Experience with a Domain Specific Language for Form-based Services

David Atkins, Thomas Ball*, Michael Benedikt,
Glenn Bruns, Kenneth Cox, Peter Mataga, Kenneth Rehor
Software Production Research Department
Bell Laboratories, Lucent Technologies
http://www.bell-labs.com/projects/MAWL/

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. A form provides a user interface that presents service data to the user, collects information from a user and returns it to the service.

Mawl is a domain-specific language for programming form-based services in a device-independent manner. We describe our experience with mawl's form *abstraction, which is the means for separating application logic and user interface description, and show how this simple abstraction addresses six issues in service creation, analysis, and maintenance: compile-time guarantees, implementation flexibility, rapid prototyping, support for multiple devices, composition of services, and usage analysis.*

1 Introduction

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 the time of day), collects information from a user (such as her name), and returns it to the service.

Both traditional interactive voice response (IVR) services and newer web services fit the form-based service paradigm. 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. A web service sends an HTML page to a user's (graphical) browser, providing information and a set of input fields to request information such as account and password.

Many services and devices fit the form-based interaction paradigm. For example, a banking service might be described by a set of forms, independent of a particular device, such as an automated teller machine, web browser, or telephone. Presentation and collection of information will differ radically, as suited to the device. A presentation of account information to a web browser might show a table of account information, including status, balance, and interest. The corresponding presentation to a telephone would provide this information, or perhaps a subset thereof, in a conversational manner by "reading" the account information to the user. Nonetheless, the basic interaction (present account information to the user) can be specified via a generic form.

Mawl is a domain-specific language (DSL) for programming form-based services in a device-independent manner [LR95, Aea97]. Mawl separates the specification of application control flow and state management from the specification of a user interface. As a result, one can code an application that is accessible via a web browser and, with minor modifications to only the user interface specification, make the application accessible via interactive voice response (IVR) platforms.

This paper describes our experience with mawl's *form* abstraction, which is the means for separating application logic and user interface descriptions. The initial impetus for the form abstraction was to simplify the creation and maintenance of dynamic web services[1] based on the Common Gateway Inter-

*Correspondence contact: tball@research.bell-labs.com, Room 1G-359, 1000 E. Warrenville Rd., Naperville, IL 60566.

[1] In this paper, we will generally use the word "service" to refer to an "application", and the word "logic" to refer to

face (CGI). In particular, mawl was constructed to provide certain compile-time guarantees about the behavior of web services, as well as platform independence and implementation flexibility by hiding the details of the CGI and HTTP (HyperText Transmission Protocol [BL95]) protocols from the programmer via language abstractions.

Our continued experience with mawl's form abstraction illuminates how a simple language abstraction can improve many parts of the software development life cycle. In addition to addressing the above two problems, the form abstraction has provided straightforward solutions to four other problems:

- *Prototyping services.* Many programmers equate "prototyping" languages with typeless, interpreted languages such as perl [WS90], tcl [Ous94] or ksh [Kor86]. In contrast, mawl supports prototyping of web services via a static type system, allowing services to be tested from a web browser without the need to write any HyperText Markup Language (HTML) documents [BLC95].
- *Supporting multiple devices.* One measure of a domain specific language is how well it supports changes that were not directly planned for in its initial design. Mawl first targeted the graphical web browser, but naturally accommodates the integration of new devices into existing services. We report on our experience in developing a number of 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 [SE95]). In most cases, the only programming interface to remote 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.

Section 2 present 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 six problems in service creation, analysis, and maintenance. Section 4 summarizes the paper and Section 5 tells how to obtain mawl.

2 Mawl: History, Service Architecture, and Language

This section describes the brief history of mawl, the mawl service architecture and its implementation in a domain-specific language, and, finally, some details of the language.

2.1 A Brief History

Mawl was created in early 1995 because of two major difficulties experienced in programming form-based web services via Common Gateway Interface (CGI) scripts.

First, when programming such scripts in a general purpose language such as C, perl or ksh, one sacrifices traditional compile-time guarantees about the consistency of the service logic and user interface code. For example, it is difficult to automatically determine if a service will generate correct HTML, or if the service is prepared to deal with whatever data may be submitted via a `FORM` mark in the generated HTML. Such basic questions may be very difficult, if not impossible, to answer in the context of a general purpose language. Another contributing factor to this problem is that many web services are programmed in an ad-hoc fashion, lacking even a basic service architecture. As a result, service logic and user interface description are often intermingled, as shown in Figure 1(a). This makes it nearly impossible to make any sort of compile-time guarantees.[2]

Second, adherence to the HTTP/CGI protocols places burdens of low-level implementation detail on the programmer. With the HTTP request/response

the specification of control flow and state management of a service.

[2] This problem is not confined to web services. Most programming environments for IVR services also have a very tight coupling of service logic and user interface description.

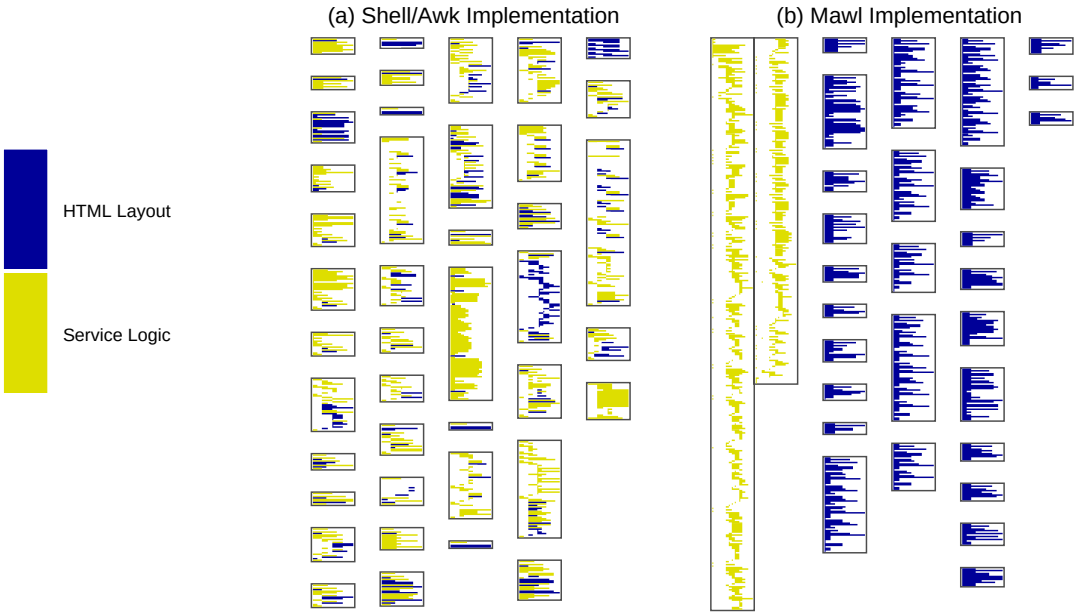


Figure 1: A global overview of two implementations of a web service called the LunchBot. Each rectangle represents a file and each line in a rectangle represents a single line of code, with length and indentation reflecting that of the underlying source code. Lines are colored to show whether they are part of service logic (yellow/grey) or user interface (blue/black). The first version of the Lunchbot (a) was programmed using shell and awk, in which service logic and HTML specification are intermixed. The second version (b) was programmed in mawl, which cleanly separates the service logic and HTML.

paradigm, a CGI process is started to respond to an HTTP request. Once the CGI process has sent the requested data, it terminates. However, many services require sequencing between pages, and the maintenance of persistent state on the server. As a result, programmers code by hand what is generated automatically by compilers for traditional sequential languages. A CGI library may provide some assistance, but the resulting program is nonetheless intertwined with a specific implementation model.

To address these problems, mawl presents an architecture for form-based service creation that is independent of the HTTP and CGI protocols. Programmers are given the illusion of a traditional imperative language in which they may code a centralized service, rather than a set of scripts coupled indirectly through HTML pages. The mawl compiler translates the program into a HTTP/CGI implementation, or into a stand-alone server implementation. As shown in Figure 1(b), mawl cleanly separates service logic and user interface (HTML layout).

2.2 The Mawl Service Architecture

Figure 2 illustrates three main abstractions in the mawl service architecture: sessions, forms, and templates. A service contains one or more *session*s. A session specifies the control flow of a service and the update of service state (persistent and per-session), which may involve concurrency control. 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 interacts with the user via the *form* abstraction. A form is an object that:

1) receives data from a session;
2) creates a document by dynamically parameterizing a template with the data;
3) presents the document to the user (via a browser);
4) accepts a response from the user;
5) returns the extracted user data to the session.

A *template* defines the static portion of a user interface as well as the dynamic portions that are param-

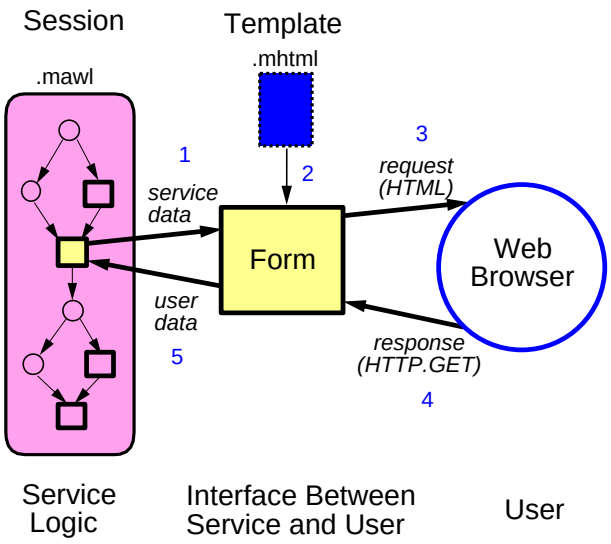


Figure 2: The mawl service architecture. Integer labels show flow of data.

eterized by values passed to the form by a session.

Figure 3 contains a simple mawl service that will be used to explain the three abstractions in more detail.

2.2.1 Sessions

A service has one or more *sessions,* each of which defines a sequence of interactions with a user. In our example, there is one session, `Greet`, that interacts twice with the user, first prompting for the user's name and then greeting the user, displaying a count of the number of visitors to the service, and the elapsed time between the presentation of the first and second pages to the user. The service logic, written in mawl, is shown in Figure 3(a). [3]

Mawl provides a persistence model that allows programmers to specify the type of storage required for mawl variables. Variables may exist on a per-session instance basis (as declared by the keyword `auto`) or persist over all session instances (as declared by the keyword `static`). In our example, the variables `GetName`, `ShowInfo`, `time_now` and `i` are per-session variables, while `access_cnt` is a persistent variable.

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

[3] Mawl has standard imperative constructs for looping, conditional control flow, procedure calls, exceptions, etc. Mawl is a statically typed language.

2.2.2 Forms

The main role of a form is to take data (service data) from a mawl session, present it to the user, collect information from the user (user data), and return it to the session. The first two lines of session `Greet` in Figure 3(a) declare two forms, `GetName` and `ShowInfo`. A form object is declared in a session with a type signature specifying the structure of the expected service data and user data records. 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.

A form has a `put` method by which a service/user interaction takes place. The session provides the form's `put` method with a record containing the service data, and in return receives a record containing the user data. This is shown in Figure 3(a), where the service first provides `GetName` with its required (empty) input record and receives back a record 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 a form 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 [Gro95] (Common Object Request Broker Architecture) and RPC [Sri95] (Remote Procedure Call) both have IDL specification languages. Just as with forms, IDL programs only express the signatures of operations, but do not describe their computation. We will return to this comparison later in Section 3.

A form is associated with a template. The linkage in the example here is by common name.

2.2.3 Templates

The service data sent to a form is used to generate an interface by parameterizing a template. Templates are specified separately, in the user-interface languages appropriate to the various devices. Figure 3(b) and (c) shows templates written in the language MHTML. MHTML is an extension of HTML that is used for creating templates. In MHTML, the

(a) Greet.mawl:

```
static int access_cnt = 0;
session Greet {
  auto form {} -> { string id } GetName;
  auto form { string id, int cnt, int time } -> {} ShowInfo;

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

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

Figure 3: 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`.

values of a form's service data may be accessed using the `MVAR` mark, among others. This mark indicates substitution of the value of the service data 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.[4]

A template represents one possible "implementation" of a form (more precisely, an implementation of a form's `put` method), for a particular browser which will "execute" it. A form may have no templates 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 3.4).

[4] Note that the MHTML in Figure 3 does not contain any `FORM` mark which specifies the CGI script to be executed upon submission of the `FORM`; the mawl compiler and run-time system 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.

2.3 Mawl Types

Mawl has four fundamental 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 specifiers and identifier pairs in braces. For example,

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

defines a record named `customer` with a field `name` that is a string and a field `age` that is an integer. Record values can be constructed "on the fly", as shown in Figure 3.

Lists in mawl behave much like arrays in C, although there is no storage allocation required. A list type is denoted by enclosing another type in brackets. For example, a list of strings would be declared as:

```
auto [ string ] names;
```

The list elements are denoted using brackets and an integer index: `names[i+1] = names[i]`. Lists grow

automatically to accommodate such references. List values may be formed by enclosing a comma separated list of values in brackets:

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

As illustrated earlier, the syntax of the form type specifier is the keyword `form` along with two type specifiers, the service type and the user type, where the types are separated by the suggestive token `->`. The service and user type specifiers must be record types.

For example,

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

might 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, the `MVAR` mark in MHTML is used to insert scalar data into a template. The `MITER` mark is used to substitute mawl list values. A natural use of a list might be to display a table with a variable number of rows and/or columns. The construct `<MITER>...</MITER>` is used to iterate over mawl lists and generate HTML that is dependent on the list element values. The `MITER` mark uses the `NAME` attribute to specify the name of a list field in the form's service data. The additional attribute `MCURSOR` 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.

3 Experience with the Form Abstraction

This section discusses how the form abstraction helps to address six different problems in form-based services. In addition, we touch upon various languages issues that arose and discuss related work.

3.1 Compile-time Guarantees

In many web services and IVR services, service logic and user interface code are inextricably interleaved, as shown in the old version of the Lunchbot (Figure 1(a)). Consequently, reasoning about one inevitably requires reasoning about another. Often, the service logic and user interface are coded in the same general purpose language, which can provide compile-time checks about the consistent use of module interfaces. However, the interface through which the user interface is specified may be too low-level and dynamic to say anything meaningful about the flow of information between service and user at compile-time (witness the Tk widget set).

Mawl's division of a service into service logic code and user interface code allows a great deal of consistency checking to be performed at compile time. First, the service logic and the MHTML may be independently analyzed to ensure that they are internally consistent. For the service code, this means standard type checking and semantics checking. For MHTML, this means verifying that a template is legal MHTML.

Additionally, the service logic (i.e., `Greet.mawl` in Figure 3) and the MHTML templates may 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 respect to the form's type signature.

For example, Figure 3(b) 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

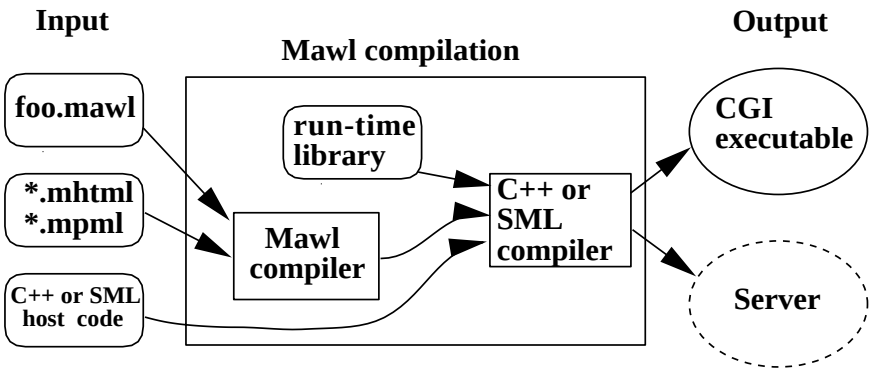


Figure 4: The mawl compilation process.

type signature of the associated `GetName` form. Similarly, the template in Figure 3(c) 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 list type 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 multi-device services, as discussed in Section 3.4.

Our combination of a declarative languge (MHTML) with a sequential imperative language (mawl) can be thought of as a language embedding, such as SQL in C. The form uses 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. This degree of compile-time checking is much greater than traditionally found in embedded languages. Of course, this comes at a price: construction of a parser and semantics checker for MHTML as well as for mawl. For a more complicated embedded language, this may be too great a luxury to afford.

3.2 Implementation Flexibility

A main advantage of the mawl architecture is implementation flexibility, which is realized by having service logic centrally specified in a session and via the form, which identifies the point at which a session relinquishes control to the user and at which control returns to the service.

Figure 4 shows the mawl compilation process, to which there are three inputs: the logic of the service, written in mawl; document templates, written in MHTML or MPML (more on this markup language in Section 3.4); and support code written in a 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++ [Str86] and Standard ML of New Jersey [MA91]. Support for Java [GA96] is planned. A compiled mawl service is linked with a run-time library to form a complete executable.

A service can be compiled either into a CGI executable or into a stand-alone server. In the CGI implementation, when a session sends out HTML via a form `put`, execution of the session is suspended and the session-instance (`auto`) state is stored on disk or in a database. This is necessary since the CGI process terminates once the HTTP request that started the process has been fulfilled. When a response is received, execution of the session picks up from the point of suspension, with the session-instance state restored. Mawl encodes the session instance in a unique identifier that is stored in the `ACTION` field of the `FORM` mark inserted into the HTML by mawl. Execution of the session continues until another form is encountered, sending another document to the user. Once a session ends, the storage for session-instance state is released. In the server implementation, each session instance is a thread and the compiled `put` method simply suspends after sending the HTML. Again, the identity of the thread is embedded in the HTML so that the server can determine which thread to awaken upon receiving another HTTP request, or if it needs to start a

new thread.

Returning to our comparison between the mawl form and interface definition languages, we see that the mawl compiler acts very much 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 into a code whose function is to respond to an HTTP request. Thus, mawl reverses the roles of client (browser) and server (web service).

As with many IDL compilers, the mawl compiler performs little transformation or optimization, assuming that the transport medium is slow. Mawl has a further advantage because a form is presented to a human to fill out, which lengthens the delay considerably. However, as we will see in Section 3.5, one advantage of the form abstraction is that it allows mawl services to interact with other web services. Thus, recent work on optimization for IDL compilers [EFF+97] might be applicable to mawl for such settings.

3.3 Prototyping Services

Section 3.1 discussed the advantages of compile-time checking of a service against MHTML, which is possible because of the separation of service logic and user interface via the form abstraction. In our experience with mawl at Bell Labs, we have found that programmers sometimes balk at using a statically typed language for web service programming, complaining that type checking impedes rapid prototyping. The requirement that a service and its MHTML type check conflicts with the demands of "Internet time", which requires that services be prototyped and deployed quickly. Programmers often refer to the advantages of type-free languages such as perl, tcl, and ksh, which are traditionally used to program CGI scripts. These languages support prototyping by offering fast turnaround in the compile-edit-debug cycle, as they perform no semantic analysis and are interpreted.

Compounding this problem was the fact that the initial implementation of mawl was overly restrictive, requiring an MHTML template for every form declared in the service logic. To address this, the mawl compiler was modified so that the service logic language could be compiled, executed and tested without any 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 a soon as a service compiles, a user 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. This points to an interesting measure for a domain-specific abstraction: does the abstraction capture some *necessary* part of the domain? In the domain of form-based services, a service programmer must decide, for each service/user interaction, what information will be exchanged between service and user. This decision is totally independent of the implementation language but is absolutely necessary in order to build a service. In general purpose languages such as perl, tcl, and ksh, the way in which this "signature" is encoded can vary widely and may be quite dynamic. With mawl, the form abstraction (via its static type signature) captures this information precisely in a localized, analyzable construct.

Developer experience with this feature has been quite positive. Mawl's static type system allows the execution of "incomplete" programs that do not contain some or any MHTML. In languages such as perl, the lack of a type system means that incomplete programs are not possible, forcing the programmer to specify some behavior for the incomplete part. The form type signature enables the automatic generation of MHTML, in addition to providing compile-time guarantees when MHTML is present. Using static types to specify "user interface types" can be a boon to prototyping.

3.3.1 Deriving MHTML from Type Signatures

We now discuss some of the issues in generating default MHTML from form types. The essence of a form is that it expresses the flow of information from service to user and back. However, it does not express 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 }
```

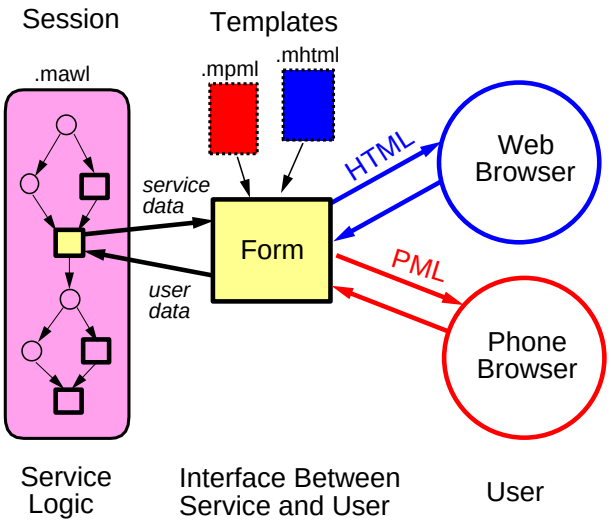


Figure 5: Supporting multiple devices with the same service logic.

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

The first two interpretations are equivalent in the sense that the returned integer is tightly coupled to the input list, but are clearly distinct from the third. Our current transformation of a form type signature to MHTML does not infer a coupling between the service data and user data. The service data is displayed via HTML tables and appropriate marks are included to get data from the user.

This points to a possibility for a third sublanguage in mawl, 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; to ensure that MHTML, when provided by the programmer, is consistent with the constraints.

3.4 Multi-Device Services

This section shows how the form abstraction allows a service to interact with multiple browsers. In particular, we focus on supporting both the graphical web browser and the telephone. The issues arising in telephone access to services include many of those that will arise when making services available to a large and diverse collection of devices; indeed, it is hard to imagine two user interfaces more dissimilar than the telephone and the graphical web browser.

The mawl architecture supports multiple devices by allowing multiple templates to be associated with a form, as shown in Figure 5. Mawl uses the `USER_AGENT` of the requester to determine whether to use an HTML template (for the web browser) or a PML template (for the telephone browser). The Phone Markup Language (PML) is a superset of HTML, extended to describe content for interpretation over a telephone. The telephone browser is provided by a system called TelePortal, developed at Bell Labs. TelePortal fetches documents from the web, and "reads" them over the telephone via interactive voice response (IVR) systems. It can also collect data from a user (typically via touchtone or automatic speech recognition).

It is clear that the graphical web browser has a much greater capacity than the telephone browser interface to present and collect data. It is an easy translation to take an IVR service and turn it into a (rather dull) web service. The other direction presents some interesting difficulties, as discussed below. In general, telephone access to a web service will typically offer a limited subset of the functionality available from a graphical web browser.

We have built a number of services that are accessible via both the web and telephone.[5] In all cases, there is one service specification that drives all devices–only the templates change. A self-service banking application (such as the Any-Time Teller) requires essentially the same set of forms for both web and telephone interfaces. However, the presentation of information and collection of the information differs radically, as suited to the device. In general, a given interaction over the telephone will present less information and collect less information than the corresponding interaction over the web.

[5] For a demonstration of a prototype banking service (the Any-Time Teller) with both web and telephone interfaces, visit
`http://www.bell-labs.com/projects/MAWL/anytime.html`

As a concrete example, consider the "login" form of the Any-Time Teller, which has type signature

```
{} -> { 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. This is accomplished by having two different templates. The MPML template uses the "hidden" attribute for the `INPUT` mark for `name`. As a result, TelePortal (the telephone browser) does not prompt the user for a name, but does return a null value for the field. The MHTML template does not use the hidden attribute, so that an input field appears for the `name` attribute.

There are many other ways to support multiple devices when programming services. For example, a general purpose language such as Java, supported by the Java Virtual Machine, allows a multitude of devices to be programmed in a single language. The Inferno operating system [SMD97] represents a similar approach, but starts with the operating system as the common denominator rather than a language. While such work definitely improves the state of programming for heterogeneous collections of devices, the question of a software architecture for service creation is left open. Form-based services will continue to be prominent even as the ends of the network become smarter. We have started to explore mawl services in which a form (or set of forms) is represented by a Java applet, which would allow mawl program to interact with Java-enabled devices.

3.5 Composing Web Services

A simple but powerful attribute of the web is that new web pages can be easily linked to existing web pages. It is similarly desirable to build new web services by combining, collating, or re-presenting information from other web services. An example of such a service is the MetaCrawler [SE95], which collates results from several search engines. Another example is a web service through which customers can order products. This web service might query a courier service (such as FedEx) to present the status of an order to the user.

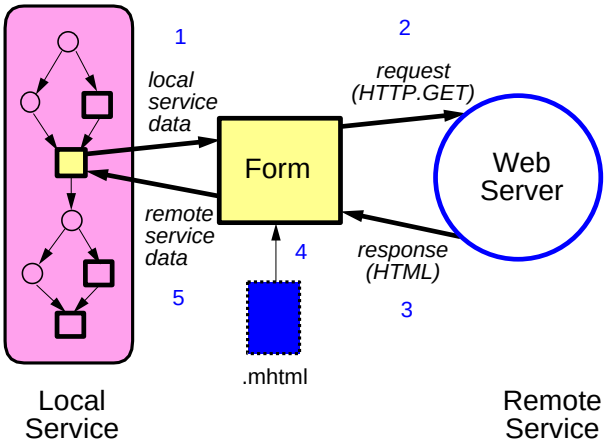


Figure 6: Using the mawl architecture to interact with remote web services. Integer labels show flow of data. Compare the use of the template with Figure 2

It is possible to compose web services like these using existing programming tools such as CGI scripts. However, the programming work is tedious, as it involves the sending of low-level HTTP messages and the parsing of HTML documents to obtain the information of interest.

The form abstraction (along with templates) allow mawl services to interact with other web services, as illustrated in Figure 6. We contrast the data flow and use of templates in this figure with that of Figure 2, where the service is interacting with a web browser. We will use the term "local service" to refer to the service the programmer is creating. As shown in Figure 2, a form interacts with a web browser by combining service data with a template to create HTML that is sent to the web browser (as a result of the current HTTP request), and receives user data in response (in the form of the next HTTP request). However, as shown in Figure 6, a form interacts with a remote web service by sending an HTTP request to the remote service, parameterized with local service data, and extracting remote service data from the HTML document returned by the remote service.

A template is used to extract the remote service data from fields in the HTML document. That is, MHTML is used as a language for pattern matching against an HTML document. When MHTML is used to generate HTML to send to a web browser (Figure 2), the `MVAR` marks specified where to substitute service data into the template. Now, MHTML is used to parse the HTML sent back from the remote web service. In this case, the `MVAR` mark of MHTML is used to bind values in the HTML doc-

ument to fields in the form's return record. Thus, the `ShowInfo.mhtml` template in Figure 3(c) 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 with the MHTML template in order to extract the relevant data.

3.6 Usage Analysis

While tools for the construction of web sites and services are numerous, most of these tools lack support for the later parts of the software life cycle: the analysis of a service, and the subsequent modification of the service. By analyzing usage, a service provider can 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, providers can improve their services.

Ideally, the analyses of service/user interactions should come for free as a side-effect of a service. Such logging may be difficult to achieve if services are programmed in an ad-hoc fashion, where the flow of information between service and user is not clear. Mawl's form abstraction provides a centralized point at which to monitor the interactions between service and user. A flag to the compiler or run-time system enables logging of each interaction.

3.6.1 Logging

When a session invokes a form, 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), as well as timing information. When a user response to a form arrives, instrumentation records the user data returned to the session. 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.

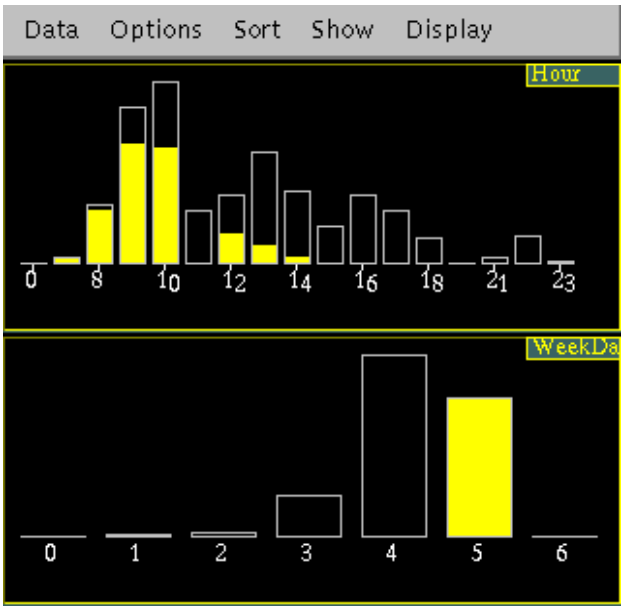
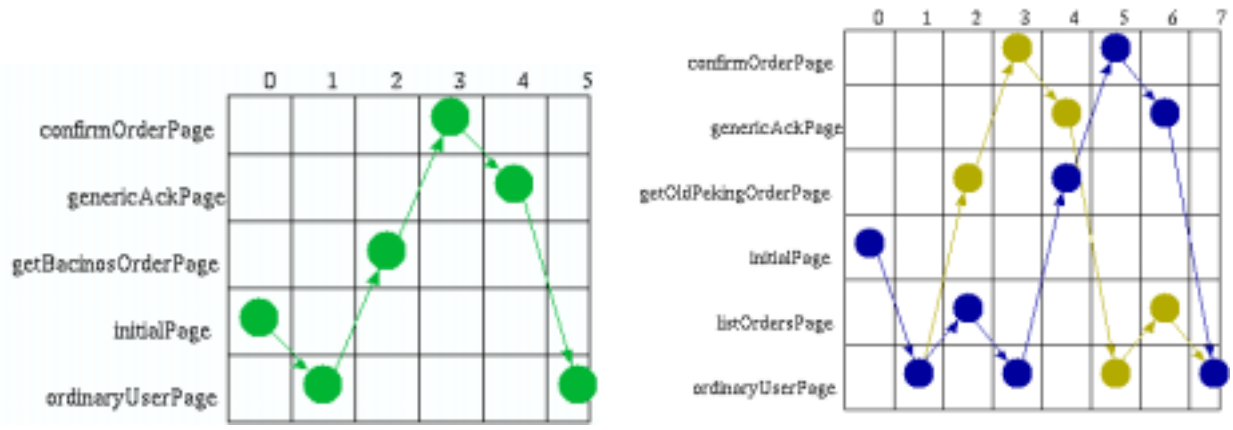


Figure 7: A bar chart view of LunchBot usage.

The template abstraction allows the data passed to and from the form to be related to the user interface that the user views. This is due to the precise mapping between service logic variables and the dynamic portions of the templates (as specified via the `MVAR` marks). This is especially useful for restructuring the templates based on data profiles. For example, profiling the user data returned by a form might show that a particular item in a `SELECT` menu was very popular. Such profile information could be used to reorder the items in a `SELECT` accordingly.

3.6.2 Visualizing the LunchBot

The Mawl system includes a data visualization component called PathView, which is a Java applet that displays user interactions with a service as paths through a graph. PathView enables analysis of the user paths in conjunction with other service statistics, using multiple views to allow exploration of various facets of user behavior. An extensible relational data interface allows new sources of data to be incorporated easily into an analysis. As an example, we use PathView to analyze the usage of the LunchBot, a web service for ordering weekly group lunches. The LunchBot was implemented in mawl shortly after the language was developed.



(a) (b)

Figure 8: A typical LunchBot scenario: ordering lunch (a). The "list orders" variant (b).

When do users access the LunchBot? The bar chart in Figure 7 shows the amount of activity by hour and weekday. The Lunchbot gets most of its use on Thursday and Fridays (days 4 and 5) every week; this is as expected, since all lunch events in this period were on Fridays. Usage on Friday is highlighted. From the hour bar chart, we can see that the major use of the Lunchbot occurs Friday morning. Most people wait until the last two hours before lunch on Friday to order, after the final warning message is sent (usually around 9 AM) announcing the imminent close of the Lunchbot.

One goal of path analysis is to identify common sequences of user interactions and tune the services to create "shortcuts" for these scenarios. Figure 8(a) shows a common pattern in the LunchBot: the "order lunch" mountain peak. With further analysis, we find a large set of paths that contain both the "order lunch" peak and a "list orders" hill, as shown in Figure 8(b). It turns out that users often list orders before ordering lunch (blue/black path). Other users list orders after ordering lunch (yellow/grey path). The frequent occurrence of the (list orders, order lunch) sequence suggests that users like to see what items are popular (a complicating factor is that the list orders page also lists the user associated with each order; a pessimist might infer that people examine the orders to see if they want to attend lunch at all; we prefer the optimistic analysis). Annotating the favorite items on the menu would provide a simple shortcut replacing the more complicated sequence of interactions.

4 Summary

Mawl was created to address two specific problems in the creation of dynamic web services: the lack of compile-time guarantees about of services, and the low-level of programming involved in coding to the CGI model. A language was necessary in order to provide these guarantees and give implementation flexibility. Neither libraries nor macros provide solutions to these two problems.

The form is the basic abstraction that helped to solve these two problems, by enforcing a separation of concerns between service logic and user interface descriptions. 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 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.

5 Availability

The mawl language (version 2.0, with C++ as a host language) is available at

`http://www.bell-labs.com/projects/MAWL/`

for SGI and Solaris platforms. Mawl is part of a larger project called Tardis, which includes TelePortal, the platform by which interactive voice response systems may be programmed using HTML. TelePortal is not currently available.

6 Acknowledgments

Christopher Ramming and David Ladd are the originators of the mawl language. 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

[Aea97] D. Atkins and T. Ball et al. Integrated web and telephone service creation. *Bell Labs Technical Journal*, 2(1), Winter 1997.

[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.

[EFF+97] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing idl compiler. In *Proceedings of the 1997 Conference on Programming Language Design and Implementation (to appear)*, June 1997.

[GA96] J. Gosling and K. Arnold. *The Java Programming Language*. Addison-Wesley, 1996.

[Gro95] Object Management Group. The common object request broker: Architecture and specification. Technical Report Edition 2.0, July 1995.

[Kor86] D.G. Korn. Ksh—a shell programming language. Technical report, AT&T Bell Laboratories, 1986.

[LR95] D. A. Ladd and J. C. Ramming. Programming the web: An application-oriented language for hypermedia service programming. In *Proceedings of the 4th International World Wide Web Conference*, pages 567–586. World Wide Web Consortium, December 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.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[SE95] E. Selberg and O. Etzioni. Multi-engine search and comparison using the Metacrawler. In *Proceedings of the 4th International World Wide Web Conference*, pages 195–208. World Wide Web Consortium, December 1995.

[SMD97] R. Pike et al. S. M. Doward. The inferno operating system. *Bell Labs Technical Journal*, 2(1), Winter 1997.

[Sri95] R. Srinivasan. Remote procedure call protocol specification version 2. Technical Report Technical Report RFC 1831, Sun Microsystems, August 1995.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[WS90] L. Wall and R. L. Schwartz. *Programming PERL*. O'Reilly & Associates, 1990.