of the SOAP XML request. The initial implementation handles just enough of the SOAP specification to operate on the messages that the graph editor sends. These rules can be extended without perturbing the existing functionality, thus providing a simple extension mechanism. This mechanism can also be employed by an application using the SOAP processor for deserializing its objects, but this route is likely to be more difficult to use than the API described above.

Limitations

We have presented four tools that help us in building Web-based applications. First we presented Active Lisp Pages (ALP), which we use for writing scripts that produce computed HTML pages. It is far too easy to misuse ALP and produce pages that are mostly Lisp code and relatively little HTML. The simplicity of ALP's implementation is also sometimes a source of problems. A syntactic error in a page can easily lead to a faulty Lisp function, which can be difficult to track down. Maintaining a policy of minimizing Lisp code is very helpful in avoiding this problem.

In our interaction manager, we have deliberately paid relatively little attention to reuse. Consider the next step selection that happens during the post-phase of a dialog step. This information is directly encoded into the post-step function itself. A more flexible implementation would adopt a declarative syntax for assembling dialogs from individual steps, perhaps even separating the presentation of each step from the computations in the pre-step phase and the post-step phase. Such an implementation would, however, lead to greater complexity with uncertain benefits for Shaken.

The interaction manager does not enforce the tasks that are performed at the pre-step or the post-step phase. It is left up to the programmer to follow the prescribed development policy. This has not posed any difficulty, as a new programmer always has many other example steps to use as models for development.

Our choice of SOAP for integrating custom interfaces over HTTP has two important consequences. While the choice of an open standard for communication brings the possibility of employing a variety of tools to help with application development, there is a loss in the simplicity of the implementation. Even in Shaken's limited SOAP processor, we must conform to an externally defined standard, ensuring that each possible situation is correctly addressed. This makes the development process considerably more difficult.

In addition, SOAP is designed to handle only the most basic data types, and the simplest function calls. The data types defined in the SOAP specification are only a small set of the possible data types in Lisp. In particular, representing lists and association lists in the arguments to the Lisp functions was particularly difficult.

Conclusions

We have presented four tools that aid in developing interactive Web-based applications:

1. Active Lisp Pages help produce HTML pages.
2. An interaction manager to aids in tracking state as a dialog evolves.
3. XML and XSLT are used for integrating component interfaces into Shaken.
4. SOAP is used for communicating structured data for custom interfaces.

The implementation of these tools would not have been much more complex without the expressiveness of Common Lisp at our disposal. ALP can compile and load a script on the fly into a running application. It can also provide an effective abstraction over CL-HTTP and AllegroServe for many of the common tasks required for a Web-based application. The interaction manager relies heavily on Lisp's ability for introspection and function application. Its utility would have been greatly reduced had the programmer been required to supply argument information explicitly. Integrating component interfaces through XML and XSLT relies on macros for providing many of the necessary abstractions. Finally, the incremental development of the SOAP processor has been possible due to a pattern matcher that can be extended as additional capabilities are required.

A possible metric for measuring the utility of these tools is by examining how applications using them perform. Both Shaken and SEAS have seen considerable use. Shaken presently has 78 registered users. The Shaken application is itself considered as a basis for other projects. SEAS has seen some deployment within DARPA, and shall soon see much wider distribution. As time progresses, we expect to build and utilize more such tools for application development.

Acknowledgments

The work presented in this paper has been supported by DARPA's Rapid Knowledge Formation Project, GENOA Project and Evidence Extraction and Link

[2] LXML is an s-expression representation of XML, thus convenient for use in Lisp applications. It is the output format of Franz Inc.'s XML parser, pxml. For more information, see `http://opensource.franz.com/xmlutils/`.

Discovery Project. We would like to thank our collaborators at NWU, ISI and University of Texas at Austin for their inputs and contributions to building Shaken. We also draw upon the efforts of many others at the AIC at SRI who have impacted these and other projects.

References

Berners-Lee, T.; Fielding, R.; and Masinter, L. 1998. Uniform Resource Identifiers (URI): Generic Syntax. http://www.ietf.org/rfc/rfc2396.txt.

Box, D.; Ehnebuske, D.; Kakivaya, G.; Layman, A.; Mendelsohn, N.; Nielsen, H. F.; Thatte, S.; and Winer, D. 2000. Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

Bray, T.; Paoli, J.; Sperberg-McQueen, C. M.; and Maler, E. 2000. Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/TR/2000/REC-xml-20001006.

Clark, P., and Porter, B. 1998. KM – the knowledge machine: Users manual. Technical report, University of Texas at Austin.

Clark, J. 1999. XSL Transformations (XSLT): Version 1.0. http://www.w3.org/TR/1999/REC-xslt-19991116.

Falkenheiner, B.; Forbus, K.; and Gentner, D. 1989. The Structure Mapping Engine: Algorithm and Examples. *Artificial Intelligence*.

Fielding, R.; Gettys, J.; Mogul, J.; Frystyk, H.; Masinter, L.; Leach, P.; and Berners-Lee, T. 1999. Hypertext Transfer Protocol – HTTP/1.1. ftp://ftp.isi.edu/in-notes/rfc2616.txt.

Foderaro, J., and Dancy, A. 2000. AllegroServe. http://allegroserve.sourceforge.net/.

Kim, J., and Gil, Y. 2001. Knowledge Analysis on Process Models. In *International Joint Conference on Artificial Intelligence*.

Lowrance, J. D.; Harrison, I. W.; and Rodriguez, A. C. 2000. Structured argumentation for analysis. In *Proceedings of the 12th International Conference on Systems Research, Informatics, and Cybernetics: Focus Symposia on Advances in Computer-Based and Web-Based Collaborative Systems*, 47–57. Baden-Baden, Germany: International Institute for Advanced Studies in Systems Research and Cybernetics and Society for App.

Mallery, J. C. 1994. A Common LISP Hypermedia Server. In *Proceedings of The First International Conference on The World-Wide Web*. Geneva: CERN.

Paley, S. M., and Karp, P. D. 1995. Adapting clim applications for use on the world wide web. In *Proceedings of the Association of Lisp Users Meeting and Workshop*, 1–9.

Raggett, D.; Le Hors, A.; and Jacobs, I. 1999. HTML 4.01 Specification. http://www.w3.org/TR/1999/REC-html401-19991224/.

Rodriguez, A. 2002. Active Lisp Pages. In *Proceedings of the International Lisp Conference*.

Thomere, J.; Rodriguez, A.; Chaudhri, V.; Mishra, S.; Eriksen, M.; Clark, P.; Barker, K.; and Porter, B. 2002. A web-based ontology browsing and editing system. In *Conference on Innovative Applications of Artificial Intelligence*. AAAI.