BRICS

Basic Research in Computer Science

The `<bigwig>` Project

Claus Brabrand
Anders Møller
Michael I. Schwartzbach

BRICS Report Series **RS-00-42**

ISSN 0909-0878 **December 2000**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/00/42/`

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.

The World Wide Web has undergone an extreme development since its invention ten years ago. A fundamental aspect is the change from static to dynamic generation of Web pages. Generating Web pages dynamically in dialogue 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. Each aspect is handled by a nearly independent sublanguage, and the entire collection is integrated into a common core language. The <bigwig> compiler uses the available Web technologies as target languages, making <bigwig> available on almost any combination of browser and server, without relying on plugins or server modules.

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. Our ambitions are twofold: to build a useful tool for Web programmers and to explore the domain-specific paradigm for language design.

Programming Web services can be a daunting task compared to programming more traditional systems. A large majority of the Web services that exist today are produced either with the Perl scripting language on top of the HTTP/CGI Web protocol [18], or with Microsoft's ASP [14] or the Open Source language PHP [4], both based on the idea of embedding program code in the Web pages. Although a vast number of Web services have been produced with these and similar languages, often the quality and development times of these services are discouraging. Provocatively, the reason being that these languages do not seem to build on the experience of thirty years of research in programming language technology. The overall goal of the <bigwig> project is to bring some of this research experience into the hands of the Web service developers.

The <bigwig> language is an intellectual descendant of MAWL, the *Mother of All Web Languages* [3, 2, 20], from which it has inherited its conceptual basis. However, we have moved further by considering more aspects of interactive Web services and by generally raising the ambitions.

We have currently released version 1.3 of the <bigwig> tool under the GPL license. This paper describes primarily that implementation but also mentions

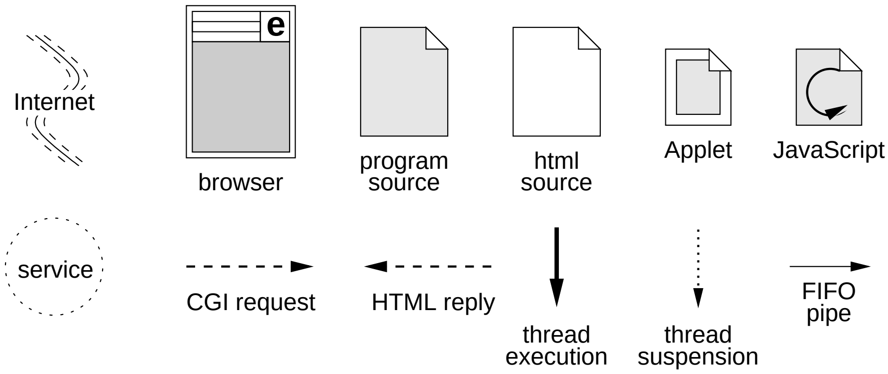

Figure 1: Legend for graphical components.

concepts and features that are scheduled for future releases.

Throughout this paper we use a graphical language to illustrate the various interactions between clients and servers that we are concerned with. The components of that notation are shown in Figure 1.

2 Language Design Principles

To a significant extent the `<bigwig>` project is an experiment in designing a programming language for a complex application domain. From the outset, we have had a clear design principle in mind.

While millions of Web services have been produced today, it is discouraging that the technologies employed frankly seem old-fashioned. In contrast, `<bigwig>` is intended to be a high-level, *domain-specific* language, meaning that it employs special syntax and constructs that are tailored to fit its particular application domain. However, within this general approach we have formulated a more specific principle to which we have faithfully adhered. The `<bigwig>` language has been designed in the following manner:

- the application domain is analyzed and fundamental aspects are identified;
- the understanding of each aspect is expressed in the design of a domain-specific sublanguage;
- the sublanguages are embedded in a simple core language; and
- a notion of syntax macros ties together the sublanguages and provides additional layers of abstractions.

Of course, many iterations have been necessary to incorporate the experiences we gained with earlier versions of the language. However, those principles have served us well and could conceivably be applicable to other application domains as well. We have focused on the following central aspects of Web services:

- *sessions* as underlying paradigm: the session concept is an essential basis for interactive Web services;
- concurrency control: Web services consist of collections of processes running concurrently and sharing resources;
- dynamic documents: HTML pages must be constructed in a flexible but safe way;

- form field validation: validating user input requires too much attention of Web programmers, so a higher-level solution is desirable;
- integration with databases: the core of a Web service is often a database with a number of sessions providing Web access; and
- security: this includes both *cryptographic security* for authenticity and confidentiality, and *information-flow security* to prevent information leaking due to inexpedient programming.

Each of these aspects is described in the following sections. Some of them are also described in previous more specialized papers [29, 8, 30, 7, 9].

An immediate consequence of our approach is that `<bigwig>` is not a small language. All in all, it has a large syntax. However, since it is composed of nearly independent sublanguages, it has a gentle learning curve. Small and simple Web services can be written without much complication, since many of the advanced features need only be introduced to match the increasing ambitions of the Web programmer.

Also, `<bigwig>` is designed to grow [32]. Our notion of syntax macros have proved remarkably successful in providing seamless extensions of the base language. This has helped each of the sublanguages to remain minimal, since desired syntactic sugar is given by the macros. During our work with `<bigwig>`, we have also discovered the idea of *very* domain-specific languages (VDSL). Those are obtained by taking macros to the extreme where they define a completely new syntax that can be viewed in isolation from the host language. When used to target specific families of applications, they offer some advantages in maintenance and accessibility.

3 Technical Design Principles

`<bigwig>` is designed with a specific user group in mind, namely ordinary programmers. Thus, the use of our tool requires programming skills on par with those for writing simple programs in standard languages such as C or Java. We have very early abandoned any idea of creating a drag-and-drop developing environment, since it seems clear to us that Web programming is just like ordinary programming, only made more complicated by the morass of technical details that are imposed by the given client-server model. In fact, our motto has been "to make Web programming as easy as writing simple C or Java programs".

The `<bigwig>` compiler uses common Web technologies as target languages. This includes HTML [26], CGI scripts [18], JavaScript [16], HTTP Authentication [6], and Java applets [1]. As new technologies become standard, the compiler will obtain corresponding opportunities for generating better code.

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

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 HTML pages and the logical structure that is necessary to define the semantics of a service. Thus, we expect that standard HTML authoring tools are used, conceivably by others than the Web programmer.

We have consciously aimed for the technological lowest common denominator of the Web. This means that `<bigwig>` services will run on almost any combination of browser and server. Thus, we do not require any privileged modules to be compiled into the server or any browser plugins to be accepted by the client. For example, we rely on the pure CGI protocol and we only use the subset of JavaScript that is known to work on all recent versions of both Internet Explorer and Netscape Navigator. This poses more challenges for our code generation in order to maintain a reasonable degree of efficiency. However, the significant upside is that `<bigwig>` is universally available. Regarding platforms, however, `<bigwig>` is currently only supported for servers running Unix or Linux, but porting to Windows NT is merely work waiting to be done. On the client side, we are of course indifferent as to the platform being used.

The core syntax of `<bigwig>` dictates the look-and-feel of programs. We wish to remain as neutral as possible, since our message is not to redefine ordinary programming practices. Consequently, we have chosen the common syntax for declarations, functions, statements, and expressions that seems to be the consensus of languages such as C and Java. Still, the details of `<bigwig>` must be learned anew, but we have not artificially introduced anything surprising.

4 Interactive 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 on which 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. These can be divided into three main paradigms: the *script-centered*, the *page-centered*, and the *session-centered* approaches. Each are supported by various tools and suggest a particular set of concepts inherent to Web services.

The Script-Centered Approach

The most primitive is the script-centered approach, which builds directly on top of the plain CGI protocol. Thus, a Web service is viewed as a collection of loosely related scripts, each of which receives form data as input and produces HTML as output before terminating. Future interactions with the server are made possible by explicitly inserting appropriate links to other scripts in the reply pages. This approach is illustrated in Figure 2.

A prototypical scripting language is Perl [34], 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 graphical designers [10].

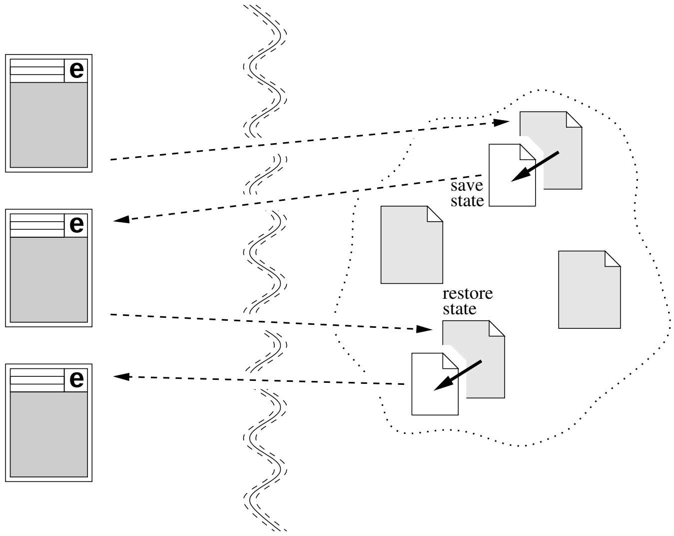

Figure 2: The script-centered approach.

The Page-Centered Approach

The page-centered approach is covered by language such as ASP [14], PHP [4], and JSP [25], where the dynamic code is embedded in the HTML pages themselves. 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.

This approach, which is illustrated in Figure 3, is 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.

However, 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 gigantic code tag that dynamically computes the entire contents. Thus, the two approaches are closely related, and the page-centered technologies are only superior to the degree in which their scripting languages are better designed.

There is generally little support for the issues related to the big picture, such as maintaining states and session threads, handling concurrency control, and providing security.

The Session-Centered Approach

The session-centered approach was pioneered by the MAWL project. 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. Important issues such as concurrency control become simpler to understand in this context and standard programming solutions are more likely to be applicable.

In all three approaches there is generous support for implementing *shared* data,

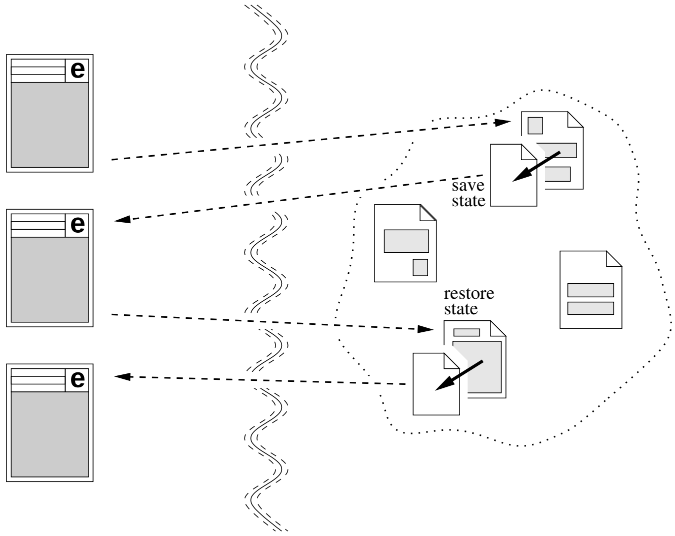

Figure 3: The page-centered approach.

usually in the form of database interfaces. However, only the session-centered approach offers the concept of data that is *private* to an individual thread. In the other approaches this concept must be emulated, sometimes by unsafe means like hidden form fields in the displayed pages.

In MAWL, all HTML templates are placed in separate files and viewed as procedures, 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. A disadvantage is that the HTML pages becomes quite static compared to the flexibility of a `print` statement.

A drawback of the session-centered approach is that some Web services are in actuality more loosely structured. If all sessions are tiny and simply does the work of a server module from the page-centered approach, then the overhead associated with sessions may seem to large. For more involved services, however, the session-centered approach makes programming easier. Figure 4 illustrates the flow of control in this approach.

5 State and Sessions

We now describe the overall structure of a `<bigwig>` service. At this stage, we are not concerned with its actual implementation on the CGI platform.

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 browses 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. In this manner, communication looks like a remote procedure call from the viewpoint of the thread. A simple session that communicates with a client is:

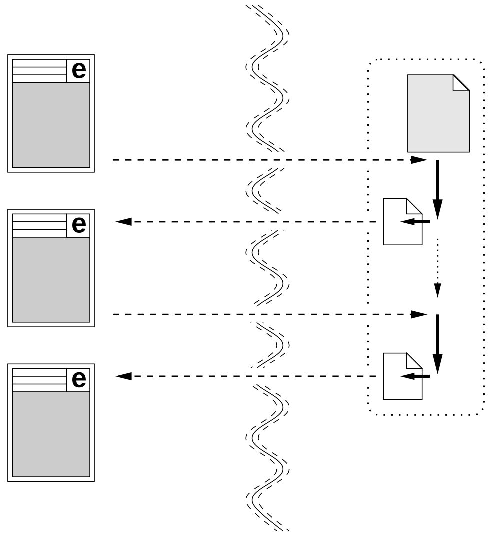

Figure 4: The session-centered approach.

```
service {
  html Please = <html>
    Please state your name:
    <input type=text name=handle>
  </html>;
  html Greeting = <html>Hello <[moniker]>, how are you?</html>;
  session Hello() {
    string s;
    show Please receive[s=handle];
    show Greeting<[moniker=s];
  }
}
```

This example also shows that HTML templates are values in `<bigwig>`; in fact, they are first-class citizens. Furthermore, they may contain *gaps* that are *plugged* with appropriate contents, as described in more detail in Section 8.

It is possible for session threads to *share* some data. A service may contain declarations that are global to all sessions or local to a particular session, including functions and variable declarations. Only variables that are declared with the modifier `shared` can be accessed by more than one thread; all other variables are private to a single thread. This is illustrated by the following example:

```
service {
  int i;               // private variable
  shared int j;        // shared variable
  session Share() {
    i++; j++;
    show (html) "["+i+","+j+"]";
  }
}
```

Since integer variables are initialized to zero, the output from subsequent invocations of the session is `[1,1]`, `[1,2]`, `[1,3]`, and so on.

As services have several concurrent threads, there is a need for synchronization and other concurrency control. Note that contention may occur between threads that are invocations of the same session as well as between threads of different session kinds. Concurrency control in `<bigwig>` is explained in detail in Section 7.

The interaction model described so far is rather rigid, in that the client and server must strictly alternate between being active and suspended. While this is

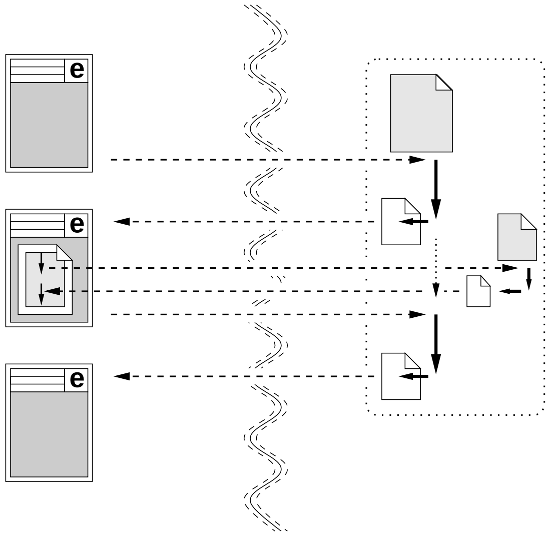

Figure 5: Seslets interacting with applets.

the typical pattern in many services, there are good reasons to allow more liberal interactions. A simple example is a chat room, where new messages should appear automatically, without the client having to resubmit the page being viewed. The essence of this concept are *client-side computations* that are able to contact the server on their own accord.

According to our design principles, we need a domain-specific sublanguage for expressing such computations. However, Java applets are perfect candidates for this task, and we have no desire to reinvent the wheel, let alone a graphical user interface toolkit. Accordingly, we will limit ourselves to providing support for applets to interface with the server. This is realized by the concept of *seslets*, which are sessions with limited capabilities that can be invoked by applets running on the HTML page that is currently being viewed by the client. A seslet is only restricted in the obvious ways that follows from not having an associated client; for example, `show` is clearly not possible from a seslet. Communication between an applet and a seslet is very simple: the applet may supply arguments to the call of the seslet, and the seslet may return a result upon termination. To facilitate the writing of such applets, the `<bigwig>` compiler generates the Java code for a superclass, itself a subclass of `Applet`, that they must inherit from in order to access the available seslets. The interaction pattern of applets and seslets is illustrated in Figure 5.

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

If the client submits the current page, then the browser terminates all running applets. However, running seslets are allowed to run until termination. This is necessary for instance to avoid breaking critical synchronization invariants.

6 The Runtime System

We want to implement our session concept on top of the plain HTTP/CGI platform, which offers a stateless protocol designed for single interactions only. However, it is very easy to make a naive but inefficient implementation of sessions, as sketched

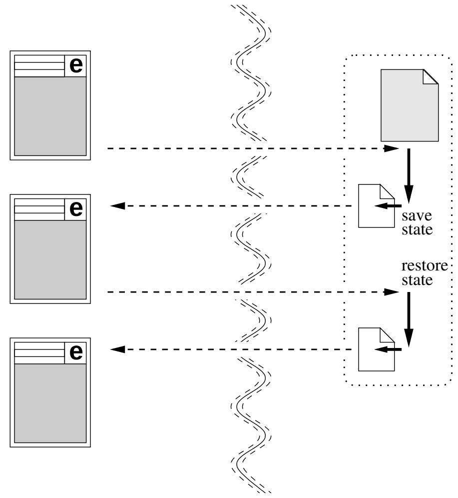

Figure 6: A naive implementation on the CGI platform.

in Figure 6, where the session is an ordinary CGI script. When sending a page to the client, the thread must first save its local state and then terminate in order to relinquish control. When the page is submitted, the thread must be started again and restore its local state in order to continue execution. These steps are prohibitively expensive. A possible remedy is to use something like FastCGI [22], although that requires specialized servers.

However, the HTTP/CGI protocol imposes other practical problems. First, in the session model, it does in general not make sense to "step back in time" using the history buffer that all browsers contain. This is no different from ordinary programs—there is no reason why Web service designs should be constrained by these history buffers. If implemented naively, the history buffer and cache gets cluttered with URLs of obsolete pages, and this will inevitably confuse and annoy the user. Second, the bookmarking feature likewise found in all browsers also becomes hazardous with a naive implementation. Adding a bookmark to a page generated directly by a CGI script causes subsequent recalls of the bookmark to rerun the script instead of just displaying its result again

The `<bigwig>` solution is to use special *connector* processes, which are the true CGI scripts, as illustrated in Figure 7. They only exist to mediate information between the client and the session thread, which is a resident server process. Connectors communicate with the thread using FIFO pipes and terminate in accordance with the CGI protocol when an HTML page is sent to the client. Each session thread is associated a unique URL, which points to an HTML document that contains the latest page shown to the client. Instead of showing the contents directly to the client, we redirect the browser to this URL.

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, and sessions can now be bookmarked for later use.

Also, with this simple solution we can 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 written to the file, reloading is no longer performed. This simple technique may prevent the client from becoming impatient

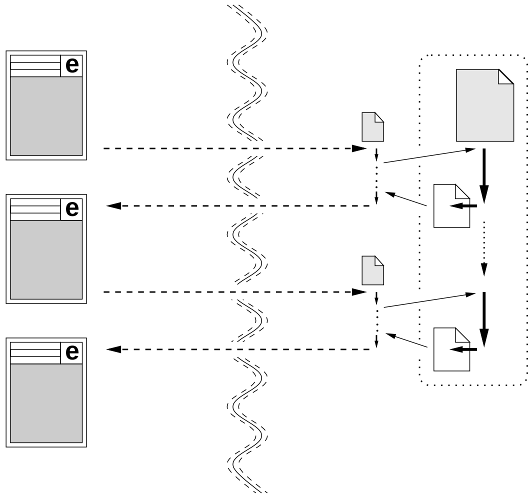

Figure 7: An implementation using connectors.

and abandoning the session.

Session threads that have been suspended for a certain length of time are automatically garbage collected by a special process on the server. Surprisingly many such abandoned threads quickly accumulate on most servers.

Seslets are handled directly by the CGI protocol, since they only involve a single interaction with the applet. The result string is encoded as having MIME type `text/plain`. The Java superclass that is generated by the `<bigwig>` compiler contains all the code necessary to marshal and unmarshal the values being communicated.

The runtime system also contains a *controller*, which is a centralized component that is used for concurrency control. Session threads may enter a number of queues, from which they leave according to some policy that the controller enforces. How `<bigwig>` uses this very general setup is explained in Section 7. The whole `<bigwig>` runtime system is described in further detail in [8].

The implementation we have described is for a single server machine. We are currently experimenting with a scalable architecture, called `<bigwulf>`, which is a small version of a Beowulf cluster [5]. Our plan is to provide a generalized version of the present runtime system that is able to use the structure of `<bigwig>` services to dynamically reconfigure itself to exploit the available resources. The result should hopefully be a cheap and scalable Web server, suitable for moderately sized enterprises.

7 Concurrency Control

We need a mechanism 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 starting. A final 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 the central controller process in the runtime system, which is general enough to enforce a wide range of safety properties [29].

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 sublanguage. We have chosen a well-known and very general formalism, namely *temporal logic*; in particular, we use a variation of monadic second-order logic [33]. 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 [19] to translate the given formula into a DFA that is used 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 non-empty 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. We have also introduced notions of *triggers* and *counters* to gain expressive power beyond regular sets of traces.

This is a very abstract approach that is a bit harsh on the average programmer. However, using our syntax macros it is possible to capture all common concurrency primitives, such as semaphores, mutex regions, the readers/writers protocol, monitors, and so on. The advantage is that `<bigwig>` can be extended with any such constructs, even some that are highly customized to particular applications.

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 and use higher-level abstractions:

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

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;
  }
}
```

Clearly, many programmers would have a hard time constructing that formula. It 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;
  }
}
```

Many `<bigwig>` applications have benefited from such tailor-made constructions.

8 Dynamic HTML Documents

All interactive Web services communicate with clients using HTML documents; most often, their contents are dynamically generated. In script-centered services, the HTML is produced by `print` statements. This is also true of page-centered services, except that some HTML may be in the form of static templates. There are several disadvantages to this approach. First, since the HTML is just the output of a program, there is no way to statically guarantee that it is well-formed. For example, the HTML could be unreadable by the browser. Second, a generated HTML form could contain form fields different from the ones the script expects to receive afterwards, leading to unpredictable behavior. Third, 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. Fourth, since the code for generating HTML is embedded in the script code, it is difficult to separate the tasks of programmers and graphical designers.

MAWL proposes a solution by viewing HTML templates as procedures. The arguments are named gaps in the template, which may be plugged by string values. The results are the form fields that the form contains. By enforcing a simple type discipline on procedure calls, the required static guarantees can be issued by the compiler. Since the templates reside in external files, they can be designed and validated by others than the programmer. We have adopted the template view in `<bigwig>`, but with some significant extensions.

For many applications, the MAWL templates are too rigid. All the HTML documents that are presented to the client are essentially static, since only some string

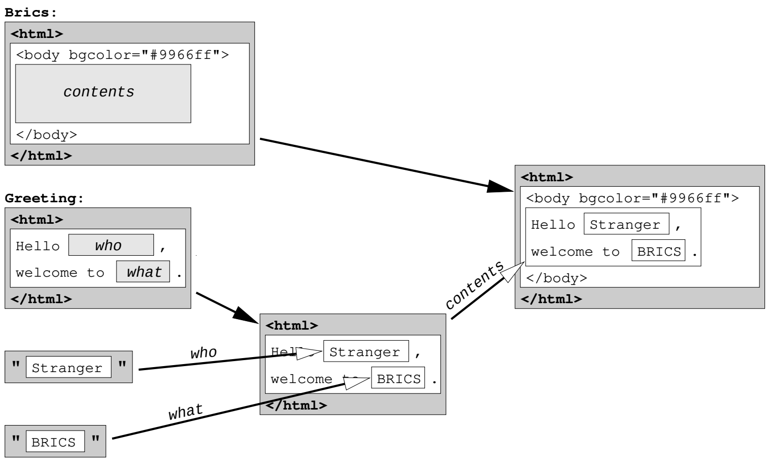

Figure 8: Building a document by plugging gaps.

values can be dynamically replaced. To alleviate this problem, MAWL includes a notion of iteration gaps than can be plugged with lists of string values. In `<bigwig>` we introduce *higher-order* templates, meaning that gaps may be plugged with other templates. This allows the construction of arbitrarily deep HTML structures, but of course complicates the task of type checking. In `<bigwig>`, HTML templates are values on equal footing with strings and integers; thus, they can be stored in variables and passed as arguments to functions. An example illustrating our use of HTML is:

```
service {
  html Brics = <html><body bgcolor="#9966ff"><[contents]></body></html>;
  html Greeting = <html>Hello <[who]>, welcome to <[what]>.</html>;
  session Welcome() {
    show Brics<[contents=Greeting<[who="Stranger",what="BRICS"]];
  }
}
```

The gradual construction of the resulting document shown to the client is illustrated in Figure 8. The following 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);
  }
}
```

It is infeasible to explicitly declare the types of higher-order templates for two reasons. Firstly, all gaps and all fields and their individual capabilities would have to be described, which may become rather voluminous. Secondly, 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 standard monomorphic interprocedural flow analysis [23], 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. We do not guarantee that only valid HTML is produced. However, the task of doing so manually is made significantly easier since all documents are combined from a finite collection of templates. The `<bigwig>` compiler only recognizes a small set of HTML tags and attributes necessary for determining the document types; in the examples, those are indicated as keywords (in boldface). All other tags are merely checked for well-formedness.

We offer explicit support for factoring out the work of graphical designers, since each HTML constant appearing in a `<bigwig>` program may have associated a URL pointing to an alternate, presumably more elaborate version:

```
service {
  session Hello {
    show <html>Hello World</html>@"http://www.brics.dk/bigwig/fancy.html";
  }
}
```

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

We also use HTML templates for pattern matching, as in the following example which shows the daily Dilbert strip without advertisements:

```
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];
    show Template<[source="http://www.dilbert.com"+s];
  }
}
```

The anonymous gaps are used as wildcards in pattern matching. Remaining gaps are always implicitly plugged with empty contents when a page is being shown.

In some situations, the page-centered approach seems more appropriate. 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 citizens, 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 12, 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 small customized Web site. In `<bigwig>` there is support for showing such *document clusters*. The difficulty is to provide a simple notation for specifying an arbitrary graph of documents connected by links. We introduce for an HTML variable `x` 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. Finally, the runtime system must be carefully extended to allow bookmarking and not to interfere with our solution to the "back button problem".

We use an optimally efficient implementation of templates. 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 $\Omega(n)$ rather than $\Omega(2^n)$.

9 Form Field Validation

A considerable effort in Web programming is expended on form field validation, that is, checking whether the data supplied by the client in 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 technique is *server-side* validation, illustrated in Figure 9, where the form fields are validated when the page has been submitted. Such a solution is automatically supported by systems such as ColdFusion [12]. They have several disadvantages, primarily being slow and wasting bandwidth. The alternative is *client-side* validation, illustrated in Figure 10, which usually requires the programmer to use JavaScript in the pages being generated.

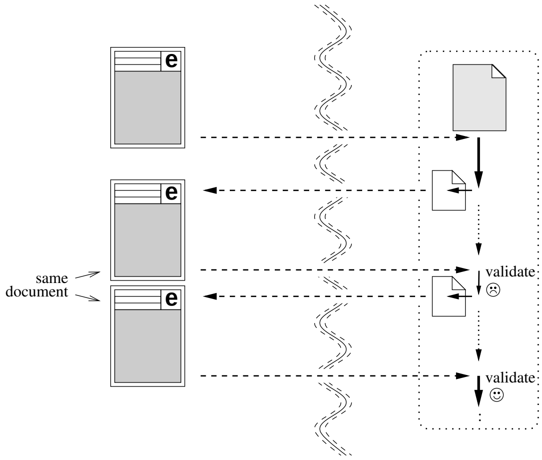

Figure 9: Server-side form field validation.

This is an illusory choice, however, since both techniques clearly must be used at the same time. 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 in JavaScript *and* in the server scripting language.

In `<bigwig>` we have introduced a domain-specific sublanguage for form field validation [7]. It also handles complex interdependencies between form fields and the compiler generates the required code for both client and server. In its simplest form, it allows regular-expression *formats* to be associated to form fields:

```
service {
  format Digit = range('0','9');
  format Number = plus(Digit);
  format Alpha = union(range('a','z'),range('A','Z'));
  format Word = concat(Alpha,star(union(Digit,Alpha)));
  format Name = concat(Word,star(concat(" ",Word)));
  format Email = concat(Word,"@",Word,star(concat(".",Word)));
  session Validate() {
    html Form = <html>
      Please enter your e-mail address:
      <input name=email type=text size=20>
      <format name=Email field=email>
    </html>;
    string s;
    show Form receive[s=email];
  }
}
```

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 performing the server-side double-check. Using “traffic-light” icons, the user is provided with continuous feedback as to whether the string entered so far is a valid prefix. We also allow the usual Perl-style syntax for regular expression, but of course only for 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 [31], 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. The reason is presumably 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 that is 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="x">x
      <option value="y">y
      <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];
  }
}
```

This is a simple language with a clean semantics that appears to handle most realistic situations.

10 Databases

For databases there is an obvious domain-specific sublanguage, namely SQL [11]. Unlike most other Web tools, however, we want our compiler to be able to statically type-check database computations. This precludes us from using the standard technique of building SQL queries as strings that are shipped to the external database engine. Instead, we introduce schemas, tuples, and relations as native concepts in <bigwig>. A simple subset of SQL, modified to blend with the core language, is then directly available in the syntax. The following simple example selects Bart and Lisa from the relation:

```
service {
  schema Person {int age; string name;}
  shared relation Person simpsons;

  session Init() {
```

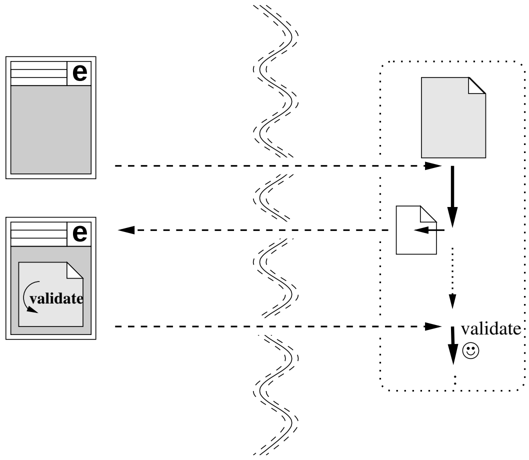

Figure 10: Client-side form field validation.

```
    simpsons = relation{
      tuple{age=38,name="Homer"},
      tuple{age=34,name="Marge"},
      tuple{age=10,name="Bart"},
      tuple{age= 8,name="Lisa"},
      tuple{age= 1,name="Maggie"}
    };
  }

  session Select() {
    relation Person result;
    result = select * from simpsons where (#.age<20 && |#.name|==4);
  }
}
```

A shared relation will always reside in an external database that is associated to the <bigwig> system through an ODBC [28] interface. Currently we have support for DB2 [21], MySQL [13], and a simple default implementation based on the UNIX file system. In contrast, private relations are viewed as data structures and are implemented directly by <bigwig> as part of the thread. As shown in the example, data can seamlessly be moved between shared and private relations. It is also possible to write mixed queries that involve both kinds. The compiler analyzes queries and tries to ship as many computations as possible to the database engine. We are still experimenting with these ideas, but certainly the query above which involves only shared relations and constants will be shipped. As a further optimization, the contents of the private `result` relation need only be retrived from the database engine in a lazy manner.

11 Security

We have not fully analyzed this enormous aspect, but some simple ideas are already in place. First of all, the implementation provides some low-level security to protect the integrity of a service. For example, each session thread is identified by a large random key that is required to gain access. Also, the compiler makes sure to catch all runtime errors to avoid malicious attacks.

At a higher level, the programmer may use a number of security modifiers that applies to communications with the client. The `ssl` modifier instructs the server to use the SSL/HTTPS protocol [17], which collaborates with the client to set up an encrypted tunnel for communication. The `htaccess` modifier protects the displayed page with HTTP Authentication [6] using a supplied password file; **<bigwig>** also supports the `getcookie` and `setcookie` functions to help manage userids. The `selective` modifier restricts access to a session to those clients whose numeric IP addresses (or alphanumeric domain names) match a given set of prefixes (or postfixes). Finally, the `singular` modifier ensures that the client has the same IP address throughout the execution of a session.

The **<bigwig>** compiler performs some simple static analysis that relates to security. Values may be classified as *secret* or *trusted*, and 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 status of a value, there are two functions `trust` and `disclose`; importantly, 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;
    }
  }
}
```

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 secrets. 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 [24] are not particularly original, but merely exist to guide the programmer.

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 [27].

12 Syntax Macros

Our syntax macros are similar to previous work in that the compiler accepts collections of grammatical rules that extend the syntax in which a subsequent program may be written [35]. Almost arbitrary extensions can be defined in a purely *declarative* manner without resorting to compile-time programming. The macros are *terminating*, *hygienic*, and *transparent* to later phases in the compiler. Error messages from later phases in the compiler are tracked through all macro invocations to pinpoint their sources in the extended syntax. We use a novel concept of *metamorphic rules* allowing the arguments of a macro to be defined in an almost arbitrary *meta* level grammar and then to be *morphed* into the host language. Collections of macros are bundled into packages.

Macros are used to provide tailor-made extensions of the language. Consider a frequent scenario, where a service wishes to publish a page that is mostly static. Once in a while the underlying data changes, and the page must be computed again. We can use macros to provide special primitives supporting an efficient implementation:

```
syntax <toplevels> publish <id d> { <exp E> } ::= {
  shared html <d>~cache;
  shared bool <d>~clean;
  session <d>() {
    if (!<d>~clean) {
      <d>~cache = <E>;
      <d>~clean = true;
    }
    show <d>~cache;
  }
}

syntax <stm> touch <id d> ; ::= {
  <d>~clean = false;
}
```

The binary operator '~' provides a mechanism for concatenating identifiers. Using this extended syntax, a service maintaining a high-score list can look like:

```
require "publish.wigmac"
service {
  shared string who;
  shared int score;
  publish HiScore {
    <html>
      The top player is <[(who)]> with <[(score)]> points.
    </html>
  }
  session Update() {
    html Winner = <html>
      Winner: <input type=text name=who><br>
      Score: <input type=text name=score>
    </html>;
    show Winner receive[who = w; score = s];
    touch HiScore;
  }
}
```

A variation updates the cached document if it has reached a certain age:

```
syntax <toplevel_list>
       publish <id d> every <intconst N> seconds { <exp E> } ::= {
  shared html <d>~cache;
  shared time <d>~timestamp;
  session <d>() {
    if (<d>~timestamp==notime || (difftime(now(),<d>~timestamp)><N>)) {
      <d>~cache = <E>;
      <d>~timestamp = now();
    }
    show <d>~cache;
```

```
  }
}
```

The macros are clearly easier to understand and maintain than the corresponding expanded code. It is even possible to use macros in an extreme way, such that an entirely new language is created. 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 students towards this goal, they 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 that 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 study program. Thus it was redesigned in the form of a VDSL, where study fields and requirements are conceptualized and defined directly in pseudo natural language style. This makes it possible for a secretary—or even the responsible faculty member—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 service. The file bachelor.wigmac is only 400 lines and yet contains a complete implementation of the new language, including "parser" and "code generator". Thus, our macro mechanism offers a rapid and inexpensive realization of new ad-hoc languages with arbitrary syntax.

13 Monitors and Managers

Once a service has been programmed and installed, it is important to monitor and manage its performance. Since <bigwig> offers a global view of all components of a service, we have unique opportunities to address these issues.

The <bigwig> compiler automatically generates a combined monitor and manager tool for each service. This tool is written in <bigwig> itself using a specially designed VDSL. Apart from the aesthetic pleasure, the advantage of this self-referential technique is that the generated code easily can be customized by the <bigwig> programmer who may want a different layout or some additional functionality.

The management tool performs several tasks. The status of all running threads can be viewed, and any thread can be paused or killed. Similarly, the status of the concurrency controller is visible, and the contents of the process queues can be manipulated. Also, the shared data can be inspected and modified. Finally, in order to maintain the service, there is support for blocking individual sessions, such that no new threads may be created by outside clients.

The management tool runs as an independent service that may be started and stopped at any time. It uses special knowledge about the implementation of the runtime system to gain access to the internal data. For security, access to the manager service is by default restricted to clients running on the Web server, but remote management is certainly an option.

A future project is to construct a tool that generates statistics about a service. Clearly, there are countless measures that could be computed and visualized by digesting the service logs that we already today maintain. Examples include total hits, hits per session kind, duration of session threads, usage of HTML templates, and more general client patterns. Many such things have already been done by MAWL.

14 Conclusions

The <bigwig> project has identified many aspects of interactive Web services and has provided solutions in a coherent framework based on programming language theory. At the same time, the <bigwig> language is a major case study in applications of the domain-specific language design paradigm. The project will continue into the foreseeable future. The development can be followed at the <bigwig> home page at http://www.brics.dk/bigwig/, and we encourage contributions and cooperations.

Acknowledgments

Tom Ball provided us with extensive and very helpful information about experiences with the MAWL language. Anders Sandholm has been a key participant during his PhD studies at BRICS. Mikkel Ricky Christensen and Steffan Olesen have worked tirelessly as student programmers during the entire project. Niels Damgaard, Uffe Engberg, Mads Johan Jurik, Lone Haudrum Olesen, Christian Stenz, and Tommy

Thorn have provided valuable feedback and suggestions. We also appreciate the efforts made by the participants of the WIG Projects course in Spring 1998.

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] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *International Conference on Parallel Processing*, 1995.

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

[7] Claus Brabrand, Anders Møller, Mikkel Ricky, and Michael I. Schwartzbach. Powerforms: Declarative client-side form field validation. *World Wide Web Journal*, 3(4), 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 and Michael I. Schwartzbach. Growing languages with metamorphic syntax macros, October 2000. submitted for publication, available from `http://www.brics.dk/bigwig/`.

[10] 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.

[11] C.J. Date and Hugh Darwen. *A Guide to the SQL Standard*. Addison-Wesley, 3rd edition, October 1992.

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

[13] Paul DuBois. *MySQL*. Macmillan Technical Publishing, 1999.

[14] Alex Fedorov et al. *Professional Active Server Pages 2.0*. Wrox Press, 1998.

[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] Nils Klarlund and Anders Møller. *MONA Version 1.3 User Manual*. BRICS Notes Series NS-98-3 (2nd revision), Department of Computer Science, University of Aarhus, October 1998.

[20] 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.

[21] IBM Corp. *DB2 Application Programming*. IBM, 1995. available from `http://www-4.ibm.com/software/data/db2`.

[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] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. In Alan Mycroft, editor, *SAS'95: Static Analysis*, volume 983 of *Lecture Notes in Computer Science*, pages 314–330, Glasgow, September 1995. Springer-Verlag.

[25] Eduardo Pelegri-Llopart. JavaServer Pages specification, version 1.2 proposed final draft. Sun Public Draft, October 2000.

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

[27] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 36(9):1278–1308, 1975.

[28] Roger E. Sanders. *ODBC 3.5 Developer's Guide*. McGraw-Hill, 1998.

[29] 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.

[30] Anders Sandholm and Michael I. Schwartzbach. A domain specific language for typed dynamic documents. In *Principles of Programming Languages, POPL'00*. ACM, 2000.

[31] Sebastian Schnitzenbaumer, Malte Wedel, and Dave Raggett. XHTML extended forms requirements. W3C Working Draft, March 1999. `http://www.w3.org/TR/xhtml-forms-req.html`.

[32] Guy Steele. Growing a language. OOPSLA invited talk, 1998.

[33] Wolfgang Thomas. *Handbook of Theoretical Computer Science*, volume B, chapter Automata on Infinite Objects, pages 133–191. MIT Press/Elsevier, 1990.

[34] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, July 2000.

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

Recent BRICS Report Series Publications

RS-00-42 **Claus Brabrand, Anders Møller, and Michael I. Schwartzbach.** ***The <bigwig> Project*. December 2000. 25 pp.**

RS-00-41 **Nils Klarlund, Anders Møller, and Michael I. Schwartzbach.** ***The DSD Schema Language and its Applications*. December 2000. 32 pp. Shorter version appears in Heimdahl, editor, *3rd ACM SIGSOFT Workshop on on Formal Methods in Software Practice*, FMSP '00 Proceedings, 2000, pages 101–111.**

RS-00-40 **Nils Klarlund, Anders Møller, and Michael I. Schwartzbach.** ***MONA Implementation Secrets*. December 2000. 19 pp. Shorter version appears in Daley, Eramian and Yu, editors, *Fifth International Conference on Implementation and Application of Automata*, CIAA '00 Pre-Proceedings, 2000, pages 93–102.**

RS-00-39 **Anders Møller and Michael I. Schwartzbach.** ***The Pointer Assertion Logic Engine*. December 2000. 23 pp. To appear in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '01 Proceedings, 2001.**

RS-00-38 **Bertrand Jeannet.** ***Dynamic Partitioning in Linear Relation Analysis: Application to the Verification of Synchronous Programs*. December 2000.**

RS-00-37 **Thomas S. Hune, Kim G. Larsen, and Paul Pettersson.** ***Guided Synthesis of Control Programs for a Batch Plant using* UPPAAL. December 2000. 29 pp. Appears in Hsiung, editor, *International Workshop in Distributed Systems Validation and Verification. Held in conjunction with 20th IEEE International Conference on Distributed Computing Systems (ICDCS '2000)*, DSVV '00 Proceedings, 2000.**

RS-00-36 **Rasmus Pagh.** ***Dispersing Hash Functions*. December 2000. 18 pp. Preliminary version appeared in Rolim, editor, *4th. International Workshop on Randomization and Approximation Techniques in Computer Science*, RANDOM '00, Proceedings in Informatics 8, 2000, pages 53–67.**

RS-00-35 **Olivier Danvy and Lasse R. Nielsen.** ***CPS Transformation of Beta-Redexes*. December 2000. 12 pp.**