The Boomerang White Paper

A Page as You Like It

Curtis E. Dyreson, Anthony M. Sloane

Abstract

Boomerang is a dynamic HTML page reconfiguration system. A user accesses Boomerang via the Common Gateway Interface. The user supplies a page name and a template, Boomerang fetches the requested page and uses the template to reconfigure it. The template is a sequence of string manipulation rules. The rules are written in a simple regular expression-based pattern matching language. Boomerang also parses HTML variables in forms and query strings and makes those variables available in the template. By taking advantage of Boomerang's dynamic page reconfiguration features, users can easily add navigational links, suppress images, redefine HTML tags, and reshape a page as desired. Since Boomerang is reached through the Common Gateway Interface and understands HTML variables, Boomerang can also be used as a general form-handling script. Several examples are given to show the utility of Boomerang. Boomerang is compatible with existing browsers and servers and does not compromise their security. ***Keywords:*** *HTML, document reconfiguration, structural regular expressions, parameterized documents, CGI*

Introduction

Hypertext document authors control how information on the World Wide Web is structured and presented. The authors write the raw text, include the appropriate HTML (HyperText Markup Language) tags, and insert all the links to other Web documents. In many cases, users are satisfied with the presentation because authors work hard to present that information in the best possible manner. But in some cases, there is a large gap between how the author has presented the information and how the user wants to view it.

An example is the lack of "return to point-of-entry" links in documents [2]. Suppose that the Web page authors at ACME On-line Shopping want to build an Internet mall. The mall will consist of a number of stores, each selling various items. It is reasonable to assume that each store is a unique hypertext document that is designed, developed, and maintained independently. Typically a store will be a sequence of forms that permit users to view items for sale, select items to purchase, enter payment information for those items, etc. The mall itself will have a single "welcome" page. The welcome page is a list of store categories (e.g., shoe stores, clothing stores, etc.) so that the user can quickly find stores selling products that the user needs. Within a mall a shopper can enter a store under several different categories. For example, a sporting goods store could be listed under athletic equipment as well as shoes and clothing. Leaving the store should return the shopper to the list of stores in a category. That is, if the customer entered from shoes, the customer should return to the list of shoe stores. But since stores are independently written and maintained, one cannot assume that they will have "exit" buttons or links. Furthermore, even if an exit button is included in each store since a store may be entered from many different locations (e.g., a sporting goods store may sell shoes, clothing, and athletic equipment and can be entered from all three places), it is impossible to anticipate in advance to where the shopper should be returned.

In other situations, a user may simply prefer a presentation different than what is provided by the document author. A common example is the

(over)use of images. Eye-popping colorful images are important for capturing the imagination of first-time document visitors, but in subsequent visits images tend to be of diminishing importance. In fact, they are often a source of irritation since images are large and slow browser response times. (Some browsers, such as Netscape, allow document navigation before images are fully retrieved, thus mitigating the problem of waiting for image retrieval.) Some authors maintain text-only and picture-only versions of documents, in part, to address this issue. (And in part to allow access by nonimage-supporting browsers such as Lynx.)

In both of these examples, supporting the desired functionality depends on the user's ability to dynamically reconfigure a document. An alternative to requiring document authors to maintain multiple versions of documents is to allow users to dynamically configure pages so that they can specify whether or not to include images [5]. In the absence of exit buttons, users should be allowed to enhance the document by adding such buttons.

This paper describes a system called Boomerang that supports dynamic reconfiguration of HTML documents. Boomerang allows users to control how information is presented. The two examples given above are only a tiny fraction of the power of reconfiguration. Users can (temporarily) add navigational buttons to pages, suppress images, combine information from multiple pages, redefine tags, and redesign poorly presented pages. But reconfiguration also helps document authors. It aids document reusability, allows individual users to be tracked, and creates a new class of hypertext documents, which we call *parameterized* documents.

This paper is organized as follows: The next section gives an overview of Boomerang and broadly discusses how Boomerang can reconfigure a page. That discussion is followed by a description of Boomerang's architecture. The heart of the architecture is a reconfiguration engine that can parse and rearrange a page. The engine is controlled by the user through a reconfiguration language, which is defined in the section "Boomerang Templates." The language borrows heavily from the sam editor [3]. Several examples are given in the section "Boomerang Examples" to demonstrate the utility of Boomerang. Finally, security issues, related work, and future work are presented.

Overview

In a typical browsing session, a user navigates from page to page. New pages are fetched through explicit user actions, e.g., a user activates a link, a submit button, or makes a "hotlist" jump.

Figure 1 shows the typical pattern of browser behavior. A user requests a page from a server, and the server returns the requested page.

Boomerang is an "intermediary" or "wrapper" script that sits between the browser and the server. When using Boomerang, a user has the same basic browsing behavior. However, when fetching new pages Boomerang serves as an intermediary agent between the user and a desired page as shown in Figure 2. Instead of directly requesting a new page, the user activates a link that starts Boomerang and instructs it to fetch the desired page. The user also tells Boomerang how to reconfigure the page. Boomerang "intercepts" the page fetch request, fetches the requested page, and reconfigures it according to user specification.

Architecture

Boomerang was initially designed to aid the construction and maintenance of a *forms document.* A forms document is a special kind of HTML document. Like an HTML document, a forms document contains a number of related *pages.* Each page is raw text interspersed with HTML tags. Unlike a standard HTML document, the pages in a forms document can also have one or more *forms.* A form consists of a number of data-entry

widgets that are manipulated by the user and then submitted to a form-handling *script*.

A form has a number of *variables*. There is one variable for each data-entry widget in the form, plus any number of "hidden" variables. Hidden variables are added by the form's author to communicate values to the script (e.g., to pass state information through a sequence of forms). When a form is submitted, the variables are formatted as a list of name/value pairs and passed as a string to the script.

A user may also invoke a script via a hypertext link. In such situations, input to the script is passed in a "query string." The query string also consists of name/value pairs.

As described in the Overview, Boomerang is an "intermediary" agent which sits between the user and the server. Boomerang is a script. A browser invokes Boomerang when a user activates the appropriate hypertext link or submit button. The browser passes to Boomerang (either via a query string or hidden variables) a URL and a comma-separated list of *page templates*. The URL is either the name of a page to fetch or a script to execute. A page template specifies how to reconfigure the page. The user may give either the name of a file containing the template or might pass the template itself. The page is successively reconfigured by each template in the list. Examples are given in the "Boomerang Examples" section.

Boomerang has three basic components: an Input Variable Handler, a Page Fetcher, and a Reconfiguration Engine. The components are shown in Figure 3. Each component is discussed below.

The Input Variable Handler parses the input to Boomerang. It strips off the URL and template and passes those values to the Page Fetcher, along with any input variables or query string (since the page to be fetched might be produced by a script). The Input Variable Handler also maps all HTML variables in the input to Boomerang variables. (Boomerang variables are described in the next section.)

The Page Fetcher fetches the desired page(s). If a page to fetch is produced by a script, Boomerang passes the appropriate input to the script, executes the script, and collects the output. Otherwise, Boomerang simply reads the page. As regards fetching pages, the Page Fetcher duplicates some server functionality, since servers usually fetch pages. This duplication could be avoided by tightly integrating the server with Boomerang, but such integration would require substantial server modification.

The fetched pages, HTML input variables, and page templates are passed (as Boomerang variables) to the Reconfiguration Engine. The Reconfiguration Engine is the heart of Boomerang. It dynamically reconfigures the pages as directed by the page template producing a final page, which is displayed by the browser.

Boomerang Templates

The Boomerang Reconfiguration Engine uses *page templates* to describe how a page should be reconfigured. A template has two parts: a *prologue* and a *body*. The prologue specifies rules that are used to instantiate Boomerang *variables*. The body is essentially raw text with embedded HTML tags just like a regular page, so it can be readily produced by HTML authors. In addition, the body can contain variable references that are expanded by Boomerang using the results of executing the prologue rules.

The rest of this section describes the form of prologue rules and the semantics of variable substitution in bodies. More detailed examples are given in the following section.

Variables

Boomerang supports a single flat name space for variable bindings. The values bound to variables are (possibly empty) lists of strings. For example, a variable `meanings` might denote the possible definitions of the word "boomerang" with the following list of three strings:

```
Australian thin curved hardwood missile
```

```
Recoil on originator
A WWW page reconfiguration tool
```

(Separate strings are shown on separate lines.)

Inputs to the reconfiguration process are supplied via predefined Boomerang variables.

- *Original page contents.* The complete contents of the original page is bound to the variable `page`, while the header and body of the page are bound to `head` and `body`, respectively.
- *Input variables.* Each input variable (i.e., data-entry variable or hidden variable in a form) is passed as the value of a Boomerang variable with the same name. Since an input variable can have only a single value, the corresponding Boomerang variables initially denote a singleton list. Note that Boomerang is always invoked with a `url` and a `template` variable. The former identifies the page to fetch and the latter the reconfiguration desired.
- *Input variable names.* The Boomerang variable `args` is bound to a list of name/value pairs of the input variables (i.e., the HTML variables in a form).

Rules

The prologue of a Boomerang template is a list of rules. Instantiation of the template consists of evaluating the rules in the order given. The result is a set of bindings to Boomerang variables that are used to substitute variable references in the body of the template.

The general form of a Boomerang rule is:

`set` *`var exp`*

and has the effect of evaluating the expression *exp* and binding the result to the variable *var*. Since it is common to simply process the original page, a rule of the form:

exp

has the same meaning as:

`set page` *`exp`*

Expressions

The operation of Boomerang expressions is based on the editing model of the sam editor[3]. The main difference is that Boomerang allows any text stored in a variable to be edited and the result to be bound to a variable, whereas sam commands operate on file buffers.

Boomerang expressions have the following general form:

source cmd

Each Boomerang/sam command operates on a region of text called *dot*. Commands can operate on the contents of dot and update its value (i.e., the region to which it refers). In Boomerang, the form of the source for an expression determines how dot is set initially for that expression and the form of the result of the expression. There are two forms of sources:

- *var.* Dot is successively set to each string in the list of strings bound to *var* and the command is run on each of them. The result of the expression is the list of strings consisting of the result of each of the command applications.
- `$` *var.* Dot is set to the single string formed by concatenating the strings in the list bound to *var* and the command is applied to it. The result of the expression is a list containing as elements the final value(s) of dot when the command finishes.

The value of the source variable is *not* modified by the evaluation of either type of expression, but it can be altered by naming the same variable as the target of an enclosing rule.

Table 1 (based on Table I in [3]) lists the Boomerang editing commands. The `a`, `c`, `d`, and `i` commands simply append to, change, delete, or insert after dot, respectively. The `<`, `>`, and `|` commands allow dot to be set by, passed to, or both passed to and set by a "safe" external script,

Table 1: Boomerang Editing Commands[1]

`a/text/`	Append text after dot
`c/text/`	Change dot to *text*
`d`	Delete dot
`g/regexp/ cmd`	If dot contains a match with *regexp*, execute {em cmd/} on dot
`i/text/`	Insert *text* before dot
`m address`	Move text in dot to after *address*
`s/regexp/text/`	Substitute *text* for each match of *regexp* in dot
`t address`	Copy text in dot to after *address*
`v/regexp/ cmd`	If dot does not contain a match with *regexp*, execute *cmd* on dot
`x/regexp/ cmd`	Execute *cmd* with dot set to each match of *regexp* in dot
`y/regexp/ cmd`	Execute *cmd* with dot set to the strings between adjacent matches of *regexp* in dot
`< prog args...`	Replace dot with the standard output of the UNIX program *prog* run with the given arguments
`> prog args...`	Send dot to the standard input of *prog*
`| prog args...`	Replace dot with the standard output of *prog* when given dot as standard input
`! prog args`	Run *prog*, ignoring any output

[1] Text can be arbitrary text possibly including newlines; regexp stands for a standard UNIX regular expression

respectively (see the section called "Security" for a discussion of safe external scripts).

The `s` command performs a text substitution on dot, allowing the usual Unix regular expression notations.

Most of the power of sam (and hence Boomerang) derives from the `x`, `y`, `g` and `v` commands. An `x` command looks through dot for matches with its regular expression argument. The command argument is repeatedly executed with dot set to each of the matches. For example, the expression

```
meanings x/WWW/ c/World Wide Web/
```

returns a list of strings the same as that bound to the variable `meanings` except that all occurrences of "WWW" have been changed to "World Wide Web." In contrast, the expression

```
$meanings x/WWW/ c/World Wide Web/
```

returns a list of "World Wide Webs," one for each "WWW" in the value bound to `meanings`. More usefully, the expression

```
meanings x/^/ i/<LI>/
```

turns the list of dictionary definitions into a list of HTML list items (`^` is a regular expression that matches the beginning of dot).

The `y` command is similar to `x` except that it executes the command argument on the strings *between* the matches. For example, the following rule deletes all text in the current page that is not an HTML tag.

```
y/<[^>]+>/ d
```

(The regular expression `[^>]+` matches any nonempty sequence of characters that does not contain a >.)

Similarly, the `g` and `v` constructs are conditional constructs. A `g` (`v`) command executes its command argument on dot if it matches (does not match) the regular expression. For example, the following expression adds boldface tags to every definition containing the word "WWW," not just to every word "WWW."

```
meanings g/WWW/ s/.*/<B></B>/
```

Table 2: Boomerang Addresses: in the + and - Forms, *a* Defaults to . and *b* Defaults to 1

•	Dot	a+b	Address b from the right end of a
n	Line n	a--b	Address b from the left end of a
/regexp/	First following match of regexp	a,b	Left end of a to the right end of b
-/regexp/	First previous match of regexp	a;b	Like a,b but sets dot after evaluating

(In an s command, & stands for the text matched by the regular expression.)

The definition of dot can be altered using *addresses* as supported by sam (summarized in Table 2 based on Table II of [3]). The most useful form of address is one that is relative to dot. For example, the address / *regexp*/ finds the next match with *regexp* after dot. Similarly, –/ *regexp*/ finds the first previous match before dot. (Addresses of this kind can be usefully used in conjunction with x commands.) If *regexp* is replaced by a number then these forms count by lines in the indicated direction. An address of the form *a,b* means from the beginning of the text denoted by *a* to the end of the text denoted by *b*.

Addresses enable complex editing tasks to be expressed concisely. For example, the following rule deletes the last item from each unnumbered list in the current page by finding the end of each list, searching backwards for the last item and deleting from there to the beginning of the line containing the end of the list.

```
x/<\/UL>/ -/<LI>/,-1 d
```

The m and t commands allow addresses to be used as targets for the text in dot to allow arbitrary text rearranging. For example, the following command copies all form submit buttons to the top of the form.

```
x/<INPUT TYPE="SUBMIT"/ .,/>/ t 0/
    <FORM>/
```

Template Bodies

Once all of the template rules have been executed, Boomerang instantiates the template body to produce the reconfigured page. The body can contain arbitrary text and will normally include HTML tags for markup. To enable variable values to be included in the final page, the body can also include text of the form *$var*, where *var* is a Boomerang variable name. During instantiation, Boomerang replaces variable references with the concatenation of the strings bound to the variable. For example, below we show a body that might be used to produce a page for inclusion in an online dictionary.

```
<TITLE>Definition of $word</TITLE>
<BODY>
$word has the following meanings:<BR>
<P>
<UL>
$meanings
</UL>
</BODY>
```

Boomerang Examples

In this section we give several examples of templates to demonstrate the power and utility of Boomerang. Each example addresses a standard reconfiguration problem faced by a typical user or author.

Suppress Images

This template finds all <IMG...> tags in a page and replaces them with the string "Image Replaced." (The lines starting with %% are comments.)

```
%% Prologue Suppress
x/<IMG[^>]*> c/Image Replaced/
%% Body
$page
```

Return to Point of Entry

The template given below adds a return to point of entry link to the bottom of a page. We assume that the link URL is passed to Boomerang in the returnlink variable (in a query string or as a hidden variable).

```
%% Prologue Return
%% Body
<HEAD>
$head
</HEAD>

<BODY>
$body
<P>
<A HREF="$returnlink"> Return to Point
    of Entry </A>
</BODY>
```

Access Counter

Access counters are popular page enhancements. An access counter is a count of the number of visitors to a page. In this example, we assume that the access counter is maintained by the script `access_count`, which updates and returns the count for a given URL. In this example, the URL is passed to `access_count` on the command line; alternatively, it could have been passed in standard input.

```
%% Prologue Counter
set count < access_count $url
%% Body
<HEAD>
$head
</HEAD>

<BODY>
This page has been visited $count times.
<P>
$body
</BODY>
```

Keep Boomerang in Control

This template reconfigures a page so that all links out of the page return control to Boomerang. We assume that the `template` variable specifies the template(s) to be executed on subsequent pages. The URL location and replacement in this example have been simplified to assume absolute links only.

```
%% Prologue Control
%% Find each link and make it a
%%  Boomerang call with the link
%%  passed in the
%%  url parameter
x/<A HREF="http:\/\/[^\/]+\// a/cgi-
    bin\/
    Boomerang?template=$template&url=/
%% Do the same for forms.  Find the
%%  action, skip the server, insert
%%  the call to Boomerang and hidden
%%  variables for the template and url
x/<FORM .* ACTION="http:\/\/[^\/]+\// a/
    cgi-bin\/Boomerang"> \
  <INPUT TYPE="hidden" NAME="template"
    VALUE="$template">\
<INPUT TYPE="hidden" NAME="url" VALUE="/
%% Body
$page
```

Track an Individual

Authors are sometimes interested in tracking individuals to determine browsing patterns for a set of pages. Individuals can be tracked by reconfiguring a page to place a unique identifier on each link and keeping Boomerang in control. When the individual moves to a new page, Boomerang is called instead and Boomerang calls the appropriate script to update the browsing database.

```
%% Prologue Track
%% Check to see if id exists, if not,
%%   then generate a new one
set id id g/^$/ < generate_unique_id
%% Update browser database to indicate
%%   where the individual is now
! update_browser_db $id $url
%% Transform the links out of the
%%   page to return control to Boomerang
%%   with tracking enabled
x/<A HREF="http:\/\/[^\/]+\// a/cgi-bin
    Boomerang?template=Track&url=/
%%  and do the same for forms.
x/<FORM .* ACTION="http:\//[^\/]+\// a/
    cgi-bin\/Boomerang"> \
<INPUT TYPE="hidden" NAME="template"
    VALUE="Track">
<INPUT TYPE="hidden" NAME="url" VALUE="/
%% Find each link and add the id to the
%%   end of the query string
x/<A HREF="[^"]*/ a/&id=$id/
%% Do the same for forms, but add it as
%%   a hidden variable just before
%%   the end of the form
x/<\/FORM>/ i/<INPUT TYPE="hidden"
    NAME="id" VALUE="$id">/
%% Body
$page
```

Template Composition

Boomerang actions can be combined by specifying multiple templates. The templates are applied one at a time from the beginning to the ending of the list. The following link combines most of the previous reconfigurations; that is, it suppresses images, adds a return to point of entry link, displays an access counter: We give the link below.

```
<A HREF="http://server/cgi-bin/
    Boomerang?template=Control,
    Counter,Return,\ Suppress&url=base.
    html&returnlink=home.html">
```

Page Decompression and Decryption

In general, the fetched page does not have to even remotely resemble HTML; it could for instance be a LaTeX document. In this example, we assume that the page is encrypted and then compressed, with .Z or .gz compression. The decryption key must be passed to Boomerang in the `key` variable. We assume that `uncompress` and `gunzip` are in the list of safe commands as described in the section "Security."

```
%% Prologue
set page $url g/.gz/ < gunzip $url |
    decrypt $key
set page $url g/.Z/ < uncompress $url |
    decrypt $key
%% Body
$page
```

A Parameterized Page

A parameterized page is a page that mentions one or more variables. When the page is retrieved, the variables are replaced by their values (the values are passed in the query string or as HTML variables). Boomerang trivially supports parameterized pages. For example, the following "Welcome" page is parameterized by the `name` variable.

```
%% Prologue
%% Body
<HEAD>
<TITLE> Welcome </TITLE>
</HEAD>
<BODY>
Good morning $name
</BODY>
```

Additional Benefits

The previous examples illustrate some of Boomerang's power. Boomerang also eases the burden of writing a sequence of WWW forms and integrating that sequence with other such sequences. One problem faced by forms document authors is that the HTML is "hidden" in scripts. Most often, changes to forms documents are to the layout of a page, i.e., to the HTML rather than to the processing of a form. Such changes are quite simple to make in HTML, but become more involved if, as is commonly the case in a forms document, the page is dynamically generated during execution of a script. Even knowledgeable HTML authors cannot make simple changes to the page layout (such as the addition of a link) since they often do not understand the scripting language or script. Only authors that know HTML, the scripting language, and the structure of the script are able to make changes to a forms document. Moreover, hiding pages in scripts makes it difficult to integrate pages designed by WYSIWYG HTML editors.

In contrast, Boomerang favors HTML over a scripting language. Boomerang presents a standardized notation for processing WWW forms, so it can be used by expert programmers to better code and document their scripts. Moreover, by using Boomerang to describe the essential HTML interaction, commonalities between scripts in a variety of languages can be identified, supporting script reuse and the mix of form-handlers written in disparate languages.

Boomerang also can be used to easily integrate output from WYSIWYG editors with scripts.

Security

Security is an important concern on any server. In Boomerang, a user can create a template that executes scripts on the server. This is a powerful

tool for users since it allows a limited form of CGI-bin programming without actually creating a script to reside in CGI-bin. Conceptually, this does not create a new security hole since servers should already ensure that each CGI-bin script, executed independently, is a "safe" script, regardless of the input it receives. However, since Boomerang makes it easier for users to execute scripts, it potentially permits outsiders to quickly probe for "unsafe" scripts. To patch this potential security hole, Boomerang can only execute scripts that are on a list of safe scripts. The list is created and maintained by the server administrators.

Related Work

Sato describes a LISP-based system for dynamically rewriting HTML [4]. In Sato's system, the author must first write a page (or simple text file, with tags of the author's choosing) and a LISP program that converts that file to HTML. When a client requests the document the server instead returns the LISP program. The LISP program is then executed on the client's machine and converts the text file (fetched from the server) into HTML. The rewriting is in cooperation with the author since the author supplies the LISP program. In contrast, Boomerang does not depend on author cooperation. Users reconfigure documents without involving the author at any stage. Boomerang also has a different implementation strategy. Sato uses MIME types and the Content meta resource in a page header to implement his system. Boomerang uses the Common Gateway Interface. Consequently, Boomerang has access to the variables in a form and supports parameterized documents. Boomerang shares the view espoused by Gleeson and Westaway [1], among others, that forms documents are important to the future of the WWW. Gleeson and Westaway lament the lack of tools to aid the design and maintenance of forms documents, Boomerang is one such tool. Sato also noted the security risk in downloading LISP code from a server to be immediately executed on a client's machine. Boomerang does not suffer from this security problem.

Sperberg-McQueen and Goldstein advocate that full SGML intelligence be added to browsers. Such intelligence would allow browsers to incorporate "style files" to interpret SGML tags in a document. So, for example, a musician could add a "`<HALF-NOTE C#>`" tag to a document. The tag's translation to HTML would be loaded into a browser by a style file, and different style files might provide different translations. Boomerang differs from the SGML solution insofar as the style file approach still depends on the author to provide the appropriate tags. Also style files do not support parameterized documents or allow document rearrangement, such as the addition of "exit" buttons. In some sense the SGML solution is attacking a different problem, that of allowing authors to work in a special-purpose notation.

Conclusion

Currently HTML page presentation is a static process. Once a page is written, the presentation of that page is fixed. The presentation may vary among browsers (e.g., a page may look different on Netscape and Lynx), but a single browser will always display a page the same way. Static page display limits both users and authors. Boomerang makes page presentation a dynamic process.

Boomerang is a Common Gateway Interface script that "intercepts" a page fetch request, fetches the desired page, and reconfigures it. The user supplies a template (or a list of templates) that informs Boomerang how to reconfigure a page. Each template has a prologue and a body. The prologue is a sequence of rules written in a regular expression-based pattern matching language.

The language supports quick matching and replacement of patterns in a page. The body of a template is raw HTML mixed with Boomerang variables. Variables are used to temporarily hold the results of rules.

After the rules have been processed the body of the template is instantiated by substituting the value for each variable.

Boomerang helps users because users can reshape and enhance pages as desired. User reconfiguration demands no special preparations by page authors. But Boomerang also helps authors. It eliminates a need to support multiple versions of pages. Moreover, authors who are unable to program scripts can still write dynamically generated pages using parameterized pages.

Boomerang is currently being implemented. More information and release versions of Boomerang are available at *http://www.cs.jcu.edu.au/ftp/pub/research/boomerang/welcome.html*. Future extensions include support for processing several pages simultaneously, transparent document migration, and parallel rule execution. ■

References

1. M. Gleeson and T. Westaway. "Beyond Hypertext: Using the WWW for Interactive Applications," in *AusWeb95—The First Australian World Wide Web Conference*, Ballina, New South Wales, Australia, April 1995.
2. D. Nicol, C. Smeaton, and A. F. Slater, "Footsteps: Trail-blazing the Web," in *The Third International Conference on the World Wide Web*, Darmstadt, Germany, April 1995.
3. Rob Pike. "The text editor sam," *Software—Practice & Experience*, 17(11):813--845, 1987.
4. S. Sato. "Dynamic rewriting of HTML documents," in *The First International Conference on the World Wide Web*, Geneva, Switzerland, May 1994. CERN.
5. C. M. Sperberg-McQueen and R. F. Goldstein. "HTML to the Max: A Manifesto for Adding SGML Inetlligence to the World Wide Web," in *The Third International Conference on the World Wide Web*, Darmstadt, Germany, April 1995.

About the Authors

Curtis E. Dyreson
[*http://www.cs.jcu.edu.au/~curtis*]
Department of Computer Science
James Cook University
curtis@cs.jcu.edu.au

Anthony M. Sloane
[*http://www.cs.jcu.edu.au/~tony*]
Department of Computer Science
James Cook University
tony@cs.jcu.edu.au