Run-time Generation of JavaScript Code by Perl CGI Programs

Stephen B. Jenkins
Senior Programmer/Analyst

Institute for Aerospace Research
National Research Council
Ottawa, Canada

YAPC::America::North 2001
McGill University, Montreal, Quebec
June 13 – 15, 2001

Abstract

Run-time generation of JavaScript code by Perl CGI programs is a technique that has been used with considerable success at the Aerodynamics Laboratory of the Institute for Aerospace Research, National Research Council of Canada. After a brief introduction to the basic concept of dynamic code generation and its role within the data acquisition, data display and test-control software of the 2m x 3m wind tunnel, four specific examples will be presented each highlighting a different application of the technique. The first and simplest example will show the generation of trivial one-line JavaScript programs. The second example will show the run-time generation of JavaScript functions that modify groups of HTML checkbox elements. The third example will cover form validation in the browser, complete with pop-up alert boxes and dynamic images to prompt the user. The final, and most complex case, will show the creation of JavaScript code that produces dynamic behavior in HTML forms. These examples not only demonstrate the primary benefit of the technique – a more interactive user interface – but also show the secondary benefits: reduced load on both the web-server and web-client computers.

1 Introduction

1.1 Run-time code generation (RTCG)

Run-time code generation is the technique of having a program create computer instructions – either another program or more code for its own thread of execution – "on the fly", using decisions made at run-time to influence the final product. Typically, the concept of run-time code generation is discussed with the aim of reducing the system resources (CPU time, memory, etc.) required by large applications (Keppel et al, 1991; Leone and Lee, 1994). There is often an implicit assumption that the same language will be used for both the generating and generated code and that both will run on the same computer. This is not unlike the most common type of RTCG in Perl: using **`eval`** to execute code that is programmatically generated.

In "*The Pragmatic Programmer*", Hunt and Thomas discuss the generation of code in more than one language (Hunt and Thomas, 2000):

> *"Whenever you find yourself trying to get two disparate environments to work together, you should consider using active code generators."*
>
> *"Another example of melding environments using code generators happens when different programming languages are used in the same application. In order to communicate, each code base will need some information in common – data structures, message formats, and field names for example. Rather than duplicate this information, use a code generator. Sometimes you can parse the information out of the source files of one language and use it to generate code in the second language. Often, though, it is simpler to express it in a simpler, language-neutral representation and generate the code for both languages."*

Closer to the topic of this paper are the comments by Kernighan and Pike in a section titled "Programs that Write Programs" from "*The Practice of Programming*" (Kernighan and Pike, 1999):

> *"One common example is the dynamic generation of HTML for web pages. HTML is a language, however limited, and it can contain JavaScript code as well. Web pages are often generated on the fly by Perl or C programs, with specific contents … determined by incoming requests."*

They go on to endorse the technique by saying:

> *"In spite of the power of program generators, and in spite of the existence of many good examples, the notion is not appreciated as much as it should be and is infrequently used by individual programmers."*

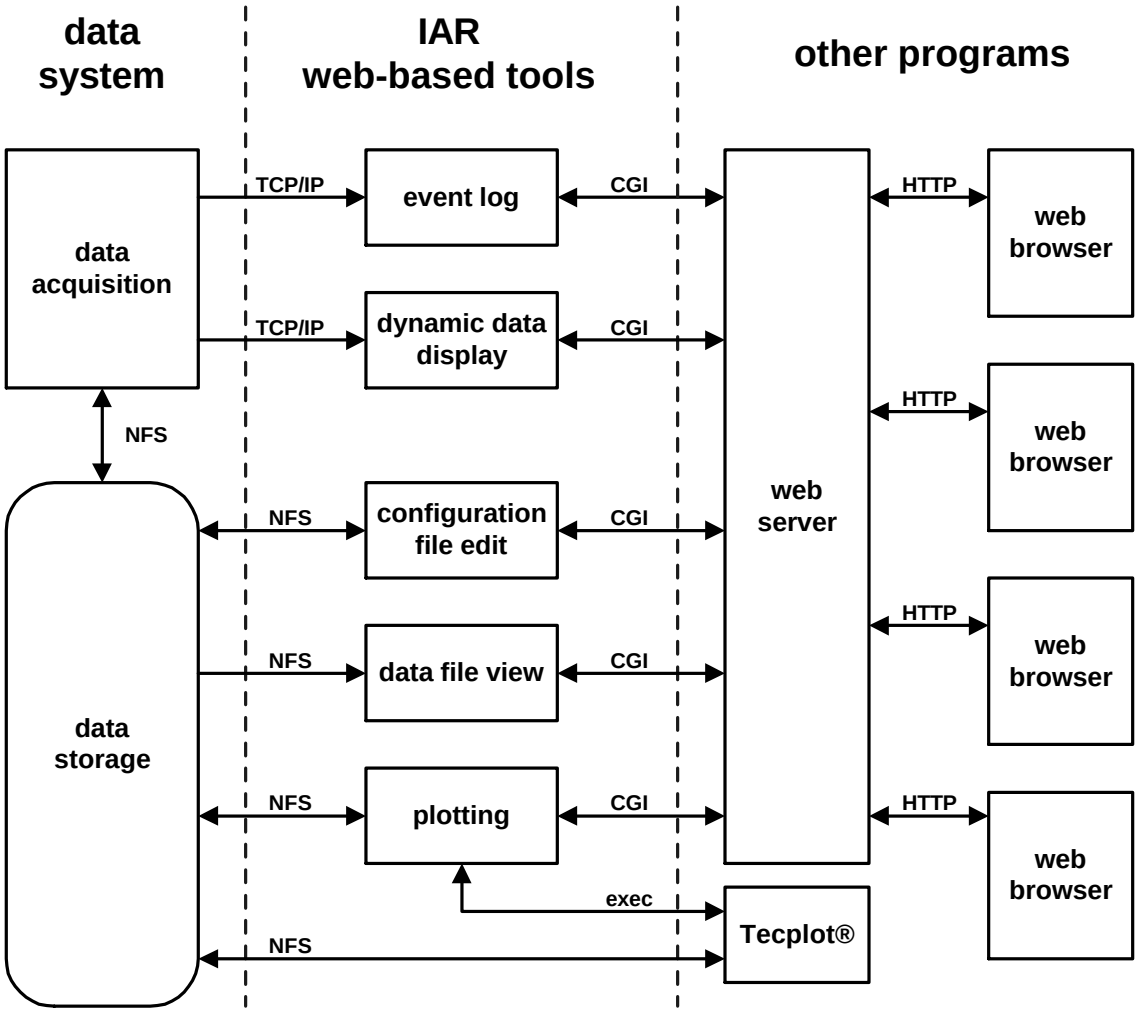

Figure 1: Operating environment

1.2 Operating environment

Several years ago a decision was made at the Aerodynamics Laboratory to move away from proprietary "home grown" user interfaces, toward the utilization of web-based systems wherever it was sensible to do so (Jenkins, 2001). This was done in order to provide clients with a more consistent and robust method of interacting with the wind tunnel data acquisition and display system.

As may be seen in Figure 1, the web interface to the wind tunnel data system is made up of the five individual software tools developed at the Aerodynamics Laboratory, the web server, the web browsers, and the commercial plotting package Tecplot®. Of the five systems developed in-house, two deal directly with the data acquisition programs: the event logging and viewing system, and the dynamic data display system. These two tools use client-server relationships and TCP/IP socket connections to receive their information and use the CGI protocol to communicate with the web server. The other three systems – configuration file editing, data file viewing, and plotting – all receive their information indirectly through disk storage, but they too use CGI to interact with the web server.

To utilize any of these tools, the user simply opens a web browser on one of the computers in the control room and the wind tunnel home page will be loaded automatically. It contains links to each of the five systems as well as links to formal documentation, to help files, and to programs that report the status of each of the servers. Because virtually all clients of the 2m x 3m wind tunnel are accustomed to 'surfing the web', they require no instruction in order to be able to make use of any of these software tools.

1.3 RTCG at the Aerodynamics Laboratory

In order to provide users at a web browser with the most responsive interface possible, the Laboratory programming staff has elected to make use of the JavaScript programming language (Flanagan, 1998). Because JavaScript runs within the user's browser (and thus on his computer), it can have a significantly shorter response time than a CGI program running on a remote web server. Since the web pages that contain the JavaScript code are dynamically generated by Perl programs, and since the contents of these pages are based on information available only at run-time, it follows that the JavaScript that manipulates these contents must also be created at run-time.

In this paper, four specific examples of RTCG will be presented, each highlighting a different application of the technique. The first and simplest example will show the generation of trivial one-line JavaScript programs. The second example will show the run-time generation of JavaScript functions that modify groups of HTML checkbox elements. The third example will cover form validation in the browser, complete with pop-up alert boxes and dynamic images to prompt the user. The final and most complex case will show the creation of JavaScript code that produces dynamic behavior in HTML forms.

In addition to the primary goal of providing a more responsive user interface, a beneficial secondary effect of this technique is the reduction of the load on web servers due to the reduced number of requests per user session. Also, one example will be shown that results in a reduction of the load on the client computer because of the re-use, rather than re-load, of a complex web page.

It should be noted that the computers of the 2m x 3m wind tunnel operate within the confines of an Intranet model: outside access is severely restricted and allowable hardware and software are tightly controlled. The techniques covered in this paper assume this model; readers are strongly cautioned to examine the suitability of these methods for their own environments.

2 Specific examples

2.1 Simple one-line JavaScript programs

The first example of code generation to be discussed is the creation of one-line JavaScript programs to accomplish simple tasks. Figure 2 shows a portion of the output of a Perl program that was written to allow users to view, with a single button click, the contents of any of the data and configuration files for a given wind tunnel test. Since tests routinely generate more than 4000 files, the maximum time required to construct, transmit, and render this web page is not insignificant (typically on the order of 10-15 seconds). This rather lengthy process would need to be repeated for each displayed file if the HTML buttons were created to use the default **`submit`** action. Instead, they were fashioned so that when pressed, each button initiates the execution of a one-line JavaScript program through its **`onClick`** event handler. This program causes a new browser window to spring into existence, and then requests the desired data file from a separate Perl CGI program (**`displayfile.pl`** in the code below) that was written to format and display a file's contents. This technique allows the web page with the long list of buttons to remain intact in the original window.

Because the Perl program searches through an entire directory tree and because it is not uncommon to have several data files of the same name in different directories, an additional feature has been added. When the user presses and holds a button, the full file path is displayed in the status line at the bottom of the browser window. This too is accomplished using a single line of JavaScript code, but with the **`onMouseDown`** event handler. If the selected file was not the one desired, the user simply moves the cursor away before releasing the mouse button.

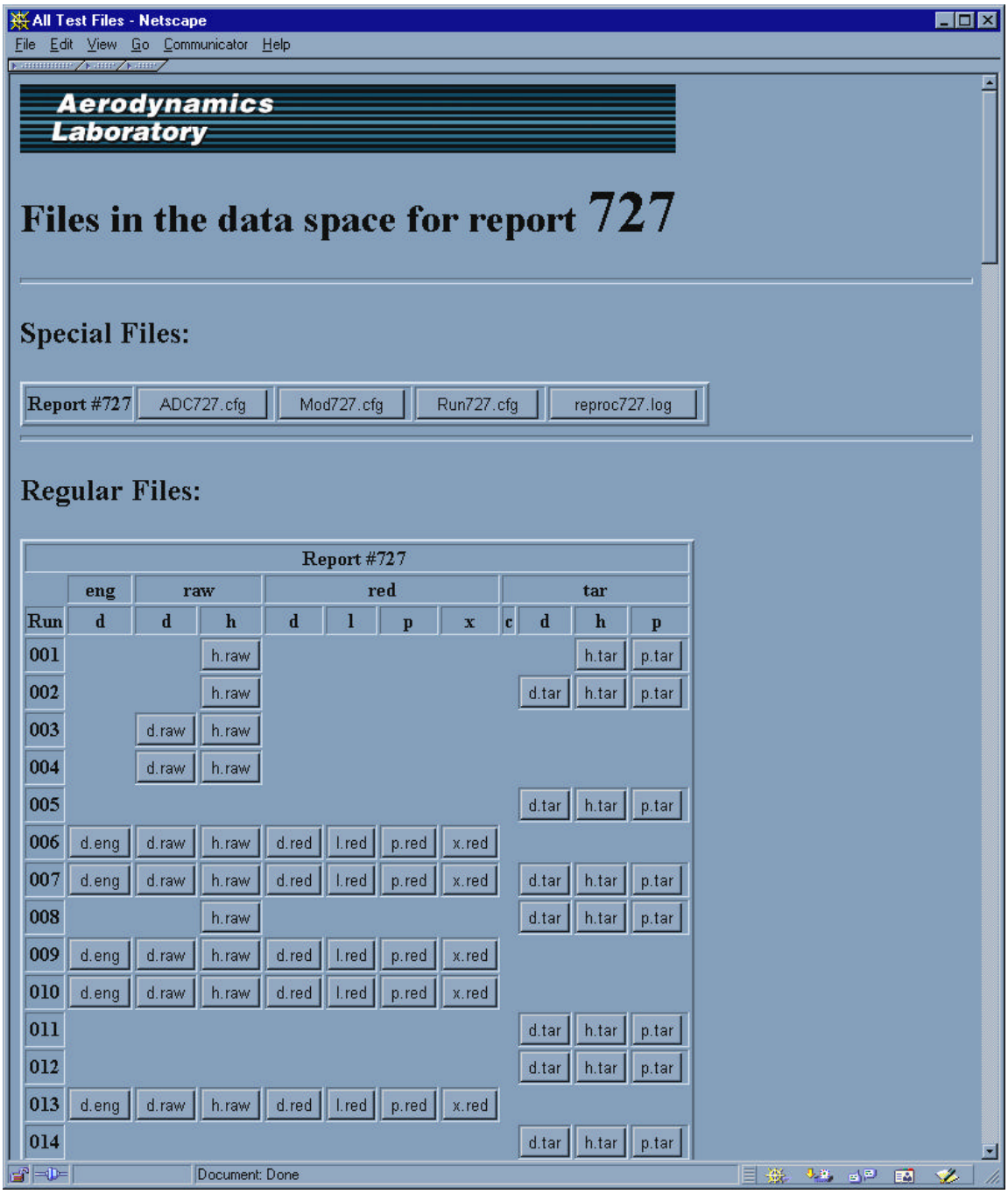

Figure 2: Data file viewing main web page

The following code is a rudimentary version of the Perl program that produces buttons as described above. In order to provide the simplest example possible, it ignores many issues such as file paths, HTML table formatting, etc.

```
#!/usr/bin/perl -wT
use strict;
use CGI qw(:standard);

# get the files to list (use File::Find in real life)
my @filenames = qw( This should really be a list of filenames );

# now create a JavaScript routine that opens the new browser window
my $JSmakepage = <<EOF;
    function makepage(filename) {
        newwin = window.open('displayfile.pl?FILE=' + filename);
        newwin.focus();
    }
EOF

# start the HTML and send the JS to the browser
print header;
print start_html( -title  => 'yapc2001-example1.pl',
                  -script => {-language=>'JavaScript', -code=>$JSmakepage} );
print start_form( -method => 'POST');

# create the buttons that will call the JS routine
foreach my $filename (@filenames) {
    (my $buttonname = $filename) =~ s/\W+/_/g; # do this to keep the browser
    my $buttonvalue = $buttonname;             #  happy with the names & values
    my $button = '<INPUT ' .
                 'TYPE="button" ' .
                 'NAME= "' . $buttonname .  '" ' .
                 'VALUE="' . $buttonvalue . '" ' .
                 'onClick="makepage(\'' . $filename . '\')" ' .
                 'onMouseDown="window.status=\'' . $filename . '\' " >';
    print "$button<BR>";
}

print end_form(), end_html();
```

Since the list of data and configuration files is a "snapshot" of the file system taken at run-time, dynamic code generation is the only method of implementing this functionality in a web-based interface. Although this is the simplest of the four examples to be discussed in this paper, it clearly demonstrates all three benefits of RTCG: a more responsive user interface, a reduction of the load on the web server, and a reduction of the load on the client computer.

2.2 Buttons that manipulate groups of checkboxes

The web page shown in Figure 3 allows users to initiate a search of data-system event logs (Jenkins, 1999). One of the search criteria that can be specified is the name of the machine reporting the event. Although a single checkbox was created to allow all machine names to be included in the search, it is common practice to require all but one or two names from the list. In order to make this process as convenient as possible a button has been provided that checks all boxes, allowing the user to de-select individual machines. Similarly, a single button click is all that is required to clear all checkboxes. In the manner described in the previous section, these buttons have been created to use the **`onClick`** event handler to call JavaScript functions, rather than cause a **`submit`** action. Since these checkboxes are generated at run-time from the names in the web server's **`hosts`** file, it follows that the functions to manipulate them must also be created at run-time.

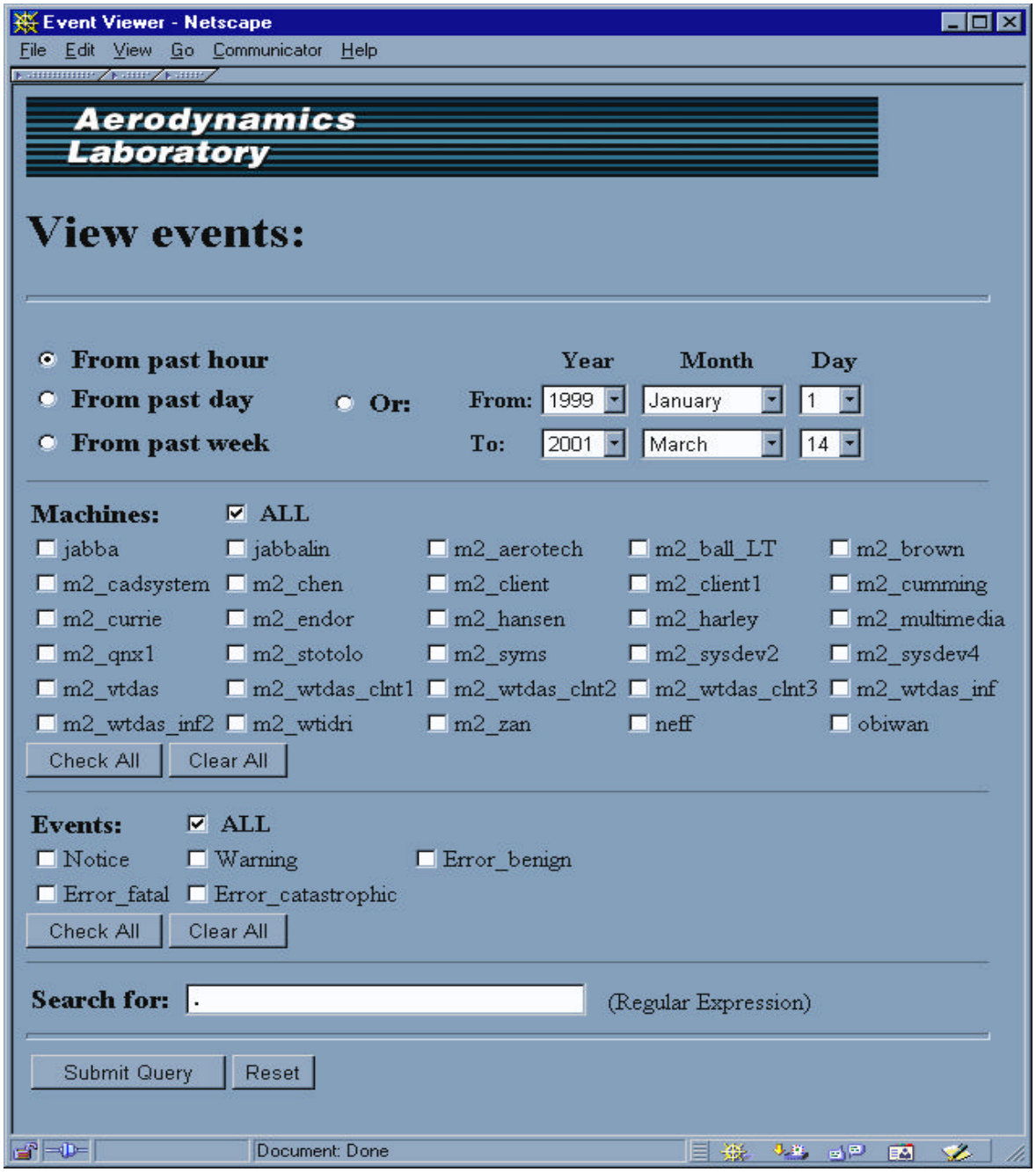

Figure 3: Event log query web page

The following code demonstrates the generation of a group of checkboxes as described above. To keep the example simple, the list of checkbox names is given in an array (**`@machinenames`**) and only elementary formatting has been performed. The process of extracting the names from a **`hosts`** file and modifying them for use in HTML elements has been omitted as it is not germane to this discussion.

```
#!/usr/bin/perl -wT
use strict;
use CGI qw(:standard);

my $groupname = 'machines';
my @machinenames = qw( This should really be a list of computer names );

# generate the JavaScript and HTML
my $js = &createJSfuncs($groupname, @machinenames);
my $html = &createHTMLcheckboxes($groupname, @machinenames);

# send the JS and HTML to the browser
print header;
print start_html( -title  => 'yapc2001-example2.pl',
                  -script => {-language=>'JavaScript', -code=>$js} );
print start_form(), $html, end_form();
print end_html();


# create the JavaScript 'check all' and 'check none' functions
sub createJSfuncs {
    my $groupname = shift;      # the name of the checkbox group
    my @boxnames =  @_;         # the list of checkbox names
    my $all =  "function ${groupname}_all()  {\n";
    my $none = "function ${groupname}_none() {\n";

    foreach (@boxnames) {
        $all .=  "\tself.document.forms[0].$_.checked = true;\n";
        $none .= "\tself.document.forms[0].$_.checked = false;\n";
    }
    $all .= "}\n";
    $none .= "}\n";
    return("\n$all\n$none");
}

# create the HTML for a group of checkboxes with all & none buttons
sub createHTMLcheckboxes {
    my $groupname = shift;      # the name of the checkbox group
    my @boxnames =  @_;         # the list of checkbox names
    my $checkboxgroup = '';

    foreach (@boxnames) {
        $checkboxgroup .= checkbox( -name=> $_ ) . "<BR>\n";
    }
    $checkboxgroup .= button( -name    => "${groupname}allbutton",
                              -value   => 'Check All',
                              -onClick => "${groupname}_all()" ) . "\n" .
                      button( -name    => "${groupname}nonebutton",
                              -value   => 'Clear All',
                              -onClick => "${groupname}_none()" ) . "\n";
    return($checkboxgroup);
}
```

Although this case does not result in any significant reduction of the load on either the web server or client, it does provide a good example of increased convenience for the user.

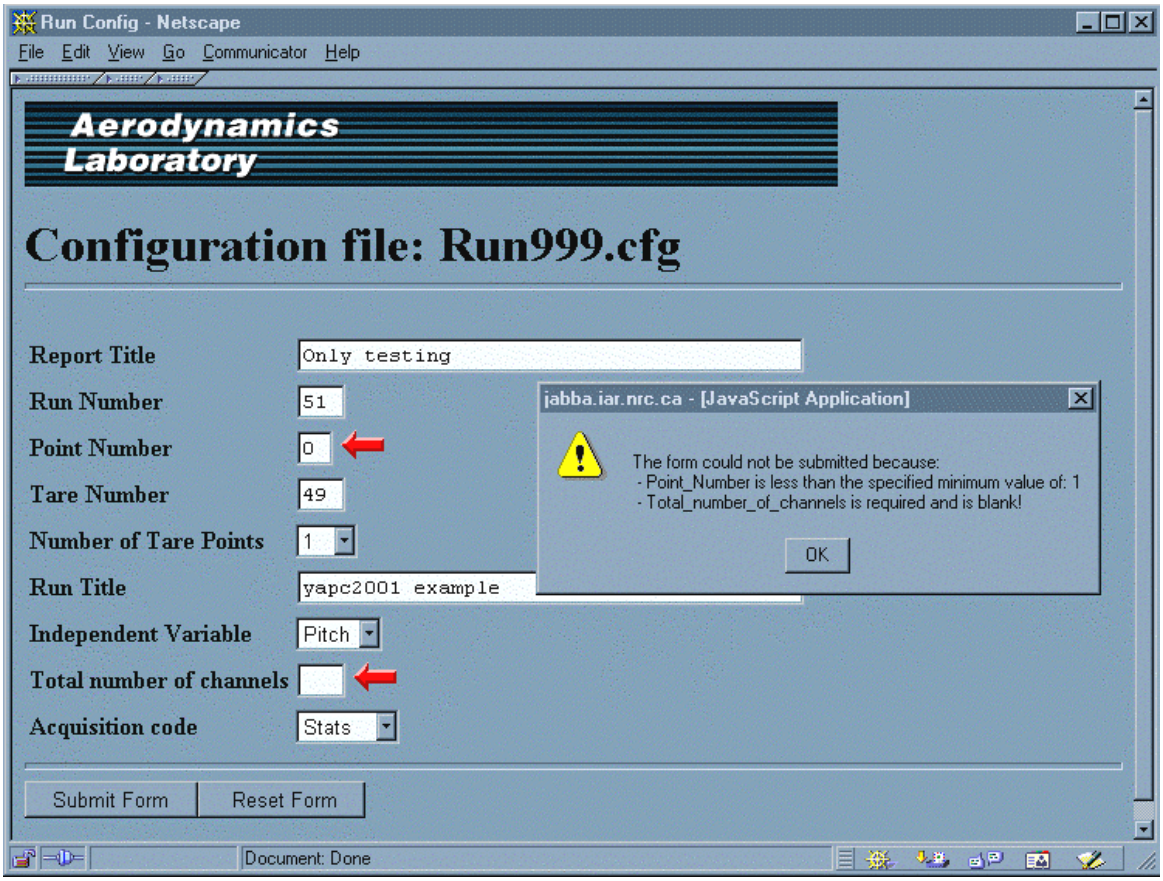

Figure 4: Configuration file editor web page

2.3 Form validation

A good introduction to the use of JavaScript to carry out form validation is contained in *JavaScript: The Definitive Guide* (pp. 311-314). Although Flanagan's example assumes traditional static programming, it is easily adapted for run-time generation by Perl programs.

Figure 4 shows the page generated by a web-based configuration file editor. The configuration files for the wind tunnel data system are made up of several lines, each with the following information: a name string, a delimiter, a description string, another delimiter, and the current value. The descriptor string contains a field type (dropdown or text) and several field constraints such as minimum value, maximum value, numeric characters only, etc. These constraints are used to generate the JavaScript validation code for each line and when the "**Submit Form**" button is pressed, that code is executed. An error string is built up from any entries that fail the validation process, and the user is notified of all problems in a single pop-up dialog box. In order to remove any potential ambiguity due to similar names, additional JavaScript code uses a standard image swapping technique to cause flashing red arrows to appear beside each of the offending entries. Once all input data passes the validation process, the form is submitted to the CGI program.

It should be noted that, although form validation in JavaScript is provided as a convenience, the Perl program should revalidate the input data before it is used. This is due to the fact that a malicious user can easily subvert the JavaScript processing, passing invalid data to the CGI program.

2.4 Dynamic forms

The final and most complex example of RTCG to be presented is the creation of forms with dynamic elements. Figure 5 shows a web page that allows clients to build motion "maps" – files that are used to control the model attitude and data acquisition processes. For the sake of brevity, this discussion will focus only on the radio box and select elements in the section of the table under the heading "**Action**" (Figure 6). It should be noted (although it does not impact the description to follow) that this software tool is not yet in regular use at the Aerodynamics Laboratory. The user interface and command generator sections have been completed, but the hardware control code is still under development.

For each step of a motion map, the user may invoke any one of three possible types of actions: a data acquisition process, an arbitrary command file, or another map process. The desired action-type can be selected with an HTML radio button. For each of these three types, there exists several command files to choose from through the use of an HTML select box. This arrangement mimics the structure of the information on disk, where each action-type has its own directory containing multiple data-system control files. This system allows complex combinations of model attitude and data system operations to be specified quite succinctly.

The Perl CGI program that creates the web pages reads the action-type directories and constructs a pair of JavaScript hashes whose keys are the action-types. The values of the first hash, called **`actionlabels`**, are strings containing a comma-separated list of simplified file names – path and extension are removed. The values of the second hash, called **`actionvalues`**, are strings containing a comma-separated list of the full file paths. When the user clicks on an action-type radio button, a JavaScript function uses that button's name to access the pair of hashes, and retrieves the strings containing the simplified and full file descriptions. These strings are parsed into their components, and the options in the "**File**" select box are replaced with the information for the action-type requested by the user.

This process may be more easily understood by way of an example. To produce the web page shown in Figures 5 & 6, the **`acquisition`** directory must have contained the following three files: **`force.acq`**, **`force_press.acq`**, and **`pressure.acq`**; and the **`map`** directory must have contained the following six files: **`basicyaw.map`**, **`japh-Erudil.map`**, **`pithchandyaw.map`**, **`probe_angle.map`**, **`probe_multi.map`** and **`yawspeep.map`**. The Perl program would have generated the following JavaScript hashes based on that information (in the code snippet below, whitespace has been added to improve readability):

```
actionlabels["Run Map"] = "none, basicyaw, japh-Erudil, pitchandyaw, probe_angle,
       probe_multi, yawsweep";

actionlabels["Run Acquisition"] = "none, force, force_press, pressure";

actionvalues["Run Map"] = "none,
       /some/appropriate/directory/map/basicyaw.map,
       /some/appropriate/directory/map/japh-Erudil.map,
       /some/appropriate/directory/map/pitchandyaw.map,
       /some/appropriate/directory/map/probe_angle.map,
       /some/appropriate/directory/map/probe_multi.map,
       /some/appropriate/directory/map/yawsweep.map";

actionvalues["Run Acquisition"] = "none,
       /some/appropriate/directory/acq/force.acq,
       /some/appropriate/directory/acq/force_press.acq,
       /some/appropriate/directory/acq/pressure.acq";
```

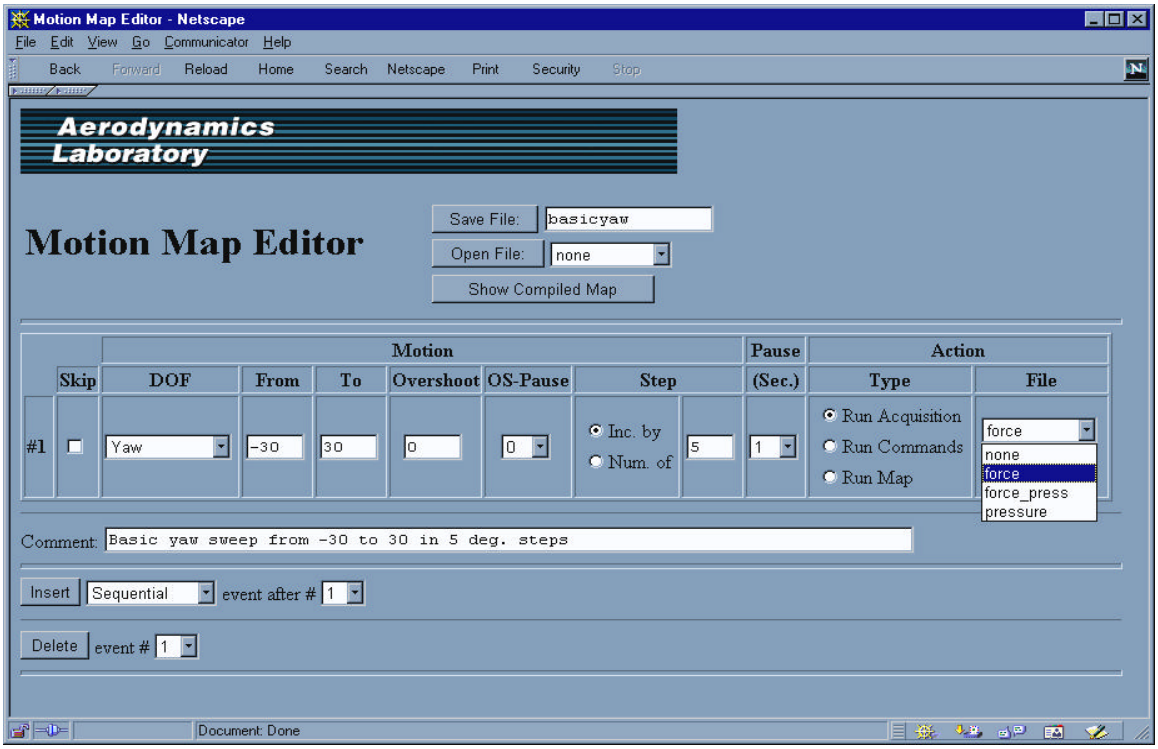

Figure 5: Motion map editor web page

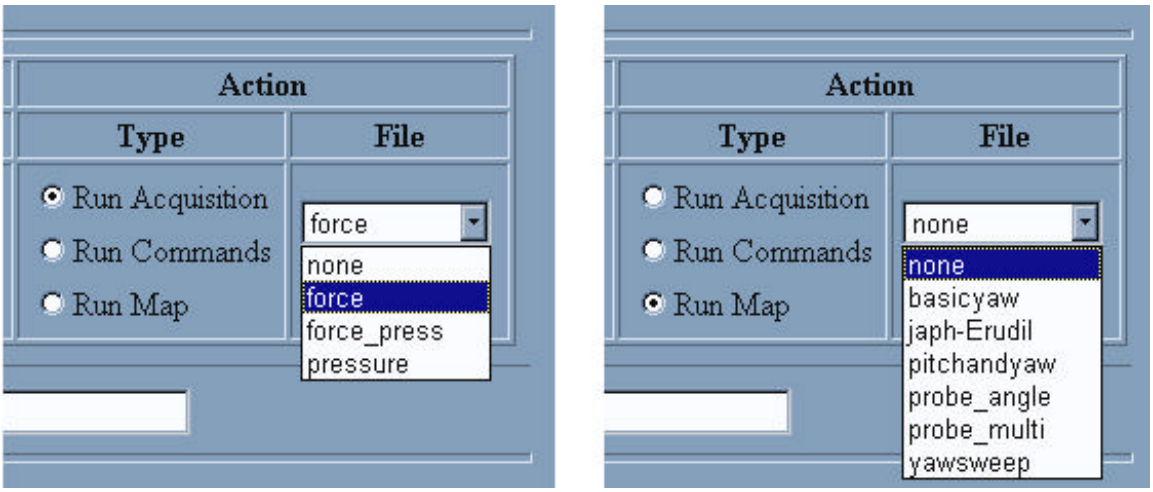

Figure 6: Motion map editor - detail

The dynamically generated hashes shown above would have been inserted into the **<head>** portion of the web page along with the following JavaScript function:

```
function action_changed(actiontype, actfilelist) {

    actfilelist.options.length = 0;              // clear the old action file list

    for( i=0; i<actiontype.length; i++ ) {       // find the current action type
        if( actiontype[i].checked ) val = actiontype[i].value;
    }

    newlabels = actionlabels[val].split(",");    // use the action type to get labels
    newvalues = actionvalues[val].split(",");    //  and the values

    for( i=0; i<newlabels.length; i++ ) {        // setup the new action file list
        actfilelist[actfilelist.length] = new Option(newlabels[i],newvalues[i]);
    }
    actfilelist[0].selected = 1;                 // set the default selection

    history.go(0);                               // redraw page to adjust the width
}
```

The simplified HTML for the "**Action**" section of the default web page is as follows:

```
<TABLE>
<TR><TD>
<INPUT TYPE="radio" NAME="ACTION1" VALUE="Run Acquisition" CHECKED
       onClick="action_changed(
            self.document.forms[0].ACTION1,self.document.forms[0].ACTFILE1)">
Run Acquisition
</TD></TR>

<TR><TD>
<INPUT TYPE="radio" NAME="ACTION1" VALUE="Run Commands"
       onClick="action_changed(
            self.document.forms[0].ACTION1,self.document.forms[0].ACTFILE1)">
Run Commands
</TD></TR>

<TR><TD>
<INPUT TYPE="radio" NAME="ACTION1" VALUE="Run Map"
       onClick="action_changed(
            self.document.forms[0].ACTION1,self.document.forms[0].ACTFILE1)">
Run Map
</TD></TR>
</TABLE>

<SELECT NAME="ACTFILE1">
<OPTION SELECTED VALUE="none">none
<OPTION  VALUE="/some/appropriate/directory/acq/force.acq">force
<OPTION  VALUE="/some/appropriate/directory/acq/force_press.acq">force_press
<OPTION  VALUE="/some/appropriate/directory/acq/pressure.acq">pressure
</SELECT>
```

When the user selects a new action-type by clicking on the "**Run Map**" radio box, it's **onClick** event handler executes the **action_changed()** function which replaces the "**File**" select options for the default (acquisition) action-type with those for the map action-type.

Since the contents of each of the three action-type directories are not known in advance, run-time generation of the data hashes is required in order to take advantage of the performance benefits afforded by client-side processing in JavaScript. Although the example has focussed on the "**Action**" section of the web page in order to highlight the dynamic form technique described above, this Perl program also makes use of the form validation method described in the previous example in order to validate the data entered into the other fields.

3 Concluding remarks

As has been shown in the four examples, the primary benefit to be derived from this type of RTCG is a more responsive user interface. At the Aerodynamics Laboratory, it is felt that this reason alone justifies the increased complexity of the CGI programs. The additional benefits in terms of reduced load on clients, servers and the network itself, while not insignificant, are "icing on the cake". This is due to the fact that, although load issues may be dealt with simply by spending money on faster hardware, the advantages gained by a better user interface – higher comfort level, increased productivity, shallower learning curve, etc. – cannot be so easily acquired.

There seem to be two major disadvantages to the RTCG techniques described in this paper. The first, and most serious, is the increase in the skill set required of the programming staff. The complexity of writing programs that, in the same file, contain three languages (two programming and one markup) may seem daunting, but proper care and the separation of tasks make the process manageable. The second drawback of this type of RTCG is the increased difficulty in debugging the programs. One language can mask or hide errors from a process in another. The best way to handle this seems to be to make small, incremental changes during development, thus minimizing the scope of the search for erroneous code. As a programmer gains experience with the techniques, these two issues become far less problematic.

Over the past several years, run-time code generation has played an increasingly important role in the development of new software at the Aerodynamics Laboratory. This trend is expected to continue. Within the context of an Intranet such as the one that exists at the Laboratory, the benefits of the described techniques far outweigh their disadvantages.

4 Acknowledgements

The author would like to thank Dr. Steven J. Zan – the leader of the Montreal Road Wind Tunnel Facilities Group – for continuing to promote an atmosphere that is conducive to the investigation and implementation of new techniques and technologies.

5 References

Flanagan, D., *JavaScript: The Definitive Guide, Third Edition.* O'Reilly & Associates, 1998.

Hunt, A. and Thomas, D., *The Pragmatic Programmer*. Addison-Wesley, 2000, pp. 104-105.

Jenkins, S.B., *A System for Recording and Viewing Events in a Distributed Data Acquisition Environment.* yapc99 Proceedings, June 1999.

Jenkins, S.B., *Using Perl to Create Web-Based Software Tools for Wind Tunnel Testing*. AIAA 2001-905, 39th Aerospace Sciences Meeting & Exhibit, Reno, NV, Jan. 2001.

Keppel, D., Eggers, S., and Henry, R., *A Case for Runtime Code Generation.* TR 91-11-04, Univ. of Washington, 1991.

Kernighan, B. and Pike, R., *The Practice of Programming*. Addison-Wesley, 1999, pp. 237-238.

Leone, M. and Lee, P., *Lightweight Run-Time Code Generation*. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, June 1994.