Mawl: Integrated Web and Telephone Service Creation

David Atkins, Thomas Ball[*] Thomas Baran, Michael Benedikt,
Kenneth Cox, David Ladd[†], Peter Mataga, Carlos Puchol,
J. Christopher Ramming[‡], Kenneth Rehor, Curtis Tuckey

Bell Laboratories

`mawlers@research.bell-labs.com`
`http://www-spr.research.bell-labs.com/~mawl/`

March 4, 1997

Abstract

Mawl is a language and compiler for programming form-based services in a device-independent manner. PML is a markup language and middleware for controlling and programming various interactive voice response (IVR) platforms using standard web infrastructure. The combination of Mawl and PML allows the creation of interactive services that users can access via a web browser or telephone. The ability to create such services in a single environment appears to be unique.

The Mawl language separates the specification of service logic from the specification of the user interface to be presented on a device. As a result, one can easily code a service that is accessible via a web browser and with minor modifications to only the user interface specification, make the service accessible via an IVR platform. Mawl draws on the principles of application language engineering to facilitate not just service creation but the entire software development life cycle, improving a service provider's ability to develop, monitor, analyze, administer, and modify form-based services.

The Phone Markup Language (PML) is a language and an architecture that enables telephone access to web services and the easy creation of IVR services using the web paradigm. The PML language is a dialect of HTML specialized to describe content for interpretation over a telephone. HTML or PML documents are served over a telephone by standard IVR platforms (or audio processing nodes); as in the hypertext model, the documents themselves may reside on any web server in the network, or may be dynamically generated. The PML middleware takes care of the tasks of fetching documents from the Internet and instructing audio processing nodes to "play" them. As a result, to program an IVR service, a programmer need only deal with a simple markup language and is insulated from the details of the network and particular audio processing nodes.

1 Introduction

The World Wide Web, originally a simple mechanism for document distribution, is rapidly becoming the standard infrastructure for diverse interactive applications. The ubiquity of the

[*]Correspondence contact: tball@research.bell-labs.com, Room 1G-359, 1000 E. Warrenville Rd., Naperville, IL 60566. Telephone (630) 979-4291. [†]Ladd's current address: Spyglass, 1240 E. Diehl Rd, Naperville, IL 60563. [‡]Ramming's current address: AT&T Research, 600–700 Mountain Avenue, Murray Hill, NJ 07974-0636.

hypertext metaphor and browser technology make a web interface attractive for a variety of business uses: intranet applications for internal processes, such as project management; internet applications that provide services to external customers, such as personal communications management or customer care; and software bundled with products, such as equipment management interfaces. (For an example of a web service, see Sidebar A.)

Mawl is an application-oriented language for producing complex interactive web services [LR95]. Mawl is supported by a number of tools, including a compiler. The design of mawl draws on the principles of application language engineering to facilitate not just service creation but the entire software development life cycle. Mawl improves a service provider's ability to develop, monitor, analyze, administer, and modify web services.

The core of a mawl service is a centralized service logic, written in a language designed to express flow of control, state management, and the content of information flow between service and user. The presentation details of the user interactions are encapsulated in templates written in an extension of the HyperText Markup Language (HTML) [BLC95] that allows dynamic customization of their content.

Mawl also provides new functionality to web services by allowing the integration of alternative user interfaces, such as the telephone. The Phone Markup Language (PML) is a dialect of HTML specialized to describe content for interpretation over a telephone. We have developed a platform that allows HTML and PML documents to be served over a telephone by special purpose audio browsers; as in the hypertext model, the documents themselves may reside on any web server in the network, and may be dynamically generated.

The combination of mawl with PML permits the creation of interactive services that users can access via web browser or telephone. The ability to create such services in a single environment appears to be unique.

1.1 Web Services and Mawl

The stateless protocol underlying the web (the HyperText Transmission Protocol or HTTP) was designed to support a single interaction in which a web server responds to a request sent across the network by a client application [BL95]. When requests are processed independently of one other, this model works well, as the proliferation of hypertext archives and simple information-collection applications shows. On the other hand, complex services require the maintenance of state and control-flow information across multiple user interactions, and this violates the assumption of independence. Since HTTP is stateless, service developers who use Common Gateway Interface (CGI) programs, invoked by client requests, are typically forced to construct their own mechanisms for remembering the history of the interaction with the client.

As shown in Figure 1, mawl eliminates this problem by providing service developers the illusion of a *centralized, sequential service logic.* From a programmer's point of view, a mawl service is composed of *sessions,* running on a web server, that interleave server-side processing with user interactions. User interactions go through agents called *arbiters* which dynamically parameterize *document templates*, send them to the user, and receive a responses which are then sent back to the session. The mechanism by which separate HTTP requests from the client are related is never explicitly addressed by the service creator; the mawl compiler provides this infrastructure. The document templates are written in a dialect of HTML that allows dynamic customization of their content, according to the current state of the service.

The separation of service logic from presentation to the user has a number of software engineering advantages. The control and information flow of all interactions with a service are specified in a central location (the session), rather than being implicit in the relationships of a

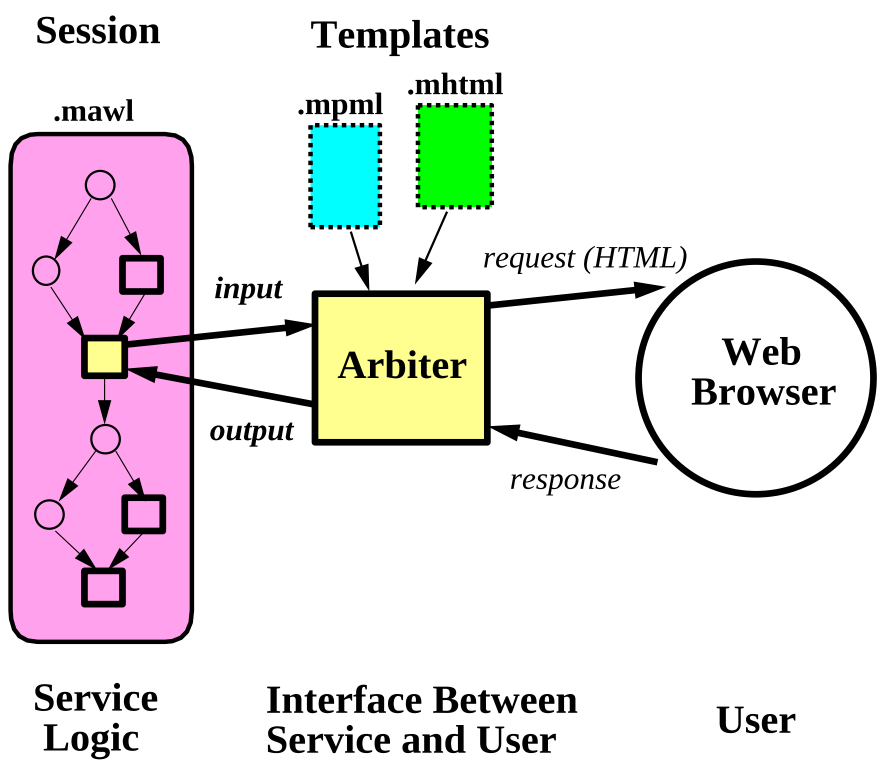


Figure 1: The three major abstractions in the mawl service architecture: sessions, arbiters, and templates.

scattered set of scripts, as is common in much web service programming. Presentation details irrelevant to service logic are specified separately in document templates, allowing developers with different skills to work on disjoint aspects of the service. The abstractions introduced by mawl allow a flexible strategy for platform independence, since the compiler and runtime system may be configured to produce different implementations of the same service logic.

Among other benefits, mawl provides:

- compile-time checking of new or modified services;
- separate update of service logic and presentation;
- instrumentation that allows dynamic tracing and visualization of service behavior and usage patterns.

It is also possible to generate administrative interfaces that allow debugging, monitoring, and modification of dynamic data.

1.2 Multiple Browsers and PML

One effect of the separation of service logic and presentation detail is the ability to accommodate existing "browsers," such as email readers and telephones, as well as future browsers, such as smart telephones or personal data assistants. The motivations for looking at the telephone as an interface to the web are fairly obvious: it is the most ubiquitous access device on the planet, is available to the most mobile of users, and has a huge user base. In addition, the issues arising in telephone access include many of those that will arise in the course of making services available to an increasingly heterogenous collection of devices; indeed, it is hard to imagine two user interfaces more dissimilar than the telephone and the graphical web browser.

The Phone Markup Language (PML) is a language and an architecture that enables telephone access to web content and the easy creation of interactive voice response (IVR) services using the web paradigm. The PML language is a dialect of HTML specialized to describe content for interpretation over a telephone. PML documents are served over a telephone by special-purpose audio browsers; as in the hypertext model, the documents themselves may reside on any web server in the network, or may be dynamically generated. Thus, in order to program an IVR service, a programmer need only deal with a simple markup language. The IVR programmer is insulated from the details of the network and particular audio nodes, just as the web programmer is insulated from details of networks and browsers.

1.3 Application Focus

Mawl is useful for the development of simple web services as well as more complicated ones, but the machinery is particularly advantageous for applications that have one or more of the following characteristics: multiple user interfaces; service analysis and administration; extended transactions.

- *Multiple user interfaces*

  Many services, especially those designed for use by external customers, require integration of different interfaces. This is particularly so for web services that replace legacy systems based on telephone access to automated services or human agents: any new system must still offer a large subset of the existing access functionality. The trend toward mobility also motivates multiple user interface support. Furthermore, enhancements of telephone-based services may require the richness of a graphical user interface. For example, for a personal communication service, the best interface for configuring user preferences is probably a web browser.

  Since PML is designed to be used within the standard web infrastructure, it is possible to integrate HTML and PML services. If mawl is used as the service logic programming environment, it is particularly easy to offer web and phone access to the same service. Thanks to mawl's separation of service logic and presentation, different user interfaces may share some or all of their service logic, and access the same state information and host language code. Together, mawl and PML make it possible to build services that integrate multiple user interfaces in a systematic, reusable and maintainable way.

- *Administration and maintenance*

  Commercial services require monitoring, modification, and other operational management. Services of business importance must reliably provide operations administration and maintenance features such as logging, performance data collection, error reporting, online administration and data update, and software upgrade of running services. Because mawl is built upon an infrastructure that maintains the complete service state, and because mawl applications are systematically generated from high-level service specifications, we have been able use mawl to automate these capabilities significantly.

- *Extended transactions*

  Complex web services require multiple, sequential, and related interactions to occur over extended periods of time. As part of such transactions, the user may wish to suspend a session and resume it later, or the service may transfer control of the session to someone else. Since mawl maintains all state on the server, including the program counter for each session, it is possible to design language features and recovery mechanisms to support extended transactions.

Section 2 describes the mawl language and its context in the Application Language Engineering research effort at Indian Hill. Section 3 discusses the goals and architecture of PML and how the combination of mawl and PML enables the creation of services accessible by both web and telephone. Section 4 describes some of the applications that have been created with mawl and PML. Section 5 reviews related work and Section 6 describes current and future research directions.

2 Mawl and Application Language Engineering

We first illustrate the main points of the mawl service architecture with a small example, and then describe the benefits of application language engineering in the context of mawl.

2.1 Mawl Architecture and Implementation

Figure 2 shows a simple mawl service illustrating the three main abstractions of mawl: sessions, arbiters, and templates.

- *Sessions*

  This service has one *session,* which defines a sequence of interactions with a user. The session interacts twice with the user, first prompting for the user's name and then greeting the user and displaying the date. The service logic, written in mawl, is shown in Figure 2(a); this code is contained in a file named `Date.mawl`.

  Mawl provides a persistence model that allows programmers to specify the type of storage required for the variables involved in a mawl program. In the example, `access_cnt` is a persistent variable. Variables may exist on a per-session basis (as declared by the keyword `auto`) or persist over all session executions (as declared by the keyword `static`). Mawl has several concurrency control mechanisms for static variables shared across instances of a session. Although not shown in this example, the mawl has standard imperative constructs for looping, conditional control-flow, procedure calls, and so on.

  The only way for the service logic to interact with the user is through simple input/output devices, called *arbiters,* which are accessed through a well-defined interface. Each arbiter has certain *input variables,* which are supplied by the service logic, and *output variables,* which are extracted from a user's response.

- *Arbiters*

  An arbiter is declared in the mawl service logic as a function from a record containing the input variables to a record containing the output variables. In the example of Figure 2(a), the arbiter `GetName` is declared as having type `{} -> {id}`.

  At run time the service provides the arbiter with a record containing the input variables, and in return receives a record containing the output variables. This is shown in Figure 2(a), where the service first provides `GetName` with its required (empty) input record and receives back a record containing the `id` variable. This record is stored in the variable

(a) Date.mawl:

```
static int access_cnt = 0;            // how many hits?
session todaysDate {
   auto arbiter {} -> { id } GetName;          // arbiter to get a user's name
   auto arbiter { id, date } -> {} ShowInfo;  // arbiter to show name and date

   auto { id } name = GetName.put({}); // get the user's name into id
   ShowInfo.put({name.id, date()});    // show user their name and date
   access_cnt = access_cnt + 1;        // they completed the service!
}
```

(b) GetName.mhtml:

```
<HTML>
   <HEAD><TITLE>Get-Name Page</TITLE></HEAD>
   <BODY>Enter your name: <INPUT NAME=id></BODY>
</HTML>
```

(c) ShowInfo.mhtml:

```
<HTML>
   <HEAD><TITLE>Time-Of-Day Page</TITLE></HEAD>
   <BODY>Hello, <MVAR NAME=id>.
         Today's date is <MVAR NAME=date></BODY>
</HTML>
```

Figure 2: A mawl service (a) that asks the user for a name through the arbiter `GetName` and then invokes the arbiter `ShowInfo` to display the name and the date. The HTML templates corresponding to the arbiters are (b) `GetName.mhtml` and (c) `ShowInfo.mhtml`. The service also tracks the number of users who have used the service.

`name`. The service then supplies `ShowInfo` with an input record containing two components, `id` from the `name` variable and `date` from a call to the function `date()`; the empty record returned by this arbiter is ignored.

- *Document Templates*

  Each instance of an arbiter may have zero, one, or many document templates associated with it. The value input to an arbiter is used to generate a document by parameterizing a document template with the value. The document templates are specified separately, in the user-interface languages appropriate to the various interfaces. Figure 2(b) and (c) shows document templates written in the language MHTML. MHTML is an extension of HTML that is used for creating document templates. When a running service detects that a screen-based browser is being used, it uses the MHTML templates to generate HTML, which is sent to the browser. In MHTML, the values of an arbiter's input variables may be accessed using the `MVAR` mark, among others. This mark indicates substitution of the value of the input variable into the generated HTML. Output variables are represented by HTML user-input marks such as `INPUT` and `SELECT`; the `NAME` attribute of these marks is the name of the arbiter output variable.

  Figure 2(b) shows the content of the file `GetName.mhtml`, which is the MHTML template

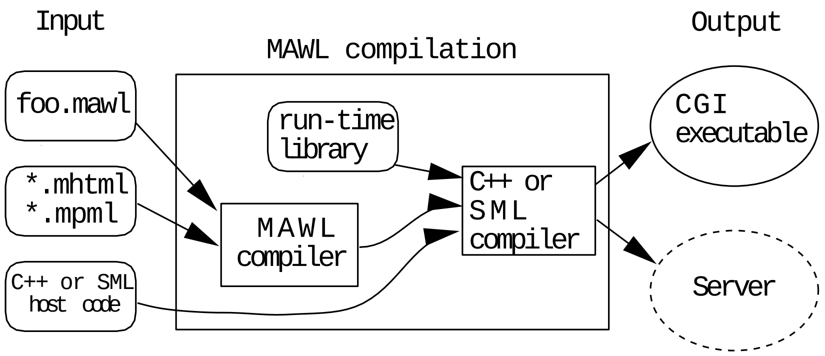


Figure 3: The mawl compilation process.

associated with `GetName` arbiter (a template may be associated with an arbiter in various ways—the linkage here is by the common name). This template contains no uses of the `MVAR` mark, so its input type is {}, while it contains one `INPUT` mark named `id`, so its output type is {`id`}. This is consistent with the type signature of the associated `GetName` arbiter. Similarly we can deduce that the template in Figure 2(c) agrees with the `ShowInfo` arbiter, since the template has input type {`id, name`} (as there are `MVAR` marks using those variables), and has output type {} (as there are no `input` marks).

The ability to determine the type of a template both from the declaration of the type of an arbiter in the service logic and from inspection of the template is very useful. At compile time, the two types can be compared to ensure that they are consistent, thus identifying a large class of run-time errors.

Figure 3 shows the mawl compilation process. Mawl relegates general-purpose computation to a host language. There are three inputs to the mawl compilation process: the logic of the service, written in mawl; document templates, written in MHTML or MPML; and support code written in the host language. The mawl compiler takes the first two inputs, which pass through the traditional compiler steps of lexing, parsing, semantic checking, and code generation. The mawl compiler back end generates code in the host language. Then, this code is compiled by the host language compiler along with the input support code. Currently supported host languages are C++ and Standard ML of New Jersey [MA91]. A compiled mawl service is linked with a run-time library to form a complete executable. The service can be compiled either as a CGI-executable or as a stand-alone server. By choosing various run-time options, service administrators can change the storage management model and user interface, as well as a host of other features that configure a mawl service, all without recompilation of the underlying service code.

Sessions are the entry points by which users enter a mawl service. Figure 2 gave an example of a service with a single session named `todaysDate`. Once the user enters a session, a new *instance* of the session, with fresh copies of local variables, is created. A session interacts with the user through a sequence of arbiters which present dynamically instantiated documents to the user, which are then filled in by the user and submitted back to the session.

In services compiled as CGI-executables, when a session sends out a document (via an arbiter) execution of the session is suspended and the local state is stored on disk or in a database. When a response is received, execution picks up from the point of suspension, with the local state restored. Execution of the session continues until another arbiter is invoked, sending another document to the user. Once a session ends, the storage for local state is released.

2.2 The Benefits of Application Language Engineering

As the example in the previous section illustrates, the core of a mawl service is a statically typed service logic, expressed in a language that frees service designers from the details of state and concurrency management. This core logic is combined with a specific user-interface technology to generate a service, as depicted in Figure 1.

The partitioning of software production into disjoint application-oriented languages is the central component of an approach to software production, called *application language engineering,* that addresses all phases of the software life cycle: requirements, design, coding, and maintenance. By addressing a problem narrower than general-purpose programming, an application-oriented language can offer a more complete solution to a software engineering problem, or a solution which addresses more precisely the goals of a software project.

The benefits of application-oriented languages come, broadly speaking, from two basic principles of language design: abstraction and restriction. Appropriate abstraction is the basis for improved support for requirements, design, coding, and maintenance. Appropriate restriction is the basis for analysis, and hence for verification, modification, and maintenance.

Mawl arranges for the programmer to deal with the abstraction of sequential programming rather than CGI-script management. State management and concurrency are abstracted as program variables shared by sessions. Separation of control flow and presentation is provided by mawl's arbiter abstraction, the input/output interface between the service logic, and the document templates. Software engineering practice has shown this kind of separation to be useful in maintaining and scaling large software projects.

Analyses enabled by mawl's separation of service logic from user interface contribute directly to service administration, are useful to service developers, and enhance the mawl infrastructure itself. Some examples:

- *Alternative implementations.* An appropriate level of abstraction allows for various alternative implementations. As illustrated in Figure 3, with mawl one can generate a CGI-based implementation, or alternatively a threaded-server implementation.
- *Type checking.* Application languages offer the opportunity for application-specific static checking. For instance, mawl's type checking ensures that the input/output characteristics of the HTML and PML templates are consistent with the service logic.
- *Logging.* Sessions, with their associated server-side state, and arbiters, which are centralized points at which services interact with users, make it possible for programmers and administrators to track users as they access mawl services. Session state may be inspected or changed on the fly, and control of a session may be transferred administratively or programmatically.
- *Visualization.* With mawl, a compiler option allows the display of the relationships between pages and the dynamic behavior of a service, whereas it might not be possible to analyze a collection of CGI scripts that implement the same service. (Sidebar B describes this capability in the context of an example.)

In addition to these analyses, the mawl abstractions also support various software engineering phases:

- *Design.* Application languages should enforce a software architecture by placing appropriate restrictions on the design space. The logic of complex services often is designed pictorially using graphs to show possible flows of events (as in Figure 6). These graphs have rather direct translations into mawl programs. Furthermore, mawl helps engineers trace from requirements to design, since one can usually identify the service logic that satisfies a given requirement for a service.

- *Coding.* The skills needed to program interactive web services are different from the skills needed to design HTML pages. Separating service programming from user-interface design allows specialists in each area to use their abilities and toolsets to full advantage. As shown in Figures 2 and 8, mawl cleanly separates these components.
- *Testing.* Because the service logic of mawl is centralized and separate from the user interface, it can be executed and tested independently of any document templates. Thus the logic of the services can be tested just once for many different user-interface technologies. Since the executable code for a service is generated by the mawl compiler, it is possible to instrument a service to generate execution traces for the running service.

3 PML: The Phone Markup Language

While mawl is a language and its implementation a compiler, we use the term PML to denote both an architecture for making web content accessible via the telephone, and the Phone Markup Language itself. There are three main issues that PML addresses:

- *Access architecture.* How does one connect the public telephone network to the web? We introduce here the *PML architecture*, which features two distinct intermediaries between the telephone user and the content host: an *audio processing node* (APN) and a set of software entities called *PML interpreters.*
- *User interface for telephone access to the web.* How does one present existing web content to a telephone user? Although helping users navigate through rich content on an interface as poor as a telephone is inherently difficult, we describe some ideas on how to leverage the structure of web content in the telephone setting.
- *Language support for telephone presentation.* In addition to developing techniques for navigating through rich web content, we wish to allow content-providers a means through which to structure new content or restructure old content for presentation over a telephone. We present a brief glimpse into the PML language.

A PML interpreter requests documents over the internet and interprets them for the telephone; it is not in any way dependent upon mawl. On the other hand, the combination of mawl and PML yields services that are accessible by both the web and the telephone. From the viewpoint of mawl, a request for a document comes from a *user agent.* If the user agent is a web browser, the service responds with HTML. On the other hand, if the user agent is a PML interpreter, the service can send back PML (presumably tuned to the telephone interface) or ordinary HTML (if no PML is available). This is illustrated in Figure 1.

3.1 Access Architecture

The PML architecture shields service programmers from the machines and processes that perform complex tasks such as text-to-speech conversion, voice recognition, document interpretation, and various interactions with the telephone network. As shown in Figure 4, the PML architecture consists of two main components—audio processing nodes, and PML interpreters—in the PML layer. These components connect the telephone network to the internet, as described below.

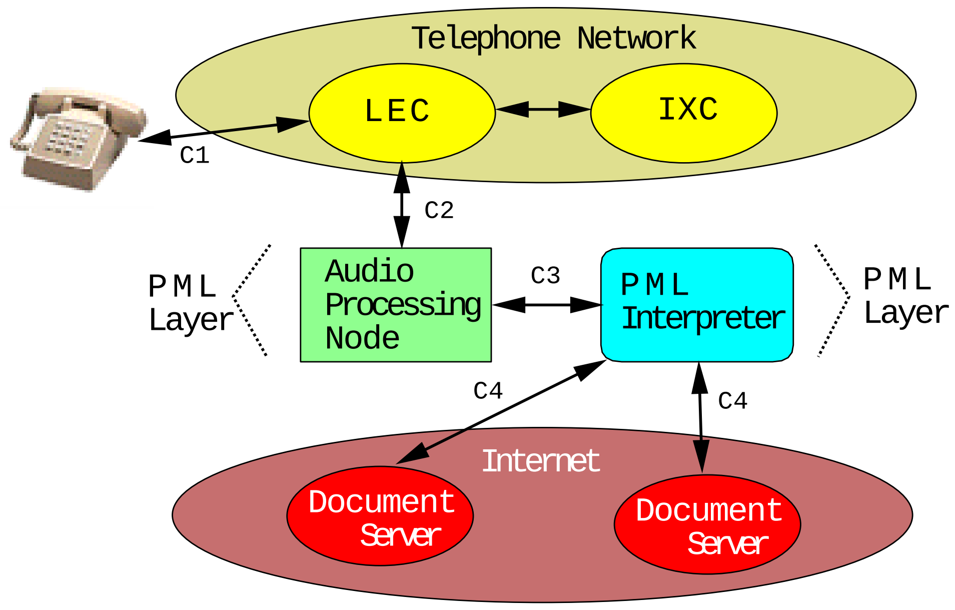


Figure 4: High-level view of the PML architecture.

- *Audio processing nodes* contain hardware for low-level interactions with telephone users, including text-to-speech conversion, touchtone signal interpretation, speech recognition, voice recording, and playing audio files. An APN might reside in a telephone company's network or on a service provider's premises. On the user side, an APN terminates dedicated lines into the telephone hardware that it controls. In our current platform, an APN is either an Lucent Conversant Voice Information System, or a Lucent AYC50-based PC running Unix. The APN communicates with a PML interpreter.
- *PML interpreters* are software processes that mediate between text documents and telephone users. They request and receive HTML or PML documents from document servers (C4). The PML interpreter can be either be co-located with the APN or located elsewhere in the telephone or computer network. Document servers store textual descriptions of information that can be interpreted for the telephone. The document servers are ordinary HTTP daemons that reside on any web server within the internet. Documents are associated with URLs using the usual conventions. The PML interpreter "reads" these documents to a caller by directing an APN (C3) to play audio documents or a string of words and to receive voice and touchtone signals (C2-C1). The communication protocol between a PML interpreter and an APN (C3) is defined by an API so that the same PML interpreter can communicate with different audio processing nodes.

To provide a telephone service using the PML platform, a service provider must obtain a telephone number, terminated at the APN, that will be associated with the URL of the service. An APN operating in conjunction with a PML interpreter forms the telephonic equivalent of a web browser: the PML interpreter requests and interprets documents, which are then "displayed" by the APN.

We use Figure 4 to illustrate a simple scenario. A telephone call (C1) to a line terminated by an APN (C2) results in the invocation of a PML interpreter with the associated URL (C3). Interaction then proceeds according to the nature of the document at that URL. A request over the internet (C4) retrieves a document that the PML interpreter parses and interprets, sending commands (C3) back to the APN to convert text to speech, collect touchtone signals, and so on. As a result of this interaction, the PML interpreter may fetch and play another document.

3.2 A User Interface for Telephone Access to Web Content

The PML interpreter supports two distinct modes of user interaction: a browsing mode in which the user has a set of commands similar to those found in graphical web browsers, and a service mode in which the user is always presented with a menu of choices.

Browser mode is the default when browsing HTML content that does not have enough structure to be presented in a hierarchical manner. In this case, the PML interpreter breaks the HTML page into paragraphs and reads the document in a linear fashion. The user has the capability to search and move around within the document, list the links on a page and visit a link, go back to a previous page, and so on.

Service mode is used when the document is a PML document or the HTML has certain structure. In this case, the user has the illusion of an IVR service; the fact that the user is actually browsing a page from the web is hidden. Figure 5 shows an example of a well-structured HTML page that the PML interpreter can read in a hierarchical manner. It first reads the top-level `h1` mark ("What do you want to hear today?") and then presents a choice of the three `h2` marks ("Press 1 for Mawl Services, 2 for Web Pages") to the user. Suppose the user selects "Mawl Services." At the level of links, the user would again be presented with a menu of choices ("Press 1 for the Lunchbot, 2 for the BookBot, 3 for the BulletinBot"). Choosing one of these links would cause the PML interpreter to access the web service, thus retrieving another page to read.

```
<html><head><title>Top level</title></head>
<body>

<h1>What do you want to hear today?</h1>

<h2>Mawl Services</h2>
  <ul>
    <li><a href="..."> The LunchBot </a>
    <li><a href="..."> The BookBot </a>
    <li><a href="..."> The BulletinBot </a>
  </ul>

<h2>Web Pages</h2>
  <ul>
    <li><a href="http://www.lucent.com"> Lucent Technologies Home Page </a>
    <li><a href="http://www.yahoo.com/"> Yahoo's Home Page </a>
    <li><a href="...">New York Times Front Page</a>
  </ul>

</body></html>
```

Figure 5: An example of an HTML page that can be read in a menu-oriented fashion by the PML interpreter.

3.3 Language Support for Telephone Presentation

The previous example might lead one to believe that HTML itself is sufficient for constructing IVR services. However, our experience has shown that to produce acceptable interactive voice response services with a minimal amount of telephone-specific code, it helps to have additional marks for describing how structured documents should be presented over the telephone. This is primarily because graphical user interfaces and textual (`tty`) user interfaces have "screen memory" to guide users through complicated logic and remind them of their choices. An audio user interface like the telephone is memoryless in this sense, and so more has to be built directly into the document in terms of markup.

The Phone Markup Language is an extension of HTML that has special marks for the telephone interface. Just as with HTML, a PML document can be retrieved by any browser that requests its URL. For example, here is a fragment of PML that specifies how many times to replay a menu selection (3) if the user does not respond in time (30 seconds). The example also includes a prompt to be read if the user makes an illegal selection ("Please choose again").

```
<h2 timeout=30s retries=3 error="Please choose again.">Mawl Services</h2>
  <ul>
    <li><a href="..."> The LunchBot </a>
    <li><a href="..."> The BookBot </a>
    <li><a href="..."> The BulletinBot </a>
  </ul>
```

PML also includes markups to control call processing capabilities on the APN, if available. For example, a mark can be used to direct the APN to place a call.

4 Applications

A crucial part of our technical plan is to build applications that test our technology, inspire new research, and provide value to Lucent. The feedback from prototyping realistic applications continues to be a critical component of our research effort.

- *Intranet Tools.* The BookBot, LunchBot, BulletinBot, and PostBot are examples of departmental services in daily use at the Software Production Research Department. We use the BookBot to order books from a variety of sources and to track the orders. The LunchBot allows members of the department to order meals for our weekly department luncheons (see Sidebar A). The BulletinBot allows individuals to administer their schedules through the web, and lets others inspect them by web or telephone. The PostBot is a telephone interface to the Post database. The service allows users to enter an employee's name using touchtone or voice signals, then reads the number and connects the call, if desired.
- *The Internet Slide Show* is a service based on the web and telephone-conference hardware. This service allows presenters to build a presentation using Microsoft Powerpoint, then present it to a geographically disparate group using the web and the telephone. Part of the voice-response component of the service allows participants who do not have computers to call and confirm their attendance, and request that the slides be faxed ahead of time. This work was originally done in collaboration with the AT&T Commercial Markets and Mobility Concepts District in Bridgewater, New Jersey.

- *5ESS-2000 Switch Customer Documentation and Training Process.* We have written an in-process decision support service in mawl that directs users through a sequence of interactions, and coordinates their actions with others in the context of an extended work-order transaction metaphor. The service is being used initially by systems engineers, feature designers, and testers in the course of software and documentation development processes. The same mawl platform can potentially be used to direct 5ESS-2000 switch customer technicians through procedural steps in the customer documentation for operation, administration and maintenance tasks on the switch. Basic features of the interface are implemented in mawl and are already part of the official product development process. This work was done in collaboration with the CD&T Requirements, Process, and Tools Group at Indian Hill.
- *Browse-by-phone.* In addition to making mawl services accessible by telephone (IVR mode), our PML interpreter provides a state-of-the-art phone browser for HTML pages anywhere in the web (browse mode). The browser consists of PML-interpreting software together with an application programming interface (API) for PML that has been implemented on both Dialogic and Conversant hardware. This browser presents content to users either by playing audio files or converting text to speech, and collects input from users by recognizing touchtone signals and voice. Touchtone signals are also used for the typical browser operations. The marks associated with HTML are parsed and interpreted appropriately.

5 Related Work

The explosive growth of the web and the emergence of its commercial potential have spurred many efforts to make up for the shortcomings of the web infrastructure for complex services. Mawl by itself may be compared to a variety of web service creation technologies. Most of these technologies were conceived of as tools for web service construction only, rather than as environments for supporting all phases of the software life cycle. While general-purpose programming-language support, such as provided by C++ or JAVA, helps *enable* the creation of service architectures, these languages do not address the problem of what such an architecture should be. In addition, once a particular architecture is defined, there are many benefits to having application-level support for enforcing and supporting this architecture, as we have discussed.

- *CGI libraries.* Libraries for the languages commonly used to write CGI "scripts" (e.g., C, ksh, tcl, perl) [Mal94] provide basic facilities for form parsing and dynamic generation of HTML. Solutions to the problems of state maintenance and concurrency usually must be constructed by hand.
- *Extending HTML with general-purpose programming constructs.* A common approach to the generation of HTML documents with dynamic content is the embedding of fragments of code executable on the server. Recent examples are WebThreads [Thr] and Netscape's JavaScript [Net]. Mawl's use of MHTML is superficially similar to these examples, but dynamic content is provided by separate service logic rather than embedded code.
- *Elaborations of CGI.* Netscape's LiveWire programming environment combines scripting and HTML code embedding, and adds support for persistent state with predefined dynamic session data objects. However, it still adheres to the CGI request/response paradigm, encouraging intermingling of HTML with scripting code.

In these examples, service logic and presentation are intermingled. As a result, the logic of services written in this manner must be inferred from the link topology of the dynamically generated HTML. We have shown that separating these two aspects of a service makes static analysis of the service possible, as well as multiple-interface access to service. Such capabilities is difficult, if not impossible, to achieve in the services constructed with these other tools.

- *Client-side enhancements.* The advent of sophisticated client-side interfaces (in particular Java-capable web browsers [GM95]) allows web services to run on the client. However, for many interesting services, the client must still interact with a server. For complex services, the programmer is left with the task of managing this client/server interaction. This suggests that client-side capabilities are in some sense complementary to mawl. The recent introduction of Java libraries that permit applet/browser/plugin communication within Netscape makes it straightforward to embed applets that support mawl's arbiter paradigm.
- *Distributed processing environments.* A number of efforts are afoot to mask or replace the web infrastructure with one that supports truly distributed applications. Examples are Java/CORBA, Java Remote Method Invocation, ActiveX, various server APIs, 'Web File System' proposals, and Inferno/Limbo. These technologies will provide some of the advantages that mawl offers, and more. However, many of the issues of service design addressed by mawl remain to be dealt with. Moreover, mawl can easily be retargeted to produce implementations based on these environments.

PML provides capabilities that can be used for traditional IVR applications, but there is a range of existing technology that combines phone and web in some fashion:

- *Web browsing by phone.* Commercial products (e.g., Netphonic's Web-on-Call) exist that support a telephone interface to web pages. The idea of extending HTML to include phone-specific markup in the style of PML has been proposed by Stylus, and used to support their version of a phone browser. PML supports these uses as well and also provides a convenient way to develop enhancements on top of these basic capabilities. Furthermore, the combination of mawl and PML provides enhanced service programming possibilities not found in existing products.
- *Web control of telephony.* There are several applications within the Company and outside that provide 'click-to-dial' call control from within a web service. Mawl has been used in an application involving audio conference control, as noted above, and can take advantage of back end and browser technology in the same way as any other web application.

  The incorporation of packet telephony with the web is the focus of research and development within the Company and outside. Mawl does not provide direct support for web browser telephony, though the PML platform will soon support playing and recording audio streams.
- *Service creation environments.* There are a number of graphical IVR service creation environments available. While some provide limited web facilities, none supports full web/phone service creation. The abstractions introduced by mawl are precisely those that one would want to capture in such a tool.

These examples show that there are existing pieces of technology that functionally overlap with mawl and PML. However, the mawl infrastructure provides a unique integration of these capabilities, together with the development and maintenance advantages that application language engineering provides.

6 Current Research Activity

Mawl's flexible architecture allows service specification and user interface issues to be investigated by quickly building prototypes and implementing language and platform features. In this way mawl provides an environment for doing research on interactive services. Some of our current research activities are as follows.

- *Enhanced session state and data management.* The Mawl Storage Server will manage all persistence storage needs for mawl-generated services. This will simplify storage operations for services by providing a protocol by which services communicate with the storage server. The Mawl Storage Server will give mawl programmers transparent access to standard database back ends, thus simplifying service logic code. The modularity of the architecture allows future enhancements to address issues of scalability and reliability.
- *Service analysis.* The mawl compiler has a logging option that collects statistics about how mawl services are used. We have developed Java applets (see Sidebar B), based on visualization technology from within our department, with which service providers can analyze usage statistics for their services and identify design or performance issues.
- *Telephonic features and services.* The PML language and interpreter permits the rapid construction of simple IVR services such as information inquiry and voicemail. We have added primitives to PML for incorporating telephonic features such as speech recognition, transfer, bridge, and conference, and we are working to incorporate other features, such as streaming audio, into mawl-based telephone services. The Conversant PML platform, currently implemented, provides a stable environment for constructing large-scale web and telephone services with call processing features, and will permit experimentation with new service specification languages and other sorts of interactive telephone services, such as call centers.
- *Interface-independent programming.* A goal of mawl is to allow service programmers to program in an interface-independent manner, insofar as that is possible. In some cases, the service logic may be independent of the user interface. Mawl supports this case transparently. In other cases, it is not realistic to expect identical service logic for different user interfaces; the sparseness of the telephone interface, for example, may preclude user interactions that the web interface permits. The goal of completely shared service logic raises many research issues related to the display of structured information using an audio interface. We will continue to incorporate research results on smart interpretation of HTML into the PML interpreter.

7 Summary

Mawl and PML provide a comprehensive solution for building complex services that are accessible via the web and the telephone. The mawl compiler and infrastructure make it easy to construct important classes of application that are difficult to create using "traditional" web programming approaches. Furthermore, mawl's architecture and session abstraction support the goals of service monitoring and administration. The trend toward moving our internal and external business processes to the web will continue to be a rich source of potential applications.

8 Acknowledgments

We would like to thank the following people for their support and assistance in the mawl effort: Jim Coplien, Dan Feistamel, Alan Hastings, Tom Jacobs, Lalita Jagadeesan, Konstantin Laüfer, Cecilia Mak, Eric E. Sumner, Jr., Ian Sutherland, Natasha Tartarchuk, and David Weiss.

References

[BL95] T. Berners-Lee. Hypertext transfer protocol (HTTP/1.0). *Working Group of the Internet Engineering Task Force*, October 1995.

[BLC95] T. Berners-Lee and D. Connolly. Hypertext markup language (HTML 2.0). *Working Group of the Internet Engineering Task Force*, August 1995.

[CBR96] Kenneth C. Cox, Thomas J. Ball, and J. Christopher Ramming. Lunchbot: A tale of two ways to program web services. Technical Report BL0112650-960216-06TM, Lucent Technologies Bell Laboratories, 28 February 1996.

[GM95] James Gosling and Henry McGilton. The java language environment: A white paper. Technical report, Sun Microsystems Laboratories, 1995. available at URL:http://java.sun.com/whitePaper/javawhitepaper_1.html.

[LR95] D. A. Ladd and J. C. Ramming. Programming the web: An application-oriented language for hypermedia service programming. In *4th International World Wide Web Conference*, 1995.

[MA91] D. B. McQueen and A. Appel. Standard ML of New Jersey. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming*, pages 1–2. Springer-Verlag, 1991.

[Mal94] John C. Mallery. A common lisp hypermedia server. In *First International WWW Conference*, 1994.

[Net] Netscape. http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/index.html.

[Thr] Web Threads. http://www.webthreads.com/.

A Sidebar: The LunchBot

We use the LunchBot to order meals for our weekly department luncheons. An administrator selects a restaurant and notifies department members of the choice. The members then inspect the restaurant menu and place their orders. Orders are faxed to the restaurant a few hours before noon. When the food is delivered, customers are notified.

This service was originally performed manually. Customers inspected paper copies of the menus and gave their orders to the department secretary, who compiled and faxed the order, and accepted delivery. The secretary also kept accounts. This approach was practical when only a few people were involved, but as the department grew it became unwieldy. Automating many of the tasks involved was the next logical step.

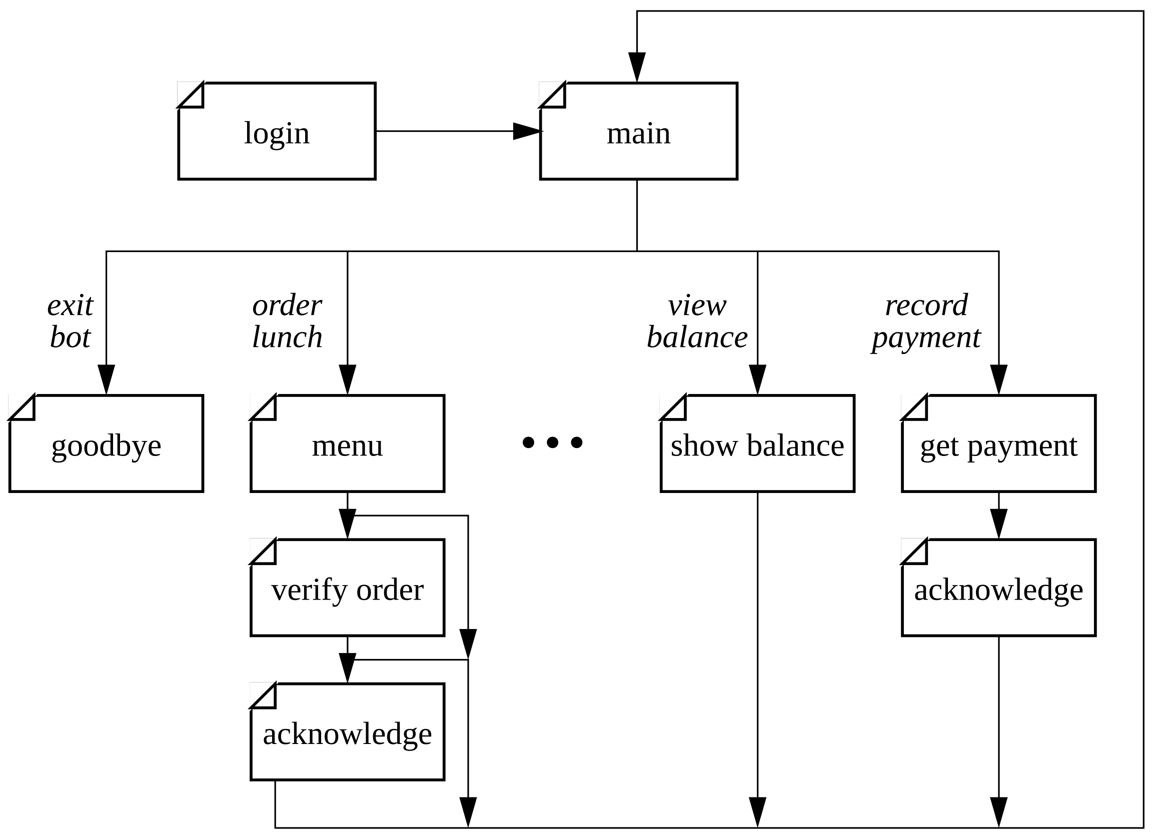


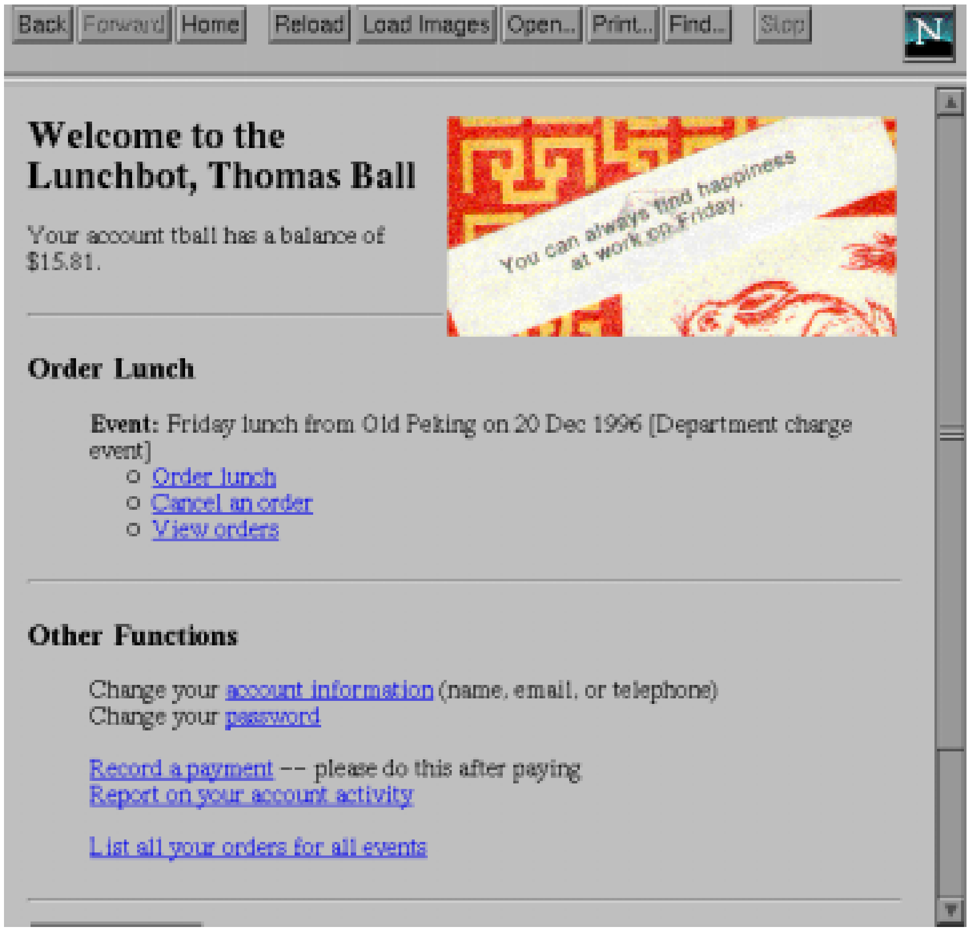
Figure 6: High-level service logic of the LunchBot. Rectangles represent HTML pages.

This led to the creation of the first web version of the LunchBot [CBR96]. This service was implemented in a "tools-based" manner for execution through the common gateway interface using awk, shell, and other Unix utilities. While this service functioned well, it was difficult to maintain. This is a common characteristic of tools-based programs: although a collection of originally unrelated tools can be orchestrated to work together, the resulting architecture is often fragile and confusing.

The LunchBot was re-implemented in mawl shortly after mawl was developed. Figure 6 shows the high-level design of the LunchBot, as depicted by relationships between LunchBot pages. The service first presents a "login" page in which the user selects either a customer account or the administrative account. The user is then presented with a main page which presents the available functions as links. Figure 7 shows a screen snapshot of the main page. The contents of the main page depend on the user's account (customer versus administrator) and the current state of the LunchBot (e.g., whether or not it is open for business). The user selects an action and is directed through a series of pages. After the action is completed, the main page is shown again.

The resulting "language-based" (mawl) version of the service has more features than the tools-based version, and is easier to maintain and enhance. For example, once the web version was up and running, a telephone-enabled version of the LunchBot took very little time to construct. Another example of enhancement is dynamic logging, which has yielded useful visualizations. See Sidebar B for an analysis of how people use the LunchBot.

Figure 8 shows the two implementations at the code level. The mawl version is on the



Welcome to the Lunchbot, Thomas Ball

Your account tball has a balance of $15.81.

Order Lunch

Event: Friday lunch from Old Peking on 20 Dec 1996 [Department charge event]

- Order lunch
- Cancel an order
- View orders

Other Functions

Change your account information (name, email, or telephone)
Change your password

Record a payment -- please do this after paying
Report on your account activity

List all your orders for all events

Figure 7: Screen snapshot of the main page of the LunchBot.

left, and the tools-based version on the right. Each file is represented by a rectangle, with one horizontal line for each line of the file. The lines are color-coded to indicate whether they are part of the service logic or the interface (e.g., HTML templates). The clean separation of service logic from interface layout in the mawl version on the left contrasts sharply with the intermixing of the tools-based version.

B Sidebar: Service Analysis

While there are many tools for the construction of web sites and dynamic web services, most of these tools lack support for a crucial part of web service maintenance: the analysis of how people access a site or use a service. For example, one might want to restructure a service in which users are forced to take many steps to perform a common task into a service in which this task is done in a single step. By identifying patterns of user access, providers can find ways to improve their services.

The mawl compiler has a logging option that compiles sessions that automatically track their usage. For each page generated by the session, the log records the unique session identifier, the page name, the time requested, and other information. This log is available through the web for analysis. In particular, mawl provides a Java applet called *PathView,* illustrated in

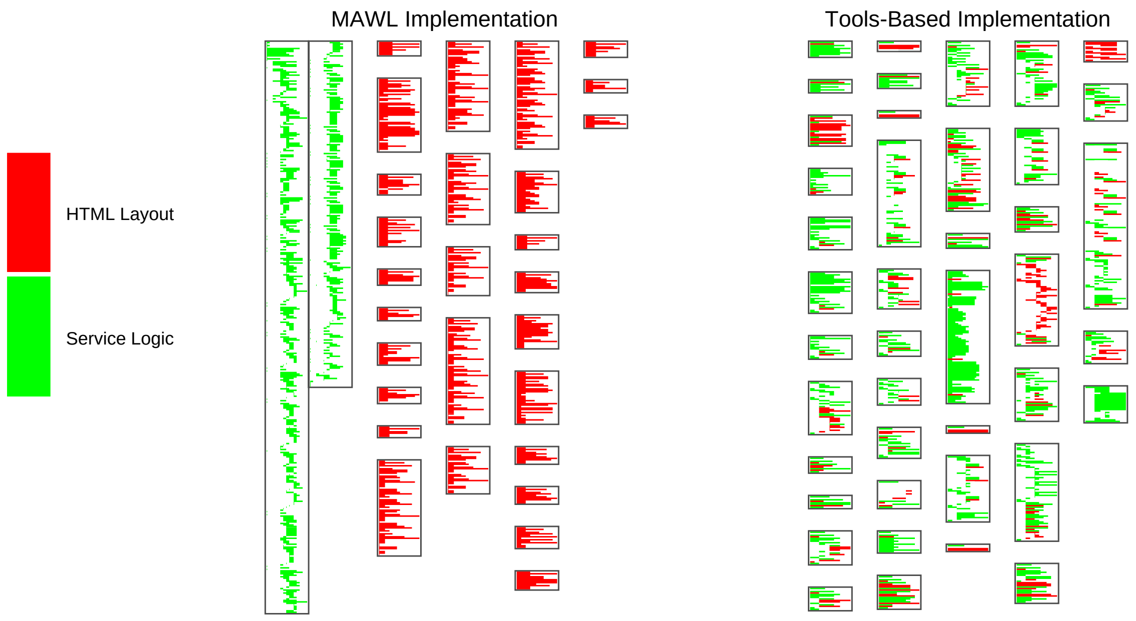
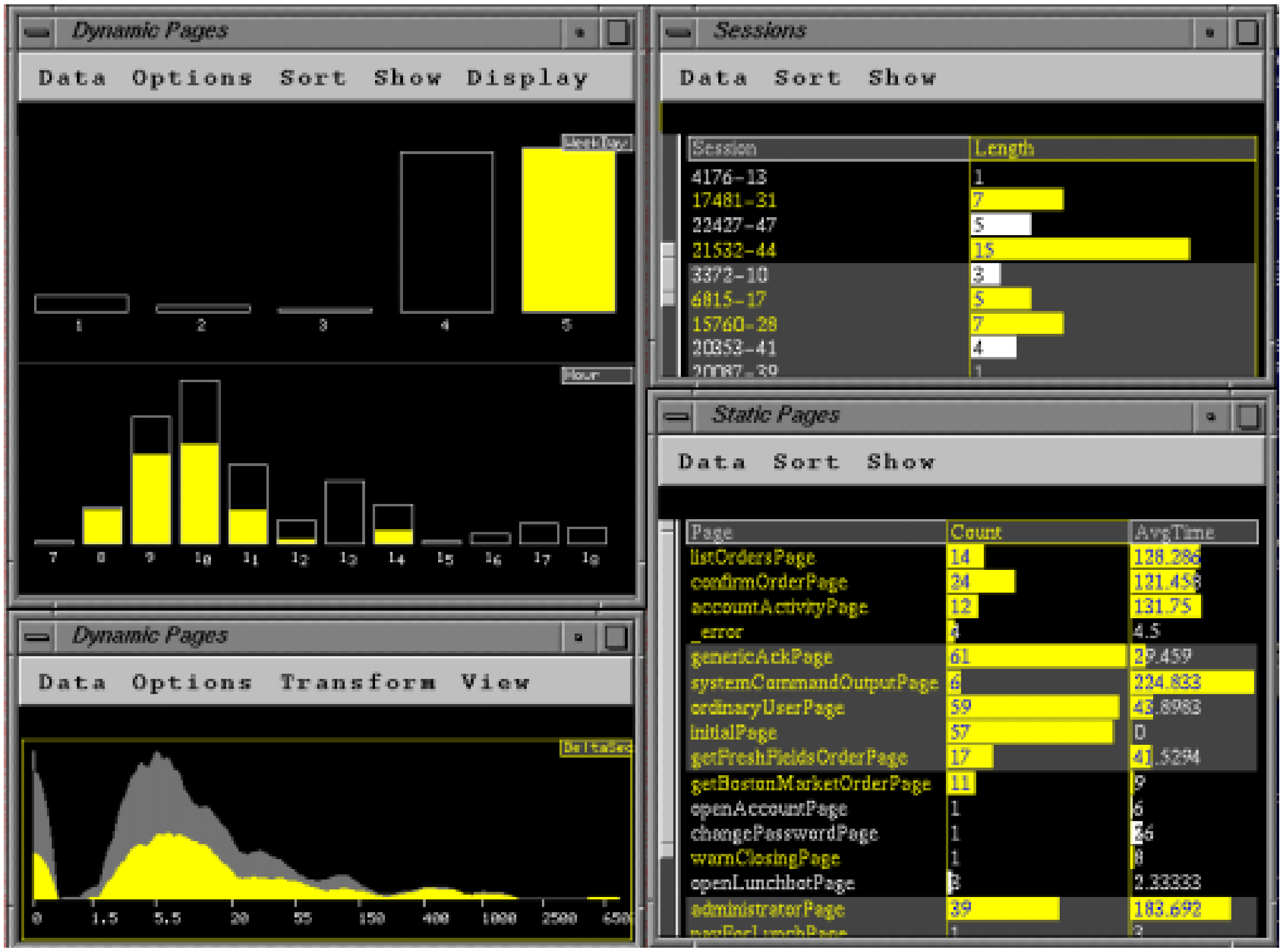


Figure 8: A global overview of the source code of the language-based (mawl) LunchBot (left) and the tools-based LunchBot (right). Lines are colored to show whether they are part of service logic (green) or interface (red).

Figures 9 and 10, for exploring logged data. A sequence of user interactions with a web service is naturally expressed as a path through an information space of pages. As users explore a site or interact with a service, each creates a path. PathView enables analysis of the set of user paths in conjunction with other page-oriented statistics, such as time spent on each page, time of day, and so on.

Figure 9 shows four statistical views of the PathView applet applied to the LunchBot. Our department luncheons are typically on Fridays. The “WeekDay” bar chart in the upper left-hand corner shows that the LunchBot is used most on Thursdays and Fridays (days 4 and 5) every week. The “Hour” bar chart shows LunchBot usage by hour, with Friday’s usage highlighted. The highlighted hourly distribution for Friday clearly dominates the overall distribution. Not surprisingly, the major use of the LunchBot occurs Friday morning, just before the LunchBot closes and lunch is ordered. The views in the upper right, lower right, and lower left show associated session-level, page-level, and temporal statistics, respectively. All of these views are linked so that selection in one view is reflected in the other views.

The statistical views shown in Figure 9 are linked with the visualization shown in Figure 10. In this view, the x-axis represents the pages served up by the LunchBot and the y-axis shows time steps. Each user creates a path through the graph. The large peak represents the sequence of pages served up when users order a meal and confirm the order. The smaller peak represents users visiting the “listOrders” page, which shows all the accumulated orders for open events. As the picture shows, some people first order lunch and then list orders, while others list orders (presumably to see what the popular menu items are) and then order lunch. The latter pattern suggests a useful new service option: when users order lunch, present the top five items ordered in the last month.



Session	Length
4176-13	1
17481-31	7
22427-47	5
21532-44	15
3372-10	3
6815-17	5
15760-28	7
20353-41	4
20087-39	1

Page	Count	AvgTime
listOrdersPage	14	128.286
confirmOrderPage	24	121.458
accountActivityPage	12	131.75
_error	4	4.5
genericAckPage	61	29.459
systemCommandOutputPage	6	224.833
ordinaryUserPage	59	43.8983
initialPage	57	0
getFreshFieldsOrderPage	17	41.5294
getBostonMarketOrderPage	11	9
openAccountPage	1	6
changePasswordPage	1	26
warnClosingPage	1	8
openLunchbotPage	3	2.33333
administratorPage	39	183.692
payForLunchPage	1	3

Figure 9: *PathView*, an applet for service analysis. PathView uses interactive visualization to enable service providers to explore user interactions with a service.

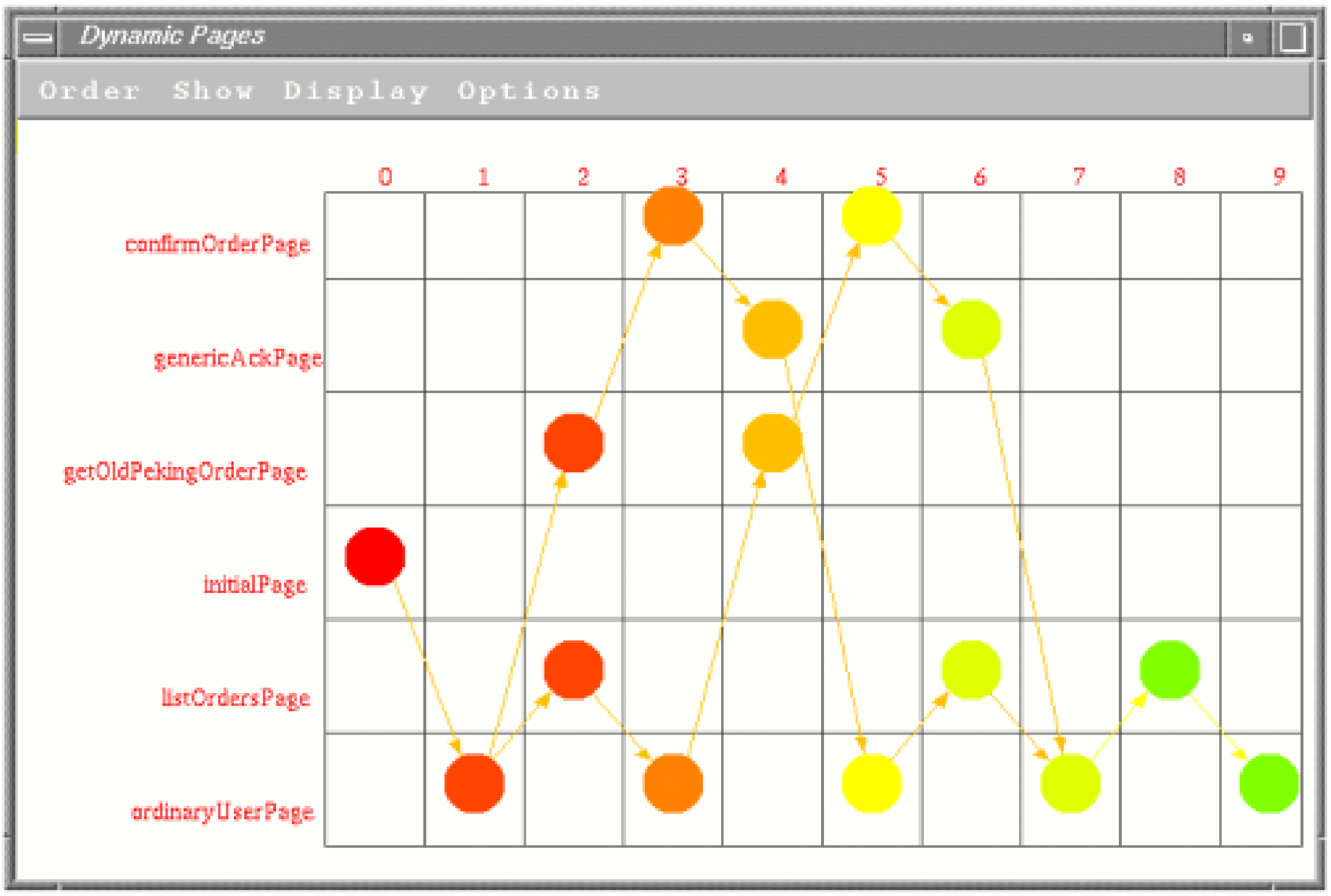


Figure 10: User interactions with the LunchBot.