Increasing the Computational Potential of the World Wide Web

S. D. Reilly, J. L. Pfaltz, J. C. French

CS-96-02
February 9, 1996

Dept. of Computer Science
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 2290l

This research was supported in part by
DOE Grant #DEFG05-95ER25254

Abstract

For many World Wide Web applications there is a need to provide session semantics so that users have the impression of a continuous interaction. This is true, for example, when one searches a database interactively. Because WWW servers are stateless some extra mechanism is necessary to give the impression of session semantics. This report discusses a strategy for implementing session semantics over a WWW application. Apart from the need to maintain state during interactive sessions, there is also the need to control the application. Under any circumstances this is a tedious activity. This report also discusses a mechanism for modeling a WWW application as a finite state automaton and describes a tool, the Stateful Server Application Tool (SSAT), built to assist in the development of these applications.

1. Introduction

The implementation of the World Wide Web is designed to minimize network traffic. In particular all web servers are *stateless*. That is, no web server can store any state regarding prior interactions with a particular client. One does not have a "web session" in which there is a "login" and a "logout". This limits the functions that web pages and browsers may provide. For example, if a user requests a document, the HTTP server returns the document to the client and then forgets about the transaction. The next request from that user (if there is one at all), is completely separate. Such a web interface is essential because it is infeasible for HTTP servers to maintain state information about clients. First, there may be an arbitrary number of clients with a valid state at any one time; to keep track of all the clients on the server site might overload the server's capacity. Second, the server has no way of knowing if a client has terminated; so it would need a timeout mechanism to determine when to release a client's state information. And third, the amount of state information for some clients may be too large; thereby overloading the server and affecting all active clients.

For all these reasons, keeping web servers stateless makes sense. The lack of state information on the server's side greatly simplifies the server process and allows "web surfing", in which the casual user touches web pages and then disappears.

But, while this might be an efficient approach to engineering the web, the stateless servers that make up the web can not support a wide variety of applications in which some retention of state is required. This has proven to be particularly true in developing a web interface for the Department of Energy's Atmospheric Radiation Measurement (ARM) archive [DOE91]. To develop these applications we have had to create *stateful* web servers, or at least web servers which behave as if they retained information about the client's state. A description of this facility is the primary focus of this report.

What we have done is: first, to implement a mechanism for creating the image of a *stateful* session through a World Wide Web interface in which a user can "login" and perform a cooperative sequence of actions that together behave as a session in which state information is preserved in between user requests for service. This allows a user's previous steps and input to be recorded, and thus significantly extends the usefulness of a World Wide Web interface to the ARM archive. This is achieved by creating a mechanism in which each client keeps track of its own state and tells the server about its current state with each interaction.[1]

A simple example of where this might be useful is a situation in which many different "levels" of users want to access information through the World Wide Web. This approach is fundamental to the current design of EOSDIS (Earth Orbiting System Data and Information System) being developed by Hughes for NASA. The server may want to present different versions of the data for different levels of users. A high school student accessing the ARM web site might need to be presented with broad brush, perspective information describing the data and be spared the technical jargon that scientists use. On the other hand, a scientist who accesses the same information won't need all of the perspective information and is happy using the specialized terminology that makes it easier for him to apply the data being accessed. In the EOSDIS design,

[1] Since this work was completed, Netscape now provides a mechanism for maintaining state on the client side. As we will see, the precise mechanism used to maintain the state is unimportant. The main contribution of this report is to show how the retained state can be used to control the ‘‘session’’ and how the process of characterizing the session sometimes can be automated.

these terminological levels are accomodated by having different thesauri for different classes of users. In this case it is a good idea to get the user level of the client and to store that information, taking it into account when serving information during subsequent references.

Another example of the utility of a stateful session would be a user running a query on a database that may require more than one step to finalize the query. Let's say a user wants to find data recorded from devices located at a particular set of sites. The query might start off with the user entering or choosing a set of sites and the server returning the kinds of devices located at those sites. The user would then pick the devices that he is interested in and re-submit the query for only those devices at the chosen sites. But, with a completely stateless session, the user would have to re-enter the desired sites because the initial query would be lost. If the set of sites were somehow retained, they could be incorporated into the final query without the user having to re-enter them, thus simulating a stateful session.

These examples are fairly simple ones, and the fact is that if some kind of state information can be retained, the operations that can be performed using web clients and servers is extended dramatically. In Section 3, we give examples and show how this is done using standard HTTP servers, HTML web browsers, and CGI scripts. Then in Section 4, we develop a tool which greatly simplifies the task of creating appropriate CGI scripts. But first, some background on HTML and HTTP is necessary to understand the implementation.

2. HTTP and HTML Background

Most people think of HTML as a simple markup language that is used simply to present documents in a pleasing manner. This was its intended purpose, but it has expanded, incorporating "forms" with which the user can interact and send information back to the server.

An HTML *form* is a fairly simple construct in which the programmer specifies a set of input fields and their properties using HTML syntax. (*C.f.* Figure 2-1.) This form cretes a window displaying 3 values, "A", "B", and "C", from which the user may select any one. Like most HTML constructs, HTML forms have a simple syntax. There are three parts to an HTML form:

(1) the form *header* `<FORM ... >` and *footer* `</FORM>`. The header specifies the location of the CGI script to which the form should be sent and the manner of submitting the form to the script, *e.g.* by `GET` or by `POST`.

```
<FORM>
Pick one:<br>
<SELECT NAME="letter" MULTIPLE>
  <OPTION>A
  <OPTION>B
  <OPTION>C
</SELECT>
</FORM>
```

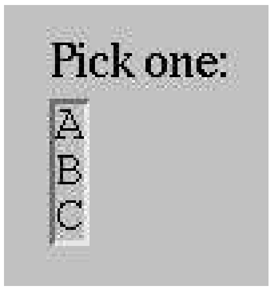
An extremely simple HTML form

Figure 2-1

(2) The *body* of the form specifies the input fields, their type, and their contents.

(3) The *initiator* of form submission actually "sends" the form from the client process to the server process. This is usually just a "submit" button.

The body of the form of Figure 2-1 consists of the single line (asking the user to "*Pick one*") followed by a selection clause supporting the choice of any of the three *letter* values A, B, or C. If one executes the form using *mosaic* or *netscape,* it will look something like Figure 2-2. Clicking the mouse on any of its three lines will select that option.



Form displayed by the HTML code of Figure 2-1
Figure 2-2

One normally creates more "elegant" forms. This is deliberately "plain" to emphasize the essential features of the form. We also note that there is no *initiator* in this form; there is no "submit" button in Figure 2-2. Consequently, there is no way of actually submitting the chosen value to the server and no exit from the form.

All forms have a method of "submitting" the form information (usually through the press of a "submit" button). The form information currently can be submitted through two methods, `POST` and `GET.` These will be discussed later. In both cases the information contained in the form is sent to a CGI script on the server's machine when the user "submits" the form. The CGI script basically consists of any executable code that can be invoked by the HTTP server.

In Figure 2-3, the URL of the CGI script has been identified in the `ACTION` clause as:

```
http://www.cs.virginia.edu/cgi-bin/post-query
```

The `method` clause specifies that the form will be submitted using the `POST` method.

The form of Figure 2-3 is only a bit more complex; but it is a complete form. It illustrates the kind of query interface used in the ARM project, in which the user can request a response based on various temperatures and/or amounts of water present. In this example, the form lets the user select multiple entries from a list of choices, and their choices will be submitted to the script with the name "variables". It also provides a mechanism so that the client can send the form to the HTML server. The `INPUT TYPE="submit"` line actually makes a push button in the client window that the user clicks on to submit the form. The `INPUT TYPE="reset"` just resets the form values to their original state (handled entirely by the browser). The following form shows how to specify a form in which the user can select several items from a list. When the form in Figure 2-3 is viewed on an HTML client, it will look something like Figure 2-4.

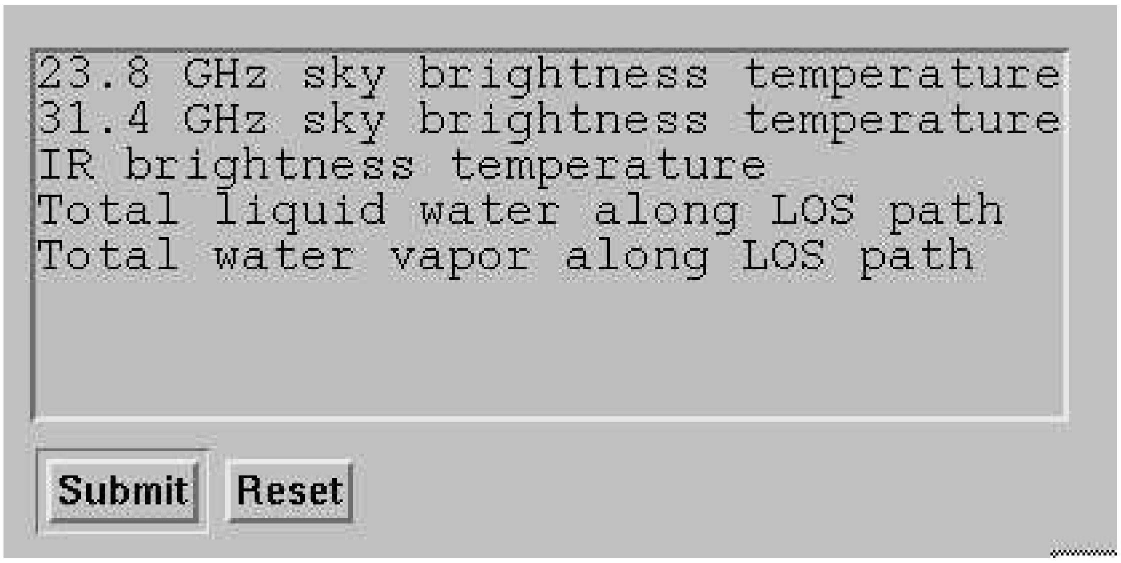
```
<FORM METHOD="POST" ACTION="http://www.cs.virginia.edu/cgi-bin/post-query">

<SELECT NAME="variables" MULTIPLE SIZE=8>
<OPTION>23.8 GHz sky brightness temperature
<OPTION>31.4 GHz sky brightness temperature
<OPTION>IR Brightness Temperature
<OPTION>Total liquid water along LOS path
<OPTION>Total water vapor along LOS path
</SELECT><br>

<INPUT TYPE="submit" VALUE="Submit">
<INPUT TYPE="reset" VALUE="Reset">
</FORM>
```

A query submission form

Figure 2-3


23.8 GHz sky brightness temperature
31.4 GHz sky brightness temperature
IR brightness temperature
Total liquid water along LOS path
Total water vapor along LOS path

Terminal display generated by Figure 2-3
Figure 2-4

Clicking the mouse on any of the five lines will select that option. We observe that there is room for three more lines in the selection window because its **`SIZE`** was specified as 8. Further the **`MULTIPLE`** keyword in the form header indicates that the user is able to select zero or more of the items specified by the <OPTION> tag. Say for example, that the user chose the last two options ("Total liquid water along LOS path", and "Total water vapor along LOS path") and submitted the form. The two name-value pairs sent to the form would be:

```
     Name         Value

   ( variables,  "Total liquid water along LOS path" )
   ( variables,  "Total water vapor along LOS path" )
```

Both the POST and GET methods send the form information to the CGI script in the form of name-value pairs which are parsed by the script and used to perform whatever function the script is intended to perform. The difference between the two methods is that the GET method sends the form information to the CGI code through the command line when the script is invoked, and the data in the POST method is sent to the standard input of the script where it can subsequently be read by the script after it is invoked. Some servers also limit access to the GET and POST operations differently. Try submitting the form and you will see the form information that is sent to the CGI script. All inputs to the scripts (i.e. all form data), is organized into name-value pairs in which every value has a name associated with it. There may be many name-value pairs with the same name such as in this case where the name "variables" is associated with several of the presented values.

One important aspect of the HTML form is that it allows "hidden" attributes. This means that you can insert a name-value pair into the form without the user seeing it (unless they view the HTML source). This means that you can have additional data that is submitted with the form, that the user doesn't see. This hidden attribute is the key to increasing the computational power of the world wide web.

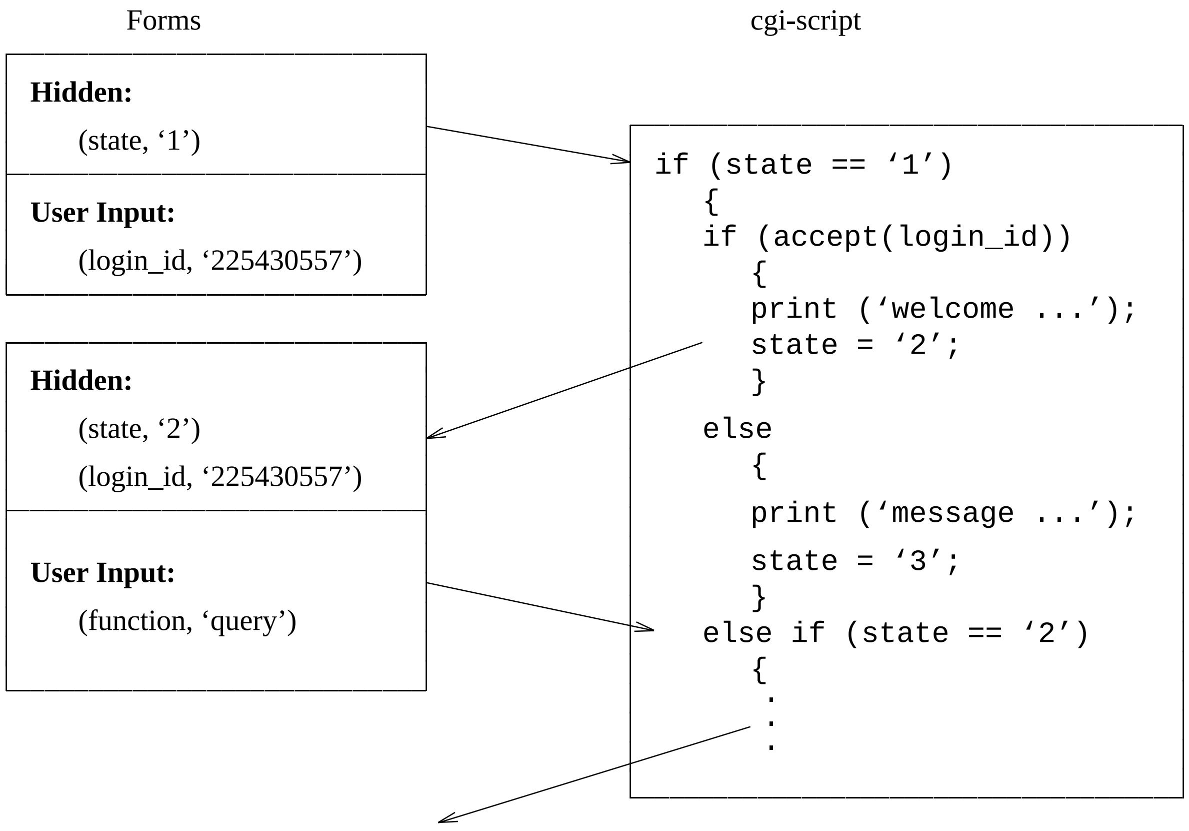
HTTP servers are crucial components of the world wide web, they take requests and serve documents to clients, and they receive form data and invoke CGI scripts. One of their biggest advantages, as well as shortfalls, is that they are stateless servers. When a request for a document or form submission is received, it is served and then forgotten. Each request is independent from all others. When a form is submitted, it sends the server the method of submitting the form (GET or POST), the location of the script to be invoked, and the name-value pairs contained in the form. The server then invokes the form and sends it the name-value pairs, encoded in the command line for the GET method, or encoded and sent to the stdin of the script for the POST method. The POST method is more generally used because data sent to a script through the command line often has a size limit.

In this section, we have examined very vanilla HTML forms being used in the manner for which they were designed — accepting and displaying information. It is essential background for the following section, in which we use these kinds of forms to simulate the states of a finite state machine, and to accept input from the user which will perform an action and transfer to another state.

3. What do we mean by 'Increasing the Computational Potential'

As we have noted, the World Wide Web is a very simple mechanism that can only perform simple stateless operations. If a finite state machine could be implemented on the web, it would increase the computational power because it could maintain state information. We have implemented a cgi-script that operates on the web which is functionally a finite automaton. This automaton creates a stateful session for the user where state information (as well as other data) is stored to create a continuous session for a user to access a remote web site. The key components to this system are recursive cgi-scripts and hidden HTML form attributes. A *recursive* cgi-script is one that outputs new forms which will be submitted back to the same cgi-script. A *hidden* attribute is just like the attribute pairs, (*a_name, a_value*) that we saw in the preceding section. The only difference is that the user never sees a hidden attribute; its value is set by the cgi-script, passed to the client with the form, *not displayed,* and returned unaltered when the user re-submits the form back to the cgi-script.

Instead of storing only state information, we can input and store arbitrary data along with the state information that may or may not affect the next state of the machine (but it should affect the function of the output). Figure 3-1 presents a schematic of the kind of interaction we might expect between a client and the ARM interface. A user must first validate that he is a registered user. Then this value is passed back and forth between the client and the server with no subsequent need for re-entry.



```
if (state == '1')
   {
   if (accept(login_id))
      {
      print ('welcome ...');
      state = '2';
      }
   else
      {
      print ('message ...');
      state = '3';
      }
   else if (state == '2')
      {
       .
       .
       .
```

An example of interaction between the client and server
Figure 3-1

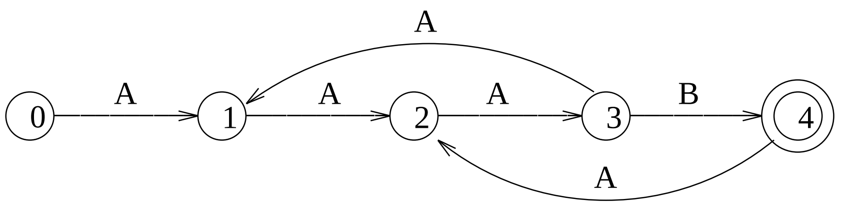
We can imagine that this registry number also encodes various privileges to perform some data-base functions, but not others.

Another example of this additional use of client input might be a database querying application in which the user specifies criteria for a query. The initial state might be the query specification state in which the user inputs the query criteria. The next state might always be the query retrieval state independent of the query criteria. The data returned from the query might be different, but the state (query retrieval) will always be the same at this point. The query information entered by the user has no impact on the state of the system, only on the function of the invoked procedure (querying the database). It is in this way that a fairly complicated application can be modeled on top of a finite automaton by adding only a small amount of extra functionality.

Even without the added functionality, we have reached our goal: "A level of computability modeled by a Finite Automaton." This is an improvement over the current level of computability of the World Wide Web -- a stateless machine.

4. A Working Example

Perhaps the simplest form of computational capability is that embodied by a finite automaton (FA) or finite state machine (FSM). Figure 4-1 illustrates a fairly simple finite state machine, in which the states $\{s_0, s_1, s_2, s_3, s_4\}$ are denoted by circles and transitions between these states denoted by arcs.



A simple finite state automaton
Figure 4-1

Above each arc is a character, in this case either "A" or "B". When in state s_i, input of one of these characters will cause a transition from state s_i to state s_k if there is such a labelled arc between the states. Otherwise, the automaton defaults into an error state. Some states can be designated as *final* states. In Figure 4-1, there is only one final state and it is denoted by the double circles.

As is well known, *c.f.* Hopcroft and Ullman [HoU79], every finite state machine defines a finite state language (or regular), and conversely. The finite state machine pictured here accepts the regular language **aaa[aaa | baa]*b**.

The source code used to implement this automaton is given in Appendix B. Each state $\{s_0, s_1, s_2, s_3, s_4\}$ in the automaton is represented by a form with a hidden state value. When the form is submitted, the input and the state is sent with the form to the script, and the next state is determined. Once the next state is determined, the new form for that state (with the corresponding hidden values) is returned to the client (html browser). If the new state cannot be determined (i.e. the automaton does not accept the input), the machine goes to the halted state and the user can start again from the initial state. If the automaton is in state 4, the user has the option to exit (or return to the beginning).

This FA was fairly easy to implement on the web. The main part of the script that represents the FA is contained in `automata.C` (see appendix B). This script is invoked for every form submission in the FA. When invoked, the script figures out which state it is in (if any) and takes the appropriate action based on the state, input, and any other data that may be available. If the input is valid for the current state, the form for the next state is output along with all of the hidden attributes that hold the state information and data. If the input is not valid for the current state, the new state becomes the halted state and the halted state form is sent back to the client.

While the finite automaton of Figure 4-1, with only 5 states, is fairly simple, this need not be the case in general. Transition tables can become far more complex. If in addition, the programmer must worry about form generation and additional operations to be performed in each state it becomes easy to lose the forest for the trees. Although the finite automaton of Figure 4-1 is almost trivial, the code of Appendix B is certainly non-trivial. If this means of extending the computational power of the net is to be practical, there must be a simpler way of generating these CGI scripts.

5. A Tool for Creating Stateful Web Sessions

5.1. Overview of SSAT — Stateful Server Application Tool

If creating a stateful machine that operates over the world wide web is a useful endeavor, then it would be practical to create a tool that would make it easy for any HTML author to create these machines. I have created a tool (SSAT - Stateful Server Application Tool) that will accept a description of a machine and output the source code for a machine that fits the description. The description of the machine includes the form contents for each state, the initial state of the machine, static variables and lists, and a transition function. The SSAT allows authors of such stateful machines to create and modify the machine without needing to be aware of all of the complexities of writing CGI scripts. They need only describe the functionality of their machine and they can have the code for the CGI script that generates the machine automatically created for them

For a sample description of a machine, see appendix A for the description of the machine that implements the automaton shown in figure 4-1. This description is read by the SSAT and the corresponding C++ source code for the script is created along with the necessary utility files. Once the code is compiled, the executable must be put in a location accessible by an HTTP server, and then the script is ready to act as a stateful server application.

One of the most important features that the SSAT provides is that the script it creates is a stateful machine. The variables and lists specified in the description of the machine are static throughout the machines execution. This means that the value a user inputs on the first "page" of his session is retained and can be used in subsequent pages without the value having to be recomputed or re-entered by the user.

Another useful feature of the SSAT is that the transitions from page to page are handled by a transition function that returns state numbers instead of the current method using hard-coded URLs which can be difficult to keep up to date and consistent. In the stateful machine, the only hardcoded URL is the location of the single CGI script.

This brings us to the next feature that is added to the current interface. In this machine, a whole set of "pages" or states is contained in one single CGI script as opposed to a set of HTML documents and scripts that may become out of date and inconsistent with each other. This increases the simplicity of maintaining such a set of pages because the relations between "pages" can be examined by simply looking at the transition function of the machine instead of traversing through many separate HTML documents looking for hard-coded URL’s.

This method of creating a stateful machine is superior to standard CGI scripts because the only source code that the author writes is that which performs the work specific to the application being written. The author need not be concerned with the details of how the code interacts with the HTTP server, environment variables, and the numerous other specifics that make writing CGI scripts tedious and error prone.

The SSAT along with the scripts that it produces allow us to implement a finite state machine on the world wide web, with the added functionality of having static data.

5.2. Using the SSAT

The SSAT expects a descriptive file (suffixed with `.web`) consisting of five parts:

(1) a specification of the initial state of the form

```
initial state: 0;
```

(2) the variables, both hidden and overt, that are to retain their values through the session, as in

```
variables: { xxx, yyy, zzz }
```

(3) a specification of the applications lists, as in

```
lists: { aaa, bbb }
```

(Lists are similar to variables in that they are maintained between states, but they contain multiple strings. Lists are represented as C++ class instantiations with the appropriate methods for the author to access and modify them.)

(4) the body of each form for each state in the application, such as

```
form0:
    {
    <h1>Welcome to this test application</h1>
    Choose "next" to transfer to the next state.
    <INPUT TYPE="submit"  Value="next">
    }
```

(The header and footer of each form will be created automatically by SSAT.)

(5) the finite state transition function, as in

```
int  transition (int current_state,  char  *output)
     {
     int     next_state;
     strcpy (output, "");
     switch (current_state)
        {
        case 0:
             next_state = 1;
        case 1:
             next_state = 0;
        default:
             next_state = 0;
        }
     return next_state;
     }
```

Readily this is the transition table of a trivial automaton which oscillates between the states s_0 and s_1 regardless of the input.

An entire test automaton is shown below

```
initial state: 0;

variables: { }

lists: { }

form0:
    {
    <h1>Welcome to this test application</h1>
    Choose "next" to transfer to the next state.
    <INPUT TYPE="submit"  Value="next">
    }
form1:
    {
    <h1>You have successfully transferred to this state</h1>
    Choosing "next" will transfer you back to the initial state
    <INPUT TYPE="submit"  Value="next">
    }

code:

#include <stdio.h>
#include <stdlib.h>
int  transition (int current_state,  char  *output)
     {
     int     next_state;
     strcpy (output, "");
     switch (current_state)
        {
        case 0:
             next_state = 1;
        case 1:
             next_state = 0;
        default:
             next_state = 0;
        }
     return next_state;
     }
```

Notice that in this code we have used neither variables nor lists.

5.3. How the SSAT works

As described earlier, when an HTML browser submits a form to the HTTP server, it submits the form name (and location) along with the all of the name-value pairs that were filled in on the form. Unless there is an error, the output of the script is then sent to the client to be viewed as another "page". This resulting page can itself include a form, and that is one aspect of how the FA works. The new form that is output is submitted to the same script that created it. This creates a recursive situation where the script creates forms that are in turn submitted to and interpreted by itself. It is because of this that a whole web of pages can be contained in one CGI script. Since it is also an executable script that creates the forms, the pages can be created dynamically making them much more powerful.

Also described earlier is the fact that a form can contain "hidden" attributes, basically name-value pairs that the user doesn't see, but that are treated just like any other form data when the form is submitted. This allows us to hold state information, as well as other data, in a form that the user doesn't see instead of on the HTTP server's side where it would create many problems.

The current state is stored in each HTML form in a variable (name-value pair) named `WEB_state.` The hidden variable `WEB_state` is automatically provided by SSAT. It is initialized to 0, and passed to *transition* as its first argument, and reset by the value returned from *transition*.[2] When each form is submitted, the script reads the WEB_state variable and figures out which state the machine is currently in. The script then reads in the rest of the name-value pairs submitted with the form (hidden or otherwise). After all of the state information and data is read by the script, the next state is determined by the transition function that the author supplies. This transition function takes as input the current state of the machine as well as any static variables and lists, or other data input by the form. The transition function computes the next state based on all of this information and returns the next state of the machine. The corresponding form, along with the new state information and other static variables and lists (stored as hidden values) is then sent to the client to be presented to the user and the cycle is repeated for the next state.

The transition function is where the function of the machine is defined. It is the only executable part of the script that is written by the user, and it can be used to implement database queries, interactive sessions, and many other applications that are not possible using current web interfaces and CGI scripts.

The FA described in figure 4-1 was created using this tool. The description of the machine is located in appendix A, and the resulting C++ code produced by the tool is located in appendix B.

6. References

[HoU79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[DOE91] *Proc. Second Atmospheric Radiation Measurement (ARM) Science Team Meeting*, U.S. Dept. Of Energy, Conf.-9110336, Denver, CO, Oct. 1991.

[2] Note: The form in figure 2-1 was not generated by this tool.

Appendix A

This is the SSAT description of the automata that was used to produce the code in appendix B, and the finite state machine in figure 4-1. Notice how the form definitions do not include the form header, and the only C++ code written by the machine's author is the transition function. Also, note how the input variable determines the next state (as computed by the transition function). Also, the machine depicted in Figure 4-1 does not have a state 5 even though there is one in the desciption of the machine. This is because state 5 is used as a halted state in this instance.

```
              /* automaton.web */

initial state: 0;

variables: {   name, input }

lists: {  }

form0:
       {
       <h1>Welcome to the initial state in the FA</h1>
       This whole set of pages is created from one script.
       To see the source code, click
       <a href="../cgi-src/">here</a>.
       <br>This page is state 0 of the finite state machine<br>

       <SELECT NAME="input" >
       <OPTION>A
       </SELECT>

       <INPUT TYPE="submit" VALUE="Step" SIZE=50>
       }

form1:
       {
       <h1> This is state 1 </h1>
       <SELECT NAME="input" >
       <OPTION>A
       </SELECT>

       <INPUT TYPE="submit" VALUE="Step" SIZE=50>
       }

form2:
       {
       <h1> This is state 2 </h1>
       <SELECT NAME="input" >
       <OPTION>A
       </SELECT>

       <INPUT TYPE="submit" VALUE="Step" SIZE=50>
       }

form3:
       {
       <h1> This is state 3 </h1>
       <SELECT NAME="input" >
       <OPTION>A
       <OPTION>B
       </SELECT>
       <INPUT TYPE="submit" VALUE="Step" SIZE=50>
       }

form4:
       {
       <h1> This is state 4 </h1>
       <SELECT NAME="input" >
       <OPTION>A
       </SELECT>

       <INPUT TYPE="submit" VALUE="Step" SIZE=50>
       }

form5:
       {
       <h1>Welcome to the halted state in the FA</h1>
       For some reason, there was an error.
       <INPUT TYPE="submit" VALUE="Go back to the initial state" SIZE=50>
       }
code:
#include<stdio.h>
#include<stdlib.h>

int transition(int current_state, char *output)
       {
       int next_state = 5;
       output = strdup("");
       switch(current_state)
         {
         case 0:
              if(!strcmp(input, "A"))
                     next_state = 1;
              break;
         case 1:
              if(!strcmp(input, "A"))
                     next_state = 2;
              break;
         case 2:
              if(!strcmp(input, "A"))
                     next_state = 3;
              break;
         case 3:
              if(!strcmp(input, "A"))
                     next_state = 1;
              else if(!strcmp(input, "B"))
                     next_state = 4;
              break;
         case 4:
              if(!strcmp(input, "A"))
                     next_state = 2;
              break;
         case 5:
              next_state = 0;
         }
       return next_state;
       }
```

Appendix B

This is the C++ source code that implements the finite automata presented in figure 4-1. This code was generated using the SSAT tool for creating stateful sessions described in this paper from the description given in Appendix A.

```
              /* masterform.C */

#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "web_util.h"

#define ERROR_FORM "Error.\n"

#define FORM0 "\
       <h1>Welcome to the initial state in the FA</h1>\
       This whole set of pages is created from one script.  \
       To see the source code, click \
       <a href=\"../cgi-src/
       <br>This page is state 0 of the finite state machine<br>\
       <SELECT NAME=\"input\" >\
       <OPTION>A\
       </SELECT>\
       <INPUT TYPE=\"submit\" VALUE=\"Step\" SIZE=50>\
        "

#define FORM1 "\
       <h1> This is state 1 </h1>\
       <SELECT NAME=\"input\" >\
       <OPTION>A\
       </SELECT>\
       <INPUT TYPE=\"submit\" VALUE=\"Step\" SIZE=50>\
        "

#define FORM2 "\
       <h1> This is state 2 </h1>\
       <SELECT NAME=\"input\" >\
       <OPTION>A\
       </SELECT>\
       <INPUT TYPE=\"submit\" VALUE=\"Step\" SIZE=50> \
        "

#define FORM3 "\
       <h1> This is state 3 </h1>\
       <SELECT NAME=\"input\" >\
       <OPTION>A\
       <OPTION>B\
       </SELECT>\
       <INPUT TYPE=\"submit\" VALUE=\"Step\" SIZE=50> \
        "

#define FORM4 "\
       <h1> This is state 4 </h1>\
       <SELECT NAME=\"input\" >\
       <OPTION>A\
       </SELECT>\
       <INPUT TYPE=\"submit\" VALUE=\"Step\" SIZE=50> \
        "
```

```
#define FORM5 "\
       <h1>Welcome to the halted state in the FA</h1>\
       For some reason, there was an error.\
       <INPUT TYPE=\"submit\" VALUE=\"Go back to the initial state\" SIZE=50>\
        "

#define WEB_init_state 0

class LIST
       {
       char   **strings;
       int    size;
       int    MAX_SIZE;
    public:
       LIST (int max_size)
              {
              strings = (char**)malloc(sizeof(char*) * max_size);
              size = 0;
              MAX_SIZE = max_size;
              }

       int add_string(char *string)
              {
              if(size < MAX_SIZE)
                     {
                     strings[size] = strdup(string);
                     size++;
                     return size-1;
                     }
              else
                     {
                     return -1;
                     }
              }

       char* get_string(int string_num)
              {
              if(string_num < size && string_num >=0)
                     {
                     return strdup(strings[string_num]);
                     }
              else
                     {
                     return NULL;
                     }
              }

       int get_number() {return size;}
       };

typedef char* VARIABLE;

VARIABLE      WEB_state;

VARIABLE      name;
VARIABLE      input;

int PROTECTED;
int ERROR_FLAG;


                              /* This is where the user code goes. */
#include<stdio.h>
```

```
#include<stdlib.h>

int transition (int current_state, char *output)
       {
       int    next_state = 5;

       output = strdup("");
       switch(current_state)
         {
         case 0:
              if(!strcmp(input,"A"))
                     next_state = 1;
              break;
         case 1:
              if(!strcmp(input,"A"))
                     next_state = 2;
              break;
         case 2:
              if(!strcmp(input,"A"))
                     next_state = 3;
              break;
         case 3:
              if(!strcmp(input,"A"))
                     next_state = 1;
              else if(!strcmp(input,"B"))
                     next_state = 4;
              break;
         case 4:
              if(!strcmp(input,"A"))
                     next_state = 2;
              break;
         case 5:
              next_state = 0;
         }
       return next_state;
       }
                             /* End user code */


int main (int argc, char *argv[])
       {
       entry  entries[MAX_ENTRIES];
       register int  x,m = 0;
       int    cl, i, WEB_got_a_state = 0, tmp;
       int    WEB_state_num, WEB_next_state;
       char   *WEB_exe_filename, *buffer;

       PROTECTED = 0;
       ERROR_FLAG = 0;

       name = strdup("");
       input = strdup("");
                             /* print out the proper cgi-script heading: */
       printf("Content-type: text/html%c%c",10,10);

                             /* Check to see if we are operating in a
                                protected environment */
       if(getenv("AUTH_TYPE") != NULL)
       PROTECTED = 1;

       WEB_exe_filename = strdup(argv[0]);   /* first acquire the name of
                                           this file so that it can be
                                           referenced in future URLs.
```

```
                           Also filter out any crap
                                  that may have come with
                                  it (i.e. PATH info).  */
for(i=0; i<strlen(WEB_exe_filename); i++)
       {
       if(WEB_exe_filename[i] == ' ')
              {
              WEB_exe_filename[i] = '\0';
              break;
              }
       }
i--;
for(i=i-1; i>=0; i--)
       {
       if(WEB_exe_filename[i] == '/')
              {
              WEB_exe_filename += (i+1);
              break;
              }
       }
fprintf(stderr,"%s\n",WEB_exe_filename);

if(strcmp(getenv("REQUEST_METHOD"),"POST"))
       {             /* By default, just go to the initial
                        state if something is screwy, the
                        generated forms will take care of it
                        from there.  */

       printf("<form method=\"POST\" action=\"%s\" >\n",
                     WEB_exe_filename);
       printf("<input type=\"hidden\" name=\"WEB_state\" value=\"%d\">\n",
                     WEB_init_state);
       printf("%s\n",FORM0);
       printf("</form>\n");
       exit(1);
       }

if(strcmp(getenv("CONTENT_TYPE"),"application/x-www-form-urlencoded"))
       {
       printf("This script can only be used to decode form results.\n");
       exit(1);
       }
                     /* use environment variable to determine
                        how much data there is */
cl = atoi(getenv("CONTENT_LENGTH"));
                     /* put name-value pairs from form into
                        an array for processing */
for(x=0;cl && (!feof(stdin));x++)
       {
       m = x;
       entries[x].val = fmakeword(stdin,'&',&cl);
       plustospace(entries[x].val);
       unescape_url(entries[x].val);
       entries[x].name = makeword(entries[x].val,'=');
       }
                     /* read values from form; put into
                        _data variables; set error flags     */
for(x=0; x<=m; x++)
       {
       if (!strcmp(entries[x].name,"WEB_state"))
              {
              WEB_state = strdup(entries[x].val);
              WEB_got_a_state = 1;
```

```
        }
        if(!strcmp(entries[x].name,"name"))
               {
               name = strdup(entries[x].val);
               }
        if(!strcmp(entries[x].name,"input"))
               {
               input = strdup(entries[x].val);
               }
        }            /* end of for loop, finished
                        going through form data */
if ( WEB_got_a_state )
        {
        WEB_state_num = atoi(WEB_state);
        }
else
        {
        fprintf(stderr,"Error:  No State information sent with POST.\n");
        exit(1);
        }
WEB_next_state = transition(WEB_state_num, buffer);

printf("<FORM METHOD=\"POST\" ACTION=\"%s\">\n",
                    WEB_exe_filename);
printf("<input type=\"hidden\" name=\"WEB_state\" value=\"%d\">\n",
                    WEB_next_state);
printf("<input type=\"hidden\" name=\"name\" value=\"%s\">\n",
                    name);
printf("<input type=\"hidden\" name=\"input\" value=\"%s\">\n",
                    input);


switch(WEB_next_state)
  {
  case 0:
        printf("%s", FORM0);
        break;
  case 1:
        printf("%s", FORM1);
        break;
  case 2:
        printf("%s", FORM2);
        break;
  case 3:
        printf("%s", FORM3);
        break;
  case 4:
        printf("%s", FORM4);
        break;
  case 5:
        printf("%s", FORM5);
        break;
  default:
        printf("%s", FORM0);
        break;
  }
printf("</form>\n");
return 1;
}
```

Appendix C

Below is a listing of the SSAT.

```
             /* main.C */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"
#define MAX_TRANSITIONS 1000
#define MAX_STATES      200
#define MAX_VARIABLES   200
#define MAX_LISTS       200
#define MAX_FORM_SIZE   3000

class LIST
       {
     public:
       char *name;
       int size;
       };

int init_state;

int states[MAX_STATES];
int num_states;

char *forms[MAX_STATES];
int num_forms;

char *variables[MAX_VARIABLES];
int num_variables;

LIST *lists[MAX_LISTS];
int num_lists;

char*  strip(char *string)
       {
       int    i;

       for(i=strlen(string)-1; i>=0; i--)
              {
              if(!(string[i]=='\n' || string[i] == ' '))
                     {
                     string[i+1]='\0';
                     break;
                     }
              }
       while(string[0]==' ' || string[0]=='\t' || string[0]=='\n')
              string++;
       return string;
       }

void   read_initial_state(FILE *fp)
       {
       char *buffer;

       buffer=getexp(fp);
       buffer=strip(buffer);
       if(strcasecmp(buffer,"initial state:"))
              {
              fprintf(stderr,"Error:  Expected \"initial state\"\n");
              exit(1);
              }
```

```
    buffer=getexp(fp);
        buffer=strip(buffer);

        if(buffer[strlen(buffer)-1] != ';' )
                {
                fprintf(stderr,"Expected the initial state at %s\n",buffer);
                exit(1);
                }
        init_state = atoi(buffer);
        }


char*   get_variable(FILE *fp)
        {
        char *buffer;
        char inchar;

        do {
           inchar = next_char(fp);
           } while( inchar == ' ' || inchar == '\n' );
        if(inchar =='}')
                return 0;
        else
                put_char_back(inchar,fp);
        buffer = getword(fp);
        buffer = strip(buffer);

        return buffer;
        }

void    read_variables(FILE *fp)
        {
        char *buffer, *var;
        char inchar;
        int i, error=0;;

        buffer = getexp(fp);
        buffer = strip(buffer);
        if(strcasecmp(buffer,"variables:"))
                {
                fprintf(stderr,"Error:  Expected \"variables:\" \n");
                exit(1);
                }
        do {
           inchar = next_char(fp);
           } while( inchar == ' ' || inchar =='\n');
        if(inchar != '{')
                {
                fprintf(stderr,"Error:  Expected '{'\n");
                exit(1);
                }
        while(var = get_variable(fp))
                {
                error=0;
                for(i=0;i<num_variables;i++)
                        {
                        if(!strcmp(var, variables[i]))
                                {
                                printf("Warning:  Duplicate variable: %s\n", var);
                                error=1;
                                }
                        }
                if(!error)
                {
                        variables[num_variables] = strdup(var);
                        num_variables++;
                        }
                }
        }

char*   get_list(FILE *fp, int &size)
        {
        char *buffer, *bracket;
        char inchar;

        do {
           inchar = next_char(fp);
           } while( inchar == ' ' || inchar == '\n' );

        if(inchar =='}')
                return 0;
        else
                put_char_back(inchar,fp);

        buffer = getword(fp);
        buffer = strip(buffer);

        bracket=strchr(buffer,'[');
        if(bracket==NULL)
                {
                fprintf(stderr,"Error:  Expected '['\n");
                exit(1);
                }
        size = atoi(bracket+1);
        bracket[0]='\0';

        return buffer;
        }


void    read_lists(FILE *fp)
        {
        char *buffer, *list;
        char inchar;
        int size, error, i;

        buffer = getexp(fp);
        buffer = strip(buffer);
        if(strcasecmp(buffer,"lists:"))
                {
                fprintf(stderr,"Error:  Expected \"lists:\" \n");
                exit(1);
                }
        do {
           inchar = next_char(fp);
           } while( inchar == ' ' || inchar =='\n');
        if(inchar != '{')
                {
                fprintf(stderr,"Error:  Expected '{'\n");
                exit(1);
                }
        while(list = get_list(fp,size))
                {
                error=0;
                for(i=0;i<num_lists;i++)
                        {
```

```
                if(!strcmp(list,lists[i]->name))
                        {
                        printf("Warning:  Duplicate list name: %s\n",list);
                        error=1;
                        }
                    }
            if(!error)
                    {
                    lists[num_lists] = new LIST;
                    lists[num_lists]->name = strdup(list);
                    lists[num_lists]->size = size;
                    num_lists++;
                    }
            }
        }

void    output_prog_header(FILE *in, FILE *out)
        {
        char *buffer;

        while(buffer = getline(in))
                {
                if(strcmp(buffer,"<code1>\n"))
                        putline(out,buffer);
                else
                        break;
                }
        }

void    output_definitions(FILE *in, FILE *out)
        {
        char *buffer;

        while(buffer = getline(in))
                {
                if(strcmp(buffer,"<code2>\n"))
                        putline(out,buffer);
                else
                        break;
                }
        }


char*   get_form(FILE *fp, int &stateno)
        {
        char *buffer;
        char form[MAX_FORM_SIZE];
        char ch;
        int i=0;

        stateno = 0;
        buffer = getexp(fp);
        buffer = strip(buffer);
        if(!strcmp(buffer,"code:"))
                return NULL;
        if(strncasecmp(buffer,"form",4))
                {
                fprintf(stderr,"Error: form#: expected, got %s\n",buffer);
                exit(1);
                }
        for(i=0;i<strlen(buffer);i++)
                {
                stateno = atoi(buffer+4);
        }
        do {
           ch = getc(fp);
           } while (ch != '{');
        i = 0;
        do {
           ch = getc(fp);
           if (ch != '}')
               {
               form[i] = ch;
               i++;
               }
           } while(ch != '}'&& i < MAX_FORM_SIZE-1);
        form[i] = '\0';
        return form;
        }

void    print_form(int num, char *buf, FILE *fp)
        {
        int i=0;

        fprintf(fp,"#define FORM%d \"",num);
        for(i=0;i<strlen(buf);i++)
                {
                if(buf[i]=='"')
                        {
                        fputc('\\',fp);
                        }
                if(buf[i]=='\n')
                        {
                        fprintf(fp,"\\");
                        }
                fputc(buf[i],fp);
                }
        fprintf(fp," \"\n\n");
        }

void    read_and_output_forms(FILE *in, FILE *out)
        {
        char *buffer;
        char inchar;
        int error, i=0, form_num, found_state;

        while(buffer = get_form(in,form_num))
                {
                for(i=0;i<num_states;i++)
                        {
                        if(states[i]==form_num)
                                {
                                fprintf(stderr,"Error:  Multiple forms defined for state %d.\n",
                                                form_num);
                                exit(1);
                                }
                        }
                states[num_states]=form_num;
                num_states++;
                print_form(form_num,buffer,out);
                }
        fprintf(out,"\n#define WEB_init_state %d\n",init_state);
        }

void    output_variables(FILE *out)
        {
```

```
   int i=0;

       fprintf(out,"/* Variable definitions */\n");
       for(i=0; i<num_variables; i++)
              {
              fprintf(out,"VARIABLE %s;\n", variables[i]);
              }
       fprintf(out,"\n\n/* List definitions */");

       for(i=0;i<num_lists;i++)
              {
              fprintf(out,"LIST %s(%d);\n",lists[i]->name,lists[i]->size);
              }
       fprintf(out,"\n\n");
       }

void   read_and_output_user_code(FILE *insrc, FILE *incode, FILE *out)
       {
       char *buffer;

       while(buffer = getline(incode))
              {
              if(strcmp(buffer,"<code3>\n"))
                     putline(out,buffer);
              else
                     break;
              }
       while(buffer = getline(insrc))
              {
              putline(out,buffer);
              }
       }

void   output_initialize_variables(FILE *incode, FILE *out)
       {
       char *buffer;
       int i=0;

       while(buffer = getline(incode))
              {
              if(strcmp(buffer,"<code10>\n"))
                     putline(out,buffer);
              else
                     break;
              }

       for(i=0;i<num_variables;i++)
              {
              fprintf(out,"%s = strdup(\"\");\n", variables[i]);
              }

       }

void   output_code1(FILE *incode, FILE *out)
       {
       char *buffer;
       int i=0;

       while(buffer = getline(incode))
              {
              if(strcmp(buffer,"<code4>\n"))
                     putline(out,buffer);
              else
```

```
             break;
             }

      fprintf(out,"  printf(\"%%s\\n\",FORM%d);\n",init_state);
      }

void  output_code2(FILE *incode, FILE *out)
      {
      char *buffer;
      int i=0;

      while(buffer = getline(incode))
             {
             if(strcmp(buffer,"<code5>\n"))
                    putline(out,buffer);
             else
                    break;
             }
      for(i=0; i < num_variables; i++)
             {
             fprintf(out,"if(!strcmp(entries[x].name,\"%s\"))\n  {\n",
                           variables[i]);
             fprintf(out,"  %s = strdup(entries[x].val);\n  }\n ",
                           variables[i]);
             }
      while(buffer = getline(incode))
             {
             if(strcmp(buffer,"<code6>\n"))
                    putline(out,buffer);
             else
                    break;
             }
      for(i=0; i < num_lists; i++)
             {
             fprintf(out,"if(!strcmp(entries[x].name,\"%s\"))\n  {\n",
                           lists[i]->name);
             fprintf(out,"  %s.add_string(entries[x].val);\n",
                           lists[i]->name);
             fprintf(out,"  }\n");
             }


/*                          list
      if ((!strcmp(entries[x].name,"platforms"))
           && (num_platforms < MAX_PLATFORMS)
             {
             platforms[num_platforms] = strdup(entries[x].val);
             num_platforms++;
             }
*/
      }

void  output_code3(FILE *incode, FILE *out)
      {
      char *buffer;
      int i=0;

      while(buffer = getline(incode))
             {
             if(strcmp(buffer,"<code7>\n"))
                    putline(out,buffer);
             else
                    break;
```

```
        }
/*
        for(i=0;i<num_transitions;i++)
                {
                fprintf(out,"WEB_state_num == %d && WEB_next_state == %d ||\n",
                        transitions[i]->from_state,
                transitions[i]->to_state);
                }
*/
        fprintf(out,"0\n");
        }

void    output_code4(FILE *incode, FILE *out)
        {
        char *buffer;
        int i=0;

        while(buffer = getline(incode))
                {
                if(strcmp(buffer,"<code8>\n"))
                        putline(out,buffer);
                else
                        break;
                }
        for(i=0;i<num_variables;i++)
                {
                fprintf(out,"    printf(\"<input type=\
                                variables[i], variables[i]);
                }
        for(i=0;i<num_lists;i++)
                {
                fprintf(out,"    for(i=0;i<%s.get_number();i++) {\n",
                                lists[i]->name);
                fprintf(out,"      printf(\"<input type=\
                                lists[i]->name, lists[i]->name);
                fprintf(out,"    }\n");
                }

        }

void    output_code5(FILE *incode, FILE *out)
        {
        char *buffer;
        int i=0;

        while(buffer = getline(incode))
                {
                if(strcmp(buffer,"<code9>\n"))
                        putline(out,buffer);
                else
                        break;
                }
        for(i=0;i<num_states;i++)
                {
                fprintf(out,"    case %d:   printf(\"%%s\",FORM%d);\n",i,i);
                fprintf(out,"               break;\n");
                }
        fprintf(out,"    default:   printf(\"%%s\",FORM%d);\n",init_state);
        fprintf(out,"               break;\n");

        }

void    output_code6(FILE *incode, FILE *out)
```

```
   {
       char *buffer;

       while(buffer = getline(incode))
              {
              putline(out,buffer);
              }
       }

void   copy_code(FILE *src, FILE *dest)
       {
       char *buffer;

       while(buffer = getline(src)) fprintf(dest,"%s",buffer)
              ;
       }

int    main(int argc, char *argv[])
       {
       char *buffer;
       FILE *src, *dest1, *dest2, *dest3, *code1, *code2, *code3;
       int i=0;

       init_state = 0;
       num_variables = 0;
       num_states = 0;
       num_lists = 0;
       num_forms = 0;
       for(i=0; i<MAX_STATES; i++)
              states[i] = -1;

       printf("<<%s>>\n", argv[1]);

       src = fopen(argv[1],"r");

                     /* read_transitions(src); */
       read_initial_state(src);
       read_variables(src);

       read_lists(src);

       dest1 = fopen("masterform.C","w");
       dest2 = fopen("web_util.C","w");
       dest3 = fopen("web_util.h","w");
       code1 = fopen("/home/sdr4p/web/src/code1.C","r");
       code2 = fopen("/home/sdr4p/web/src/code2.C","r");
       code3 = fopen("/home/sdr4p/web/src/code3.C","r");

       output_prog_header(code1,dest1);

       read_and_output_forms(src,dest1);

       output_definitions(code1,dest1);

       output_variables(dest1);

       read_and_output_user_code(src, code1, dest1);
       output_initialize_variables(code1,dest1);
                     /* Need to copy the supporting     */
                     /* files to the source directory */
       output_code1(code1,dest1);
       output_code2(code1,dest1);
       output_code3(code1,dest1);
   output_code4(code1,dest1);
       output_code5(code1,dest1);
       output_code6(code1,dest1);

       copy_code(code2,dest2);
       copy_code(code3,dest3);

       printf("<<<<<<<<<<<<<<>>>>>>>>>>>>>>\n");
       for(i=0;i<num_states;i++)
              printf("State: %d\n",states[i]);
       for(i=0;i<num_variables;i++)
              printf("Variable: %s\n", variables[i]);
       for(i=0;i<num_lists;i++)
              printf("List: %s {%d}\n",lists[i]->name,lists[i]->size);
       return 0;
       }
```