Mawl: A Domain-Specific Language for Form-Based Services

David L. Atkins, *Member*, *IEEE*, Thomas Ball, *Member*, *IEEE Computer Society*, Glenn Bruns, and Kenneth Cox, *Member*, *IEEE*

Abstract—A form-based service is one in which the flow of data between service and user is described by a sequence of query/response interactions, or forms. Mawl is a domain-specific language for programming form-based services in a device-independent manner. We focus on Mawl's form abstraction, which is the means for separating service logic from user interface description, and show how this simple abstraction addresses seven issues in service creation, analysis, and maintenance: compile-time guarantees, implementation flexibility, rapid prototyping, testing and validation, support for multiple devices, composition of services, and usage analysis.

Index Terms—World Wide Web, web services, programming languages, forms, HTML.

1 Introduction

Domain-specific languages (DSLs) offer a more complete solution to software engineering problems than general purpose programming languages can offer. The benefits of DSLs derive from two basic principles of language design: abstraction and restriction. The choice of appropriate abstractions aids the phases of requirements, design, coding, and maintenance by providing high-level entities and relationships that fit the domain closely. Restriction of language expressiveness allows for greater automated analysis and hence supports verification, modification, and maintenance.

We present a DSL called Mawl for creating form-based services for the web and telephone. We focus on Mawl's *form* abstraction and show how it supports the software life cycle through appropriate abstraction, restriction, and compiler support. We discuss how the DSL approach has helped solve several software engineering problems that arise with the creation of web and telephone services. When applicable, we consider how these results generalize to other domains. Finally, we consider the shortcomings and pitfalls of Mawl in particular and of the DSL approach in general.

1.1 The Domain: Form-Based Interactive Services

Our domain of interest is that of services in which a user interacts with a remote computer for the purpose of completing some transaction, which may involve browsing an information space, selecting items, providing responses to queries, etc. A service might be as simple as entering a name to get a telephone number or as complex as ordering from a catalog.

• *D.L. Atkins, T. Ball, G. Bruns, and K. Cox are with Software Production Research Department, Bell Laboratories, Lucent Technologies, 263 W. Shuman Blvd., Naperville, IL 60566.*
E-mail: {datkins, tball, grb, kcc}@research.bell-labs.com.

Manuscript received 22 June 1998; revised 10 Dec. 1998.
Recommended for acceptance by D. Wile.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 109126.

A form-based service is one in which the flow of data between service and user is described by a sequence of query/response interactions, or *forms*. A form provides a user interface that presents service data to the user (such as a list of accounts), collects information from a user (such as the selected account), and returns it to the service.

Both web services (such as FedEx's package tracking service and Amazon's bookstore) and traditional interactive voice response (IVR) telephone services fit the form-based service paradigm. A web service sends an HTML (HyperText Markup Language [5]) page to a user's graphical browser, providing information and a set of input fields to request information such as account and password. Upon receiving a response from the user, the service sends another page (or form, in our parlance). An IVR service typically presents a user with a menu of choices ("For Jazz Music, press 1; for Classical Music, press 2; ..."), collects a sequence of digits or performs automatic speech recognition, and then presents information or another menu.

1.2 Mawl: A DSL for Form-Based Services

Mawl is a DSL for programming form-based services in a device-independent manner [13], [2], [1]. Mawl defines a software architecture that separates the specification of service control flow and state management, which we refer to as "service logic," from the specification of a user interface. Mawl's *form* abstraction is the means for enforcing this separation of concerns through a functional interface similar to an interface definition language (IDL).

Mawl's form abstraction illuminates how DSLs can improve many parts of the software development life cycle. The form abstraction has provided straightforward solutions to several software engineering problems:

- *Well-formedness of web services.* The Mawl compiler can ensure that the service logic and HTML input to it are both internally consistent and also that they are consistent with one another.

- *Implementation flexibility and platform independence.* Because Mawl's language abstractions hide the details of the Common Gateway Interface (CGI) and HyperText Transmission Protocol (HTTP) [4]) from the programmer, the compiler is free to choose different implementation strategies for the same service logic to address issues of scalability and portability.
- *Prototyping services.* Many programmers equate "prototyping lanugages" with typeless, interpreted languages such as perl [23], tcl [16], or ksh [11] that support a fast edit-compile-debug cycle. In contrast, Mawl supports prototyping of web services via a static type system, allowing services to be run from a web browser without the need to write any HTML.
- *Testing and validation.* Automated testing of web services is difficult and tedious if the only way to interact with a service is through a web browser. Furthermore, web services are inherently concurrent programs requiring coordination of access and updates to persistent data, since different users can simultaneously access a service. Mawl's separation of service code from user interface description allows the service code to be run and tested in a batch mode, independent of a particular browser.
- *Supporting multiple devices.* One measure of a DSL is how well it supports unplanned changes. Mawl first targeted the graphical web browser, but naturally accommodates the integration of new devices into existing services. We will describe how Mawl allowed us to develop services accessible via both the graphical web browser and the telephone.
- *Composing web services.* Many web services query, collect, and integrate information from other web services (i.e., see MetaCrawler [19]). The only programming interface available to most web services is via an HTTP request, which returns HTML that must be parsed to extract the desired information. The Mawl form abstraction substantially simplifies the programming of services that must interact with remote web services.
- *Usage analysis.* The creation of a service is just a small part of the software life cycle. Analysis and maintenance of a service are necessary to keep the service up-to-date and functioning properly. The form abstraction is a natural place to monitor all user/service interactions and record a log of interesting events. A Java applet allows programmers of Mawl services to analyze usage patterns once a service has been deployed.

1.3 Overview

Section 2 presents a brief history of Mawl and describes the basics of the Mawl service architecture and language. Section 3 shows how the Mawl form abstraction has helped address seven problems in service creation, analysis, and maintenance. Section 4 further evaluates Mawl and the DSL approach in the context of a large Mawl application, the LunchBot. Section 5 summarizes the paper.

2 Mawl: History, Software Architecture, and Language

This section describes a brief history of Mawl, the Mawl service architecture and its implementation in a DSL, and some necessary details of the language.

2.1 A Brief History

Mawl was created in early 1995 because of difficulties experience in programming form-based web services using CGI programs and the HTTP protocol.

With the HTTP request/response protocol and CGI, a program is executed to respond to an HTTP request from a browser. Once the program has sent the requested data (usually an HTML document), it terminates. However, many services require sequencing between pages, and the maintenance of persistent state on the server. The HTML page created by one CGI program may contain a URL (Uniform Resource Locator) pointing to the next program to execute. Thus, the control flow of the service becomes split across pages and the multiple programs associated with these pages, and intermingled with the user interface description (that is, the HTML).

A direct analogy is helpful: HTTP, CGI are the GOTOs [7] of the 1990s; they are low-level mechanisms that, when used directly by programmers, unnecessarily complicate the creation of web services. The idea of structured control flow common to modern sequential languages is lost. CGI programmers encode control flow as assembly language programmers do, with direct jumps rather than structured control flow constructs.

To address this problem, Mawl presents an architecture for form-based services that is independent of the HTTP and CGI protocols. Programmers are given the illusion of a traditional imperative sequential language in which they may code a centralized service, rather than a set of programs coupled indirectly through HTML pages. This language is independent of a particular markup language such as HTML.

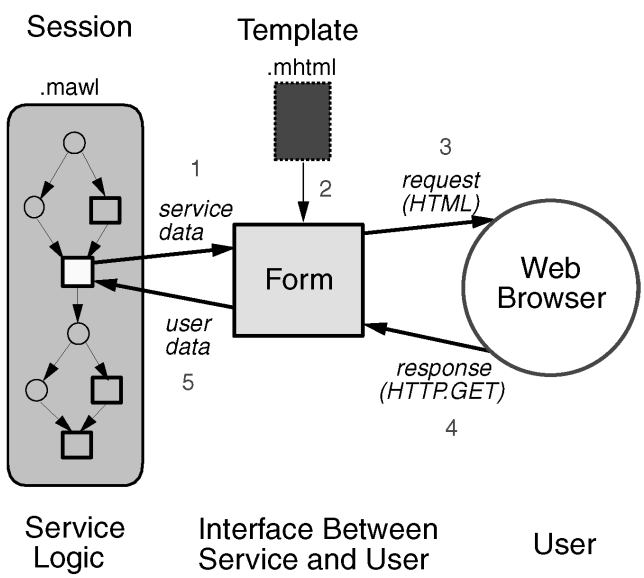


Fig. 1. The Mawl software architecture. Integer labels show flow of data.

2.2 The Mawl Software Architecture

Fig. 1 illustrates the Mawl software architecture and the flow of information (integer labels). A Mawl service consists of one or more *sessions*. A session specifies the control flow of a service and the update of service variables. Typically, each session controls a different aspect of the service (e.g., there may be a session for general users and another session for the administrators of a service). A session communicates with the user via a *form*, which defines the data to be presented to the user and the data to be accepted from the user. A *template* defines the static portion of a user interface as well as the dynamic portions that are parameterized by values passed to the form by a session.

Fig. 2 contains a simple Mawl service that will be used to explain the three abstractions in more detail.

2.2.1 Sessions

A *session* is a sequential program that communicates with the user by making method calls on forms. A session is specified using standard imperative constructs for looping, conditional control flow, procedure calls, exceptions, etc.

In our example, there is one session, `Greet`, that communicates twice with the user, first prompting for the user's name and then greeting the user, displaying a count of the visitors to the service, and the elapsed time between the presentation of the first and second forms to the user. The service logic, written in Mawl, is shown in Fig. 2a.

Mawl provides a persistence model that allows programmers to specify the type of storage required for Mawl variables. Each session instance has its own environment for *local* variables (declared by the keyword `local`). A global environment contains *global* variables (declared by the keyword `global`) that are shared and persist across all session instances. As multiple sessions may be active at the same time, concurrency control is required for updating global variables. A block-structured `region` statement provides mutual exclusion of statement execution for concurrency control.

In our example, the variable `GetName`, `ShowInfo`, `time_now`, and `i` are local variables, while `access_cnt` is a global variable. In this simple example, concurrency control is not needed since the variable `access_cnt` is updated atomically in the expression `++access_cnt`.

2.2.2 Forms

The only way for a session to interact with the user is through a simple input/output abstraction called a *form*.

A form is an object with a parameterized method `put`, which takes a record of data as input and returns a record of data as output. The input record represents *service data* to be sent to the user and the output record represents *user data* collected from the user. The Mawl form declaration specifies the type of the service data and user data records.

The first two lines of session `Greet` in Fig. 2a declare two forms, `GetName` and `ShowInfo`. A form is declared with a type signature specifying the structure of the expected service data and user data. The form `GetName` has type `{} -> {string id}`, meaning that it expects no data from the service (thus, the empty record `{}`) and returns a record containing a string named `id`.

The session `Greet` first provides `GetName` with its required (empty) input record and receives back a record

```
(a) Greet.mawl:

  global int access_cnt = 0;
  session Greet {
    local form {} -> { string id } GetName;
    local form { string id, int cnt, int time } -> {} ShowInfo;

    local int time_now = minutes();
    local string i = GetName.put({}).id;
    ShowInfo.put({i, ++access_cnt, minutes()-time_now});
  }

(b) GetName.mhtml:

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

(c) ShowInfo.mhtml:

  <HTML><HEAD><TITLE>Show-Info Form</TITLE></HEAD>
  <BODY>Hello <MVAR NAME=id>, you are visitor number <MVAR NAME=cnt>.<BR>
        Time elapsed since first form is <MVAR NAME=time> minutes.
  </BODY></HTML>
```

Fig. 2. A Mawl service: (a) that asks the user for a name through the form `GetName` and then uses the form `ShowInfo` to display their name, how many visitors to the service there have been, and the time elapsed between the presentation of the first and second forms. The HTML templates corresponding to the forms are: (b) `GetName.mhtml` and (c) `ShowInfo.mhtml`.

containing the string field id. This string is extracted into the variable i. The service then supplies ShowInfo with a record containing three values: the user's name, the updated access count, and the elapsed time. The empty record returned by this form is ignored by the session.

There is a close connection between the form notation and interface definition languages (IDLs). An IDL is a language for describing the interfaces of a software component. An IDL specification describes the input/output signature of an operation, where a set of operations comprises an interface. CORBA [10] (Common Object Request Broker Architecture) and RPC [21] (Remote Procedure Call) both have IDL specification languages. Just as with forms, IDL programs only express the signatures of operations, but do not describe this computation. We will return to this comparison later in Section 3.2.

Another way to view a form is as a contract between the programmer creating the session code and the graphic designer creating the HTML templates for the service. The form enables both people to work independently, confident that if they respect the form's type signature then the service logic and the HTML can be combined with no conflict. The Mawl compiler will check this "contract," as described later.

Support for the development of form-based database applications is discussed in [18], [17]. In [18], a Form Application Development System (FADS) is described. Here a form is a database tuple, relation, or sequence of tuples and relations. A form does not capture, as Mawl's forms do, how data flow from service to user and vice versa. Instead, forms have associated operations that are defined in a database query language. Communication between users in a FADS application is through a shared database. In [17], the form-based approach is taken further by making the development enivornment itself totally forms-based, so that a developer defines the forms of an application interactively by filling in forms.

The use of C-like syntax in Mawl forms exemplifies the DSL design tenet that familiar notation should be used when they are suitable. Note however, that Mawl programs are not simply translated by the Mawl compiler into C, with the C-like parts of a form unaffected. Instead, C-like syntax is used in Mawl as a familiar notation for types and expressions.

2.2.3 Templates

Each form may have one or more *templates* associated with it. The association in the example here is by common name —the form name used in the Mawl code is that of the template file.

A template is a document in some markup language (such as HTML) that has been extended with marks that refer to service variables. These marks allow documents to incorporate service data at run-time.

MHTML is an extension of HTML that is used for creating HTML templates. Fig. 2b and Fig. 2c show templates written in the language MHTML. The additional marks found in MHTML allow for substitution of service data into a form.

MVAR is an MHTML mark for accessing form's service data. This mark indicates substitution of scalar service data (integer, string, etc.) into the generated HTML. User data are represented by the standard HTML user-input marks such as INPUT and SELECT; the NAME attribute of these marks is the name of the user data variable.[1]

A template represents one possible "implementation" of a form's put method, for a particular browser that will "execute" it. A form may have no template associated with it, which has interesting implications for prototyping services (Section 3.3). Furthermore, a form may have multiple templates associated with it, which is useful for supporting multiple devices or browsers (Section 3.5).

2.3 Mawl Types

Mawl has four basic types: integers, floats, Booleans, and strings; and three complex types: records, lists, and forms. Mawl records are similar to C structures. The syntax for defining a record type is to enclose a list of (type specifier, identifier) pairs in braces. The example below declares a record variable named customer with a field name that is a string and a field age that is an integer.

```
local { string name, int age } customer;
```

Record values can be constructed "on the fly," as shown in Fig. 2a in the argument to put.

Lists in Mawl behave much like arrays in C, although no storage allocation is required. A list type is denoted by enclosing another type in brackets. For example, a variable to hold a list of strings is declared as:

```
local [ string ] names;
```

List elements are denoted using brackets and an integer index: names [i + 1] = names [i]. Lists grow automatically to accommodate such references. List values are formed by enclosing a list of values in brackets:

```
charlist = [ a, b, c ];
```

As illustrated earlier, the syntax for form types is the keyword form along with a service type and a user type, separated by the token ->. The service and user types must be record types. For example,

```
form { [ float ] temps } -> {} show_temps;
```

can be used to display a list of temperature values, as shown below:

```
show_temps.put ( { [0, 10.5, 20, 30] } );
```

2.4 MHTML

As explained previously, scalar data is inserted into a template with the MVAR mark. List data is inserted with the MITER mark. The construct <MITER>...</MITER> iterates over Mawl lists and generates HTML that is dependent on the list element values. A natural use of MITER is to display a table with a variable number of rows. The NAME attribute of MITER specifies the name of a list in a form's

1. Note that the MHTML in Fig. 2 does not contain any FORM mark which specifies the CGI program to be executed upon submission of the FORM; the Mawl compiler and run-time systems takes care of inserting a FORM mark and ensuring that control returns to the appropriate point in the session with the correct state, as discussed in Section 3.2.

service data. The additional MCURSOR attribute names a new cursor variable over the list. The MHTML enclosed between <MITER> and </MITER> is repeated for each element in the list, with the value of the cursor variable set to the index for that iteration. Other MHTML marks may then use Mawl's list element notation to refer to list elements.

The MHTML below shows how a list of temperatures might be displayed in a single column table:

```
<HTML><BODY>
 <TABLE>
  <MITER NAME = temps MCURSOR = i>
   <TR><TD><MVAR NAME = temps[i]></TD></TR>
  </MITER>
 </TABLE>
<BODY><HTML>
```

The <MITER> mark iterates over the temps list and i is the name chosen for the index used in the subsequent MVAR mark.

The extension of HTML to MHTML illustrates another basic DSL design tenet. Here the need for a language with capabilities similar to HTML was designed by extending HTML in a style consistent with HTML. This approach contrasts with the use of C-like syntax in Mawl, because there a small fragment of C was adopted, rather than extending C.

3 Benefits of the Form Abstraction

This section focuses on several software engineering problems that the DSL approach helped to address. We describe each problem, our solution, and how the domain-specific approach helped provide a solution. We also discuss whether these problems and solutions would apply in other domains.

3.1 Compile-Time Guarantees

For complicated web services, we would like to guarantee at compile time that a service will generate only valid HTML. Furthermore, we would like to know that this HTML is consistent with the service logic. That is, that the service is prepared to deal with the values that may be entered by the user for a given page of HTML. Such consistency checking is difficult to achieve in general purpose languages, in which HTML is generated using print statements.

Mawl's division of a service into sessions, forms, and templates allows consistency checking to be performed at compile time. First, the sessions and the MHTML can be independently analyzed to ensure that they are internally consistent. For the sessions, this means standard type checking and semantics checking. For MHTML templates, this means verifying that a template is legal MHTML.

Additionally, a session (i.e., Greet.mawl in Fig. 2) and the MHTML templates can be checked against one another. The form abstraction makes this possible by providing a type signature expressing the structure of a service/user interaction. The MHTML represents the body of a form's put method and can be analyzed to ensure it is consistent with the form's type signature.

For example, Fig. 2b shows the content of the file GetName.mhtml, which is the MHTML template associated with GetName form. This template contains no uses of the MVAR mark and contains one INPUT mark named id, which is consistent with the type signature of the associated GetName form. Similarly, the template in Fig. 2c agrees with the ShowInfo form, since the template has MVAR marks referring to the service data id, cnt, and time and has no INPUT marks.

Another example of a consistency check is to ensure that only values of type list are used in MITER marks. It is not required that the MHTML refer to all the service data passed to a form, which is useful for multidevice services, as discussed in Section 3.5.

Consistency checking of two languages against one other is an interesting problem that arises in the area of embedded languages, which is a common approach to constructing DSLs. An example of such an embedding is that of the declarative database query language SQL [20] in the general purpose language C [12]. Generally an embedding of SQL in C uses some escape character to prefix a line containing SQL and the SQL can contain references to C variables that are input to the SQL query. A preprocessor over the C code extracts the lines needed to build the SQL query. The intermingling of the two languages makes static type checking of the embedded language (SQL) difficult.

Rather than embed MHTML directly in the Mawl language, we used the form, a functional interface, to moderate between the two languages. The HTML language was extended (to MHTML) so that values in the Mawl type system can be substituted into MHTML and so that static checking can be performed on MHTML and between Mawl and MHTML. The MHTML is the body of the form's put method and can be type checked against the form's type signature. This degree of compile-time checking is much greater than traditionally found in embedded languages.

3.2 Implementation Flexibility

A common problem with web services is that of scaling to larger hit rates. While the CGI protocol is conceptually simple, it has high overhead because of the large amount of process creation and destruction that can take place, as detailed later. As a result, a web service that commits to a particular implementation model may run into trouble later if the service becomes popular.

We consider how Mawl code is compiled to an CGI-based implementation and also show how it can be compiled to a more efficient server-based implementation. The Mawl service architecture supports implementation flexibility via the centrally specified control flow of a session and via the form, which identifies where a session relinquishes control to the browser and where control returns to the session.

A Mawl service can be compiled either to a CGI executable (which is run by an HTTP server) or to an HTTP server. Fig. 3 shows the Mawl compilation process, to which there are three inputs: the Mawl code; document templates, written in MHTML or MPML (find more on MPML in Section 3.5); and support code, written in a host language. The Mawl compiler takes the first two inputs and performs the traditional compiler steps of lexing, parsing,

semantic checking, and code generation. The Mawl compiler generates code in the host language. This code is compiled by the host language compiler along with the input support code. Currently supported host languages are C++ [22] and Standard ML of New Jersey [14]. A compiled Mawl service is linked with a run-time library to form a complete executable.

In the CGI implementation, the Mawl compiler ensures that session state is properly stored between HTTP requests. A session is started by an initial HTTP request, creating a CGI process. When a session provides service data to a form's `put` method (step 1 of Fig. 1), HTML is created using an appropriate template and sent back to the browser (steps 2 and 3). At this point, the session execution is suspended and its local variables are stored on disk or in a database. Finally, the CGI process exits. When the user submits the HTML page (resulting in a new HTTP request—step 4 of Fig. 1), a new CGI process is started, resuming the session from the point of suspension (step 5), with its state restored. Mawl inserts a session identifier into the HTML so that the subsequent HTTP request will be mapped to the correct session instance. This mapping could also be accomplished using cookies. Sessions that have been dormant for longer than a parameterized time-out (typically, two days) are garbage collected.

The CGI model has high overhead because of the amount of process creation and destruction that takes place (one process for every form `put` executed by a session). Since the Mawl language makes no commitment to a particular implementation model, the Mawl compiler can generate other implementations to address issues of overhead, as was done with the server implementation. In this implementation, a Mawl service compiles to an HTTP server, in which each session instance is a lightweight thread. The compiled `put` method of a form simply suspends the thread after sending the HTML to the browser, as compared to the CGI implementation which requires saving session state to disk. In the server implementation, session state remains in memory and the thread is resumed upon receiving the next HTTP request.

Returning to our comparison between Mawl forms and interface definition languages, we see that the Mawl compiler acts like an IDL compiler. An IDL compiler takes an IDL program and produces the code to manage the transfer of data between the sender of a message (client) and the receiver (server). In the Mawl programming model, the sender of a message is the web service, although this "send" operation compiles to code that responds to an HTTP request. Stated another way, from the programmer's point of the view, the service initiates a request to the user even though it is actually responding to a request from the user. Thus, Mawl reverses the roles of client (browser) and server (web service). Mawl provides a valuable service to programmers by shielding them from the low-level details of the client/server interaction.

3.3 Prototyping Services

The initial implementation of Mawl required that a programmer provide a correctly-typed MHTML template for every form declared in a session. We found that programmers often complained about this requirement, stating that it conflicts with the need to prototype and deploy services quickly. They refer to the advantages of type-free languages such as perl, tcl, and ksh, which are traditionally used to create CGI programs. These languages support prototyping by offering fast turnaround in the edit-compile-debug cycle, as they perform little to no semantic analysis and are interpreted.

To address this problem, the Mawl compiler was modified so that the sessions could be compiled and executed without MHTML templates. This required no change to the Mawl language. The Mawl compiler now generates a default MHTML template when none exists for a form, using the form's type signature (more on this below). Thus, as soon as a service compiles, a programmer can interact with it via a web browser.

With the new implementation, we get the best of both worlds. Static type checking not only prevents a large class of run-time errors in Mawl services, but also assists in prototyping since the programmer is not required to code MHTML to execute a service. Developer experience with this feature has been positive. Mawl's static type system allows the execution of "incomplete" services that contain little or no MHTML. In languages such as perl, programmers are forced to specify some behavior for the incomplete part of a service.

The problem of generating default MHTML from a form's type signature highlights aspects of service specification that the form abstraction does not address. The essence of a form is that it expresses the flow of information

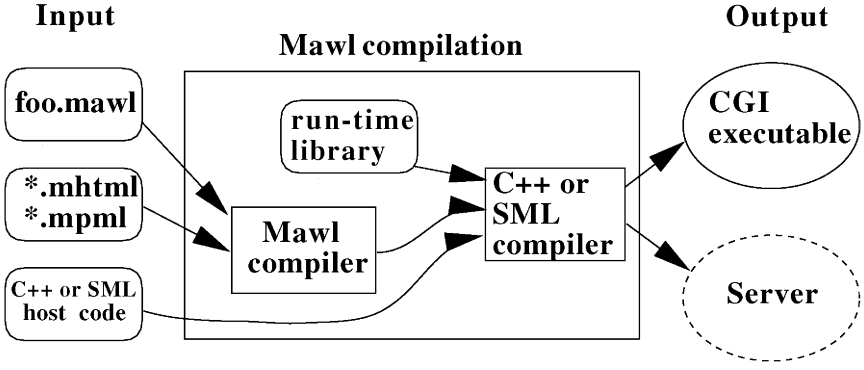


Fig. 3. The Mawl compilation process.

from service to user and back. However, it does not capture any coupling between the outgoing and incoming data that is often expressed in user interfaces. For example, given a form type signature

```
{ [int] intlist } -> { int i }
```

what user interface should be generated? There are at least three possible interpretations:

- select an integer from a list of integers, returning the selected integer
- select an integer from a list of integers, returning the index of the selected integer
- present a list of integers, collect an integer from the user, and return it

In the first two interpretations the returned integer is coupled to the input list. In the third it is not. Our current translation of a form type signature to MHTML does not assume a coupling between the service data and user data. For the above example, the translation presents the list of integers and a separate input field for collecting an integer from the user.

This points to a possibility for a third sublanguage in Mawl, or the use of dependent types [15], which would express the constraints between the service data and user data of a form. An example constraint might state a user field is "one-of" a list field in the service record. Such constraints would be optional and could serve two purposes: to generate better default user interfaces, and to ensure that MHTML, when provided by the programmer, is consistent with the constraints.

The need for the integration of multiple languages is a key aspect of the domain-specific language approach. A domain typically consists of multiple subdomains, each of which may require its own particular language. In the case of Mawl, there are two core sublanguages: the imperative C-like sublanguage of sessions and the declarative sublanguage of MHTML. A constraint sublanguage that would allow the programmer to express expected or required properties of sessions and forms would be quite valuable.

3.4 Testing and Validating

Testing of interactive services is an onerous task. One difficulty is that the only way to test a service may be through the graphical user interface it provides. While a GUI may aid the end-user, it is an obstacle to the tester who is concerned with exercising some aspect of the underlying service logic. Another difficulty is that a web service is inherently a concurrent program, as it is accessed simultaneously by many users. This requires coordinating access and updates to persistent data and ensuring such properties as the absence of deadlock, livelock, etc.

Mawl's separation of a service into an executable set of sessions and a set of HTML templates (joined through the form abstraction) makes it possible to execute a Mawl service under the control of entities other than web browsers. In particular, a Mawl service can be combined with a testing harness to allow batch testing and even state-space exploration [9] of Mawl sessions executing in parallel.

A testing harness simply provides an alternative implementation for each form's `put` method. The harness can check that the input to a `put` method meets various requirements. If it does, the harness can supply the return value of the `put` method from a test file associated with the form. A tester can construct scenarios, develop the necessary test files and drive the service through the scenario, checking invariants along the way. A web browser is not required. While it is possible to test a web service by writing test programs that interact with the service via HTTP requests, these programs must parse the HTML produced by service to determine if the HTML contains valid values. Since Mawl test harnesses receive values directly from the service (via the put method of a form, rather than in HTML), they are much simpler to write and maintain.

Mawl enforces a classic separation of control and data, analogous to that found in some finite state machine formalisms (such as VFSM [8]). Finite state machine formalisms do not model data structures, thus enabling efficient state space exploration. Although Mawl is not finite state, it does separate the specification of control flow and state management from user interface, allowing Mawl sessions to be tested independently of a particular user interface.

3.5 Multidevice Services

Accessibility is a key problem for interactive services. One may access a banking service at home from a graphical web browser. How can one access this service with a cell phone? If two versions of the service are provided, the problem of maintaining consistency between them arises. Typically, programming of interactive voice response (IVR) and web services is done by separate development teams using different programming environments. Coordinating the activities of the two teams to ensure that the two services present a consistent view of the bank and its services to the customer can be difficult in such a situation.

The Mawl architecture supports access from various devices by allowing multiple templates to be associated with a form, as shown in Fig. 4. Mawl uses the identity of

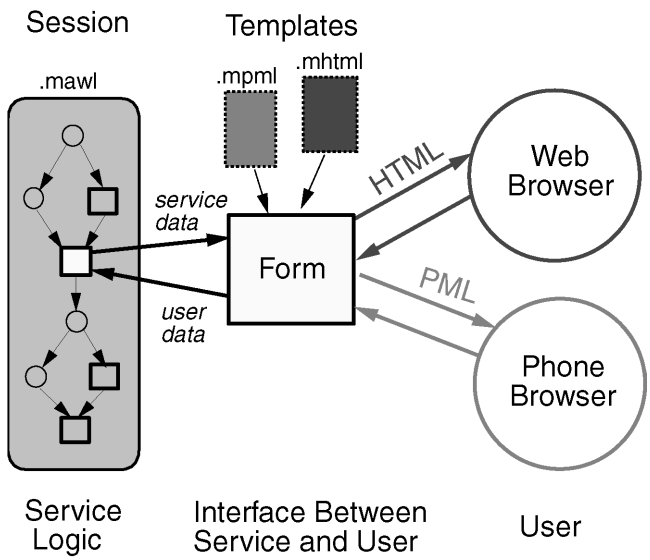


Fig. 4. Supporting multiple devices with the same service logic.

the browser to determine whether to use an HTML template (for the web browser) or a Phone Markup Language (PML) template (for the telephone). PML is a superset of HTML, extended to describe content for interpretation over a telephone. Telephone access to web content is provided by a system called TelePortal, developed at Bell Labs. TelePortal fetches documents from the web, and "reads" them over the telephone via IVR systems. It can also collect data from a user, typically via touchtone or automatic speech recognition.

Clearly, a web browser has much greater capacity than a telephone to present and collect data. While it is easy to turn an IVR service into a web service, creating an IVR service from a web service presents some interesting difficulties. Telephone access to a web service will typically offer less functionality than a web browser.

We have built services that are accessible via both the web and telephone, including the Any-Time Teller, a prototype banking service. In these services, there is one service specification that drives all devices—only the templates change to accomodate new devices. A self-service banking application (such as the Any-Time Teller) uses the same set of forms for both web and telephone interfaces. However, the presentation of information and collection of the information differ radically. In general, a given interaction over the telephone will present less information and collect less information than the corresponding interaction over the web.

As a concrete example, consider the form used for authentication in the Any-Time Teller, which has type signature

$$\{\} -> \{\ \texttt{string name, int acctid, int pin}\ \}$$

On a web browser, an HTML page with three input fields corresponding to the three record fields above is presented; the user may enter either her name or account id, and a PIN to login. Entering alphabetic characters over the telephone touchpad is tedious and error-prone. Thus, the login form should prompt the user only for an account id and PIN, which are integers. Different templates for the form are used to achieve this. The MPML template does not contain an `INPUT` mark for `name`. As a result, TelePortal does not prompt the user for a name, but returns a null value for the field `name`. The MHTML template does include an `INPUT` mark for the field `name`.

3.6 Composing Web Services

A simple but powerful attribute of the web is that new web pages can be easily linked to existing web pages. However, sometimes one wants not merely to link to other web pages, but to combine, collate, or present information from other pages. For example, the MetaCrawler [19] collates results from several search engines. Another example is a web service through which customers can order products. This service might query a courier service (such as FedEx) to show the order's status to a customer.

Web services like these can be composed using existing programming tools such as CGI programs. However, the programming work is tedious, as it involves sending low-level HTTP messages and parsing the retrieved HTML documents to extract the information of interest.

With forms and templates Mawl services can extract data from other web services and treat it as user data. Fig. 5 illustrates the scheme. We use the term "local service" to refer to the service the Mawl programmer is creating. Compare the data flow and use of templates in Fig. 5 with that of Fig. 1. In Fig. 1, a form interacts with a web browser by combining service data with a template to create HTML that is sent to the browser (as a result of the current HTTP request), and receives user data in response (from the next HTTP request). In Fig. 5, a form interacts with a remote web service by sending an HTTP request (parameterized with local service data) and extracts remote service data from the HTML document returned by the remote service.

Templates play a special role in composing web services. Fig. 1 shows that in the usual case MHTML templates are used to generate HTML that is parameterized by service data. Here the `MVAR` marks specify where service data should be inserted into the template. In Fig. 5, MHTML is used as a language for pattern matching against an HTML document. Here, the `MVAR` mark of MHTML is used to bind values in the HTML document to fields in the form's output record. For example, the `ShowInfo.mhtml` template in Fig. 2c can be used to extract the name, count and time information from an HTML document of this structure.

The implementation of this feature requires only that a new `query` method for forms be added to Mawl. To access the remote service, `query` is used instead of `put`. The method `query` takes as input the URL of a remote service, and a record of local service data. Invoking the method causes an HTTP request to be sent to the remote web server. The received HTML is then matched against the MHTML template to extract the relevant data.

We have described the composition of web services in Mawl to show how the use of separate sublanguages supports the composition of web services. Composition is provided by templates, which were originally intended to define a device-dependent mapping of data to presentation,

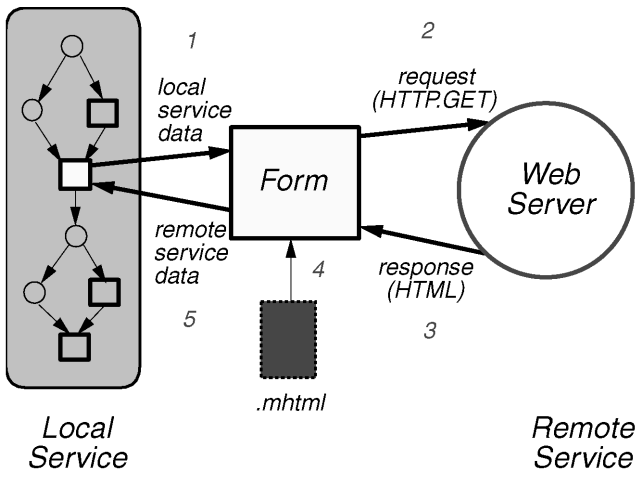


Fig. 5. Using the Mawl architecture to interact with remote web services. Integer labels show flow of data. Compare the use of the template with Fig.1.

but can also be used to map a presentation to the data it contains.

There are other tools and approaches that are more sophisticated in their approach to handling the problem of composition. WIDL, the Web Interface Definition Language [3], is an instance of the eXtensible Markup Language (XML, [6]). Using WIDL one can describe services, their locations, and their inputs and outputs. WIDL output descriptions allow data to be extracted from specific points within a page of HTML. WIDL is more specialized than Mawl, as it is used to describe existing web services, not to create new ones. Also, WIDL combines the information that Mawl separates into forms and templates. However, WIDL's features for extracting data from web pages are more sophisticated than Mawls. For example, with WIDL one can combine pattern matching with references to the structure of a web page. Also, it has an exception-handling mechanism that allows an alternative description of a page's outputs to be used if a first description does not match a page.

3.7 Usage Analysis

While tools for the construction of web sites and services are numerous, most of these tools lack support for the analysis and modification of a service. Usage analysis can help one restructure a service to meet the needs of users better, or improve its performance. For example, analysis of a service might show that users routinely follow paths through a service that are more complicated than necessary. By identifying the pattern and restructuring the service, the service can be improved.

Mawl's form abstraction provides a centralized point to monitor the interactions between service and user. Such monitoring may be difficult to achieve if services are programmed in an ad hoc fashion.

When a session invokes a form's `put` method, instrumentation records the service data sent to the form, the template used to create the user interface, other session-specific information (such as the session identifier and current source line), and timing information. When a user responds to a form, instrumentation records the user data. With forms, we can record not only the amount of time between a request and a response, but the amount of time between the response to one form request and the next. This allows measurement of service performance.

The Mawl system includes a data visualization component called PathView, which is a Java applet that displays user interaction with a service as paths through a graph. As an example, we use PathView to analyze the usage of the LunchBot, a web service for ordering weekly group lunches, which will be described in more detail in Section 4.

Fig. 6 is a bar chart showing LunchBot activity by hour and weekday. Friday usage is highlighted, since all lunches in the period covered by the charts were held on Friday. The weekday chart shows that the LunchBot was used mostly on Thursday and Friday. The hour chart shows that most use occurred on Friday morning. Most people wait until the last two hours before lunch on Friday to order (after an e-mail message is sent announcing the imminent closing of the Lunchbot).

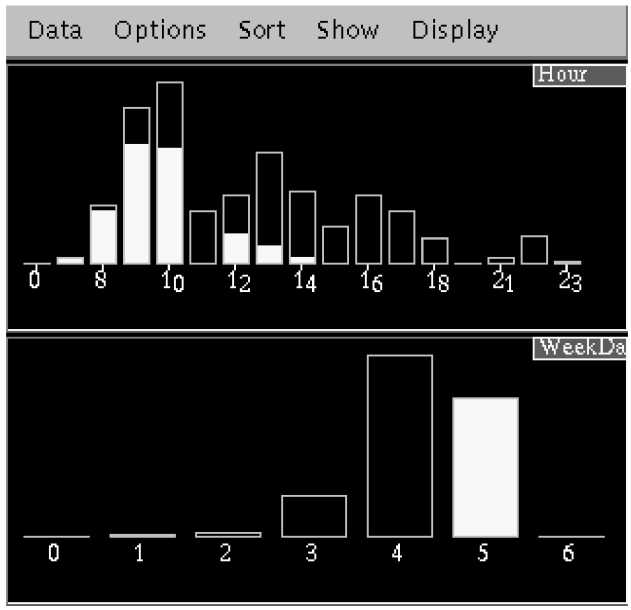


Fig. 6. A bar chart view of LunchBot usage.

One goal of path analysis is to identify common sequences of user interactions and tune the services to create "shortcuts" for these scenarios. Fig. 7a shows a common scenario in the LunchBot: one user ordering lunch. The x-axis represents the sequence of form puts. The y-axis shows the name of the form presented to the user. The visual pattern is that of a mountain peak.

Fig. 7b shows several users ordering lunch. Here, we see the mountain peak pattern as users order lunch. However, in these scenarios, users use the "list order" capability of the LunchBot, which presents the items that have been ordered so far by all users. Some users list orders before ordering lunch (black path). Other user's list orders after ordering lunch (gray path). The frequent occurrence of the (list orders, order lunch) sequence suggests that users like to see what items are popular. Annotating the favorite items on the menu would provide a simple shortcut replacing the more complicated sequence of interactions.

4 Evaluation of Mawl

In this section, we describe our experiences with applying Mawl to web service programming. The description will primarily focus on our experience developing the LunchBot, a service for collecting orders from customers. As the name suggests, it was originally conceived as a means to gather lunch orders, with the customers choosing items from menus. However, we have since applied it to many related areas, including catalog orders, ticket sales, and surveys.

The LunchBot uses password-protected accounts to control access to the system. A special administrator account is used by the operator of the service. This account lets the administrator create menus, which represent lists of items. (The term menu is again historical; a menu can be the luncheon dishes supplied by a restaurant, the items

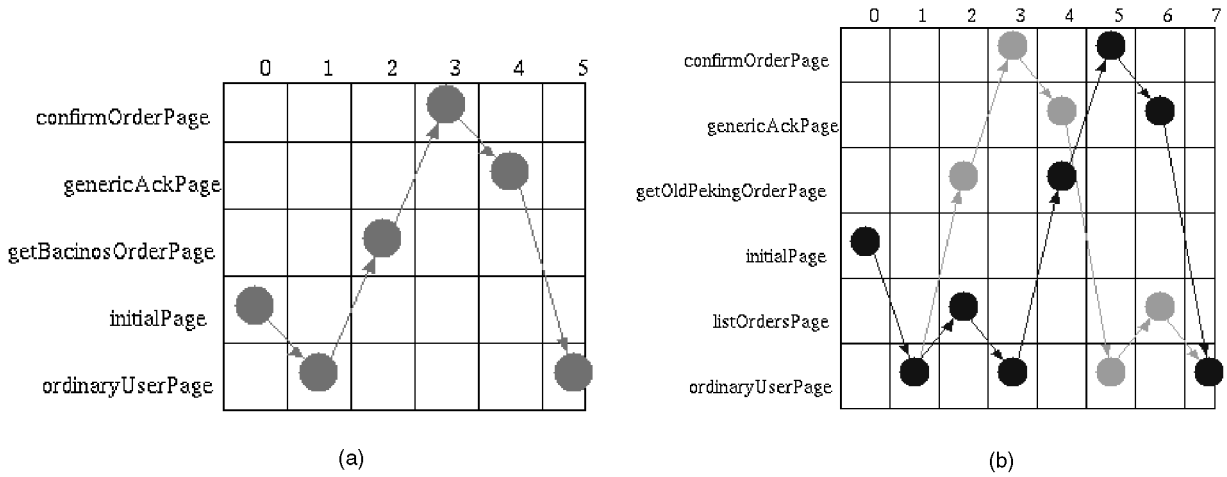


Fig. 7. A typical LunchBot scenario. (a) ordering lunch; (b) the "list orders" variant.

available through a catalog, or the questions to be answered in a survey.)

The administrator can also open an event, selecting a menu from which orders are to be taken. The service collects the necessary information about the event, such as the date of the event and the date by which orders must be placed, and e-mails an announcement to the potential customers. Customers use their accounts to log in to the service and place orders by selecting items from the menu. The LunchBot computes the total charge for the order and deducts it from the customer's balance.

Once the orders have been collected, the administrator closes the event so that no more orders may be taken. The administrator then places the order (for example, by faxing it to a restaurant). When the order has arrived, the administrator uses the service to inform the customers via e-mail. The administrator also enters the amount paid for the order. The LunchBot does all the necessary bookkeeping to keep track of customer balances and the amount that should be in the lunch kitty.

The LunchBot consists of several smaller subservices, such as ordering lunch, examining orders, creating menus, opening events, and so forth. This led to a simple top-level code architecture, as shown in Fig. 8. After logging into the system through an initial series of forms, the user is presented with a primary form that lists the various subservices as links. The set of links displayed depends on whether the user logs in as the administrator or as an ordinary customer. Each link leads to a short series of forms that gather the necessary information for that subservice and update the service data as appropriate. A typical use of the service consists, as shown in Fig. 7, of a sequence of these subservices, possibly with repetitions of individual subservices.

The service data is held in several tables. For example, one table holds account information, with each row corresponding to a single user and the columns holding the various information fields (account name, encrypted password, user name, e-mail address, balance, and so forth). Other tables hold the menus, events, and orders. These tables are represented in Mawl as arrays of structures, with each array element being a table row.

The complete LunchBot consists of about 2,800 lines of Mawl code and 1,800 lines of host (C++) code. Much of the host code performs operations on the lists that represent the service data tables, for example selecting those rows associated with a particular user.

4.1 Mawl Compared with CGI Programming

The utility of Mawl as a service programming language may be evaluated by either of two criteria. The first is to compare the programming effort for a Mawl program with that for an equivalent CGI-based program (e.g., using a combination of shell scripts and Perl or awk programs). When considered in this way, we have found Mawl to be clearly superior to CGI-based programming.

Fig. 9 presents a graphical comparison of a previous implementation of the LunchBot (in shell script and awk) with the Mawl implementation. Each rectangle represents a file and each line in a rectangle represents a single line of code, with length and indentation reflecting that of the underlying source code. Lines are colored to show whether they are part of service logic (gray) or user interface (black). The shell/awk version of the Lunchbot (Fig 9a) shows how service logic and HTML typically are intermixed in services created this way. The Mawl version (Fig. 9b) cleanly separates the service logic and HTML.

Mawl's imperative programming model was successful. Programmers could develop services using familiar flow-of-control constructs, rather than having to chain together many separate programs. This was, of course, greatly facilitated by the invisible way in which Mawl handles state saving and restoration across CGI process boundaries.

The Mawl form-based I/O model, in which the service programmer can treat a form as a function from one type of data structure to another, also was effective. Again, the way in which Mawl completely hides the complexities of the HTTP transport protocol and the CGI interface are a significant improvement over CGI-based programming.

The data-sharing model of Mawl, in which program variables may be either local (local) or shared (global), was adequate for the data-sharing requirements of the LunchBot. The LunchBot data tables are stored as global shared variables, and are thus accessible by all instances of the

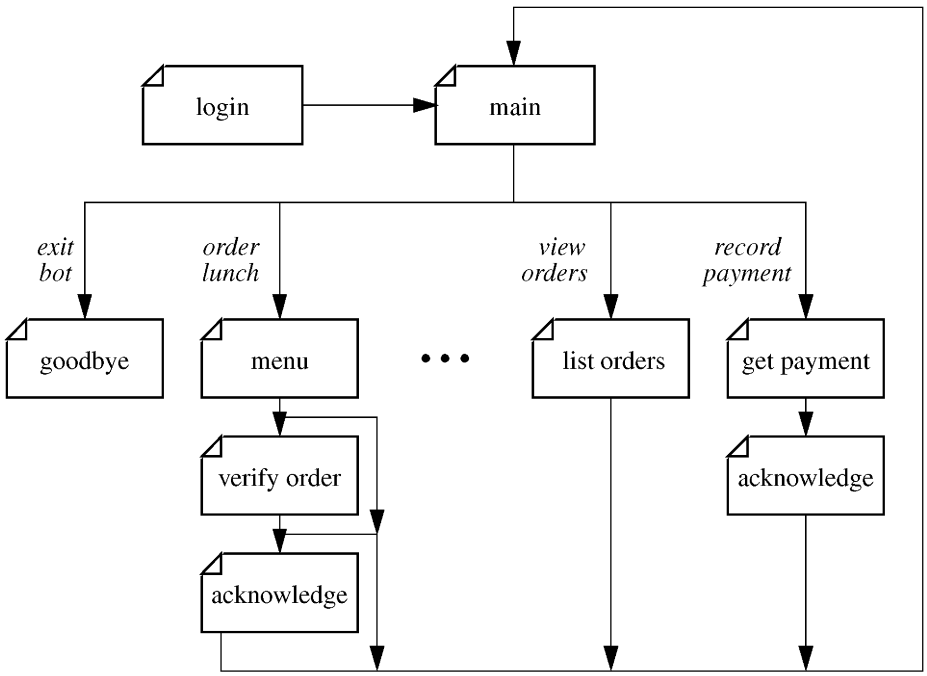


Fig. 8. High-level service logic of the LunchBot. Rectangles represent HTML pages.

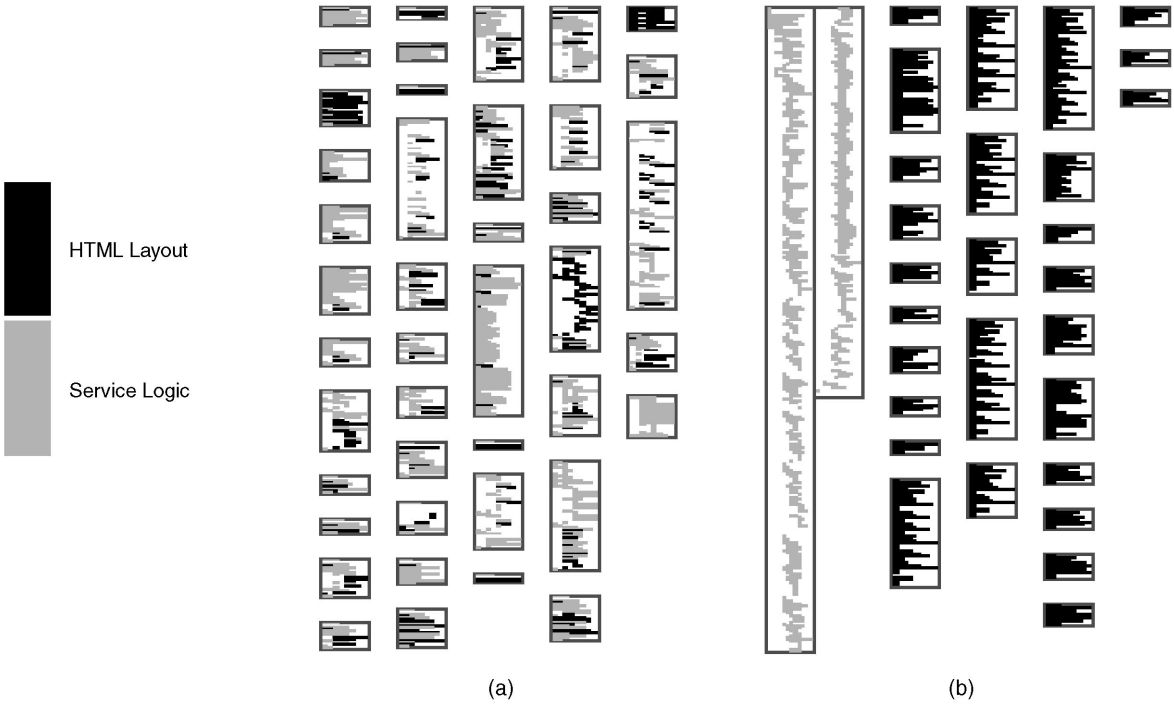


Fig. 9. A global overview of two implementations of the LunchBot. (a) Shell/Awk implementation; (b) Mawl implementation.

service. Mawl's *region* construct, which allows control over mutual exclusion of code execution, was sufficient to manage access to the data and prevent corruption due to concurrent modifications by different users.

Finally, the compilation cycle of Mawl was sufficiently fast to allow an incremental prototyping of the service. This had been a concern, especially as the cycle requires two compilation steps (translation of Mawl to C++, followed by compilation of the C++). However, even for the relatively large LunchBot service, the whole compile time is under one minute on our SGI web server. The testing portion of the cycle was much easier in Mawl than in CGI-based systems, where it is often difficult to test the individual programs without first creating a major portion of the service.

4.2 Mawl as a Service Programming Language

The second means to evaluate Mawl is to consider it as a DSL for programming web services, and to ask whether it

performs this role adequately. Using this criterion, we do identify certain shortcomings of Mawl. However, these shortcomings are largely because Mawl did not go far enough as a DSL—it does not provide suitable abstractions and language constructs for some aspects of web programming.

4.2.1 Data Transformation

One major shortcoming of Mawl is in the area of data representation and data transformation. In retrospect, it has become obvious that many web services are actually interactions with a database system, typically a relational database. A complete DSL for web service programming should, therefore, provide language constructs for declaring, accessing, and manipulating such databases. In this respect Mawl is deficient; while it provides structured types, all data transformations must be accomplished using imperative programming constructs rather than a declarative query language.

We illustrate this with the LunchBot. Recall that the system data is held in tables, exactly as used in a relational model. However, because of the limitations of Mawl data types, each of these tables had to be represented as a global Mawl variable containing a list of structures. Searching these lists is slow in Mawl, so a large portion of the host code is devoted to such searches; this means, for example, that simply looking up a user account requires reading the entire accounts list so it can be passed to a host function. Similarly, changing a row of the table is an assignment to a list element, requiring the list to be saved to disk. (During development of the LunchBot, we actually modified the system runtime to permit efficient saving and restoring of single elements of lists, precisely to address this problem.) These operations could be performed more efficiently if the appropriate database concepts were built into the Mawl language as data types and operations, and were compiled into appropriate calls on the runtime library API.

One might argue that a Mawl program could make use of an external database by placing the necessary access functions in host code. To a great extent, this is precisely what is done in the LunchBot. However, this is likely to result in needless programming effort, particularly given our observation of the centrality of database manipulations in Web services. In particular, it seems almost perverse to require the programmer to write a host-language function as a wrapper for each database access (as is currently required), when a relatively simple SQL-like extension to the language would allow the access to be written in the Mawl program itself.

4.2.2 The Back Button Problem

The other main shortcoming of Mawl was not encountered in the programming of services, but in their subsequent use. A major complaint of LunchBot users in the first few weeks after the service was available was that they cannot use the browser BACK button in Mawl. Users of Mawl services must adapt to pressing a Mawl-supplied "Continue" button on a page to get to the next page of the service. Previous pages represent old program states that are no longer accessible.

This is a serious problem, in that users like to use the BACK button—for example, it is the natural way to correct an error in form entry, rather than cancelling the entire operation and restarting the entry process. The main reason the BACK button cannot be used is that certain transitions from page to page are accompanied by data manipulations that are irreversible. For example in the Lunchbot it does not make sense to allow the user to back up after having confirmed an order, as that operation causes several changes which cannot easily be undone. However, it would be perfectly reasonable to permit the user to back up from the page where she must confirm her order to the previous page with the menu so she can change the order.

Restated, while the sequential paradigm of Mawl services aids programmers, it places a burden on the users of a Mawl service. When interacting with a service from a web browser, users expect to be able to use the browser commands to skip around from page to page. However, in Mawl services the only valid page is the current page. The end result is that the service controls the browsing, rather than the user.

We must also distinguish between the BACK operation, whereby the user *views* a previous state, with an "undo" operation, whereby the user *returns* to a previous state. As browsers have no UNDO button, and indeed most browsers do not even notify the service when the BACK button is used, the only practical way to recognize an undo operation is when the user attempts to resume execution from a previous page. The overall problem is additionally complicated because most browsers provide a history list, so the case where the user tries to undo more than one step of execution must be handled.

A possible solution, working within the sequential programming model of Mawl, is to augment the transitions with syntactic marks that indicate whether it is possible to back the state into or across them. These marks might also form part of an improved database interface, as they bear obvious resemblances to database checkpointing and rollback operations. These marks would determine whether the user could resubmit a previous form, that is, roll back the state. If the operation is allowed, the state is rolled back and execution resumed from the new point. Otherwise, the user is notified and execution resumes at the last-reached state.

A different approach to the BACK button problem would be to reconsider the imperative model used by Mawl. For example, the model might be replaced with a declarative language model and reactive execution model, e.g., in the style of rule-based systems. The use of the BACK button could then be handled by writing specific rules. Such a model might also simplify other services, for example survey services in which a user must fill out a number of forms but the sequence in which the forms are completed is not important.

5 Summary

Mawl was created to address two specific problems in the creation of dynamic web services: the lack of compile-time guarantees about services, and the low level of programming involved in coding CGI programs. The form is the basic abstraction that helped to solve these problems by

enforcing a separation of concerns between service logic and user interface. The Mawl service architecture and form abstraction have been quite stable since the language's inception and have been used to solve several new problems quite different in nature from the initial two: prototyping, testing, and composing services, accommodating multiple devices, and enabling usage analysis. The only solution that required a change to the language (and a minor change, at that) was the composition problem. All the other solutions only changed the compiler analysis or runtime infrastructure.

ACKNOWLEDGMENTS

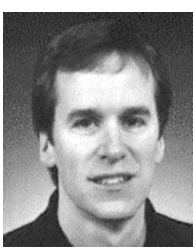
Christopher Ramming and David Ladd invented the Mawl language. Thanks to other members of the Mawl team for their input, including Michael Benedikt, Peter Danielsen, Peter Mataga, Ken Rehor, and Curt Tuckey. Thanks to Natasha Tatarchuk for her work on the visualization applets. Thanks also to Mooly Sagiv and Mike Siff for their perceptive comments and recommendations.

REFERENCES

[1] D. Atkins and T. Ball et al., "Experience with a Domain Specific Language for Form-Based Services," *Proc. Usenix Conf. Domain Specific Languages,* pp. 37–49, Oct. 1997.
[2] D. Atkins and T. Ball et al., "Integrated Web and Telephone Service Creation," *Bell Labs Technical J.,* vol. 2, no. 1, Winter 1997.
[3] C. Allen, *WIDL: Application Integration with XML,* O'Reilly, 1997.
[4] T. Berners-Lee, "Hypertext Transfer Protocol (HTTP/1.0)," *Working Group of the Internet Eng. Task Force,* Oct. 1995.
[5] T. Berners-Lee and D. Connolly, "Hypertext Markup Language (HTML 2.0)," *Working Group of the Internet Engineering Task Force,* Aug. 1995.
[6] T. Bray, J. Paoli, and C.M. Sperberg-McQueen, eds., " Extensible Markup Language (XML) 1.0, 1998. http://www.w3.org/TR/1998/REC-xml-19980210
[7] E.W. Dijkstra, "Go to Statement Considered Harmful," *Comm. ACM,* vol. 11, no. 3, pp. 147–148, Mar. 1968.
[8] A.R. Flora-Holmquist and M.G. Staskauskas, "Formal Validation of Virtual Finite State Machines," *Proc. Workshop Industrial-Strength Formal Specification Techniques (WIFT'95),* pp. 122–129, Boca Raton, Fla. Apr. 1995.
[9] P. Godefroid, "Model Checking for Programming Languages Using Verisoft," *Proc. 24th ACM Symp. Principles of Programming Languages,* pp. 174–186, Paris, Jan. 1997.
[10] Object Management Group. "The Common Object Request Broker: Architecture and Specification," Technical Report, edition 2.0 July 1995.
[11] D.G. Korn, "ksh—A Shell Programming Language," Technical Report, AT&T Bell Laboratories, 1986.
[12] B.W. Kernighan and D.M. Ritchie, *The C Programming Language.* Englewood Cliffs, N.J.: Prentice Hall Software Series, second edition, 1988.
[13] D.A. Ladd and J.C. Ramming, "Programming the Web: An Application-Oriented Language for Hypermedia Service Programming," *Proc. Fourth Int'l World Wide Web Conf.,* pp. 567–586, World Wide Web Consortium, Dec. 1995.
[14] D.B. McQueen and A. Appel, "Standard ML of New Jersey," *Proc. Third Int'l Symp. Programming Language Implementation and Logic Programming,* pp. 1–2, Springer-Verlag, 1991.
[15] D.B. MacQueen, "Using Dependent Types to Express Modular Structure," *Proc. 13th Ann. ACM Symp. Principles Of Programming Languages,* pp. 277–286, Jan. 1986.
[16] J.K. Ousterhout, *Tcl and the Tk Toolkit* Addison-Wesley, 1994.
[17] L.A. Rowe, "Fill-in-the-Form Programming," *Proc. Conf. Very Large Data Bases,* pp. 394–404, 1985.
[18] L.A. Rowe and K.A. Shoens, "A Form Application Development System," *Proc. 1982 ACM SIGMOD Conf. Management of Data,* pp. 28–38, 1982.
[19] E. Selberg and O. Etzioni, "Multi-Engine Search and Comparison Using the Metacrawler," *Proc. Fourth Int'l World Wide Web Conf.,* pp. 195–208, World Wide Web Consortium, Dec. 1995.
[20] A. Silberschatz, H.F. Korth, and S. Sudarshan, *Database System Concepts.* McGraw-Hill, 1997.
[21] R. Srinivasan, "Remote Procedure Call Protocol Specification Version 2," Technical Report RFC 1831, Sun Microsystems, Aug. 1995.
[22] B. Stroustrup, *The C++ Programming Language.* Addison-Wesley, 1986.
[23] L. Wall and R.L. Schwartz, *Programming PERL.* O'Reilly & Assoc., 1990.

David L. Atkins received a BA degree in mathematics from the College of Wooster in Ohio and MA and PhD degrees in mathematics from the University of Kansas. Atkins is a member of technical staff in the Software Department Research Department at Bell Labs, Naperville, Illinois, and telecommutes full time from Eugene, Oregon. His research interests include programming languages, version management, and software visualization as they relate to software production. He is a member of the IEEE.

Thomas Ball received a BA degree in computer science from Cornell University in 1987 and the MS and PhD degrees in computer science from the University of Wisconsin at Madison in 1989 and 1993, respectively. Ball is a member of the technical staff in the Systems and Software Research Center of Lucent Technologies' Bell Laboratories. His research focuses on improving the software production cycle through the application of domain specific programming languages, the dynamic and static analysis of programs, and software visualization. He is program chair of the 1999 Conference on Domain Specific Languages. He is a member of the ACM and IEEE Computer Society.



Glenn Bruns received the PhD degree in computer science from the University of Edinburgh, Scotland. Dr. Bruns is a member of the technical staff at Bell Laboratories, Lucent Technologies. From 1985-1989 he was a member of the technical staff at MCC, where he worked in the Software Technology Program. From 1990-1996 he was a senior research fellow at the Laboratory for Foundations of Computer Science at the University of Edinburgh. Since 1996 he has been at Bell Laboratories, where he works in the Software Production Research Department. He is interested in the application of concurrency theory and temporal logic to the design and analysis of distributed systems. He is the author of Distributed Systems Analysis with CCS (Prentice Hall, 1997).

Kenneth Cox received the BS degree (1985) in computer science and electrical engineering from Washington University in St. Louis Missouri, and the MS and DSc degrees (1993) in computer science, also from Washington University in St. Louis. He is currently employed as a member of technical staff in Bell Laboratories research, part of Lucent Technologies, in the Software Production Research Department in Naperville Illinois. His research interests include information visualization with applications to user interactions.