The `<bigwig>` Project

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach
BRICS, Department of Computer Science
University of Aarhus, Denmark
{brabrand,amoeller,mis}@brics.dk

Abstract

We present the results of the `<bigwig>` project, which aims to design and implement a high-level domain-specific language for programming interactive Web services.

A fundamental aspect of the development of the World Wide Web during the last decade is the gradual change from static to dynamic generation of Web pages. Generating Web pages dynamically in dialog with the client has the advantage of providing up-to-date and tailor-made information. The development of systems for constructing such dynamic Web services has emerged as a whole new research area.

The `<bigwig>` language is designed by analyzing its application domain and identifying fundamental aspects of Web services inspired by problems and solutions in existing Web service development languages. The core of the design consists of a session-centered service model together with a flexible template-based mechanism for dynamic Web page construction. Using specialized program analyses, certain Web-specific properties are verified at compile time, for instance that only valid HTML 4.01 is ever shown to the clients. In addition, the design provides high-level solutions to form field validation, caching of dynamic pages, and temporal-logic based concurrency control, and it proposes syntax macros for making highly domain-specific languages.

The language is implemented via widely available Web technologies, such as Apache on the server-side and JavaScript and Java Applets on the client-side. We conclude with experience and evaluation of the project.

1 Introduction

The `<bigwig>` project was founded in 1998 at the BRICS Research Center at the University of Aarhus to design and implement a high-level domain-specific language for programming interactive Web services. Such services are characterized by involving multiple interactions with each client, mediated by HTML forms in browsers. In the following we argue that existing Web service programming languages in various ways provide only low-level solutions to problems specific to the domain of Web services. Our overall ambitions for the project are to identify the key areas of the Web service domain, analyze the problems with the existing approaches, and provide high-level solutions that will support development of complex services.

Motivation

Specifically, we will look at the following Web service technologies: the HTTP/ CGI Web protocol [18], Sun's Java Servlets [27] and their JavaServer Pages (JSP) [28], Microsoft's Active Server Pages (ASP) [19], the related Open Source language PHP [4], and the research language MAWL [3, 2, 21].

CGI was the first platform for development of Web services, based on the simple idea of letting a script generate the reply to incoming HTTP requests dynamically on the server, rather than returning a static HTML page from a file. Typically, the script is written in the general-purpose scripting language Perl, but any language supported by the server can be used. Being based on general-purpose programming languages, there is no special support for Web specific tasks, such as generation of HTML pages, and knowledge of the low-level details of the HTTP protocol are required. Also, HTTP/CGI is a stateless protocol that by itself provides no help in tracking and guiding users through a series of individual interactions. This can to some degree be alleviated by libraries. In any case, there are no compile-time guarantees of correct runtime behavior when it comes to Web-specific properties, for instance ensuring that invalid HTML is never sent to the clients.

Servlets are a popular higher-level Java-specific approach. Servlets, which are special Java programs, offer the common Java advantages of network support, strong security guarantees, and concurrency control. However, some significant problems still exist. Services programmed with servlets consist of collections of request handlers for individual interactions. Sessions consisting of several interactions with the same client must be carefully encoded with cookies, URL rewriting, or hidden input fields, which is tedious and error-prone even with library support, and it becomes hard to maintain an overview of large services with complex interaction flows. A second, although smaller, problem is that state shared between multiple client sessions, even for simple services, must be explicitly stored in a name–value map called the "servlet context", instead of using Java's standard variable declaration scoping mechanism. Thirdly, the dynamic construction of Web pages is not improved compared to CGI. Web pages are built by printing string fragments to an output stream. There is no guarantee that the result will always become valid HTML. This situation is slightly improved by using HTML constructor libraries, but they preclude the possibility of dividing the work of the programmers and the HTML designers. Furthermore, since client sessions are split into individual interactions that are only combined implicitly, for instance by storing session IDs in cookies, it is not possible to statically analyze that a given page sent to a client always contains exactly the input fields that the next servlet in the session expects.

JSP, ASP, PHP, and the countless homegrown variants were designed from a different starting point. Instead of aiming for complex services where all parts of the pages are dynamically generated, they fit into the niche where pages have mostly static contents and only small fragments are dynamically generated. A service written in one of these languages typically consists of a collection of "server pages" which are HTML pages with program code embedded in special tags. When such a page is requested by the client, the code is evaluated and replaced by the resulting string. This gives better control over the HTML construction, but it only gives an advantage for simple services where most of every page is static.

The MAWL language was designed especially for the domain of interactive Web services. One innovation of MAWL is to make client sessions explicit in the program logic. Another is the idea of building HTML pages from templates. A MAWL service contains a number of sessions, shared data, and HTML templates. Sessions serve as entry points of client-initiated session threads. Rather than producing a single HTML page and then terminating as CGI scripts or Servlets, each session thread may involve multiple client interactions while maintaining data that is local to that thread. An HTML template in MAWL is an HTML document containing named gaps where either text strings or special lists may be inserted. Each client interaction is performed by inserting appropriate data into the gaps in an HTML template and then sending it to the client, who fills in form fields and submits the reply back to the server.

The notions of sessions and document templates are inherent in the language and, being compilation-based, allow important properties to be verified statically, without actually running the service. Since HTML documents are always constructed from the templates, HTML validity can be verified statically. Also, since it is clear from the service code where execution resumes when a client submits form input, it can be statically checked that the input fields match what the program expects. One practical limitation of the MAWL approach is that the HTML template mechanism is quite restrictive, as we cannot insert markup into the template gaps.

We describe more details about the existing languages in the following sections. By studying services written in any of these languages, some other common problems show up. First of all, often surprisingly large portions of the service code tend to deal with form input validation. Client-server interaction takes place mainly through input forms, and usually some fields must be filled with a certain kind of data, perhaps depending on what has been entered in other fields. If invalid data is submitted, an appropriate error message must be returned so that the client can try again. All this can be handled either on the client-side—typically with JavaScript [16], in the server code or with a combination. In any case, it is tedious to encode.

Second, one drawback of dynamically generated Web pages compared to static ones is that traditional caching techniques do not work well. Browser caches and proxy servers can cause major improvements in saving network bandwidth, load time, and clock cycles, but when moving towards interactive Web services, these benefits disappear.

Third, most Web services act as interfaces to underlying databases that, for instance, contain information about customers, products, and orders. Accessing databases from general-purpose programming languages where database queries are not integrated requires the queries to be built as text strings that are sent to a database engine. This means that there is no static type checking of the queries. As known from modern programming languages, type systems allow many programming bugs to be caught at compile time rather than at runtime, and thereby improve reliability and reduce development cost.

Fourth, since running Web services contain many concurrently executing threads and they access shared information, for instance in databases on the server, there is a fundamental need for concurrency control. Threads may require exclusive access to critical regions, be blocked until certain events occur, or be required to satisfy more high-level behavioral constraints. All this while the service should run smoothly without deadlocks and other abrupt obstacles. Existing solutions typically provide no or only little support for this, for instance via low-level semaphores as Perl or synchronized methods in Servlets. This can make it difficult to guarantee correct concurrent execution of entire services.

Finally, since Web services usually operate on the Internet rather than on secure local networks, it is important to protect sensitive information both from hostile attacks and from programming leaks. A big step forward is the Secure Sockets Layer (SSL) protocol [17] combined with HTTP Authentication [5]. These techniques can ensure communication authenticity and confidentiality, but using them properly requires insight into technical protocol and implementation details. Furthermore, they do not protect against programming bugs that unintentionally leak secret information. The "taint mode" in Perl offers some solution to this. However, it is runtime based so no compile-time guarantees are given. Also, it only checks for certain predefined properties, and more specialized properties cannot be added.

The `<bigwig>` Language

Motivated by the languages and problems described above, we have identified the following areas as key aspects of Web service development:

- *sessions*: the underlying paradigm of interactive Web services;
- *dynamic documents*: HTML pages must be constructed in a flexible, efficient, and safe fashion;
- *concurrency control*: Web services consist of collections of processes running concurrently and sharing resources;
- *form field validation*: validating user input requires too much attention from Web programmers so a higher-level solution is desirable;
- *database integration*: the core of a Web service is often a database with a number of sessions providing Web access; and
- *security*: to ensure authenticity and confidentiality, regarding both malicious clients and programming bugs.

To attack the problems, we have designed from scratch a new language called `<bigwig>`, as a descendant of the MAWL language. This language is a high-level, domain-specific language [30], meaning that it employs special syntax and constructs that are tailored to fit its particular application domain and allow specialized program analyses, in contrast to library-based solutions. Its core is a C or Java-like skeleton, which is surrounded by domain-specific sub-languages covering the above key aspects. A notion of *syntax macros* tie the sub-languages together and provide additional layers of abstraction. This macro language, which operates on the parse tree level, rather than the token sequence level as conventional macro languages, has proved successful in providing extensions of the core language. This has helped each of the sub-languages remain minimal, since desired syntactic sugar is given by the macros. Syntax macros can be taken to the extreme, where they, with little effort, can define a completely new syntax for *very*-domain-specific languages tailored to highly specialized application domains.

It is important that `<bigwig>` is based on compilation rather than on interpretation of a scripting language. Unlike many other approaches, we can then apply type systems and static analysis to catch many classes of errors before the service is actually installed.

The `<bigwig>` compiler uses common Web technologies as target languages. This includes HTML [24], HTTP [5], JavaScript [16], and Java Applets [1]. Our current implementation additionally relies on the Apache Web server. It is important to apply only standard technologies on the client-side in order not to place restrictions on the clients. In particular, we do not use browser plug-ins, and we only use the subset of JavaScript that works on all common browsers. As new technologies become standard, the compiler will merely obtain corresponding opportunities for generating better code. To the degree possible, we attempt to hide the low-level technical details of the underlying technologies.

We have made no effort to contribute to the graphical design of Web services. Rather, we provide a clean separation between the physical layout of the HTML pages and the logical structure of the service semantics. Thus, we expect that standard HTML authoring tools are used, conceivably by others than the Web programmer. Also, we do not focus on efficiency, but on providing higher levels of abstraction for the developers. For now, we regard it as less important to generate solutions that seamlessly scale to thousands of interactions per second, although, of course, scalability is an issue for the design.

The main contributions of the `<bigwig>` project are the following results:

- The notion of client sessions can and should be made explicit in Web service programming languages;

- dynamic construction of Web pages can at the same time be made flexible and fast, while still permitting powerful compile-time analyses;

- form field validation can be made easier with a domain-specific language based on regular expressions and boolean logic;

- temporal logic is a useful formalisms for expressing concurrency constraints and synthesizing safety controllers; and
- syntax macros can be used to create very-domain-specific high-level languages for extremely narrow application domains.

We focus on these key contributions in the remainder of this article, but also describe less central contributions, such as a technique for performing client-side caching of dynamically generated pages, a built-in relational database, and simple security mechanisms. The individual results have been published in previous more specialized articles [25, 26, 8, 7, 9, 6, 10]. Together, these results show that there is a need for high-level programming languages that are tailor-made to the domain of Web service development.

Overview

We begin in Section 2 by classifying the existing Web service languages as script-, page-, or session-centered, arguing for the latter as the best choice for complex services. In Section 3, we show how the HTML template mechanism from MAWL can be extended to become more flexible using a notion of higher-order templates. Using novel type systems and static analyses, the safety benefits of MAWL templates remain in spite of the increased expressibility. Also, we show how our solution can be used to cache considerable parts of the dynamically generated pages in the browser. In Section 4, we address the problem of validating form input more easily. Section 5 describes a technique for generating concurrency controllers from temporal logic specifications. Section 6 gives an introduction to the syntax macro mechanism that ties together the sub-languages of `<bigwig>`. In Section 7, we mention various less central aspects of the `<bigwig>` language. Finally, in Section 8 we describe our implementation and a number of applications, and evaluate various practical aspects of `<bigwig>`.

2 Session-Centered Web Services

Web programming covers a wide spectrum of activities, from composing static HTML documents to implementing autonomous agents that roam the Web. We focus in our work on *interactive Web services*, which are Web servers where clients can initiate sessions that involve several exchanges of information mediated by HTML forms. This definition includes large classes of well-known services, such as news services, search engines, software repositories, and bulletin boards, but also covers services with more complex and specialized behavior.

There are a variety of techniques for implementing interactive Web services, but they can be divided into three main paradigms: the *script-centered*, the *page-centered*, and the *session-centered*. Each is supported by various tools and suggests a particular set of concepts inherent in Web services.

The Script-Centered Approach

The script-centered approach builds directly on top of the plain, stateless HTTP/CGI protocol. A Web service is defined by a collection of loosely related scripts. A script is executed upon request from a client, receiving form data as input and producing HTML as output before terminating. Individual requests are tied together by explicitly inserting appropriate links to other scripts in the reply pages.

Perl is a prototypical scripting language, but almost any programming language has been suggested for this role. CGI scripting is often supported by a large collection of library functions for decoding form data, validating input, accessing databases, and realizing semaphores. Even though such libraries are targeted at the domain of Web services, the language itself is not. A major problem is that the overall behavior is distributed over numerous individual scripts and depends on the implicit manner in which they pass control to each other. This design complicates maintenance and precludes any sort of automated global analysis, leaving all errors to be detected in the running service [15, 2].

HTML documents are created on the fly by the scripts, typically using `print`-like statements. This again means that no static guarantees can be issued about their correctness. Furthermore, the control and presentation of a service are mixed together in the script code, and it is difficult to factor out the work of programmers and HTML designers [12].

The Java Servlets language also fits this category. The overall structure of a service written with servlets is the same as for Perl. Every possible interaction is essentially defined by a separate script, and one must use cookies, hidden input fields, or similar techniques to connect sequences of interactions with the clients. Servlets provide a session tracking API that hides many of the details of cookies, hidden input fields, and URL rewriting. Many servlet servers use cookies if the browser supports them, but automatically revert to URL rewriting when cookies are unsupported or explicitly disabled. This API is exemplified by the following code inspired by two Servlet tutorials:[1]

```
public class SessionServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    ServletContext context =  getServletContext();
    HttpSession session = request.getSession(true);
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML><HEAD><TITLE>Servlet Demo</TITLE></HEAD><BODY>");
    if (session.isNew()) {
      out.println("<FORM ACTION=SessionServlet>" +
                  "Enter your name: <INPUT NAME=handle>" +
                  "<INPUT TYPE=SUBMIT></FORM>");
      session.putValue("state", "1");
    } else {
```

[1] `http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/` and `http://java.sun.com/docs/books/tutorial/servlets/`

```
        String state = (String) session.getValue("state");
        if (state.equals("1")) {
          String name = (String) request.getParameter("handle");
          int users =
            ((Integer) context.getAttribute("users")).intValue() + 1;
          context.setAttribute("users", new Integer(users));
          session.putValue("name", name);
          out.println("Hello " + name + ", you are user number " + users);
          session.putValue("state", "2");
        } else /* state.equals("2") */ {
          String name = (String) session.getValue("name");
          out.println("Goodbye " + name);
          session.invalidate();
        }
      }
      out.println("</BODY></HTML>");
    }
  }
```

Clients running this service are guided through a series of interactions: first, the service prompts for the client's name, then the name and the total number of invocations is shown, and finally a "goodbye" page is shown. The `ServletContext` object contains information shared among all sessions, while the `HttpSession` object is local to each session. The code is essentially a `switch` statement that branches according to the current interaction. An alternative approach is to make a servlet for each kind of interaction. In spite of the API, we still need to explicitly maintain both the state and the identity of the session.

The model of sessions that is supported by Servlets and other script-centered approaches tends to fit better with "shopping basket applications" where the client browses freely among dynamically generated pages than with complex services that need to impose more strict control on the interactions.

The Page-Centered Approach

The page-centered approach is covered by languages such as ASP, PHP, and JSP, where the dynamic code is embedded in the HTML pages. In a sense, this is the inverse of the script-centered languages where HTML fragments are embedded in the program code. When a client requests a page, a specialized Web server interprets the embedded code, which typically produces additional HTML snippets while accessing a shared database. In the case of JSP, implementations work by compiling each JSP page into a servlet using a simple transformation.

This approach is often beautifully motivated by simple examples, where pages are mainly static and only sporadically contain computed contents. For example, a page that displays the time of day or the number of accesses clearly fits this mold. The following JSP page dynamically inserts the current time together with a title and a user name based on the CGI input parameters:

```
<HTML><HEAD><TITLE>JSP Demo</TITLE></HEAD><BODY>
Hello <%
  String name = request.getParameter("who");
  if (name==null) name = "stranger";
  out.print(name);
%>!
<P>
This page was last updated: <%= new Date() %>
</BODY></HTML>
```

The special `<%...%>` tags contain Java code that is evaluated at the time of the request. As long as the code parts only generate strings without markup, it is easy to statically guarantee that all pages shown are valid HTML and other relevant properties. But as the services become more complex, the page-centered approach tends to converge towards the script-centered one. Instead of a mainly static HTML page with some code inserted, the typical picture is a single large code tag that dynamically computes the entire contents. Thus, the two approaches are closely related, and the page-centered technologies are superior only to the degree in which their scripting languages are better designed.

The ASP and PHP languages are very reminiscent of JSP. ASP is closely tied to Microsoft's Internet Information Server, although other implementations exist. Instead of being based on Java, it defines a language-independent connection between HTML pages and scripting languages, typically either Visual Basic Script or Microsoft's version of JavaScript. PHP is a popular Open Source variant whose scripting language is a mixture of C, Java, and Perl.

These languages generally provide only low-level support for tracking client sessions and maintaining session state. Cookies, hidden input fields, and some library support is the common solution. For other Web service aspects also, such as databases and security, there is often a wide range of libraries available but no direct language support.

The Session-Centered Approach

The pure session-centered approach was pioneered by the MAWL project. Here a service is viewed as a collection of distinct *sessions* that access some shared data. A client may initiate a session *thread*, which is conceptually a process running on the server. Interaction with the client is viewed as remote procedure calls from the server, as known from classical construction of distributed systems but with the roles reversed.

The flow of an entire session is programmed as a single sequential program, which is closer to ordinary programming practice and offers the compiler a chance to obtain a global view of the service. Figure 1 illustrates the flow of control in this approach. Important issues such as concurrency control become simpler to understand in this context and standard programming solutions are more likely to be applicable.

The following MAWL program is equivalent to the previous Servlet example:

```
static int users = 0;

session GreetingSession {
  auto form {} -> {handle} hello;
  auto string name = hello.put().handle;
```

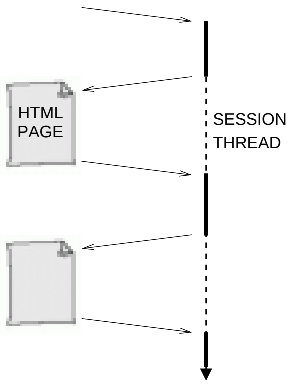

Figure 1: Client-server sessions in Web services. On the left is the client's browser, on the right a session thread running on the server. The tread is initiated by a client request and controls the sequence of interactions.

```
   auto form {string who , int count} -> {} greeting;
   users++;
   greeting.put({name, users});

   auto form {string who} -> {} goodbye;
   goodbye.put({name});
}
```

The HTML templates *hello*, *greeting*, and *goodbye* are placed in separate files. Here is `hello.mhtml`:

```
<HTML><HEAD><TITLE>MAWL Demo</TITLE></HEAD><BODY>
Enter your name: <INPUT NAME=handle>
</BODY></HTML>
```

and `greeting.mhtml`:

```
<HTML><HEAD><TITLE>MAWL Demo</TITLE></HEAD><BODY>
Hello <MVAR NAME=who>, you are user number <MVAR NAME=count>
</BODY></HTML>
```

The template for *goodbye* is similar. A form tag and a continue button are implicitly inserted. Variables declared `static` contain persistent data, while those declared `auto` contain per-session data, also called *local* data. The `form` variables are declared with two record types. The former defines the set of gaps that occur in the template, and the latter defines the input fields. In the templates, gaps are written with `MVAR` tags. Template variables all have a `put` method. When this is executed, the arguments are inserted in the gaps, the resulting page is sent to the client who fills in the fields and submits the reply, which is turned into a record value in the program. Note how the notion of sessions is explicit in the program, that private and shared state is simply a matter of variable declaration modifiers, and that the templates are cleanly separated from the service logic. Obviously, the session flow is clearer, both to the programmer and to the compiler, than with the non-session based approaches. One concrete benefit is that it is easy to statically check both validity and correct use of input fields.

The main force of the session-centered approach is for services where the control flow is complex. Many simple Web services are in actuality more loosely structured. If all sessions are tiny and simply do the work of a server module from the page-centered approach, then the overhead associated with sessions may seem too large. Script-centered services can be seen as a subset of the session-centered where every session contains only one client interaction. Clearly, the restriction in the script-centered and the page-centered languages allows significant performance improvements. For instance, J2EE Servlet/JSP servers employ pools of short-lived threads that store only little local state. For more involved services, however, the session-centered approach makes programming easier, since session management comes for free.

Structure of `<bigwig>` Services

The overall structure of `<bigwig>` programs is directly inspired by MAWL. A `<bigwig>` program contains a complete specification of a Web *service*. A service contains a collection of named *sessions*, each of which essentially is an ordinary sequential program. A client has the initiative to invoke a thread of a given session, which is a process on the server that executes the corresponding sequential code and exclusively communicates with the originating client. Communication is performed by *showing* the client an HTML page, which implicitly is made into a form with an appropriate URL return address. While the client views the given document, the session thread is suspended on the server. Eventually the client submits the form, which causes the session thread to be resumed and any form data entered by the client to be *received* into program variables. A simple `<bigwig>` service that communicates with a client, as in the Servlet and MAWL examples, is the following:

```
service {
  html hello = <html>Enter your name: <input name=handle></html>;

  html greeting =
    <html>Hello <[who]>, you are user number <[count]></html>;

  html goodbye = <html>Goodbye <[who]></html>;

  shared int users = 0;

  session Hello() {
    string name;
    show hello receive[name=handle];
    users++;
    show greeting<[who=name,count=users];
    show goodbye<[who=name];
  }
}
```

The program structure is obviously as in MAWL, except that the session code and the templates are wrapped into a `service` block. For instance, the `show-receive` statements produce the client interactions similarly to the `put` methods in MAWL. However, `<bigwig>` provides a number of new features. Most importantly, HTML templates are now *first-class values*. That is, `html` is a built-in data type, and its values can be passed around and stored in variables like any other data type. Also, the HTML templates are *higher-order*, meaning that instead of only allowing text strings to be inserted into the template gaps, we also allow insertion of other templates. This is done with the special *plug* operator, x`<[`y=z`]` which inserts a string or template z into the y gaps of the x template. Clearly, this constitutes a more flexible document construction mechanism, but it also calls for new ideas for statically verifying HTML validity, for instance. This is the topic of Section 3. Other new features include the techniques for improving form field validation and concurrency control, together with the syntax macro mechanism, all of which are described in the following sections.

A Session-Based Runtime Model

The session-based model can be implemented on top of the CGI protocol. One naive approach is to create session threads as CGI scripts where all local state is stored on disk. At every session interaction, the thread must be started again and restore its local state, including the call stack, in order to continue execution. A better approach is to implement each session thread as a process that runs for the whole duration of the session. For every interaction, a tiny transient CGI script, called a *connector process*, is executed, acting as a pipe between the Web server and the session process. This approach resembles FastCGI [22], and is described in detail in [8]. Our newest implementation is instead based on a specialized Apache server module.[2] Naturally, this is much faster than the CGI solutions since it does not create a new process for every single interaction, but only for the session processes.

Two common sources of problems with standard implementations of sessions are history buffers and bookmarking features found in most browsers. With history buffers and the "back" button, the users can step back to a previous interaction, and either intentionally or unintentionally resubmit an old input form. Sometimes this can be a useful feature, but more often this causes confusion and annoyance to the users who may, for instance, order something twice. It is a general problem that the information shown to the user in this way can be obsolete since it is tailor-made only for the exact time of the initial request. Since the information was generated from a shared database that may have changed entirely, it does generally not make sense to "step back in time" using the history buffer. This is no different from ordinary programs. Even if the programmer was aware of this and added serial number checks, the history buffer will be full of URLs to obsolete requests. If the service really needs a "back" feature, it can be programmed explicitly into the flow of the sessions.

[2]See `http://httpd.apache.org/`.

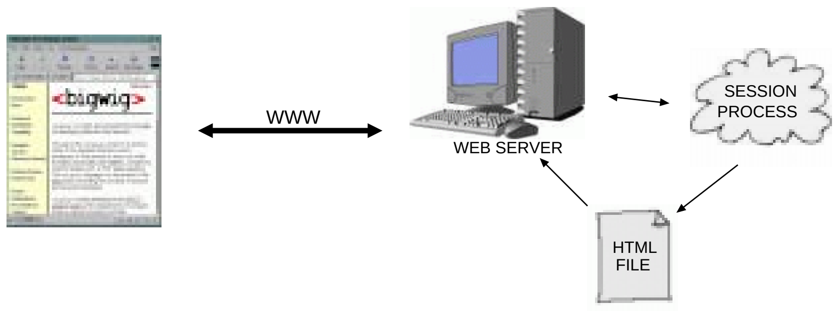

Figure 2: Session-based runtime model with reply indirection. Each session thread is implemented as a separate process that writes its HTML reply to a designated file.

It also becomes hazardous to try to use bookmarks to temporarily suspend a session. Invoking the bookmark will typically cause a CGI script to be executed a second time instead of just displaying its results again.

`<bigwig>` provides a simple but unique solution to these problems: Each session thread is associated with a URL which points to a file on the server containing the latest HTML page shown to the client. Instead of sending the contents directly to the client at every `show` statement, we redirect the browser to this URL, as illustrated in Figure 2. Since the URL serves as the identification of the session thread, this solves the problems mentioned above: The history list of the browser now only contains a single entry for the duration of the session, the sessions can now be bookmarked for later use, and in addition, the session identity URL can be passed around manually—to another browser, for instance—without problems. When using URLs instead of cookies to represent the session identity, it also becomes possible for a single user to simultaneously run multiple sessions in different windows but with the same browser.

Furthermore, with this simple solution we can automatically provide the client with feedback while the server is processing a request. This is done by, after a few seconds, writing a temporary response to the HTML file, which informs the client about the status of the request. This temporary file reloads itself frequently, allowing for updated status reports. When the final response is ready, it simply overwrites the temporary reply file, causing the reloading to stop and the response to be shown. This simple technique may prevent the client from becoming impatient and abandoning the session.

Additionally, the `<bigwig>` runtime system contains a garbage collector process that monitors the service and shuts down session processes abandoned by the clients. By default, this occurs if the client has not responded within 24 hours. The sessions are allowed to execute some clean-up actions before terminating.

3 Dynamic Construction of HTML Pages

In MAWL, all HTML templates are placed in separate files and viewed as procedures of a kind, with the arguments being strings that are plugged into gaps in the template and the results being the values of the form fields that the template contains. This allows a complete separation of the service code and the HTML code. Two benefits are that static guarantees are possible and that the work of programmers and HTML designers can be separated, as previously mentioned. A disadvantage is that the template mechanism becomes too rigid compared to the flexibility of the `print`-like statements available in the script-centered languages. However, those languages permit essentially no static guarantees or work separation. Furthermore, with the script-centered solutions the HTML must often be constructed in a linear fashion from top to bottom, instead of being composed from components in a more logical manner. The `<bigwig>` solution provides the best of the two worlds. Higher-order HTML templates as first-class values are in practice as flexible as `print` statements, and the MAWL benefits are still preserved.

We define *DynDoc* as the sub-language of `<bigwig>` that deals with document construction, that is, the control structures, HTML template constants, variables and assignments, plug operations, and `show-receive` statements. Template constants are delimited by `<html>`...`</html>`. Gaps are written with special `<[`...`]>` tags. Special *attribute gaps* can be used in place of attribute values, as shown in the example below. Of course, only strings should be plugged into such gaps, not templates with markup. The plug operation x`<[`y=z`]` creates a new template by inserting a copy of z in the y gaps of a copy of x. When used in a `show-receive` statement, a template is converted to a complete document by implicitly plugging empty strings into all remaining gaps. Also, it is automatically wrapped into a `form` element whose action is to continue the session, unless the session terminates immediately after. And finally, it is inserted into an outermost template like

```
<html><head><title>service </title></head><body>...</body></html>
```

unless already inside a `body` element. The following example illustrates that documents can be built gradually using higher-order templates:

```
service {
  html brics = <html>
    <head><title>Hi!</title></head>
    <body bgcolor=[color]><[contents]></body>
  </html>;
  html greeting = <html>Hello <[who]>, welcome to <[what]>.</html>;
  session Welcome() {
    html h = brics<[contents=greeting];
    show h<[color="#9966ff",who="Stranger",what="BRICS"];
  }
}
```

The construction process is shown in Figure 3. Note that gaps may be plugged in any order, not necessarily "bottom up". MAWL provides little functionality

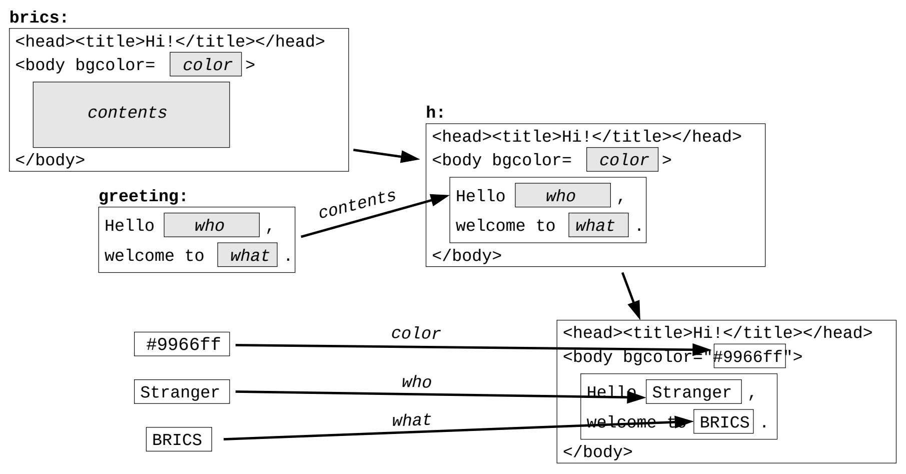

Figure 3: Building a document by plugging into template gaps. The construction starts with the five constants on the left and ends with the complete document on the right.

beyond plugging text strings into gaps. The special `MITER` tag allows list structures to be built iteratively, but still precludes general tree-like structures. The following `<bigwig>` example uses a recursive function to construct an HTML document representing a binary tree:

```
service {
  html list = <html><ul><li><[gap]><li><[gap]></ul></html>;
  html tree(int i) {
    if (i==0) return <html>foo</html>;
    return list<[gap=tree(i-1)];
  }
  session ShowTree() {
    show tree(10);
  }
}
```

Something similar could not be done with MAWL's first-order templates. In a script-centered or a page-centered language it is of course possible, but not with such a simple program structure reflecting the logical composition of the document, since it must be generated linearly by printing to the output stream. An alternative is to use an HTML tree constructor library, but that forces documents to be built bottom-up, which is often inconvenient.

The use of higher-order templates generally leads to programs with a large number of relatively small template constants. For that reason it is convenient to be able to inline the constants in the program code, as in these examples, rather than always placing them in separate files. However, we do offer explicit support for factoring out the work of graphical designers using a `#include` construct as in C. Alternatively, each HTML constant in a `<bigwig>` program may have an associated URL, pointing to an alternate, presumably more elaborate, version:

```
service {
  session Hello {
    show <html>Hello World</html> @ "fancy/hello.html";
  }
}
```

The compiler retrieves the indicated file and uses its contents in place of the constant, provided it exists and contains well-formed HTML. In this manner, the programmer can use plain versions of the templates while a graphical designer simultaneously produces fancy versions. The compiler checks that the two versions have the same gaps and fields. In order to accommodate the use of HTML authoring tools, we permit gaps to be specified in an alternative syntax using special tags.

The DynDoc sub-language was introduced in [26] where it is also shown how this template model can be implemented efficiently with a compact runtime representation. The plug operation takes only constant time, and showing a document takes time linear in the size of the output. Also, the size of the runtime representation of a document may be only a fraction of its printed size. For example, a binary tree of height n shown earlier has a representation of size $O(n)$ rather than $O(2^n)$.

Analysis of Template Construction and Form Input

We wish to devise a type checker that allows as liberal a use of dynamic documents as possible, while guaranteeing that no errors occur. More precisely, we would like to verify the following properties at compile time:

- at every plug operation, x`<[`y=z`]`, there always exists a y gap in x;
- the gap types are compatible with the values being plugged in, in particular, HTML with markup tags is never inserted into attribute gaps;
- for every `show-receive` statement, the fields in the `receive` part always exist in the document being shown;
- the field types are compatible with the `receive` parts, for instance, a select menu allowing multiple items to be selected yields a vector value; and
- only valid HTML 4.01 [24] is ever sent to the clients.

The first four properties are addressed in [26] as summarized below. The last property is covered in the following section.

It is infeasible to explicitly declare the exact types of higher-order templates for two reasons. First, all gaps and all fields and their individual capabilities would have to be described, which may become rather voluminous. Second, this would also imply that an HTML variable has the same type at every program point, which is too restrictive to allow templates to be composed in an intuitive manner. Consequently, we rely instead on a flow analysis to infer the types of template variables and expressions at every program point. In our experience, this results in a liberal and useful mechanism.

We employ a monovariant interprocedural flow analysis, which guarantees that the form fields in a shown document correspond to those that are received, and that gaps are always present when they are being plugged. This analysis fits into standard data-flow frameworks [23], however it applies a highly specialized lattice structure representing the template types. For every template variable and expression that occurs in the given program, we associate a lattice element that abstractly captures the relevant template properties. The lattice consists of two components: a *gap map* and a *field map*. The gap map records for every occurring gap name whether or not the gap occurs at that point, and in case it does occur, whether it is an HTML gap or an attribute gap. Similarly, the field map records for every occurring input field name information about the input fields, which can be of type text, radio, select, or checkbox, representing the different interaction methods.

Given a `<bigwig>` program we construct a flow graph. This is quite easy since there are no higher-order functions or virtual methods. All language constructs that are not included in DynDoc are abstracted away. It is now possible to define transfer functions that abstractly describe the effect of the program statements. This produces a constraint system which we solve using a classical fixed point iteration technique. From this solution, we can see that the first three properties mentioned above are satisfied, and, if not, generate error messages indicating the cause.

With this approach, the programmer is only restricted by the requirement that at every program point the template type of an expression must be fixed. In practice, this does not limit the expressibility, rather, it tends to enforce a more comprehensible structure of the programs. Also, the compiler silently resolves conflicts at flow join points by implicitly plugging superfluous gaps with empty content.

HTML Validity Analysis

The fifth property, HTML validity, is addressed with a similar but more complicated approach, as described in [9].

The main idea is the following: We define a finite structure called a *summary graph* that approximates the set of templates that a given HTML expression may evaluate to. This structure contains the plug operations and the constant templates and strings that are involved.

As an example, consider the summary graph in Figure 4. The nodes correspond to program constants and the edges correspond to plug operations. For instance, the `li` template may be plugged into the *`items`* gaps in the `ul` template here. The node labeled • represents arbitrary text strings and ϵ is the empty string. The root of the graph corresponds to the outermost template. By "unfolding" this graph according to the plug edges, this summary graph defines a possibly infinite set of HTML fragments without gaps (in this case the set of all

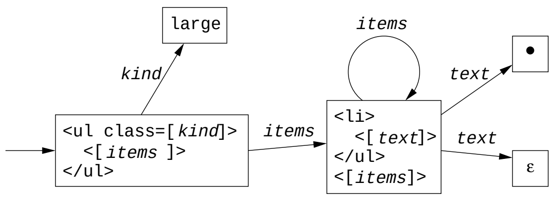

Figure 4: A summary graph representing a set of HTML fragments.

ul lists of class large with one or more character data items). This structure turns out to provide an ideal abstraction level for verifying HTML validity.

Again, we apply a data-flow analysis to approximate the flow of template values in the program. This time we use a lattice consisting of summary graphs. It is possible to model plug operations with good precision using transfer functions; however, two preliminary analyses are required: one for tracking string constants, and one, called a *gap track analysis*, for tracking the origins of gaps. The latter tells us, for each template variable and gap name, which constant templates containing such a gap can flow into that variable at any given program point. Clearly, these analyses are highly specialized for the domain of dynamic document construction and for <bigwig>'s higher-order template mechanism, but they all fit into the standard data-flow analysis frameworks. For more details see [9].

Once we have the summary graphs for all the show statements, we need to verify that the sets of document fragments they define are all valid HTML according to W3C's official definition. To simplify the process we reformulate the notion of Document Type Definition (DTD) as a simpler and more convenient formalism which we call *abstract DTD*. An abstract DTD consists of a number of *element declarations*, whereof one is designated as the root. An element declaration defines the requirements for a particular type of element. Each declaration consists of an element name, a set of names of attributes and subelements that may occur, and a boolean expression constraining the element type instances with respect to their attribute values and contents. The official DTD for HTML is easily rewritten into our abstract DTD notation. In fact, the abstract DTD version captures more validity requirements than those expressible by standard DTDs and merely appear as comments in the HTML DTD. As a technicality, we actually work with XHTML 1.0 which is an XML reformulation of HTML 4.01. There are no conceptual differences, except that the XML version provides a cleaner tree view of documents for the analysis.

Given a summary graph and an abstract DTD description of HTML, validity can be checked by a recursive traversal of the summary graph starting at the roots. We memoize intermediate results to ensure termination, since the summary graphs may contain loops. If no violations are encountered, the summary graph is valid. Since all validity properties are local to single elements and their contents, we are able to produce precise error messages in case of violations. Analysis soundness is ensured by the following property: if all summary graphs corresponding to `show` expressions are verified to be valid with respect to the abstract DTD, then all concrete documents are guaranteed to be valid HTML.

The program analyses described here all have high worst-case complexities due to complex lattices. Nevertheless, our implementations and experiments show that they work well in practice, even for large intricate programs. These experiments are mentioned in Section 8.

Caching of Dynamically Generated HTML

Traditional Web caching based on HTTP works by associating an expiration time to all documents sent by the servers to the clients. This has helped in decreasing both network and server load and response times. By default, no expiration is set, and, by using "now", caching is effectively disabled. This technique was designed primarily for documents whose contents rarely or never change, not for documents dynamically generated by interactive Web services. The gradual change from statically to dynamically generated documents has therefore caused the impact of Web caching to degrade.

Existing proposals addressing this include Active Cache, HPP, and various server-based techniques, as explained in the survey in [6]. Server-based techniques aim to relieve the server of redundant computations, not to decrease network load. They typically work by simplifying assumptions, for instance that many interactions can be handled without side-effects on the global service state, that interactions are often identical for many clients, or that the dynamics of the pages is limited to, e.g., banner ad rotation. None of this applies to complex interactive services. Active Cache is a proxy-based solution that employs programmable cache applets. This can be very effective, but it requires both specialized proxy servers and careful programming to ensure consistency between the proxies and the main server.

HPP tries to separate the constant parts from the dynamic parts of the generated documents. We apply a similar technique. In contrast to HPP, our solution is entirely automatic, while HPP requires extra programming. The idea is to exploit the clear division between the service code and the HTML templates in `<bigwig>`. In our normal implementation of DynDoc, the internal template representation is converted to an HTML document on the server when the `show` statement is executed. Instead, we now store each template constant in a fixed file on the server, and defer the conversion to the client using a JavaScript representation of the dynamic parts. The template files can now be cached by the ordinary browser caches. More details of the technique can be found in [6]. We summarize our evaluation results in Section 8.

Code Gaps and Document Clusters

In the following, we describe two extensions to the DynDoc language. Occasionally, the page-centered approach is admittedly more appropriate than the session-centered one. Consider the following example, which gives the current time of day:

```
service {
  session Time() {
    html h = <html>Right now, the time is <[t]></html>;
    show h<[t=now()];
  }
}
```

An equivalent but less clumsy version can be written using *code gaps*, which implicitly represent expressions whose values are computed and plugged into gaps when the document is being shown:

```
service {
  session Time() {
    html h = <html>Right now, the time is <[(now())]></html>;
    show h;
  }
}
```

Documents with code gaps remain first-class values, since the code can only access the global scope. Note that code gaps in `<bigwig>` are more powerful than the usual page-centered approach, since the code exists in the full context of sessions, shared variables, and concurrency control. In fact, with the idea of *published* documents described in Section 6, the page-centered approach is now included as a special case of `<bigwig>`.

Some services may want to offer the client more than a single document to browse, for example, the response could be a tiny customized Web site. In `<bigwig>` we have experimented with support for showing such *document clusters*. The difficulty is to provide a simple notation for specifying an arbitrary graph of documents connected by links. For an HTML variable `x`, we introduce the *document reference* notation `&x`, which can be used as the right-hand side of a plug operation. It will eventually expand into a URL, but not until the document is finally shown. Until then, the flow analysis just records the connection between the gap and the variable. When a document is shown, the transitive closure of document references is computed, and the resulting cluster of documents is produced with references replaced by corresponding URLs. The following example shows a cluster of two documents that are cyclically connected. Notice that the cluster can be browsed freely without cluttering the control-flow:

```
service {
  session Cluster() {
    html greeting = <html>
      Hi! Click <a href=[where]>here</a> for a kind word.
    </html>;
    html kind = <html>How nice to see you! <a href=[there]>Back</a></html>;
    kind = kind<[there = &Greeting];
    show greeting<[where=&kind];
  }
}
```

The compiler checks that all cluster documents with submit buttons contain the same form fields. It is also necessary to perform an escape analysis to ensure that document variables are not exported out of their scope.

4 Form Field Validation

A considerable effort in Web programming is expended on form field validation, that is, on checking whether the data supplied by the client in the form fields is valid, and when it is not, producing error messages and requesting the fields to be filled in again. Apart from details about regular expression matching, the main problem is to program a solution that is robust, efficient, and user friendly.

One approach is *server-side* validation, where the form fields are validated on the server when the page has been submitted. None of the languages mentioned in Section 1 provide any help for this, except for the regular expression matching in Perl. Therefore, the main logic of the service often becomes cluttered with validation code. In a sense, every program part that sends a page to a client must be wrapped into a while-loop that repeats until the input is valid. Other disadvantages include wasting bandwidth and causing delays to the users.

The alternative is *client-side* validation, which usually requires the programmer to use JavaScript in the pages being generated. This permits more sophisticated user interactions and reduces the communication overhead. However, client-side validation should not be used alone. The reason is that the client is perfectly capable of bypassing the JavaScript code, so an additional server side validation must always be performed. Thus, the same code must essentially be written both in JavaScript and in the server scripting language. In practice, writing JavaScript input validators that capture all validity requirements and at the same time are also user friendly can be very difficult, since, unfortunately, most browsers differ in their JavaScript support. Whole Web sites are dedicated to explaining how the various subsets of JavaScript work in different browsers.[3]

In `<bigwig>` we have introduced a domain-specific sub-language, called PowerForms, for form field validation [7]. It handles complex interdependencies between form fields, and the compiler generates the required code for both client and server. By compiling into JavaScript, only the PowerForms implementors need to know the details of how browsers support JavaScript, rather than all Web service programmers. Also, the programmer does not need to write essentially the same code in a server-side version and a client-side version anymore.

PowerForms is a declarative language. Informally, this means that the programmer specifies what the input requirements are, not how to check them. In its simplest form, PowerForms allows regular-expression *formats* to be associated to form fields:

```
service {
  format Digit = range('0','9');
  format Alpha = union(range('a','z'),range('A','Z'));
  format Word = concat(Alpha,star(union(Digit,Alpha)));
  format Email = concat(Word,"@",Word,star(concat(".",Word)));
  session Validate() {
    html form = <html>
      Please enter your email address:
      <input name=email type=text size=20>
```

[3]See e.g. http://www.webdevelopersjournal.com/articles/javascript_limitations.html or http://www.xs4all.nl/~ppk/js/version5.html.

```
      <format name=Email field=email>
    </html>;
    string s;
    show form receive[s=email];
  }
}
```

This example shows how to constrain input in the `email` field to a certain regular expression. The `<bigwig>` compiler generates the JavaScript code that checks the user input on the client-side and provides help and error messages, and also the code that performs the server-side double-check. “Traffic-light” icons next to the input fields provide the user with continuous feedback about the string entered so far. “Green” means valid, “yellow” means invalid but a prefix of something valid, and “red” means not a prefix of something valid. Other alternatives can be chosen, such as checkmark symbols, arrows, etc. We also allow the usual UNIX-style syntax for regular expressions in the subset of our notation that excludes the intersection and complement operators.

Formats can be associated to all kinds of form fields, not just those of type `text`. For `select` fields, the format is used to filter the available options. For `radio` and `checkbox` fields, only the permitted buttons can be depressed.

As noted in [14], many forms contain fields whose values may be constrained by those entered in other fields. A typical example is a field that is not applicable if some other field has a certain value. Such interdependencies are almost always handled on the server, even if the rest of the validation is performed on the client. Presumably, the reason is that interdependencies require even more delicate JavaScript programming. The `<bigwig>` solution is to allow such field interdependencies to be specified using an extension of the regular expressions: the `format` tags are extended to describe boolean decision trees, whose conditions probe the values of other form fields and whose leaves are simple formats. The interdependence is resolved by a fixed-point process computed on the client by JavaScript code automatically generated by the `<bigwig>` compiler. A simple example is the following, where the client chooses a letter group and the `select` menu is then dynamically restricted to those letters:

```
service {
  format Vowel = charset("aeiouy");
  format Consonant = charset("bcdfghjklmnpqrstvwxz");
  html form = <html>
    Favorite letter group:
    <input type=radio name=group value=vowel checked>vowels
    <input type=radio name=group value=consonant>consonants
    <br>
    Favorite letter:
    <select name=letter>
      <option value="a">a
      <option value="b">b
      <option value="c">c
      ...
      <option value="z">z
    </select>
    <format field=letter>
```

```
        <if><equal field=group value=vowel>
        <then><format name=Vowel></then>
        <else><format name=Consonant></else>
        </if>
      </format>
    </html>;
    session Letter() {
      string s;
      show form receive[s=letter];
    }
}
```

ColdFusion [13] provides a mechanism reminiscent of PowerForms. However, it does not support field interdependencies or validation of non-`text` fields. PowerForms is shown to be a simple language with a clean semantics that appears to handle most realistic situations. We have implemented it both as part of the `<bigwig>` compiler and as a stand-alone tool that can be used to add input validation to general HTML pages.

5 Concurrency Control

As services have several session threads, there is a need for synchronization and other concurrency controls to discipline the concurrent behavior of the active threads. A simple case is to control access to the shared variables using mutex regions or the readers/writers protocol. Another issue is enforcement of priorities between different session kinds, such that a management session may block other sessions from running. Another example is event handling, where a session thread may wait for certain events to be caused by other threads.

We deal with all of these scenarios in a uniform manner, based on a central controller process in the runtime system, which is general enough to enforce a wide range of safety properties [25]. The support for concurrency control in the previously mentioned Web languages is limited to more traditional solutions, such as file locking, monitor regions, or synchronized methods.

A `<bigwig>` service has an associated set of *event labels*. During execution, a session thread may request permission from the controller to pass a specific event checkpoint. Until such permission is granted, the session thread is suspended. The policy of the controller must be programmed to maintain the appropriate global invariants for the entire service. Clearly, this calls for a domain-specific sub-language. We have chosen a well-known and very general formalism, temporal logic. In particular, we use a variation of monadic second-order logic [29]. A formula describes a set of strings of event labels, and the associated semantics is that the trace of all event labels being passed by all threads must belong to that set. To guide the controller, the `<bigwig>` compiler uses the MONA tool [20] to translate the given formula into a minimal deterministic finite-state automaton that is used by the controller process to grant permissions to individual threads. When a thread asks to pass a given event label, it is placed in a corresponding queue. The controller continually looks for nonempty queues whose event labels correspond to enabled transitions from the current DFA state. When a match is found, the corresponding transition is performed and the chosen thread is resumed. Of course, the controller must be implemented to satisfy some fairness requirements. All regular trace languages can be expressed in the logic.

Applying temporal logics is a very abstract approach that can be harsh on the average programmer. However, using syntax macros, which are described in Section 6, it is possible to capture common concurrency primitives, such as semaphores, mutex regions, the readers/writers protocol, monitors, and so on, and provide high-level language constructs hiding the actual formulas. The advantage is that `<bigwig>` can be extended with any such constructs, even some that are highly customized to particular applications, while maintaining a simple core language for concurrency control.

The following example illustrates a simple service that implements a critical region using the event labels `enter` and `leave`:

```
service {
  shared int i;
  session Critical() {
    constraint {
      label leave,enter;
      all t1,t3: (t1<t3 && enter(t1) && enter(t3)) =>
                 is t2: t1<t2 && t2<t3 && leave(t2);
    }
    wait enter;
    i = i+1;
    wait leave;
  }
}
```

The formula states that for any two `enter` events there is a `leave` event in between, which implies that at any time at most one thread is allowed in the critical region. Using syntax macros, programmers are allowed to build higher-level abstractions such that the following can be written instead:

```
service {
  shared int i;
  session Critical() {
    region {
      i = i+1;
    }
  }
}
```

We omit the macro definitions here. In its full generality, the `wait` statement is more like a `switch` statement that allows a thread to simultaneously attempt to pass several event labels and request a timeout after waiting a specified time.

A different example implements an asynchronous event handler. Without the macros, this could be programmed as

```
service {
  shared int i;
  constraint {
    label handle,cause;
    all t1: handle(t1) => is t2: t2<t1 && cause(t2) &&
                               (all t3: t2<t3 && t3<t1 => !handle(t3));
  }
  session Handler() {
    while (true) {
      wait handle;
      i++;
    }
  }
  session Application() {
    wait cause;
  }
}
```

This nontrivial formula allows the handler to proceed, without blocking the application, whenever the associated event has been caused at least once since the last invocation of the handler. Fortunately, the macros again permit high-level abstractions to be introduced with more palatable syntax:

```
service {
  shared int i;
  event Increment {
    i++;
  }
  session Application() {
    cause Increment;
  }
}
```

The runtime model with a centralized controller process that ensures satisfaction of safety constraints is described in [8]. The use of monadic second-order logic for controller synthesis was introduced in [Sandholm and Schwartzbach 1998] where additionally the notions of *triggers* and *counters* are introduced to gain expressive power beyond regular sets of traces, and conditions for distributing the controller for better performance are defined.

The session model provides an opportunity to get a global view of the concurrent behavior of a service. Our current approach does not exploit this knowledge of the control flow. However, we plan to investigate how it can be used in specialized program analyses that check whether liveness and other concurrency requirements are complied with.

6 Syntax Macros

As previously mentioned, `<bigwig>` contains a notion of macros. Although not specific to Web services, this abstraction mechanism is an essential part of `<bigwig>` that serves to keep the sub-languages minimal and to tie them together.

A macro language can be characterized by its level of operation which is either *lexical* or *syntactic*. Lexical macro languages operate on sequences of tokens and conceptually precede parsing. Due to the independence of syntax, macros often have unintended effects, and parse errors are only discovered at invocation time. Consequently, programmers are required to consider how individual macro invocations are being expanded and parsed. Syntactic macros amend this by operating on parse trees instead of token sequences [31]. Types are added to the macro arguments and bodies in the form of nonterminals of the host language grammar. Macro definitions can now be syntax-checked at definition time, guaranteeing that parse errors no longer occur as a consequence of macro expansion. Using syntax macros, the syntax of the programming language simply appears to be extended with new productions.

Our macros are syntactic and based entirely on simple declarative concepts such as grammars and substitution, making them easy and safe to use by ordinary Web service programmers. Other macro languages, such as MS^2, Scheme macros, and Maya, instead apply full Turing complete programming languages for manipulating parse trees at compile time, making them more difficult to use.

As an initial example, we extend the core language of `<bigwig>` with a `repeat-until` control structure that is easily defined in terms of a `while` loop.

```
macro <stm> repeat <stm S> until ( <exp E> ) ; ::= {
  {
    bool first = true;
    while (first || !<E>) {
      <S>
      first = false;
    }
  }
}
```

The first line is the header of the macro definition. It specifies the nonterminal type of the macro abstraction and the invocation syntax, including the typed arguments. As expected, the type of the `repeat-until` macro is `<stm>`, representing statements. This causes the body of the macro to be parsed as a statement and announces that invocations are only allowed in places where an ordinary statement would be. We allow the programmer to design the invocation syntax of the macro. This is used to guide parsing and adds to the transparency of the macro abstractions. This particular macro is designed to parse two appropriately delimited arguments, a statement `S` and an expression `E`. The body of the macro implements the abstraction using a boolean variable and a `while` loop. When the macro is invoked, the identifiers occurring in the body are α-converted to avoid name clashes with the invocation context.

With a concept of *packages*, macros can be bundled up in collections. Our experience with `<bigwig>` programming has led us to develop a “standard macro package”, `std.wigmac`, that extends the sub-languages of `<bigwig>` in various ways and has helped keep the language minimal. For instance, the form field validation language is extended with an `optional` regular expression construct, and database language macros transform SQL-like queries into our own iterative `factor` construction. Also, various composite security modifiers are defined, and concurrency control macros, such as the `region` from Section 5, gradually build on top of each other to implement increasingly sophisticated abstractions.

Macros are also used to tie together different sub-languages, making them collaborate to provide tailor-made extensions of the language. For instance, the sub-languages dealing with sessions, dynamic documents, and concurrency control can be combined into a `publish` macro. This macro is useful when a service wishes to publish a page that is mostly static, yet needs to be recomputed once in a while, when the underlying data changes. The following macros efficiently implements such an abstraction:

```
macro <toplevels> publish <id D> { <exp E> } ::= {
  shared html <D>~cache;
  shared bool <D>~cached;
  session <D>() {
    exclusive if (!<D>~cached) {
      <D>~cache = <E>;
      <D>~cached = true;
    }
    show <D>~cache;
  }
}

macro <stm> touch <id d> ; ::= {
  <d>~cached = false;
}
```

The `publish` macro recomputes the document if the cache has expired and then shows the document, while the `touch` macro causes the cache to expire. The `~` operator is used to create new identifiers by concatenating others. Using this extended syntax, a service maintaining a high-score list, for example, can look like this:

```
require "publish.wigmac"
service {
  shared int record;
  shared string holder;
  publish HiScore {
    computeWinnerDoc(record, holder)
  }
  session Play() {
    int score = play();
    if (score>=record) {
      show EnterName receive[holder=name];
      record = score;
      touch HiScore;
    } else {
      show <html>Sorry, no record.</html>;
    }
  }
}
```

Here the high-score document is only regenerated when a player beats the record. This code is clearly easier to understand and maintain than the corresponding expanded code.

The expressive power of syntax macros is extended with a concept of *metamorphisms*, as explained in [10]. This declaratively permits tree structures to be transformed into host language syntax without compromising syntactic safety, something not possible with other macro languages. Using this mechanism in an extreme way, it is possible to define whole new languages. We call this concept a *very* domain-specific language, or VDSL.

At the University of Aarhus, undergraduate computer science students must complete a bachelor's degree in one of several fields. The requirements that must be satisfied are surprisingly complicated. To guide them towards this goal, the students must maintain a so-called "bachelor's contract" that plans their remaining studies and discovers potential problems. This process is supported by a Web service which, for each student, iteratively accepts past and future course activities, checks them against all requirements, and diagnoses violations until a legal contract is composed. This service was first written as a straight `<bigwig>` application, but quickly became annoying to maintain due to constant changes in the curriculum. It was redesigned in the form of a VDSL, where study fields and requirements are conceptualized and defined directly in a more natural language style. This makes it possible for non-programmers to maintain and update the service. An small example input is

```
require "bachelor.wigmac"
studies
  course Math101
    title "Mathematics 101"
    2 points fall term
  ...
  course Phys202
    title "Physics 202"
    2 points spring term
  course Lab304
    title "Lab Work 304"
    1 point fall term
  exclusions
    Math101 <> MathA
    Math102 <> MathB
  prerequisites
    Math101,Math102 < Math201,Math202,Math203,Math204
    CS101,CS102 < CS201,CS203
    Math101,CS101 < CS202
    Math101 < Stat101
    CS202,CS203 < CS301,CS302,CS303,CS304
    Phys101,Phys102 < Phys201,Phys202,Phys203,Phys301
    Phys203 < Phys302,Phys303,Lab301,Lab302,Lab303
    Lab101,Lab102 < Lab201,Lab202
    Lab201,Lab202 < Lab301,Lab302,Lab303,Lab304
  field "CS-Mathematics"
    field courses
      Math101,Math102,Math201,Math202,Stat101,CS101,CS102,CS201,CS202,CS203,
      CS204,CS301,CS302,CS303, CS304,Project
    other courses
      MathA,MathB,Math203,Math204,Phys101,Phys102,Phys201,Phys202
    constraints
      has passed CS101,CS102
      at least 2 courses among CS201,CS202,CS203
      at least one of Math201,Math202
      at least 2 courses among Stat101,Math202,Math203
    has 4 points among Project,CS303,CS304
    in total between 36 and 40 points
```

None of the syntax displayed is plain `<bigwig>`, except the macro package `require` instruction. The entire program is the argument to a single macro, `studies`, that expands into the complete code for a corresponding Web service. The file `bachelor.wigmac` is only 400 lines and yet defines a complete implementation of the new language. Thus, the `<bigwig>` macro mechanism offers a rapid and inexpensive realization of new ad-hoc languages with almost any syntax desired. Similar features do not occur in any of the Web service languages mentioned in the previous sections.

7 Other Web Service Aspects

There are of course other features in `<bigwig>` that are necessary to support Web service development, but for which we have no major innovations. These are briefly presented in this section.

HTML Deconstruction

The template mechanism is used to construct HTML documents, but when "run in reverse" it also allows for deconstruction. This is realized by using the templates as patterns in which the gaps play the role of variables, as illustrated in this example:

```
service {
  html Template = <html>
    <[]><img src=[source] alt="today's Dilbert comic"><[]>
  </html>;
  session Dilbert() {
    string data = get("http://www.dilbert.com/");
    string s;
    match(data,Template)[s =source];
    exit Template<[source ="http://www.dilbert.com"+s];
  }
}
```

which grabs the daily strip from the Dilbert home page. Gaps without names serve as wildcards.

Seslets

For some interaction patterns a strict session model can be inappropriate, since the client and server must alternate between being active and suspended. Furthermore, information cannot be pushed on the server's initiative while the client is viewing a page. A simple example is a chat room where new messages should appear automatically, without the client having to reload the page being viewed, and where only the new message and not the entire new page is transmitted. The essence of this concept is *client-side computations*, which are able to contact the server on their own accord.

The `<bigwig>` solution is a notion of *seslets*. A seslet is a kind of lightweight session that is allowed to do anything an ordinary session can do, except perform `show` operations. It is invoked by the client with some arguments and eventually returns a reply of any `<bigwig>` type. Typically, it performs database operations or waits for certain events to occur, and then reports back to the client.

Since we are limited by the existing technologies on the client-side, our current implementation is restricted to using Java applets or JavaScript. To facilitate the writing of applets, the `<bigwig>` compiler generates the Java code for an abstract class extending `Applet`, which must be inherited from in order to access the available seslets. Alternatively, we have experimented with a JavaScript interface. However, this approach is limited by the lack of client-server communication support from JavaScript, so we currently apply cookies for the communication.

An important use of seslets is to allow client-side code to synchronize with other active threads on the server. For example, the chat room solution could employ a seslet that uses the concurrency control mechanisms of `<bigwig>` to wait until the next message is available, which is then returned to the applet. In this way, no client pulling or busy waiting is required.

Databases

Most Web services are centered around a database. In the general case, this is an existing, external database which the service must connect to. The `<bigwig>` system supports the ODBC interface for this purpose. In most other Web service languages, database queries are built dynamically as strings that must be parsed by the database engine. In `<bigwig>`, queries are not built as strings but are written in a query language that is part of the `<bigwig>` syntax. This allows for compile-time checking of the syntax and types of queries, eliminating another source of errors. Since many smaller services use only simple data, we also offer an internal database that is implemented on top of the file system.

Security

There are many aspects of Web service security.[4] The security in `<bigwig>` can be divided into two categories, depending on whether it is generically applicable to all services or specific to the behavior of a particular service.

The former category mostly relates to the runtime environment and communication, dealing with concepts such as integrity, authenticity, and confidentiality. Integrity of a session thread's local state is achieved by keeping it exclusively on the server. Integrity of shared data is provided by the database. An interaction key is generated and embedded in every document shown to the client and serves to prevent submission of old documents. Clients and session threads are associated through a random key which is created by the server upon the first request and carried across interactions in a hidden input field. This mechanism may optionally be combined with other security measures, such as SSL,

[4]See `http://www.w3.org/Security/faq/`.

to provide the necessary level of security. Authenticity and confidentiality are addressed through general declarative security modifiers that the programmer can attach on a `service`, `session`, or `show` basis. The modifiers `ssl` and `htaccess` enforce that the SSL and HTTP Authentication protocols are used for communication. The `selective` modifier restricts access to a session to those clients whose IP numbers match a given set of prefixes. Finally, the `singular` modifier ensures that the client has the same IP address throughout the execution of a session.

We envision performing some simple static analyses relating to the behavioral security of particular services. Values are classified as *secret* or *trusted*, and, in contrast to tainting in Perl, the compiler keeps track of the propagation of these properties. Furthermore, there are restrictions on how each kind of data can be used. Form data is always assumed to be untrusted and gaps are never allowed to be plugged with secret values. Variables can be declared with the modifiers `secret` or `trusted` and may then only contain the corresponding values. The `system` function can only be called with a trusted string value. To change the classification of a value, there are two functions, `trust` and `disclose`. The programmer must make the explicit choice of using these coercions. An example involving trust is the following service:

```
service {
  session Lookup() {
    html Error = <html>Invalid URL!</html>;
    html EnterURL = <html>Enter a URL: <input type=text name=URL></html>;
    string u,domain;
    show EnterURL receive[u = URL];
    if (|u|<7 || u[0..7]!="http://") show Error;
    for (i=7; i<|u| && u[i]!='/'; i++);
    domain = u[7..i];
    if (system("/usr/sbin/nslookup '" + domain + "'").stderr!="") {
      show Error;
    }
  }
}
```

This code performs an *nslookup* on the URL supplied by the user to check whether its domain exists. Since the value of `domain` is derived from the form field `URL` it should not be trusted, and its use in the call of `system` will be flagged by the compiler. And, indeed, it would be unfortunate if the client enters `"http://foo';rm -rf /'"` in the form. A similar analysis is performed for `secret`. Consider the example:

```
service {
  shared secret string password;
  bool odd(int n) { return n%2==1; }
  session Reveal() {
    if (odd(|password|)) show <html>foo</html>;
  }
}
```

The compiler is sufficiently paranoid to reject this program, since the branching of the `if`-statement depends on a function applied to information derived from a secret value. These analyses are not particularly original, but are not seen in other Web service programming languages.

There is still much work to be done in this area. So far, we have not considered using cryptological techniques to ensure service integrity, the role of certificates, or more sophisticated static analyses.

8 Evaluation

The `<bigwig>` language should be evaluated according to two different criteria. First, the quality of our language design as seen from concrete programming experiences. This is necessarily a somewhat intangible and subjective criterion. Second, the performance of our language implementation as seen from observed benchmarks.

Experience with `<bigwig>`

`<bigwig>` is still mainly an experimental research tool, but we have gained experiences from numerous minor services that we have written for our own edification, a good number of services that are used for administrative purposes at the University of Aarhus, and a couple of production services on which we have collaborated. Apart from these applications, we estimate that `<bigwig>` has been downloaded roughly 2500 times from our Web site, and we have mainly received positive feedback from the users.

One production service is the Web site of the European Association for Theoretical Computer Science (`www.eatcs.org`), handling newsletters, webboards, and several membership services. It is written in 5,345 lines of `<bigwig>`, using 133 HTML templates and 114 `show` statements. Another is the Web site of the JAOO 2001 conference (`www.jaoo.dk`), handling all aspects of advertisement, schedules, registration, and attendance services. It is written is 7,943 lines of `<bigwig>`, using 248 HTML templates, and 39 `show` statements.

These experiences have shown that `<bigwig>` has two very strong points. First, the session concept greatly simplifies the programming of complicated control flow with multiple client interactions. Second, the HTML templates are very easy and intuitive to use and the static guarantees catching numerous errors, many of which are difficult to find by any other means. It is particularly helpful that the HTML analyzers provide precise and intuitive error messages.

The JAOO application has been particularly interesting, since it involves collaboration with an external HTML designer. This experience confirmed that our templates are successful in defining an interface between programmers and designers and that gaps and fields define a useful contract between the two.

The main weak point that we identified is the core language, which is often found to lack minor features. We plan to address this in future work, as mentioned in Section 9.

The stand-alone version of the PowerForms sub-language has been surprisingly popular in its own right. It has many active users, and has been integrated into a proprietary Web deployment system.

Performance

When evaluating the performance of the `<bigwig>` implementation, we want to focus on the areas where we tried to provide improvements. We are not aiming for simple high-load services, but are focusing on services with intricate control-flow. Still, informal tests show that the throughput of our services is certainly comparable with that of straight CGI-based services or Servlet applications running on J2SE.

The automatic caching scheme based on our HTML templates is designed to exploit their intricate structure to cache static fragments on the client side. We have obtained real benefits from this approach. The experiments reported in [6] show that the size of the transmitted data may shrink by a factor of 3 to 30, which on a dial-up connection translates into a reduction in download time by a factor of 2 to 10.

It is also relevant to evaluate the performance of the `<bigwig>` compiler, since we employ a series of theoretically quite expensive static analyses. However, in practice they perform very well, as documented in [26, 9]. The EATCS service is analyzed for HTML validity in 6.7 seconds and the JAOO service in 2.4 seconds.

9 Conclusion

The `<bigwig>` project has identified central aspects of interactive Web services and provided solutions in a coherent framework based on programming language theory. At the same time, the `<bigwig>` project is a case study in applications of the domain-specific language design paradigm.

We argue that the notion of sessions is essential to Web services and should constitute the basic structure of a Web service programming language. Together with higher-order document templates, such as in the DynDoc sub-language, the dynamic construction of Web pages becomes flexible at the same time, making it easy to use, and safe by compile-time guarantees regarding document validity and the use of input forms. We have shown that form field validation, compared to traditional approaches, can be made easier with a domain-specific sub-language, such as PowerForms, which automatically translates high-level specifications into a combination of more low-level server-side and client-side code. We have examined how temporal logics can be used to synthesize concurrency controllers. Finally, we have demonstrated how macro mechanisms can be made effective for extending and combining languages, in the context of the sub-languages of `<bigwig>`.

Version 2.0 of the `<bigwig>` compiler and runtime system is freely available from the project home page at `www.brics.dk/bigwig/` where documentation and examples can also be found.

Regarding the future development of `<bigwig>` we now move towards Java. We are developing JWIG [11] as an extension of Java, where we add the most successful features of `<bigwig>`, such as the session model, dynamic documents, form field validation, and syntax macros. Since the design of `<bigwig>` has focused on the Web specific areas, we hope that the many standard programming issues of Web services become easier to develop with JWIG. However, a number of new challenges arise. For instance, the program analyses described in Section 3 all assume that we have access to precise control-flow graphs of the programs. This is trivial for `<bigwig>`, but certainly not for Java. Other future plans include type-safe support for XML document transformation, WML and VoiceXML support, and broadening the view towards development and management of whole Web sites comprising many services.

Acknowledgments

Tom Ball provided us with extensive and very helpful information about experiences with the MAWL language. Anders Sandholm was a key participant during his Ph.D. studies at BRICS. Mikkel Ricky Christensen and Steffan Olesen worked tirelessly as student programmers during the entire project. Niels Damgaard, Uffe Engberg, Mads Johan Jurik, Lone Haudrum Olesen, Christian Stenz, and Tommy Thorn provided valuable feedback and suggestions. We also appreciate the efforts made by the participants of the WIG Projects course in Spring 1998. Finally, we are grateful for the insightful comments we received from the anonymous reviewers.

References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language.* Addison-Wesley, 3rd edition, 2000.

[2] David Atkins, Thomas Ball, Michael Benedikt, Glenn Bruns, Kenneth Cox, Peter Mataga, and Kenneth Rehor. Experience with a domain specific language for form-based services. In *Usenix Conference on Domain Specific Languages*, October 1997.

[3] David Atkins, Thomas Ball, Glenn Bruns, and Kenneth Cox. Mawl: a domain-specific language for form-based services. In *IEEE Transactions on Software Engineering*, June 1999.

[4] Leon Atkinson. *Core PHP Programming.* Prentice Hall, 1999.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. RFC1945, May 1996. `http://www.w3.org/Protocols/rfc1945/rfc1945`.

[6] Claus Brabrand, Anders Møller, Steffan Olesen, and Michael I. Schwartzbach. Language-based caching of dynamically generated HTML, May 2001. Submitted for publication.

[7] Claus Brabrand, Anders Møller, Mikkel Ricky, and Michael I. Schwartzbach. Powerforms: Declarative client-side form field validation. *World Wide Web Journal*, 3(4):205–314, 2000.

[8] Claus Brabrand, Anders Møller, Anders Sandholm, and Michael I. Schwartzbach. A runtime system for interactive Web services. *Computer Networks*, 31:1391–1401, 1999. Also in Proceedings of the Eighth International World Wide Web Conference.

[9] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering, PASTE 2001*. ACM, 2001.

[10] Claus Brabrand and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros. In *Proceedings of Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2002*. ACM, 2002.

[11] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction, 2002. Submitted for publication.

[12] K. Cox, T. Ball, and J. C. Ramming. Lunchbot: A tale of two ways to program web services. Technical Report BL0112650-960216-06TM, AT&T Bell Laboratories, 1996.

[13] John Desborough. *Cold Fusion 3.0 Intranet Application*. International Thomson Publishing, 1997.

[14] Micah Dubinko, Sebastian Schnitzenbaumer, Malte Wedel, and Dave Raggett. XHTML extended forms requirements. W3C Working Draft, April 2001. `http://www.w3.org/TR/xhtml-forms-req.html`.

[15] Mary Fernandez, Dan Suciu, and Igor Tatarinov. Declarative specification of data-intensive Web site. In *USENIX Conference on Domain-Specific Languages*, October 1999.

[16] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, June 1998.

[17] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol version 3.0. Internet Draft, November 1996. `http://home.netscape.com/eng/ssl3/draft302.txt`.

[18] Shishir Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly & Associates, Inc., 2000.

[19] Alex Homer, John Schenken, Mathew Gibbs, Jan D. Narkiewicz, Jason Bell, Mike Clark, Andy Elmhorst, Bruce Lee, Matt Milner, and Adil Rehan. *ASP.NET Programmer's Reference*. Wrox Press, 2001.

[20] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual.* BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001.

[21] David A. Ladd and J. Christopher Ramming. Programming the Web: An application-oriented language for hypermedia services. In *4th Intl. World Wide Web Conference*, 1995.

[22] Open Market, Inc. FastCGI: A high-performance Web server interface, April 1996. Technical White Paper, www.fastcgi.com.

[23] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.

[24] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 specification. W3C Recommendation, December 1999. `http://www.w3.org/TR/html401`.

[25] Anders Sandholm and Michael I. Schwartzbach. Distributed safety controllers for Web services. In *Fundamental Approaches to Software Engineering, FASE'98*, number 1382 in LNCS, 1998.

[26] Anders Sandholm and Michael I. Schwartzbach. A type system for dynamic Web documents. In *Principles of Programming Languages, POPL'00.* ACM, 2000.

[27] Sun Microsystems. Java Servlet Specification, Version 2.3, 2001. `http://java.sun.com/products/servlet`.

[28] Sun Microsystems. JavaServer Pages Specification, Version 1.2, 2001. `http://java.sun.com/products/jsp`.

[29] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.

[30] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[31] Daniel Weise and Roger F. Crew. Programmable syntax macros. In *Programming Language Design and Implementation, PLDI'93*, 1993.