Intensional HTML

by

Taner Yildirim
B.Sc., Bilkent University, 1994

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. W. W. Wadge, Supervisor (Dept. of Computer Science)

[redacted]

Dr. M. Levy, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. B. Freeman-Benson, Outside Member (Adjunct, Dept. of Comp. Sci.)

[redacted]

Dr. P. Driessen, External Examiner (Dept. of Elec. & Comp. Engineering)

Supervisor: Dr. W. W. Wadge

Abstract

Intensional HTML (IHTML) is a high-level authoring language for the World Wide Web that makes it practical to specify pages and sites that exist in many different versions or variants using standard client and server software.

Each IHTML page defines an intension - an indexed family of actual (extensional) HTML pages which varies over a multi-dimensional author-specified version space. The version space is partially ordered by a refinement/specialization ordering. For example, *platform:macintosh* can be refined to *platform:macintosh+language:french* or to *platform:macintosh%68k* and the latter two both refine to *platform:macintosh%68k + language:french.*

Authors can create multiple labeled versions of the IHTML source for a given page. Requests from clients specify both a page and a version, and the server-side software selects the appropriate source page and uses it to generate the requested actual HTML page. The clients request versions of pages simply by clicking on intensional/transversion links, which look like standard hypertext links, provided by IHTML authors. From the client point of view, a multi-versioned site is nothing but a huge site with static clones created for each existing version of all pages.

Authors do not, however, have to provide separate sources for each version. If the server-side software can not find a source page with the exact version requested, it uses the page whose label most closely approximates the requested version. In other words, it treats the refinement ordering as an (reverse) inheritance ordering. Thus different versions can share source, and authors can write generic, multi-version code.

Examiners:

[redacted]

Dr. W. W. Wadge, Supervisor (Dept. of Computer Science)

[redacted]

Dr. M. Levy, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. B. Freeman-Benson, Outside Member (Adjunct, Dept. Computer Science)

[redacted]

Dr. P. Driessen, External Examiner (Dept. of Elec. & Computer Engineering)

Table of Contents

List of Figures

Acknowledgements

I would like to thank everyone who supported me and offered me guidance throughout this research. In particular, the assistance and advice by Dr. W. W. Wadge, who also suggested the topic of this thesis, was much appreciated. I wish to thank Gordon Brown and Monica Schraefel for their creative suggestions and proof reading this thesis.

I am also grateful for the financial support from the University of Victoria and Dr. W. W. Wadge.

Finally, I would like to thank my family and friends for all their encouragement.

To my family and friends.

Chapter 1

Introduction

In this thesis I describe the design and implementation of Intensional HTML (also referred as IHTML), a high-level authoring language for the World Wide Web that provides an intensional solution to the problems encountered in producing, viewing and managing Web documents that exist in multiple versions. Specifically, I am interested in representing a large number of versions of a Web page by a small number of generic source files, so that the components that make up the page do not necessarily have to have separate source files for each possible version.

To demonstrate this intensional solution, I also implemented a prototype site using IHTML, which looks like a conventional site consisting of a huge number of Web documents whereas in fact, the number of underlying source files is far less.

In IHTML, links to other pages, graphics images, java applets and server-side includes have *intensional* interpretations. In other words, these elements are context dependent, and they point to different pages, images, class and include files depending on the current version of the page they reside in. Furthermore, if any component (an image, a class or an include file) of a document does not exist in the exact version requested, a refinement ordering is used to approximate the most relevant version among the existing ones.

1.1 Problem Definition

Computer technology advances very rapidly and creates new needs and requirements. To satisfy these requirements, computer software almost always is produced in groups of versions, each version specially adapted to a particular use. These groups of versions are very similar to each other, yet different in certain aspects. Hence, the easiest approach to creating new versions of software is to copy and modify the original source.

The rapid growth of the Web increases the pressure on Web site designers to provide multiple versioned sites, for a number of reasons:

- The Web is international, and most of the sites must be available in many different languages.
- Users have a wide range of bandwidths, so that some appreciate high quality graphics while others prefer purely textual pages.
- Different browsers have different capabilities and some common browsers like Netscape's Navigator and Microsoft's Internet Explorer have their own extensions to HTML such as frames, Java applets, scripts and ActiveX controls, which can cause formatting or functionality problems if used with the wrong browser.
- Commercial sites offer more material to paying subscribers while others may want certain information hidden from non-members.
- Some sites would offer local information (e.g. weather reports) which would differ in different parts of the world.
- Site designers might want to offer sites which are customizable to take personal preferences (fonts, backgrounds, text and link colors) into account.

The easiest way to produce a variant of a Web site is to make a copy of the existing HTML source and modify it. Unfortunately, the cloning (copying/modifying) approach to version creation can produce severe problems. As the number of families of versions increase, maintenance costs increase as well. Any change made to the original source has

to be propagated in the sources for all the versions which used the original source. This approach will eventually result in a large family of clones, almost impossible to change in a uniform way.

The obvious solution is to ensure that the different members of the family share the same source (and not copies thereof) so that necessary changes are made only once and need not be propagated. This not only reduces maintenance costs but also increases interface consistency at the same time.

Source sharing in conventional HTML is done by hyperlinks and image elements where two or more different pages are linked to the shared page, or where two or more pages include the shared graphics image. However, this is a very crude form of sharing when the shared object is not self-contained, because hyperlinks, for example, are essentially pointers, and sharing a pointer results in sharing everything the pointer itself points to.

To illustrate the problem, consider the task of supporting English and French versions of a simple slide show. The slide show consists of sequences of pages of text and/or graphics, each linked to the next page in sequence. Obviously, the author has to create separate English and French versions of pages with text on them. But the author also has to make clones of any pages that have only images, even though the French and English versions will appear identical on the screen (assuming that the images do not have any text). The problem is that the English version of the page in question must be linked to the rest of the English version of the show, while the French version of the page must be linked to the rest of the French version of the show. Two separate source files are required, which will only differ in the link to the next page.

1.2 The IHTML Solution

In this thesis I present an intensional variant of HTML, IHTML, a new Web authoring language which incorporates an inheritance based approach to hypertext versioning. IHTML allows authors to define, with a single source file, a whole indexed family of HTML variants based on the file in question. These variants are generated on demand, then discarded after use. In a sense, IHTML automates the cloning process, and eliminates the maintenance problem by ensuring that the clones are short lived.

IHTML is intensional because IHTML source has both *intensional* and *extensional* meanings. The intension is the whole indexed *family* of HTML pages whereas the extensions are the different *individual*, automatically generated HTML pages.

IHTML authors can define a whole family of HTML pages by only one generic source file, which is used as a template by the server-side software to create all the existing versions of the page in question. These generic source files define the structure and content of an intensional page by specifying the order and source of the components that make up the page. Usually there are multiple source files for these components in which case the page exists in multiple versions. It is also possible to provide multiple source files for the whole page if the structure and content of the page is completely different in each version.

The server-side software uses a coding scheme for version expressions, which are embedded in source file labels. The server-side software accepts requests for particular versions of particular pages, and generates the actual HTML from the appropriate IHTML source file(s) by combining the corresponding versions of the components of the page.

IHTML authors do not, however, have to provide separate source files for every possible version. The IHTML index space ("version space") is partially ordered by a refinement relation, and the source for a more refined version is by default inherited from the

less refined (more generic versions).

When the server-side software receives a request for a particular version of a particular page (or part thereof), it looks for a source file labeled with the requested version. If there is no such source, it looks for a source file whose label most closely approximates the requested version. More refined versions can therefore by default inherit source from more generic ones, and a relatively small number of source files can define a very large family of pages.

For example, in the case of the slide show, the author could name the pages *slide1*, *slide2*, *slide3*... and provide two source files for each of these pages, one in English (E) and one in French (F) as shown in Figure 1.1.

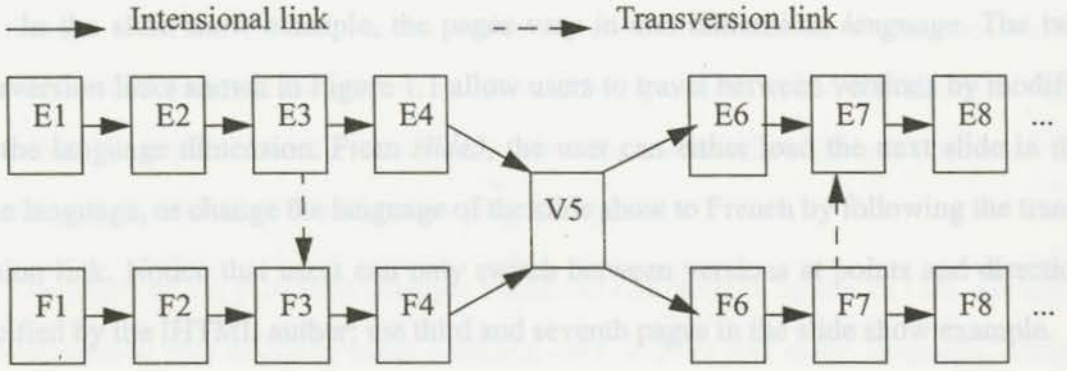

Figure 1.1 Intensional and transversion links of a slide show.

Suppose that the fifth slide is purely graphical. The IHTML author can provide a *single* source file, V5, labeled as the standard (so-called *vanilla*) version. When a request comes for, say, the French version of *slide5*, the server-side software first looks for a source file for that page labeled as the *French* version, F5. When it finds none, uses the more general standard version, V5. Requests for the English version are similarly referred

to the single standard source. Note that there is only *one* intensional link to the next page in *slide5*. When the current version is French, this link points to F6 and when the current version is English it points to E6.

The English and French versions of the site coexist as sort of parallel universes and intensional links are used to travel in the same universe, from E1 to E2 and from E5 (approximated by V5) to E6.

To allow users travel between different versions of a site, IHTML is equipped with *transversion links* which serve as context switching operators. Transversion links are interpreted as leading from a given version of the source page to the modified version of the target page - the modifications resulting from altering one or more of the *dimension* values that make up the version.

In the slide show example, the pages vary in one dimension, *language*. The two transversion links shown in Figure 1.1 allow users to travel between versions by modifying the language dimension. From *slide3*, the user can either load the next slide in the same language, or change the language of the slide show to French by following the transversion link. Notice that users can only switch between versions at points and direction specified by the IHTML author: the third and seventh pages in the slide show example.

Similar to intensional links, IHTML image elements and Java applets represent families of image elements and Java applets, each including or running the corresponding version of the image or the Java program they point to. Transversion images (Java applets) include (run) the modified version of the target image (Java program), however the current version of the site remains unmodified.

In addition to links, images and Java applets, IHTML also supports *server-side includes* which can also be either intensional or transversion. An IHTML include incorporates (by copying) the contents of the corresponding (or modified) version of the source

file they point to. For example, each page in the slide show might include a *header* at the top and a *footer* at the bottom, and the English version of a page will include the English versions of these files. One could even include the French version of the header in the English version of a page.

The include facility is very important for IHTML because it allows authors to break the source files into smaller components, resulting in higher modularity. In this way, authors can isolate the parts of the whole page that actually vary, and write more generic source for the parts (such as headers and footers) that do not.

IHTML also supports unversioned webware (HTML files, images, Java applets and other internet sources) by treating the conventional HTML elements as extensional elements.

It might seem that IHTML requires its own version of the server and client software. However, I used standard techniques to allow existing clients to browse multiple versioned Web sites published by existing servers. The server-side software performs all needed functions and there is no browser plug-in required.

The pages of the prototype multiple versioned site (created by the IHTML features described above) vary in *background*, *category*, *date*, *display*, *link color*, *language*, *order*, and *text color* dimensions. The site consists of thirteen intensional pages each available in all versions - a total of more than 4,5 million virtual HTML pages supported by 192 IHTML files. In other words, 107K bytes of IHTML source supports a virtual site which would correspond to 12M bytes of cloned HTML.

In summary, IHTML extends conventional HTML by incorporating an inheritance based approach in order to allow Web site developers to write more generic code that applies to many versions of a page. The resulting sites are much more modular, consistent in interface and much easier to maintain. Users can view different versions of a site simply

by following links, without having to type in complex expressions. The dimensions used in the IHTML version space are customizable by authors to suit the needs of the site in question, hence IHTML can be applied to any indexable family of pages.

1.3 Thesis Overview

In Chapter 2, I review some necessary background on HTML, CGI (Common Gateway Interface) programming, intensional logic and intensional programming, and the version algebra which forms the basis of the version control system used in IHTML.

In Chapter 3, I provide an overview of IHTML and describe its distinguishing features: intensional and transversion links, images, applets and include files. In this chapter I also present the extended version space used in IHTML and the object-oriented and data-flow aspects of this new language.

In Chapter 4, I describe the current implementation of IHTML, which includes the data structures used, the approximation algorithm, the coding scheme used for version expressions, the scripts that help authors to create multiple versioned sites, and the CGI script to do IHTML to HTML transformation. In this chapter I also describe the implementation of my sample site and its comparison to the HTML based alternative.

In Chapter 5, I discuss the conclusions and possible extensions of my work.

Chapter 2

Background

2.1 HTML and CGI Programming

IHTML extends HTML by introducing new elements that have intensional interpretations. These intensional elements are implemented using a CGI script which intercepts the requests from clients, and executes a server-side program to find the most relevant IHTML source file. The CGI script is also used to parse IHTML source files and create conventional HTML files on demand. In the following sections, I introduce the basic concepts of HTML and CGI programming related to IHTML.

2.1.1 HTML

HTML is a Web authoring language designed to specify the logical organization of hypertext on the Web. HTML contains commands, called *tags*, to mark text as headings, paragraphs, lists and so on. It also has tags for including images, applets within documents and tags for including hypertext links which connect the document they reside in to other documents or internet resources.

HTML was not designed using What You See Is What You Get (WYSIWYG) model. Instead, HTML gives users a way to mark structural parts of a document without having to worry about the appearance of the document. The programs that render HTML

documents (Web browsers) use these mark-up tags to display HTML documents in a readable form. This organization allows for a separation of a document's structural specification in the HTML code from its formatted appearance in a Web browser.

The basic difference between an HTML document and a simple text document lies in the mark-up *tags* of HTML, which are always enclosed in angle-brackets (< >). Tags typically occur in begin-end pairs which are in *<tag> ... </tag>* form, where the *<tag>* indicates the beginning of a tag pair, and the *</tag>* indicates the end. These pairs define *containers* which are also referred as *elements*. Any content within a container has the rules of that container applied to it. One can argue that an HTML document is composed of these elements.

IHTML extends four elements of conventional HTML, namely *anchor*, *image*, *applet* and *body* and adds a new element that serves as an intensional equivalent of conventional server-side includes. In the following sections, I briefly describe the conventional usage of these elements and their tags.

Throughout the rest of this thesis, I will use the following conventions to formalize the syntax of HTML and IHTML elements:

- *[a | b]* represents either *a* or *b*,
- *[a]?* represents zero or one occurrence of *a*,
- *[a]+* represents one or more occurrences of *a*,
- *[a]** represents zero or more occurrences of *a*.

2.1.1.1 Anchors

The main feature of the Web is that documents can be linked to each other, or to other types of files such as movies or sound clips, through the use of hyperlinks. These links allow authors to link documents together in intuitive ways, as opposed to traditional linear texts. They are also referred to as *anchors* and have the following format:

```
<a href="URL_address">highlighted_text</a>
```

where `URL_address` is a Uniform Resource Locator.[1] The area between the beginning `<a>` and ending `</a>` tags becomes a *hot* part of the text and is highlighted or underlined by most browsers. Clicking on the `highlighted_text` transports the users to the `URL_address`, which is usually another HTML document.

The `name` attribute of the anchor element can be used to specify distinct targets of hypertext references. For example:

```
<a name="fragment_identifier">hyperlink_target</a>
```

marks the string as a possible target of a hyperlink. In URLs, these named locations are called *fragments* and can be referenced by appending the `fragment_identifier` to the document URL, separated by the "#" sign. The `hyperlink_target` is assumed to be within the *same* document if the document URL is replaced by the `fragment_identifier` (prepended by #):

```
<a href=#fragment_identifier>highlighted_text</a>.
```

2.1.1.2 Images

Besides hyperlinks, the other great advantage of the Web is the ability to integrate graphic images into a document. Some would argue that this represents one of the greatest strengths of the Web. Graphic images are certainly used as heavily as hyperlinks. They can be placed almost anywhere within the body of the document and represent most of the data which is transferred. Images are placed in Web documents using `img` tags, which have the following format:

```
<img src="graphics_file" [alt="..."]? [align="..."]? [ismap]?>.
```

Notice that the `graphics_file` can be a full URL address, which can be totally different from that of the HTML document. Visually speaking, graphic images are part of a Web document, but in reality graphic images are actually all separate files which get

included in the HTML document that refers to them. Furthermore, Web browsers have to make additional connections to the server to obtain the required images of a document. Thus, a single document containing ten images requires eleven distinct connections to complete the document.

2.1.1.3 Applets

An applet is a small Java program that can be embedded in another application. Java applets provide interactive, executable content on a Web page. A Java applet is included in a Web page with the `applet` tag, which has the following syntax:

```
<applet [codebase="..."]? code="..." width="..." height="..." >
    [param name="..." value="..."]* [alternate-html] </applet>
```

where:

- `codebase`'s value specifies the base URL (absolute or relative) of the applet to be displayed. This should be a directory, not the Java class file itself. Its default value is the URL of the current document.
- `code`'s value specifies the compiled Java class file. It must be relative to the `codebase` if that attribute is specified, or relative to the current document's URL.
- `width`'s value specifies the initial width in pixels.
- `height`'s value specifies the initial height in pixels.
- `param` tag with its `name` and `value` attributes, specifies a named parameter and a string value that are passed to the applet. This tag is used to customize applets.

Browsers which do not support Java and do not understand the applet tag ignore this tag and displays any `alternate-html` specified.

2.1.1.4 Body Elements

HTML documents are separated into two major sections; `head` and `body`. The head elements contain information about documents, whereas body elements contain the actual content of documents. All the attributes that describe how an HTML document should look in the browser window are enclosed in the opening tag of body elements:

```
<body [[background | bgcolor]="..."]? [text="..."]? [link="..."]?
      [vlink="..."]? [alink="..."]? > ... </body>
```

where:

- `background`'s value is the URL of the graphic image that will be tiled as the background of the page.
- `bgcolor` allows the author to specify a solid background color. The color is specified using a hexadecimal color code of the form; #RRGGBB, where RR, GG, BB are the hexadecimal digits specifying the Red, Green, and Blue values of the color.
- `text` specifies the color of the document's text.
- `link` specifies the color of the document's hotspots (hyperlinks).
- `vlink` specifies the color of the visited links.
- `alink` specifies the color of the active link (the color that it appears while the user is selecting it).

These attributes are actually extensions understood by the most commonly used Web browsers: Netscape's Navigator and Microsoft's Internet Explorer.

I will discuss server-side includes after I present the necessary background for CGI programming.

2.1.2 Gateway Programming

The Common Gateway Interface (CGI) is a standard for interfacing external applications with information servers such as HTTP. A plain HTML document that the Web server daemon (e.g. HTTPD) retrieves is static, which means it exists in a constant state: a text file that doesn't change. CGI programs extend this static model by allowing the server to provide different documents depending on the client's request. CGI programs can also create new documents on demand. The CGI specification allows the Web server to communicate with other programs running on the server.

> With CGI, the Web server can call up a program, while passing user-specific data to the program (such as what host the user is connecting from, or input the user has supplied using HTML form syntax). The program then processes that data and the server passes the program's response back to the Web browser.[2]

A CGI program is basically the equivalent of letting the world run a program on your server. A schematic overview of data flow in CGI programming is shown in Figure 2.1. [3]

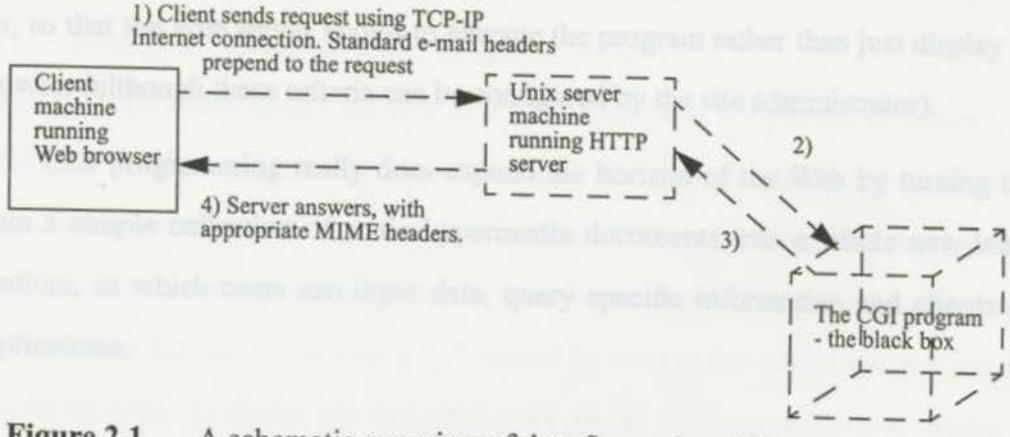

Figure 2.1 A schematic overview of data flow using CGI.

Once the CGI program starts running, it can either create and output a new document, or provide the URL to an existing one. On Unix, programs send their output to standard output as a data stream. The data stream consists of two parts. The first part is either a full or partial HTTP header that (at minimum) describes what format the returned data is in (e.g., HTML, plain text, GIF, etc.). A blank line signifies the end of the header section. The second part is the body, which contains the data conforming to the format type reflected in the header. The body is not modified or interpreted by the server in any way.

A CGI program can be written in any language that is capable of understanding standard input, output, and environment variables, although certain languages are more suited for CGI programming than others. Most CGI applications involve manipulating text some way or another, hence inherent pattern matching is very important. The ability to interface with other software and the ease of access to environmental variables are also among the important features of a language to be popular in CGI programming. Some of the most used languages for CGI programming include C/C++, C Shell, Frontier (AppleScript), Perl, Tcl, and Visual Basic.

CGI programs usually reside in a special directory (*cgi-bin*) and have the extension *cgi*, so that the Web server knows to execute the program rather than just display it to the browser (although these criteria can be configured by the site administrator).

CGI programming really does expand the horizon of the Web by turning the Web from a simple collection of static hypermedia documents into a whole new interactive medium, in which users can input data, query specific information and clients can run applications.

2.1.3 Server-side Includes

Server-side includes (SSI) are directives which can be placed into HTML documents to execute other programs or output data such as environment variables, the current date, the file's last modification date, and the size or last modification of other files. Unfortunately, these directives are not supported by all servers; the servers from NCSA and Netscape can handle server-side includes whereas the CERN server can not.

Server-side includes are not really CGI, however they can be used for incorporating CGI-like information, as well as output from CGI programs, into HTML documents on the Web. Server-side includes make use of special extensions to HTML tagging.[4]

Upon request, an SSI-enabled server parses the specified document and returns the evaluated document. The server does not automatically parse all files looking for SSI directives. Depending on the configuration of the server, only the files with the SSI extension, usually *shtml*, are searched for directives. The parsing represents a performance hit that the client must suffer. The advantage of server-side includes is that the included information is included on the fly at request time.

The standard SSI directive (that includes a file) has the following format:

```
<!--#include virtual="..." file="..." -->.
```

The SSI directive will insert the text of a document into the parsed document. Any included file is subject to the usual access control. The `virtual` tag gives a virtual path to a document on the server. Only normal files (not CGI scripts) can be accessed in this fashion. However, another parsed document can be accessed. The `file` tag gives a path name relative to the current directory. "../" cannot be used in this pathname, nor can absolute paths be used. As above, you can send other parsed documents, but you cannot send CGI scripts.

The include directives can be used to increase the reusability of HTML code. For

example, all the pages in a site could include a common footer file containing information common to all pages: name of the author/corporation, copyright information etc.

2.2 Intensional Programming and The WWW

As implied by its name IHTML is an intensional language. In this section I introduce the basic concepts of intensional logic and intensional programming and discuss the relationship between the World Wide Web and intensional programming.

2.2.1 Intensional Logic and Intensional Programming

Intensional logic is a mathematical formal system for describing entities whose value depends on implicit contexts.[5] For example an HTML page titled as "course listings for the present term" denotes a page containing a list of course names. The contents of this page depends on a number of parameters; department, present date, level (undergraduate / graduate) and so on. The contents of the page depends on *all* of these parameters and it is impossible to conclude anything about the list of courses unless the values of all the parameters are known, because the contents of the page will change every term for all levels in all departments. In fact, the quoted expression can be expressed as a mathematical expression of the form *course_list(d, t, l)*; where department is represented by *d*, term by *t* and level by *l*.

To evaluate context sensitive expressions like these, a distinction has to be made between what is called the *extension* and the *intension* of expressions.[6] The extension of an expression is the value in a given context. For example, course_list(CS, "1996 spring", "U") would be the extensional value of the quoted expression which would denote the undergraduate course listings of the computer science department for 1996 spring term. Natural language expressions are inherently context dependent and can have different extensions (values or meanings) in different contexts. The *intension* of an expression rep-

resents all the possible extensions and also defines the relationship between these extensions and their contexts. In other words, the intension of an expression can be regarded as a mapping function which assigns a value to the expression in a given context.

Intensional programming is a programming paradigm that is based on intensional logic. The main characteristic of intensional languages is that they have context switching operators, called *intensional operators*. These operators can combine the values from different contexts without explicit context manipulation.[7]

Eduction is the standard operational model for intensional programming. It is the mode of operation for dynamic demand-driven dataflow.[8] In eduction, the requested value of an object in a particular context is not computed unless it is not already available. During the computation the values of other variables with different contexts might be needed, which can be requested in parallel.

2.2.2 Possible Worlds

The Web consists of a family of pages indexed by their URLs. The intensional operators of the Web are the *hyperlinks*, since they switch the context to the URL of the page they link to.

In addition, the Web model is nothing but *eduction* - pages are only retrieved if they are requested. Furthermore, if graphics links must be followed to complete a page, separate connections must be made to (possibly) different remote hosts to retrieve the images. Also in case of server-side includes, separate connections must be made to retrieve the contents of the include files.

Hence, the intensional programming community considers the Web as the first large-scale experiment in intensional programming and eduction.[9]

However, new intensional operators can be added to the Web to make it more useful.

Having a family of systems (in the case of the Web, a family of pages) requires a family of implementations (a family of HTML files). Yet there is no need to produce separate implementations, because these systems are already members of a family, and have strong family resemblances. Instead, a family of implementations can be build in which related members share code.[10]

Furthermore, this family of Web pages can be considered as versions of a single Web page. Therefore a version control system which allows code sharing between versions is required. In the following sections I give a brief overview of conventional version control systems and introduce an innovative approach that is suitable for intensional systems.

2.3 Version Control

Software systems, like most other systems, evolve. Requirements, specifications, and hardware platforms change; optimizations are made; different versions are created to suit different needs. These different versions can be regarded as families of systems, all very similar, yet different. Handling these different versions of a large system is not an easy task, as each of its components evolve independently. Hence version control systems are developed.

Version control systems solve some of the problems of dealing with this evolution. Among the most common version control systems are SCCS [11] and RCS [12, 13, 14], which keep track of changes made to a file by different programmers and save storage by keeping a single copy for each file and a family of changes made.

All version control systems have their own languages to express the possible versions that a file, a module or a system may exist in. For example, SCCS and RCS use strings of numbers, such as 1.2.1.1, to describe the possible versions that a file may take.

Most of the version control systems use a tree structure to represent the possible ver-

sions that a system may have. Yet, when two or more versions of a module are combined to produce a new version, merging information is lost. For example, a module can exist in French and graphics versions and there might be a need for the graphics version of the module in French. French and graphics versions are represented as branches of the tree and when a graphics version in French is created, it is merged into the trunk of the tree (main version). However, the tree structure cannot model how or even the fact that the merge took place.

Another problem with version control systems that arises is the lack of correspondence between the same versions of different components (modules or parts that form the whole software). The versions of a component have only local significance, and this makes it difficult to talk about versions of complete systems. The users can use different versions of different components to build any desired combination. However, this freedom brings the responsibility of deciding which of the huge number of possible combinations result in a working system.

In J. A. Plaice and W. W. Wadge's approach[15], however, version labels (which are not necessarily numbers) are intended to have a global, uniform significance. Thus the graphics version of a component is meant to be combined with the graphics version of the rest of the components of the system. This approach makes it possible to talk of versions of the *complete system.* Any version of the complete system can be formed, by choosing the corresponding versions of the individual components. Suppose, for example, that every page of a Web site exists in French version, too. Then the French version of the site is formed by linking the French versions of all the pages to each other.

In general, all the pages do not have to exist in the French version (purely graphics pages do not have to exist in a French version). It may be possible to create the French version of the whole site by altering only the textual pages, which will have French versions. Therefore, to create the French version of a Web site, the French version of each

page is used, if it exists; otherwise the ordinary "vanilla" version is used instead.

Plaice and Wadge generalize this approach by defining a partially ordered algebra of version labels:

> The partial order is the refinement relation: $V \subseteq W$, read as "V is refined by W," or "V is relevant to W,", which means (informally) that W is the result of further developing version V. The basic principle is that in configuring version W of a system, we can use version V of a component as long as that component does not exist in version V' with $V \subseteq V' \subseteq W$, where $V' \neq W$.

This refinement ordering is used to form versions of complete systems. The desired version of the complete system is formed by combining the most relevant versions of each component.

In the following sections, I describe the version algebra that is the basis of the version space of IHTML. The simplest possible algebra would allow only one version, named by the empty string, which is called the vanilla version and represented by ε.

2.3.1 Project History

The simplest and most common versions of a component are those that represent the successive stages in the development process. In general, these versions are represented by numeric subsequences: 1.1, 1.3, or 3.4.1.

In Plaice and Wadge's version space, numeric versions can only build one branch. The initial set of possible versions can be described by the following grammar:

V ::= ε | N

N ::= n | N.n

where n is a nonnegative integer and "." is the *sub-number* operator. The *refinement order*, written $\subseteq$, indicates how one version is derived from others:

$$\varepsilon \subseteq V$$

$$\frac{(V \subseteq V') \, and \, (V' \subseteq V'')}{V \subseteq V''}$$

where $\frac{a}{b}$ means *a therefore b*. For the current set of versions, intuitive dictionary order is used:

$$\frac{n \leq m}{n \subseteq m}$$

$$N \subseteq \mathrm{N.n}$$

$$\frac{N \subseteq M \text{ and N is not a prefix of M}}{N.\mathrm{n} \subseteq M}$$

So, for example, $1.2.3.4 \subseteq 1.3 \subseteq 2.4.5$.

Numeric versions can be used to represent the date; 1997.05.18 would represent a date that is more specific than 1997.04.30 and the most up-to-date version of a site can be build by requesting a version labeled with the current date.

2.3.2 Sub-versions

Since the Web is international, most certainly the requirements of the users will differ. One of the most important differences would be at the level of user interface. Some aspects are a necessity: a Turkish reader would want to view Web pages in Turkish and a Canadian would want to be able to choose between English and French and so on. Some aspects are a matter of personal taste, such as a preference for graphics, or for pure text.

Again, if one prefers graphics, which backgrounds/colors does he/she prefer? Even if the essential functionality were the same, the differences in user interface would make the Web sites look significantly different.

To handle these kind of variety of requirements Plaice and Wadge use *sub-versions* (also called *variants*). Sub-versions were partially addressed in SCCS and RCS, with the introduction of branches that were identified with numbers. In this version space, *names* are used instead and they can be described by the following grammar:

$$V ::= \varepsilon \mid N \mid x \mid V\%V$$

$$N ::= n \mid N.n$$

$$V \subseteq V\%V'$$

where x is any alphanumeric string and "%" is the sub-version operator. For example, the *image%sky* version of a Web site background would be the *sky* sub-version of the *image* version of the background.

Unlike in RCS, in IHTML names are not variables, but constants. Under the refinement relation they are all incomparable, i.e: *graphics* and *text* versions of a component cannot be compared. The sub-version operator is associative and has ε as identity:

$$\varepsilon\%V \equiv V$$

$$V\%\varepsilon \equiv V$$

$$(V\%V')\%V'' \equiv V\%(V'\%V'')$$

Sub-versions can be very powerful. For example, consider an international Web site for weather forecast. The page to display the weekly forecast can be versioned as follows: *canada%ontario, canada%bc%victoria, us%california%sanfrancisco* and so on. Depending on the request from users, the most relevant page can be viewed.

2.3.3 Joins of Versions

Sometimes different sub-versions can be compatible. Plaice and Wadge's system has a join operator, "+", for joining (merging) compatible versions. For example, a French user can request the graphics version of a Web page with a sky pattern as the image used for the background. The version for the page would be *french+graphics+image%sky*. With the join operator, +, the grammar for the version algebra is refined to:

$$V ::= \varepsilon \mid N \mid x \mid V\%V \mid V+V$$

$$N ::= n \mid N.n$$

Plaice and Wadge introduce the following axioms:

$$V \subseteq V + V'$$

$$\frac{(V_1 \subseteq V_1') \; and \; (V_2 \subseteq V_2')}{V_1 + V_2 \subseteq V_1' + V_2'}$$

The join operator, "+", is idempotent, commutative and associative, and left distributes %:

$$V + V \equiv V$$

$$V + V' \equiv V' + V$$

$$(V + V') + V'' \equiv V + (V' + V'')$$

$$V\%V' + V\%V'' \equiv V\%(V' + V'')$$

The + operator is defined to be the least upper bound operator induced by the $\subseteq$ relation. Version $V_1 + V_2$ is the least upper bound of V_1 and V_2 if and only if for all V such that $V_1 \subseteq V$ and $V_2 \subseteq V$, relation $V_1 + V_2 \subseteq V$ also holds. Consider such a V: then $(V_1 + V_2) \subseteq (V + V) \equiv V$. So $V_1 + V_2$ is the least upper bound.

Versions of complete systems are formed by using the most relevant components and if the + operator was not defined as the least upper bound operator, then the term "most relevant" would have no meaning. For example, if the requested version of a Web site is V and the most relevant versions of the pages of the site are V_1, V_2, ... V_n, where for all $V_i \subseteq V$ and $1 <= i <= n$. Then the least upper bound of these versions is the join of all the relevant versions: $V_1 + V_2 + ... + V_n$.

2.3.4 Canonical Form

The equality axioms allow a canonical form for all version expressions. In fact, except for the commutative and associative rules of +, the equations simply become rewrite rules:

$$\varepsilon \% V \rightarrow V$$

$$V \% \varepsilon \rightarrow V$$

$$(V \% V') \% V'' \rightarrow V \% (V' \% V'')$$

$$V \% V' + V \% V'' \rightarrow V \% (V' + V'')$$

$$(V + V') + V'' \rightarrow V + (V' + V'')$$

$$\frac{V \subseteq V'}{V + V' \rightarrow V'}$$

Some form of a dictionary order, represented by « , is used to introduce a total order on sub-versions.

$$N \ll x$$

$$\frac{x \quad \text{alphabetically precedes} \quad y}{x \ll y}$$

$$\frac{V \ll V'}{V \ll V' \% V''}$$

$$\frac{V \ll V''}{V\%V' \ll V''}$$

With the dictionary order, the canonical form for the joins is:

$$\frac{V \ll V'}{V' + V \rightarrow V + V'}$$

$$\frac{V \ll V'}{V' + (V + V'') \rightarrow V + (V' + V'')}$$

Note that the precedence order of the version space operators is ".", "%", "+" in descending order and parenthesis can be used at will to change interpretation order.

The following examples illustrate typical canonical form transformations:

x%y%z + x%a%b → x% (a%b + y%z)

(x + 4.1) + a%b → 4.1 + a%b + x

last + first → first + last

(x%y%z + a%(b%c + b%d%e)) + x%r → a%b%(c + d%e) + x%(r + y%z)

2.3.5 Variant Structure Principle

In summary, when constructing version V of a system, for each component C the most relevant version is used. To select the appropriate version of a component C, let S be the set of versions in which C is available. The set of relevant versions is $\{V' \in S | V' \subseteq V\}$. The most relevant version is the maximum element of this set, if there is one. If there is no maximum element, there is an error condition and there is no version V of the given system.

It is possible to apply this principle to a multiple versioned Web site as follows: suppose a site has pages $P_1, P_2, \ldots, P_n$, then the version that most closely approximates ver-

sion V of the site is formed by joining versions V_1 of P_1, V_2 of P_2 ... where in each case V_i is the version of P_i most relevant to version V, and the version constructed is $V_1 + V_2 + \ldots + V_n$, where $V_1 + V_2 + \ldots + V_n \subseteq V$.

Plaice and Wadge call this principle the variant structure principle, which describes the way sub-versions of a system can "inherit" components from a more generic version and the join operator, +, solves the problem of combining versions. In the following chapters, I will introduce an alternative to the variant structure principle, which aggregates *all* the relevant versions (in a user specified manner) to form the most relevant version.

Detailed discussions on this version language, the variant structure principle and applications using these principles, Lemur and Marmoset, can be found in [15, 16].

Chapter 3

Intensional HTML

In this chapter, I describe the intensional features of IHTML, which include links, images, Java applets and server-side includes. I conclude this chapter by describing the extended version space, the object-oriented and dataflow aspects of IHTML.

3.1 Overview

Intensional HTML, like conventional HTML, is a high-level Web authoring language. Its advantage over conventional HTML is that it allows practical specification of pages that can exist in many different versions. Web sites created by IHTML are easier to maintain and require less space when compared to the sites created by cloning conventional HTML files.

A typical IHTML page, like a conventional HTML page, consists of two major parts: the *head* and the *body*. The example IHTML code below illustrates the overall structure of an IHTML document and some of its intensional elements.

```
<ihtml>
        <head>
                ...
        </head>
        <body <include file=background> <include file=text> ... >
                <include file=header>
                ...
                <include file=filename/body>
                ...
                <a href=options from=filename>
                ...
                <include file=footer>
                <include file=update date=YYYY.MM.DD>
        </body>
</ihtml>
```

The outermost element, which encompasses the entire document, is named *ihtml*, to indicate that the enclosed text is an IHTML document. An IHTML document can be viewed by a standard browser as it is preprocessed and transformed into a conventional HTML document by a CGI script. The *head* element, as in conventional HTML, is a container for information about the document, which is not displayed as part of the document. The *body* element, where IHTML differs from conventional HTML, contains all the text and other material that is to be displayed in the browser window. The opening body tag has some attributes, extensions of common browsers, which affect the appearance of the document's background, color of text, links, active links, visited links. Using *intensional includes*, described later, these attributes can be versioned which allows a wide variety in the appearance of IHTML documents.

The set of elements that IHTML supports is a superset of the elements supported by conventional HTML. In the following sections I describe IHTML specific elements in detail.

3.2 Intensional Elements

3.2.1 Links

Intensional links in IHTML source look exactly like the links in conventional HTML, except the value of the `href` attribute:

```
<a href="URL_address">highlighted_text</a>
```

where *URL_address* is the URL of the *directory* that contains all the versions of the target page. The intensional links are interpreted as denoting a whole family of links, each connecting a given version of the page they appear in to the *corresponding* version of the page they link to.

For example, considering the slide show example from Chapter 1, suppose that the generic IHTML source for *slide5* contains the link:

```
<a href=slide6>Slide6</a>.
```

This is interpreted as meaning that the English version of *slide5* is linked to the English version of *slide6*, and that the French version of *slide5* is linked to the French version of *slide6*. When the server-side software generates the French version of *slide5* (a conventional HTML file) from the generic source (IHTML file), it transforms the generic (intensional) link into an extensional (conventional) link to the French version of *slide6* (this might be the only change made). If the value of `href` starts with a `#`, then the intensional link serves as a hyperlink to the fragment identifier of the current page. Intensional hyperlinks to fragments of a different intensional page can be created by specifying the URL of its directory before the fragment identifier.

IHTML also allows links to conventional (unversioned) HTML webware. These links are called *extensional* (or *external*) links which have exactly the same format of conventional hyperlinks except that `href` attribute is replaced with `xhref`:

```
<a xhref="URL_address">highlighted_text</a>
```

The server-side software produces the final hyperlink by replacing `xhref` by `href` (`URL_address`, the URL of the target *file*, is left unchanged). Following these links would take users to different, possibly unversioned, sites. I could have used `href` for extensional links and `ihref` for intensional links. However, I chose the path of using the conventional attribute names for intensional elements: since extensional elements constitute smaller portions of intensional source files, and they should be the ones distinguished by different (new) syntax.

3.2.2 Images

Similar to intensional links, intensional image elements in IHTML source look exactly like the image elements in conventional HTML, the only difference being the value of the `src` attribute.

```
<img src="graphics_file" [alt="..."]? [align="..."]? [ismap]?>.
```

They are interpreted, however, as denoting a whole family of images which get included in the *corresponding* version of the pages that include them.

For example, considering the slide show example again, suppose that the generic IHTML source for *slide5* contains the image tag:

```
<img src=image5>.
```

This is interpreted as meaning that the high-resolution version of *slide5* will include the high-resolution version of *image5*, if there is any, and that the low-resolution version of *image5* will be included in the low-resolution version of *slide5*. Currently, IHTML supports *jpg* and *gif* formatted image files. Any of the two formats of the same graphics file can be kept under the same directory, hence IHTML authors do not have to supply both formats for each version of the same image. In fact, there won't be a best approximation, i.e. an error condition will occur, if both formats of the same version are supplied.

For image sources that do not exist in different versions, extensional image elements

can be used:

```
<img xsrc="graphics_file" [alt="..."]? [align="..."]? [ismap]?>.
```

These image elements are interpreted as conventional image elements and server-side software produces the final image element by replacing `xsrc` with `src`. Similar to extensional links, `graphics_file` is the URL of the image *file*.

3.2.3 Java Applets

Similar to intensional images, intensional Java applets in IHTML source look exactly like the Java applets in conventional HTML, the only difference being the value of the `code` attribute, which specifies the directory that contains all the versions of the Java applet in question.

```
<applet code="class_file" [codebase="..."]? width="..."
        height="..."> [<param name="..." value="...">]* </applet>.
```

They are interpreted, however, as denoting a whole family of Java applets which get executed in the *corresponding* version of the pages that include them.

For Java applets that do not exist in different versions, extensional Java applets can be used:

```
<applet xcode="class_file" [codebase="..."]? width="..."
        height="...">[<param name="..." value="...">]* </applet>.
```

These applet elements are interpreted as conventional applets and server-side software produces the final applet element by replacing `xcode` with `code`. In extensional Java applets, `class_file` is the full name of the Java class file (i.e. "`.class`" extension is included).

3.2.4 Includes

IHTML also has an intensional server-side include feature which causes the contents of the included file to be incorporated (by copying) into the HTML page under construction.

The IHTML includes are also generic. For example, the English version of any page will include the English version of the include file. When processing an include, the server-side software looks for the version of the named source file whose version label most closely approximates the current version, i.e. the version requested of the file in which the include appears.

The included files can have intensional elements and can include IHTML files, too. There is no limitation on the number of nested includes. However, caution must be used as infinite loops can occur if a file includes itself or part of it directly or indirectly.

The include facility is very important for IHTML because it allows authors to break the source components into pieces smaller than a whole page. This allows authors to isolate the parts of a page that actually vary, and write more generic source for the parts (such as headers and footers) that do not. In other words, intensional includes increase the level of code sharing.

For example, each page of the slide show can include a header file at the top with

```
<include file=header>
```

which can contain the logo of the company/conference and date etc., and include a footer file at the bottom with

```
<include file=footer>
```

which can contain links to various parts of the slide show and name of the author/presenter etc., at the bottom.

Extensional (external) includes are not implemented for a number of reasons. An extensional include would be specified by one of the following two possible directives:

```
<!-- #include file="file.html" --> or

<!-- #include virtual="/virtual_path/file.html" -->
```

The first directive can be used to include a file in the current directory and the second directive can be used to include a file from a directory relative to the server root. The basic problem with the use of conventional server-side include directives is that they are not supported by all servers. Even if the IHTML author's server supports server-side includes (which would require additional configuration and parsing), these directives can only include files that reside on the *same* server, because the *virtual* attribute of the include directives can only specify the virtual path to the include file on the same server. (NCSA limits the included file to something relative to, but not above, the current directory. Thus, ../ is disallowed, as is any absolute path, even if the HTTPd server process would normally have access there. However, there are a few servers (not very commonly used) that let you specify any path at all. Purveyor, for example, lets you use UNC file specifications, thus allowing the include to pull its data from anywhere reachable on the network. Nevertheless, I do not consider it as a good practise to include files from remote hosts as the integrity of your site would depend on other servers states: authorization issues, up and down times, etc.) On the other hand, any unversioned file that resides on the same server can be included by IHTML's intensional includes. As long as the file is stored as an intensional file would, under a directory with the same name ("*/filename/filename.ihtml*"), the server-side software would consider this file as the vanilla (default) version of the (theoretically intensional) file and use it in all versions of the site.

3.3 IHTML Version Space

The families of pages specified by IHTML are indexed by (subspaces of) the algebraically defined version space described in Chapter 2. In the terminology of intensional logic, the elements of this space are *possible worlds*; each individual possible world (version) determines a particular extension, i.e. an actual conventional HTML page.

The elements of this version space are equivalence classes of expressions together with the coarsest order which satisfies the axioms.

The IHTML version space extends the version space defined in Chapter 2 in one important way: it allows explicit dimensions. [17]

For example, the term *french* in the examples of previous chapters is interpreted as referring to the interface language. What if the information on the pages were about cooking and the cuisine also had to be specified?

In IHTML, versions consist of an arbitrary number of distinct *dimensions*. Dimensions are represented by arbitrary identifiers specified by IHTML authors. For example, the author of a multiple versioned cooking site can define the following dimensions: *language*, *cuisine*, *platform*, and *language:french* + *cuisine:chinese* + *platform:Mac%K68* would represent a valid version of this site.

The values of each dimension is specified by a *version expression* (french, Mac%K68). Version expressions are represented by the rules of the version algebra defined in Chapter 2. In other words, IHTML version space represents a version as a group of distinct dimensions and the coordinates of each dimension is represented by the original version space.

The extra rules needed to extend the version space described in Chapter 2 are:

$$D{:}\varepsilon \equiv \varepsilon$$

$$D{:}(V+V') \equiv \mathrm{D{:}V} + \mathrm{D{:}V'}$$

$$\mathrm{D{:}V} \subseteq \mathrm{E{:}V'} \leftrightarrow D \equiv E \text{ and } V \subseteq V'$$

It should be clear now how to compare two versions in IHTML version space. In general

$$D_0{:}V_0+D_1{:}V_1+\ldots \subseteq E_0{:}W_0+E_1{:}W_1+\ldots$$

if and only if each D_i is equal to E_j for some j and in each such case $V_i \subseteq W_j$. (Here the assumption is that the D's and E's are distinct among themselves, ordered in dictionary order and no V_i is ε.) I will discuss the comparisons of versions and version expressions in further detail in the next chapter.

3.4 Transversion Elements

One of the main characteristics of intensional programming languages, is *context-switching (intensional)* operators. In IHTML, context-switching is achieved by *transversion elements*; links, images, applets and includes.

Transversion links are intensional links that lead from the current version of the source page to a *different* version of the target page - different in a way specified in the link element.

Transversion links allow visitors to the site to move from one version to another, without necessarily filling in forms or composing complex URLs. At the same time, transversion links give the author full control over the way in which different versions of the site are interconnected. Transversion links can point to any versioned page of the site.

A transversion link has the same format as an intensional link, except that it can contain arbitrary number of assignments to dimension identifiers:

```
<a href="URL_address" [dimension_identifier="..."]+ > ... </a>.
```

The link is interpreted as leading from a given version of the source page to a modified version of the target page (or a fragment of the target page) - the modifications resulting from altering the coordinates of the given dimensions as specified. Note that if the modified version of the target page does not have the specified fragment, browsers display the page starting from the top by default.

In the IHTML template for the English version of the title page of the slide show the author can include a transversion link of the form:

```
<a href=slide1 language=french>French version of the site</a>.
```

This link will be interpreted as a link to the *language:french* version of the title page hence the French version of the whole slide show. The reason is that the French version of *slide1* is linked to the French version of *slide2*, and so on. The English and French versions coexist as sort of parallel universes, and the transversion links let the reader move from one of the universes to the other.

Furthermore, transversion links can refine the current version of the page they reside in, since they can also specify coordinates for unset dimensions, which are considered to be initialized to their default (vanilla) values. For example a link of the form:

```
<a href=slide1 background=plain%blue>Plain blue background</a>
```

in *language:english* version of *slide1* will take the users to the *language:english+background:plain%blue* version of *slide1* (if it exists) and the *background* dimension will be set from then on (i.e. used by the rest of the pages in the site).

Transversion image elements and Java applets give IHTML authors the ability to display versions of an image or a Java applet different from the current version. These transversion elements have the same syntax as the intensional ones with additional attributes to set the dimension identifiers:

```
<img src="graphics_file" [dimension_identifier="..."]+ ... >
```

```
<applet code="class_file" [codebase="..."]?
        [dimension_identifier="..."]+ width="..." height="...">
        [<param name="..." value="...">]* </applet>.
```

IHTML also allows transversion includes, with a similar syntax:

```
<include file="URL_address" [dimension_identifier="..."]+ >.
```

For example:

```
<include file=footer language=english>
```

will include the footer file in a version like the current version except that the *language* component will be *english*.

The difference between transversion links and the other transversion elements is the scope of the modifications to the current version. The modified version is global in case of links and local in case of images, applets and includes. For example, in the following IHTML template for the *language:english* version of *slide5*:

```
...
<img src=image5 resolution=256>
...
<a href=slide5 language=french> French version of the site </a>
...
<include file=slide5/body display=text>
...
<a href=slide6> Next slide </a>
...
```

the *language:english+resolution:256* version of the *image5* is used, the transversion link, when activated, would link the users to the *language:french* version of *slide5* (hence *language:french* version of the whole site), the transversion include element includes the *display:text+language:english* version of the body part of *slide5*. Any *intensional* element in this included text would point to *display:text+language:english* version of their source. However, the intensional link

```
<a href=slide6> Next slide </a>
```

in *slide5* still links the users to the *language:english* (not *display:text+language:english+resolution:256*) of the next slide.

External links, applets and images in IHTML can be considered as transversion links applets and images (respectively) in which the coordinates of all dimensions are set to the vanilla version, ε.

3.5 An Intensional Variable

As the number of dimensions supported by a site increases, the number of possible transversion links that make sense for every intensional page increases too. Usually, the transversion links are placed close to the end of intensional pages, after the contents of the usual body part and before the contents of the footer file. However, it may become impractical to list all the possibilities for all pages considering the number of different versions all pages might have.

Instead, transversion links that are common to all pages (the ones that modify dimensions like background, text and link colors for instance) can be (as in the prototype site) grouped into a special page called the *options* page. This page's sole purpose is to present to the users a set of possible worlds and upon their request take them to the corresponding world.

The basic idea is that, by following an intensional link from any page of a versioned site, the user will be presented with a page that displays the possible worlds in some fashion. Options pages are versioned, too. Hence, in the *display:graphics* version, possible color and image patterns for backgrounds are displayed using small anchored images, whereas in the *display:text* version, only possible colors for backgrounds (in the form of colored text links) are presented. All the links in options page are transversion links. Upon

following any of these links, users are taken back to the page they were viewing prior to the options page, however the whole site is now in the requested (possibly modified) version.

Intensional and transversion links, introduced in the previous sections, do not carry any information about the page they reside in. Hence, there is no way for a transversion link to take users back to the previous page without hardcoding the name of the previous page into its `href` tag. Without the previous page information, clones of all the versions of the options page must be created for all the pages that exist in a site, the only difference being the value of the `href` tags.

IHTML syntax is extended by introducing an intensional variable, *prev_page*, which can be used instead of the static values of href tags. This variable contains URLs of previous pages at any location of the site. In this way, authors can create generic transversion links for a particular version of the options page which will do the same modification to the current version for all the pages in the site. In other words, there will be only *one* IHTML template for *language:english* version of the options page, which will be available from the *language:english* version of all the pages in the site.

For example, in the IHTML source for the *background:image%sky* version of *slide5,* the author can include the following link:

```
<a href=options from=slide5> Options </a>
```

which would link the page to the same version of the options page. A transversion link in the options page of the form:

```
<a href=#prev_page# background=plain%blue> Plain Blue </a>
```

would be equivalent to:

```
<a href=slide5 background=plain%blue> Plain Blue </a>
```

which would link to the *background:plain%blue* version of *slide5*. The same transversion link of the options page would be equivalent to:

```
<a href=slide1 background=plain%blue> Plain Blue </a>
```

if the options pages was visited from *slide1*.

3.6 IHTML as an Object-Oriented/Dataflow Language

It should be clear that IHTML incorporates a fundamental concept of object-oriented programming - inheritance - in order to allow Web site designers to write generic code which applies to many versions of a page.

In the object-oriented model, first the receiving object is searched for the code for the message in question. If the object does not have the code, the search is continued up the inheritance hierarchy. Once the code for the message is found, it is executed at the original object.

Correspondingly, in IHTML, the server-side software might receive a request for the V version of page P, but the page does not exist in that version: the V' version is the most relevant copy of page P. Then the V' version of page P is taken to be equal to the V version of page P. In other words, by default the V version of the whole site inherits its components from those of version V'. Version inheritance allow IHTML authors to reuse code in much the same way that class inheritance allows object-oriented programmers to reuse program source.

On the other hand IHTML may seem to lack many of the features of an object-oriented language: objects, messages, instance variables and methods. However, the versions (elements of the version space) correspond to classes. A demand for version V of page P can be thought of as sending the "message" P to object V. In this analogy, the source files are the "methods" associated with the version (object) with which they are labeled. The

procedure for selecting the right source file then corresponds exactly to the usual procedure for locating the appropriate method to handle a message (assuming that multiple inheritance is supported).

In fact, a better paradigm to assign to IHTML is *dataflow* rather than object-oriented, in particular, *dynamic demand-driven* dataflow.

Demand-driven dataflow has two major characteristics. The evaluation order is dynamic and an operation is performed only if it is required. This improves performance and errors can be avoided if they only occur in unnecessary operations. In dynamic demand driven dataflow, i.e. eduction, there is a two-way traffic in the communication lines. In one direction demands are sent to the server, and in the other direction the data produced on the fly by the server is sent back to the user.

In IHTML, each demand, produced by following links, is for a particular page together with a particular version. This demand can (when there are includes such as image or text) generate additional demands, possibly with different versions. Eventually all the sub-demands are met and the required HTML page is returned to the user. The (intensional) pages correspond to the dataflow program variables, and the IHTML source files play the role of the defining expressions. The crucial difference is that the same variable (page) can have several defining expressions, with the choice of the relevant expression determined by the version of the demand.

Also there are no unnecessary operations in the sense that the most relevant source file is not found and the conventional HTML code is not produced until there is a demand, in this case an activated intensional or transversion link. The time spent on a search for the most relevant source file depends on the number of versions supported by the site (versions have to be mapped to their integer codes) and the number of existing versions for that particular file (all the different versions of the source file must be compared to the version requested). A typical search takes about one second (0.56s - 0.7s) to complete.

Hence, if an intensional page has n intensional or transversion links, then the demand-driven approach reduces the conventional HTML creation time by n seconds. This increased speed becomes remarkable especially in the case of the options page which can have hundreds of transversion links.

The fact that IHTML has object-oriented *and* dataflow features is not coincidental. In fact, there is a general analogy between object-oriented computation and extended (multiple definition) dataflow described in [10].

Chapter 4

An Implementation of IHTML

I have implemented the intensional HTML system described in the previous chapter. The implementation consists of three relatively independent parts: A CGI script to do IHTML to HTML transformation, software to find the most relevant versions of IHTML templates, Java class files and image files, and a group of script-like programs to help IHTML authors set up multiple versioned Web sites.

4.1 Overview

It might seem from what has been said that IHTML requires its own version of the server and client software. However, I preferred using standard techniques to allow existing clients to browse multiple versioned Web sites published by existing servers. My ultimate goal was to allow IHTML authors to create Web sites that would look like conventional sites consisting of a huge number of HTML pages whereas in fact, the number of underlying IHTML pages would be far less.

On entry to my multiple versioned Web site, the user is presented with the default version of the home page, which looks like a conventional HTML page (*display:graphics+language:english* version is chosen as the default version, however any of the existing versions of the site could be used instead). From then on, the user can browse the site in

exactly the same way he/she would in case of a conventional Web site. However, when the user clicks on intensional links, a CGI script is executed which is passed parameters including, a relative URL (name of the intensional page) taken directly from the IHTML source of the link, and a representation of the current version. In the case of transversion elements, an additional piece of information, modifications to the current version (again taken directly from the IHTML source of the elements), is also passed to the CGI script. Finally, an optional parameter, the name of the current page can be passed to the CGI script which would be used to replace any possible *prev_page* variables in the next page.

Satisfying requests from clients involves searching for the appropriate IHTML source file and then transforming the generic source file into the particular HTML file corresponding to the version in question.

The basic idea is to ensure that all links to an IHTML page go through a CGI script. The CGI script itself ensures that all intensional and transversion links are transformed to CGI calls. When the CGI script generates HTML from IHTML, it transforms the normal-looking generic IHTML links into CGI calls with the appropriate parameters. In case of includes, the CGI script executes the server-side software to find the most relevant source file and uses its contents to replace the include element in the HTML file under construction. Intensional and transversion image (or applet) elements are replaced with conventional HTML image (applet) elements with the `src` (`code`) attributes of the image (applet) element set to the most relevant image (Java class) file.

All the IHTML template files for a particular intensional page are stored in a directory labeled with the name of that intensional page. Most of the these files *include* a header, a body, a footer, and an update file. In most cases the body file is represented by a sub-directory. Usually, the footer, header files for all the pages have similar contents, containing links to the index page, copyright information, name of company, date, address, and etc., hence directories for footers, headers are at the same level with the other directo-

ries that represent the intensional pages of the site. Similarly, all the image files (or Java class files) used in a multiple versioned site are usually kept in a directory (labeled as *images* or *applets*) at the same level with directories for intensional pages and each image (or Java class) is represented as a sub-directory labeled with the name of the image (or Java class). One other directory that can be shared among the whole site is the *update* directory. This directory would contain files which would consist of last update dates (dates would be represented in *yyyy.mm.dd* format so that version comparisons are easier) and text which would vary in each language. In this way, any page of the site can include one of these files by specifying the date of last update.

Each IHTML template file is named in the form *pagename.nnn*.ihtml where *pagename* is the name of the intensional page (also the name of the directory) and *nnn* is the integer code which represents the version label of the source file. Using the same convention, image files are labeled with either *imagename.nnn*.jpg or *imagename.nnn*.gif depending on the format of the image and the Java class files are labeled with *classname.nnn*.class. I describe the relationship between the integer codes and versions in the next section.

4.2 Server-side Software

The server-side software, implemented in C, is based on LEMUR[15], an intensionalized form of SLOTH[18]. LEMUR is a Unix-based tool for creating and maintaining large C programs built from reusable modules represented as Unix directories. Individual modules can be compiled without any knowledge of the application that uses, directly or indirectly, the module in question. It also allows users to create and label different versions of the individual files that make up a module, where labels can be any element of the version space described in Chapter 2.

The server-side software uses a similar scheme to create and maintain multiple ver-

sioned Web sites, intensional pages being the reusable components. However, the server-side software uses the extended version space described in the previous chapter. Furthermore, integer codes are used for versions in file labels instead of character strings, which can be very long and impractical with the repeated use of + and %. The coding scheme, a variant of the hash-consing technique[19], assigns unique small integer codes to versions on a demand-driven basis. In hash-consing, a list is stored in a hash table as pairs *(head, position of tail)*. The list is then represented by the position (in the table) of the first tuple in the list.

In IHTML, versions are represented as ordered lists of dimension identifier and value pairs, allowing the use of hash-consing. The following functions are supported by the server-side software to implement this scheme:

- *insert(version)*: Inserts a version into the hash table if it does not exist, returns the position of the version in the table.
- *getversion(code)*: Returns the version represented by *code*.
- *formversion(version, code)*: Merges *version* and the version represented by *code* into a new version, inserts the new version into the hash table and returns the code for the new version.

For example, calling *insert(v)*, *insert(v')* and *insert(v")* on an empty table where: v is *display:graphics+language:english+text:blue*, v' is *background:image%sky+language:english+text:blue* and v'' is *display:graphics+language:french+text:blue*, would result in the table shown in Figure 4.1. Furthermore, the last insert has the same effect as calling *formversion(language:french, 1)*. Hence, *getversion(1)*, *getversion(4)*, *getversion(5)* would return versions v, v', v'' respectively.

	head		tailcode
0	null	null	0
1	display	graphics	2
2	language	english	3
3	text	blue	0
4	background	image%sky	2
5	display	graphics	6
6	language	french	3
⋮			

Figure 4.1 The hash-consing technique.

As new versions are inserted into the table, first pairs of their corresponding lists, in the previous example positions 1, 4 and 5, are marked as head pairs to indicate that versions can be formed starting at those positions of the table. This method prevents the need to keep track of the integer codes representing the current versions supported at different times.

The use of hash-consing ensures that the representation is compact and version comparisons (especially equality checks: plain integer comparisons) can be performed efficiently. It also keeps the size of the parameters passed to the CGI script small, avoids restrictions on the symbols which can appear in version expressions, and also provides a

measure of security. Readers cannot jump into an arbitrary chosen version of the site. Instead, the IHTML author, using only transversion elements, controls the access of the readers to the different versions of the site.

4.2.1 Map file

The server-side software makes use of a map file to load the hash table which contains the versions currently supported by the site. The map file consists of version records of the format shown below:

```
version
        dimension_identifier1 dimension_value1
        dimension_identifier2 dimension_value2
        ...
        dimension_identifierN dimension_valueN
endversion
```

The dimension identifier and value pairs are case insensitive and do not have to be in any particular order. Modifications to existing versions are saved as new version records leaving the original records unchanged. The map file can be created manually or by using *packv*, a script to aid IHTML authors in creating multiple versioned sites. IHTML authors must grant read and write permissions to the map file to everybody so that each execution of the server-side programs can read and update the current versions. (Note that CGI scripts run as nobody, hence has very restricted access to local files.)

There is only one map file for each site which is, therefore, shared among all the clients accessing the site. Consequently, a locking scheme is required to prevent any inconsistencies that might occur. Temporary lock files are used to simulate locks on the map file. The standard file locking function, *flock()*, is not used as it is not reliable on NFS mounted Web sites. In my scheme, the server-side software checks to see if the map file is locked by any another process by searching for lock file(s) named `.lock.`*pid* (under a

dedicated lock directory, usually a sub-directory of the cgi-bin directory), where *pid* is the process identifier of the process which created the lock file. The server-side software is blocked until there is no lock file in the lock directory. When the server-side software creates its lock file it again checks for any other lock files, to prevent race conditions that might occur when two or more processes check for lock files, find none and create their lock files at the same time. In such situations, the process with the smaller process identifier proceeds and the others remove their lock files. Processes remove their lock files as soon as they finish updating the map file. In case the process terminates abnormally (the memory on the server might not be enough to execute large number of executables at the same time), its lock file is removed by the next process which determines that the lock file is too old to be created by a process that is still executing.

4.2.2 Data Structures

Version expressions (dimension values) are represented by binary trees. Binary trees were chosen because of the recursive nature of version expressions and the fact that all the version space operators are binary operators. Nodes of a tree can be *join* nodes, *sub-version* nodes, *numeric sub-version,* nodes, *leaf* or *numeric leaf* nodes. An empty tree represents the expression for the vanilla version. The right child of a leaf node, can be empty, representing an expression that does not contain + and/or % operators. However, the left child of any node is either a plain text (a simple expression like; english) or a sub-tree. For example, the version expression *french+graphics%256+4.1* is represented as shown in Figure 4.2.

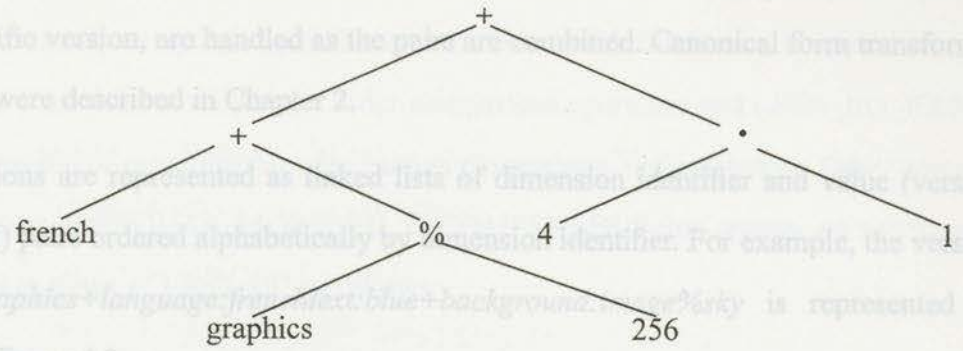

Figure 4.2 Representation of a version expression in IHTML.

Version expressions are parsed by a simple recursive descent parser. The parser creates and stacks sub-trees (sub-expressions) as operators with higher precedences (than the top most operator in the stack) are met, leaving the left child and (possibly) the right child of the final tree in the stack. Parentheses, if any, are checked to match before parsing and handled by recursive calls to the parser.

After version expressions are parsed, their binary trees are checked to contain valid expressions. The following conditions are checked:

- leaf nodes must have no children,
- non-leaf nodes must have exactly two children,
- sub-numeric nodes must have a leaf (numeric) node as its left child and either a numeric or a sub-numeric node as its right child.

Before the version expressions can be compared correctly, they must be transformed into their canonical forms. A version expression is turned into its canonical form by breaking the expression into a list of sub-expressions of the same type, sorting the elements of the list in dictionary order and building a (possibly) new expression by combining the sub-expression pairs as the sorted list is traversed in reverse dictionary order. Cases like

extracting common initial subsequences of sub-versions and merging of joins into the more specific version, are handled as the pairs are combined. Canonical form transformation rules were described in Chapter 2.

Versions are represented as linked lists of dimension identifier and value (version expression) pairs ordered alphabetically by dimension identifier. For example, the version *display:graphics+language:frenchtext:blue+background:image%sky* is represented as shown in Figure 4.3.

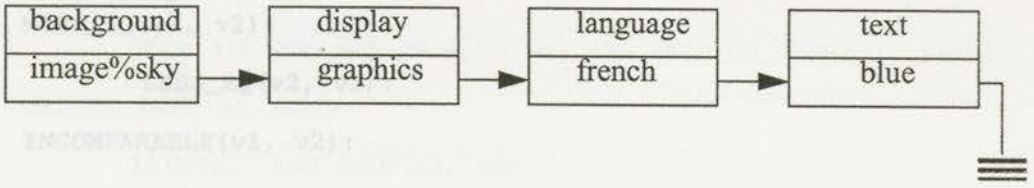

Figure 4.3 Representation of a version in IHTML.

The major data structures used in the server-side software are included in Appendix A.

4.2.3 Version Comparisons

Version (dimension sum) comparisons are very important to the server-side software as its main purpose is to find the most relevant versions of given pages. The server-side software uses the following predicates to compare two versions, `v1` and `v2`:

```
LESS_EQ(v1, v2):
        id(head(v2)) < id(head(v1)) AND LESS_EQ(v1, tail(v2)).
LESS_EQ(v1, v2):
        id(head(v1)) = id(head(v2)) AND
        LESS_EQ_EXPR(value(head(v1)), value(head(v2))) AND
        LESS_EQ(tail(v1), tail(v2).
```

where `head(v)` and `tail(v)` are the head and tail of the list that represents the version `v`, `id(d)` and `value(d)` are the identifier and the value of dimension `d`. It should be noted that < and = are used as dictionary order comparison operators and LESS_EQ_EXPR is the less specific or equal operator for version expressions, defined below. Other comparison operators like EQUAL, MORE_EQ, INCOMPARABLE that operate on versions can be defined in terms of LESS_EQ as follows:

```
EQUAL(v1, v2):
        LESS_EQ(v1, v2) AND LESS_EQ(v2, v1).
MORE_EQ(v1, v2):
        LESS_EQ(v2, v1).
INCOMPARABLE(v1, v2):
        NOT LESS_EQ(v1, v2) AND NOT LESS_EQ(v2, v1).
```

Version expression comparisons are more complex than version comparisons. A number of predicates are required to check different cases depending on the structure of the binary trees that represent the expressions. Two version expressions, `e1` and `e2`, can be compared using the following predicates:

```
LESS_EQ_EXPR(e1, e2):
        leaf(e1) AND leaf(e2) AND label(e1) <= label(e2).
LESS_EQ_EXPR(e1, e2):
        leaf(e1) AND sub-version(e2) AND
        LESS_EQ_EXPR(e1, right(e2)).
LESS_EQ_EXPR(e1, e2):
        leaf(e1) AND numeric(e1) AND sub-numeric(e2) AND
        LESS_EQ_EXPR(e1, left(e2)).
LESS_EQ_EXPR(e1, e2):
        leaf(e1) AND join(e2) AND
        LESS_EQ_EXPR(e1, left(e2)) OR LESS_EQ_EXPR(e1, right(e2)).
```

```
LESS_EQ_EXPR(e1, e2):
        sub-version(e1) AND sub-version(e2) AND
        (LESS_EQ_EXPR(left(e1), left(e2)) OR
        LESS_EQ_EXPR(right(e1), right(e2)) OR
        LESS_EQ_EXPR(e1, left(e2))).
LESS_EQ_EXPR(e1, e2):
        sub-version(e1) AND join(e2) AND
        (LESS_EQ_EXPR(e1, left(e2)) OR LESS_EQ_EXPR(e1, right(e2))).
LESS_EQ_EXPR(e1, e2):
        join(e1) AND sub-version(e2) AND LESS_EQ_EXPR(e1, left(e2)).
LESS_EQ_EXPR(e1, e2):
        join(e1) AND join(e2) AND
        (LESS_EQ_EXPR(left(e1), left(e2)) OR
        LESS_EQ_EXPR(right(e1), right(e2)) OR
        LESS_EQ_EXPR(e1, left(e2)) OR LESS_EQ_EXPR(e1, right(e2))).
LESS_EQ_EXPR(e1, e2):
        sub-numeric(e1) AND leaf(e2) AND numeric(e2) AND
        LESS_EQ_EXPR(left(e1), e2).
LESS_EQ_EXPR(e1, e2):
        sub-numeric(e1) AND sub-version(e2) AND
        LESS_EQ_EXPR(e1, left(e2)).
LESS_EQ_EXPR(e1, e2):
        sub-numeric(e1) AND sub-numeric(e2) AND
        LESS_EQ_EXPR(left(e1), left(e2)) AND
        LESS_EQ_EXPR(right(e1), right(e2)).
LESS_EQ_EXPR(e1, e2):
        sub-numeric(e1)AND join(e2) AND
        LESS_EQ_EXPR(e1, left(e2)) OR LESS_EQ_EXPR(e1, right(e2)).
```

where `right(e)`, `left(e)` refer to right and left children of the binary tree used to represent the version expression `e`. `Leaf(e)`, `sub-version(e)`, `sub-numeric(e)`, `numeric(e)`, `join(e)` are used to check the type of the version expression, and `label(e)` is the label of the version expression. "<=" is the dictionary comparison order when `label(e)` is alphanumeric and integer comparison operator when `label(e)` is numeric (or `numeric(e)`). EQUAL_EXPR, MORE_EQ_EXPR and INCOMPARABLE_EXPR can be defined in terms of LESS_EQ_EXPR in the same way as above.

The underlying C code has one comparison function for versions and one for version expressions both of which return one of the four possibilities; equal, less specific, more specific or incomparable. The code for the version comparison function is included in Appendix B. The server-side software makes use of the version comparison and canonical form transformation algorithms of Lemur to compare and canonicalize its version expressions.

Given the path to a directory, the type of the file and the version V requested, the server-side software goes through all the files of the requested type in the directory, extracting version information from file names. The versions that are incomparable to or more specific than V are discarded and the most specific of all the relevant files is chosen. When a page does not exist in the requested version the CGI script displays the conventional "404 File Not Found Error". This type of error occurs when either the page does not exist in any version less specific (including the vanilla version) than the requested version or there is no best match for the most relevant version. (If a page exists in V and V' versions and the requested version is $V + V'$, then there is no best match.)

The directories with names of the form *pagename_a* are treated differently. These directories are called *aggregate* directories which can contain IHTML, image and Perl script files with names different from *pagename*. For this type of directories a different version of the variant structure principle is used. The basic idea in aggregate directories is

to choose *all* the relevant files instead of the *most* relevant one and aggregate the relevant files in the order specified by author supplied (local) scripts. These scripts, which serve as filters, can also be versioned. Usually include files are aggregated, but whole pages can be aggregated too, where the local script must not only aggregate the relevant templates but make sure that the resulting template is a valid IHTML page.

When there is a request for a page that contains an include which should be aggregated, the server-side software passes the names of all the relevant files to the most relevant version of the local script. The script creates a temporary file (temp.*pid*.ihtml) by aggregating all the relevant files in an order depending on its version. An example local script is included in Appendix C. Temporary files are discarded after they are processed by the CGI script. Aggregate files (temp.*pid*.ihtml) are only created on demand and discarded after they are no longer needed, since the resulting aggregate file will be different every time a relevant file is added to or removed from the directory.

In other words aggregate directories allow authors to define dynamically created IHTML templates. This can be useful when joins of versions are requested. For example, in case of an intensional page represented by an aggregate directory, when version $V + V'$ is requested and the page exists only in versions V and V', it is no longer an error condition. Instead, the local script aggregates these two versions of the file (both relevant to version $V + V'$) which is then used as the most relevant file. The Perl scripts are versioned in order to allow authors define various aggregation rules. Note that vanilla version of a file in an aggregate directory cannot contain any data that is also contained in another version of the file because of replication possibilities.

4.2.4 Packing and Unpacking Version Expressions

The server-side software has two supporting script-like programs to pack (transform version into integer codes) and unpack (transform integer codes to version) version information.

packv [-fMap_file]? is the script to pack version information. The user is presented with the corresponding integer code for any valid version information supplied. If the version does not exist, it in inserted into the *Map_file*, hence this script can be used repeatedly to create the *Map_file*. For any intensional page, the user is also given the option to find the most relevant version of the file in comparison to the version that is just packed. The user has to specify the type of the file he/she is looking for since versioned files can be IHTML templates, images, Java class files or Perl scripts. This usage can be practical for creating different versions of files, since one can start editing the copy of the most relevant file rather then starting from scratch.

unpackv -vVersion_code [-fMap_file]? is the script to unpack version codes, which basically has the reverse functionality of *packv*; displaying the corresponding version information for any valid *Version_code*.

Both scripts use the map file of the site (in the cgi-bin directory) by default, however for testing purposes or in case of a server supporting multiple sites, user specified map files can be used instead.

4.3 CGI Script

The CGI script that activates the server-side software to find the most relevant IHTML file and transform it into HTML is written in Perl. Perl, by far the most widely used language for CGI programming, was chosen for the following reasons:

- It is highly portable and readily available.
- It contains extremely powerful string manipulation operators, as well as func-

tions to deal with binary data.

- It contains very simple and concise constructs.
- It makes calling shell commands very easy, and provides some useful equivalents of certain UNIX system functions

Once the script is running, the first thing it needs to generate is a valid HTTP header, ending with a blank line. The header contains a content type, also known as a MIME type. In my case the content type of the data that follows is *text/html*. This header is needed to notify the client what type of data it is sending, so that the browser can format and display the document properly. After the MIME content type is output the script proceeds to display output in HTML.

The script uses a default version, set by the author, to display the home page. For any other page requested, the script starts computation by finding the most relevant version of the requested intensional page. This information is passed to the script through the PATH_INFO environment variable. The PATH_INFO environment variable, also called extra path information, depends on the server knowing where the name of the program ends, and understanding that anything following the program name is extra. Here is how the script is called with extra path information:

```
href=cgi-bin/script_name.cgi/page_path,V,[M,prev_page_path]?
```

where `page_path` is relative path to the directory which contains all different versions for the requested page, `V` is the integer code for the version requested, `M` is the integer code for the modifications to the version `V` and `prev_page_path`, is the relative path to the directory which represents the page that was displayed prior to the current page.

The script computes the version requested using `V` and `M` (if there is any) and invokes the server-side software with the appropriate parameters which in turn returns the full path of either the most relevant file or the temporary file which is an aggregate of all the relevant files. The communication between the CGI script and the server-side software is done

through a pipe. After the file is located and opened the script starts parsing the template. IHTML templates are scanned element by element (< ... >). This is achieved by enabling multiple lines in pattern matching and setting the paragraph separator to "<" and ">" back and forth. The elements are pre-processed by replacing multiple spaces with single spaces. The attributes are forced to have:

```
attribute_name=attribute_value
```

format. Conventional and intensional attributes can be specified in any arbitrary order. Intensional attributes are removed in the final HTML element and invalid attributes are ignored.

Intensional links like:

```
<a href=page_path> ... </a>
```

are transformed to:

```
<a href=cgi-bin/script_name.cgi/page_path,V> ... </a>
```

where v is the integer code for the current version.

Intensional image elements like:

```
<img src=images/image_name ... >
```

are transformed to:

```
<img src=images/image_name/image_name.V'.[jpg | gif] ... >
```

where V' is the integer code for the version that is most relevant to the version represented by V and `images` is the directory that contains all the images of the site.

Intensional applet elements like:

```
<applet code=class_name codebase=applets ... >
```

are transformed to:

```
<applet code=class_name.V'.class
```

```
codebase=www_url/applets/class_name ... >
```

where *V'* is same as above and *applets* is the directory that contains all the Java class files of the site and *www_url* is the URL of the www directory of the server. (Absolute URLs must be used to specify codebase attribute values as CGI scripts are unaware of current directory information.)

Intensional includes like

```
<include file=page_path ... >
```

are handled by invoking the server-software which finds the most relevant include file and runs the CGI script which in turn processes the IHTML template and prints the HTML contents. The intensional elements contained in include files are processed in the same way, except that #prev_page# tags are replaced by the URL of the previous page of the first page that includes this file. Since includes can be nested, previous page information should be forwarded through the include chain.

Transversion links like:

```
<a href=page_path language=english> ... </a>
```

are transformed to:

```
<a href=cgi-bin/script_name.cgi/page_path,V,M> ... </a>
```

where V is the integer code for the current version and M is the integer code for the dimension modifier *language=english*.

Transversion image elements like:

```
<img src=images/image_name ... color=256 ... >
```

are transformed to:

```
<img src=images/image_name/image_name.V'.[jpg | gif] ... >
```

where *V'* is the most relevant version to the version formed by setting the color dimension of the current version to *256*.

Extensional links, images and applet elements in IHTML source files are transformed to conventional HTML links, images and applet elements by replacing `xhref`, `xsrc` and `xcode` with `href`, `src` and `code` respectively.

Intensional links with a `from` attribute:

```
<a href=page_path from=current_page_path>
```

are transformed to:

```
<a href=cgi-bin/script_name.cgi/page_path,V,current_page_path>
```

where, *V* is the integer code for the current version and *current_page_path* is the name of the page containing the link, which will be used to replace any `#prev_page#` occurrences in the target page. Transversion links with `from` tags are transformed to similar links which include the additional parameter *M* between *V* and *current_page_path* parameters.

The base URL and WWW root directory settings, name of the server-side executable and the default version code are configurable from the header of the script allowing the use of relative paths and URLs in IHTML source files.

The script is 539 lines of Perl code and it uses two Perl files; *dimensions.pl* and *modifiers.pl*, which are explained below.

Dimensions.pl functions as an include file for both the CGI script and the server-side software. IHTML authors can define the dimensions they want to support in this file. The file consists of a Perl array definition of the following format:

```
@dimension_list = (
        'dimension_identifier1',
        'dimension_identifier2',
        ...
        'dimension_identifierN'
```

```
);
```

To distinguish between dimensions and other keyword attributes, the CGI script does a linear search for each attribute name (specified in an intensional or transversion element) in `dimension_list` array. If the search fails, the script assumes that it is a conventional HTML keyword attribute and leaves the attribute name-value pair as it is. If the pair is not a valid conventional HTML attribute then it is ignored by the browser. Hence, if a new dimension identifier (an identifier that is not included in `dimension_list` array in dimensions.pl) is used in a transversion element, then the name-value pair is ignored and left as it is by both the CGI script and the browser.

Modifiers.pl is another Perl file that consists of an array of strings which contain the dimension modifiers used in all IHTML source files of the site:

```
@modifiers_list = ([“dimension_list[i] = ... “,]*);
```

where 0 < i < N-1, N being the size of the `dimension_list` array. Duplicates are not allowed in the `modifiers_list` array to make sure that integer codes used for the dimension modifiers are unique. Modifiers.pl is created automatically by a Perl script, *crtmdfrs.pl,* which recursively scans through all IHTML files under a given directory looking for transversion elements which has any valid modification(s) to the dimensions specified in dimensions.pl. In this scheme, when the CGI script comes across a transversion element, it uses its position in the `modifiers_list` array as the integer code, M, and use it as part of the extra path information of the transformed transversion link. In case of multiple modifiers in a transversion element, the script separates the integer codes with “+” symbols.

For example, the transversion element:

```
<a href=page_path language=english background=image%sky
        display=text>
```

would be transformed into:

```
<a href=cgi-bin/script_name.cgi/page_path,V,M1+M2+M3>.
```

where *M1*, *M2* and *M3* are the indices of `language=english`, `background=image%sky` and `display=text` in the `modifiers_list` array, respectively.

This scheme hides the internal representation details from the user and the dimension modifiers are passed to the CGI script in a concise way preventing excessively long URLs. Also, the alternative of this approach, using the dimension modifiers as part of the parameters would involve decoding inside the CGI script since some of the symbols of the version algebra would be automatically encoded. The only disadvantage of my scheme is that the author has to run *crtmdfrs.pl* every time a new IHTML source (containing new dimension modifiers) is included to the site.

As far as security is involved, the CGI script eliminates the common security pitfalls of CGI programming by making sure that the only input from the users can be in the form of clicking on hyperlinks. Most of the security problems with CGI scripts arise when untrusted data is passed to a program executed by the CGI script. In my case, the parameters passed to the server-side software are created by the CGI script on the fly and the user has no way of interfering or modifying the data.

4.4 A Sample Intensional Site

A multiple versioned site using the currently available IHTML features can be reached at URL:

```
http://lucy.uvic.ca/~taner/cgi-bin/scan.cgi
```

At first sight, it looks like a fairly normal home page. A simple ticker applet displaying a welcome message and basic information about the site, some links to related pages with my biographical details, resume, course work, and my favorite bands and beers. How-

ever, at the bottom of each of these pages is a link anchored to the phrase Turkish version of this *site*. When clicked, the text on the page and the welcome message displayed by the applet changes from English to Turkish.

The words are well chosen: if we proceed to explore the site again we find the resume, the course work, the bands and so on, but all these pages are in Turkish. We are now in the Turkish version of the whole site, not just of the home page. At the bottom there is an anchored phrase containing the words *ingilizce versiyonu* and, not surprisingly, it leads us back to the English version of the site. Each page also offers us a similar transversion to the text only version (actually, versions) of the site.

Finally, all the pages have links to an *options* page. These links take us to pages which contain the options for changing the version of the site. In the *display:graphics* version of this page thirty small anchored images (background colors/patterns) are displayed. Following, say, the link anchored to the orange square takes us to the home page of a version of the site in which all the pages (including the options page) have orange backgrounds. The *display:text* version of the options page contains only six links (instead of images) to change the background color of the site. Similarly, there are twenty-seven options for the color of the text and the color of the hyperlinks of the site. These options are grouped in the same page since it is guaranteed that all the pages of the site exist in all the possible extensions. This is simply because the IHTML source files have the following body tags:

```
<body <include file=background> <include file=text>
      <include file=link> ... >
```

which include the current version of the `background`, `text` and `link` files, e.g. bgcolor="#FF0000", text="#FFFFFF", link="#FF00FF". Transversion includes can be used to force specific values to be used for any of the body attributes when the exact value is not known. For example:

```
<body <include file=background background=red> <include file=text>
    <include file=link> ... >.
```

would force the background color to be red regardless of the contents of the current version of the background file. Alternatively, the author can use a hard-coded value if he/she knows the hexadecimal representations:

```
<body bgcolor="#FF0000" <include file=text> link="#FF00FF" ... >.
```

Note that to be able to version the body attributes in this manner, all the extensional values for these attributes must be supplied in separate files labeled with proper version codes.

A sample IHTML template and snapshots of some of the versions of HTML pages it supports are included in Appendix D.

In the intensional page (surf) which lists a number of links to various sites from different categories, the links are aggregated. All the links are stored as individual files in a directory called *links_a* under *surf* directory. These files are named with their link labels. For example, the file that contains a link to MTV

```
<a xhref=http://www.mtv.com/>MTV</a>
```

is labeled as *mtv.V.ihtml* where V represents the version: *language:english+category:music*. In this page, below the usual transversion links (to English, Turkish or text, graphics versions), there are additional transversion links labeled such as: *humor, movies and music links*, *sports and weather links*, *links from all categories*, *links in ascending alphabetical order*, *links in descending alphabetical order*. When the user clicks on *humor, movies and music links*, the requested version is *language:english + category:humor+movies+music* and all the link files from humor, movies and music categories are displayed in default (ascending alphabetical) order. Similarly when *links from all categories* link is activated all the links from all the categories are displayed. The links

relevant to the current category are ordered by one of the two versions of a Perl script (local to *links_a* directory) labeled with versions *order:ascending* and *order:descending*. However, as shown in Appendix C, these scripts are generic enough to be used by any intensional page without any change.

4.5 Comparisons and Evaluation

The site described above uses a tiny eight-dimensional subset of the version space. The dimensions are *language, display, background, date, category, order, text, and link*. The language and display dimensions have two coordinates each: english turkish and text graphics, respectively. The background dimension has thirty and the last two dimensions have twenty seven coordinates. Four distinct dates are used as the update dates. The category dimension has six coordinates: humor, sports, music, movie, software, weather. The order dimension has two coordinates: ascending and descending. The site consists of thirteen intensional pages each available in all versions - a total of more than 4,5 million virtual HTML pages.

The original one-version HTML program for the site consisted of about 38K bytes divided into 13 different files. The IHTML source consists of 192 files, but most of them, 94, are tiny files - consisting of a single line (like background=image/sky.gif, text="#FF00FF", link="#0000FF", Last update: 1997.01.07 etc.). There is a total of 107K bytes of IHTML source - and most of the extra is Turkish versions of the original English text. These 107K bytes of IHTML source supports a virtual site which would correspond to over 13G bytes of cloned HTML.

The IHTML version is also much more modular. For example, there is exactly one file containing the list of beers, labeled with the standard (vanilla) version. However, some of the movies have alternate Turkish titles, so there are two source files for this information, labeled with *language:english* and *language:turkish*, which are also used for ver-

sions like *display:text*, *display:graphics* with any background image/color, text and link colors. Adding extra dimensions involves minimal changes: modifying the *dimensions.pl* file and creating the source files.

Chapter 5

Conclusions and Further Work

5.1 Conclusions

I have described the principles of an intensional variant of HTML and its implementation. Based on intensional logic, IHTML is a high-level Web authoring language intended to ease the specification of Web sites that can exist in many versions or variants. IHTML source files can be viewed by existing browsers since IHTML code is preprocessed by a CGI script and converted to conventional HTML code.

In addition to supporting all features of conventional HTML, IHTML makes it possible to define a whole family of conventional HTML pages using a single generic template, hence automating the cloning process which would be required otherwise to achieve the same effect. By creating the clones on demand and discarding them after they are no longer needed, IHTML eliminates maintenance problems and offers a space efficient alternative to Web site developers.

The intensional approach described in this thesis makes it possible to version any component of a Web page. Text components of a Web page can be versioned by intensional server-side includes. Furthermore, any element that points to a source file (HTML file, image file, Java class file, sound file, movie file, zip file, doc file, etc.) can be versioned by labeling all the (existing) versions of the source file (using the coding scheme

described) and storing these source files under a directory.

The current implementation can version most of the major Web page components that are used presently. However, there will be new extensions to HTML which will create the need to version these new components as well. Any component that the current implementation is not versioning directly can be versioned by storing all the HTML elements that point to different source files (in conventional HTML syntax) in properly labeled IHTML files which can be included from the Web page that is being versioned. Actually, this method makes more sense especially for applet elements since the name of a Java class file has to be the same with the main class defined in the file and embedding version information into the Java class file name would result embedding version information into the main class definition as well. If for any reason the version codes change, not only the labels of the Java class files have to be modified, but also the file contents (class definitions) have to be modified and the class files have to be compiled again.

The dimensions used in the IHTML version space can be customized, allowing Web authors to apply the intensional approach to variation described in this thesis to any indexable family of pages, even if the pages don't seem to exist in different versions. This allows a wide variety of possible applications for IHTML.

For example, in case of the slide show, the slides are indexed by the natural numbers. By introducing a new dimension called *page_number*, all the slides in the slide show can be considered to be variants of 'the' slide, which can be represented by a single generic IHTML page. This generic template would have headers, footers, logos etc. which would be common to all the slides and include an intensional *body* file, which would vary over the new *page_number* dimension. If we have a number of different slide presentations, we could in turn consider them to be versions of 'the' presentation, and write an even more generic page for all our presentations.

At a university site, the pages for different departments could be produced as ver-

sions of a single generic department page that varies over the *department* dimension. Similarly, the different faculty pages could be treated as a family of pages varying over a *professor* dimension.

A page which changes every day, e.g. that of a newspaper, can clearly be indexed by the set of dates. Using the *date* dimension, we can write generic source which specifies parts of the layout that are invariant. We can extend our scheme by allowing source pages to be labeled by *intervals*, with the understanding that the source is valid for requests whose *date* coordinates lies in the interval. This idea is described in more detail in [9], where it is pointed out that it amounts to treating the Web as a kind of reactive system.

Any site that contains substantial information on a particular subject can present the information in different *levels*. For example, a site can present a programming language to users of different levels (beginner, intermediate and advanced) in different ways. By defining a *level* dimension, the contents of each page and even the structure of the whole site can be tuned to the level of the current user. In the beginners level the whole site might consist of a single page, a very brief overview of the language, whereas in the advanced level different aspects of the language can be described in a number of chapters with references to other sources and so on. IHTML source files can be organized in such a way that source files of higher levels include either whole or parts of the source of lower levels, either by aggregating them in some user specified fashion or using nested includes.

Despite the advantages listed above, there is an inevitable disadvantage of using IHTML. The CGI script must find the most relevant source file for each intensional element (except links) in a page, which represents a performance hit that clients must suffer. Although the number of IHTML templates found per page has been reduced by using a demand-driven approach which eliminates searches if the intensional/transversion links are not activated, IHTML can still be slow when pages exist in a large number of versions. Loading of an intensional page in my demo site might take up to 10 seconds to complete,

whereas the corresponding conventional HTML page takes about 2-3 seconds to load. The server-side software has to open and read *map.dat* to load the hash table so that it can create version codes and find the most relevant file and then it has to download the hash table contents back to map.dat to make sure any new versions and version codes created by transversion elements are stored in map.dat for future use.

5.2 Further Work

The current implementation is only a demonstration version and can be improved in a number of ways.

To solve the performance problem mentioned above, some sort of a caching mechanism can be implemented. However, in case of any change to the (order of) existing versions in the map file, the disk cache should be cleared to avoid inconsistency, since the hash table that contains the versions supported is loaded from the map file every time a search is required and a different order used in loading the table may cause different integer codes for the same version. Furthermore, the possibility that the exact same version of a page being retrieved over and over again is very low in most cases.

The server-side software can be implemented as a daemon, which given a version code and path of a directory would return the most relevant file. This would avoid the use of a local map file and the file sharing and performance problems it brings with. However, a locking mechanism should be used for hash table access and the contents of the table should be saved properly in case of an abnormal termination.

Another possible way of increase in performance is to implement the server-side software as part of the Web server itself. However, implementing all the functionality of a general purpose server is a nontrivial task. Although a basic server would be sufficient as long as only the elements supported by the server are used by IHTML authors, IHTML would no longer handle all the elements of conventional HTML.

The scripts to help Web authors to set their intensional sites, *packv, unpackv and crt-mdfrs.pl* can be grouped in a single application with a graphical user interface.

The server-side software can be re-implemented in Java to eliminate portability problems (which would increase in case of the daemon approach).

I decided to use current technology for viewing multiple versioned sites, but intensional browsers can be implemented that would suit the needs of multiple versioned sites. One major difference of an intensional browser from the conventional ones would be intensional navigation. An example would be the *back* and *forward* buttons of the browsers, which would take users to the previous or next page in the current version. For example, in the case of the slide show, after switching to another version of the site by following a transversion link from the *language:english* version of *slide6* to go to the *language:french* version of *slide6*, if the user clicks on the back button, the *language:french* version of *slide5* should be created and displayed. This effect can also be achieved by including intensional back links in IHTML source in the case of linear sites. (By a linear site I mean that any page has one previous and next page at most.) However, for non-linear sites, a list of pages visited, which can be arbitrarily long, would have to be passed as yet another argument to the CGI script.

The options page can be displayed in a separate window and clicking on any of the transversion links would load the new version of the page in the original window. In this way, going back and forth between pages and the options page can be avoided.

Composing images, just like text, on demand would increase the variety of images used in HTML and save space at the same time. The idea is to represent graphical images as a number of transparent layers stacked on top of each other. In this way intensional text can be incorporated to graphical images. For example, when composing the *language:french* version of an image, the vanilla version of the background layer, which is purely graphics, and *language:french* version of the text layer, which is obviously purely

text, can be used to form the final version of the image. In this way, Web site developers would not have to re-create images with text from scratch to use them in multiple versioned sites.

Bibliography

[1] D. Connolly, "Names and addresses, URIs, URLs, URNs, URCs", W3 web site at http://www.w3.org/pub/WWW/Addressing/Addressing.html, 1990.

[2] S. Gundavaran, *CGI Programming on the World Wide Web*, O'Reilly & Associates, Inc., U.S.A., 1996, p. 1.

[3] J. December and M. Ginsburg, *HTML & CGI Unleashed*, Sams.net Publishing, U.S.A, 1995, p. 383.

[4] NCSA HTTPd Development Team, "Server side includes (SSI)", NCSA HTTPd web site at http://hoohoo.ncsa.uiuc.edu/docs/tutorials/includes.html,1995.

[5] A. A. Faustini and W. W. Wadge, "Intensional Programming," Technical Report-DCS-55-IR, Department of Computer Science, University of Victoria, 1986.

[6] R. Thomason, editor, *Formal Philosophy, Selected Papers of R. Montague*, Yale University Press, new Haven, Conn, 1974.

[7] P. Rondogiannis, *Higher-Order Functional Languages and Intensional Logic*, Ph.D. Dissertation, University of Victoria, 1994, p. 4.

[8] J. A. Plaice and J. Paquet, "An introduction to intensional programming", in Mehmet A. Orgun, Edward A. Ashcroft, editors, *Intensional Programming I*, Singapore:World Scientific, 1996, pp. 1-14.

[9] W. W. Wadge and A. Yoder, "The possible - World Wide Web", in Mehmet A. Orgun, Edward A. Ashcroft, editors, *Intensional Programming I*, Singapore: World Scientific, 1996, pp. 207-213.

[10] W. W. Wadge, "Possible woorlds", in Mehmet A. Orgun, Edward A. Ashcroft, editors, *Intensional Programming I*, Singapore: World Scientific, 1996, pp. 56-62.

[11] Marc F. Rochkind, "The Source Code Control System", *IEEE Transactions on Software Engineering*, SE-1(4), 1975, pp. 364-370.

[12] Walter F. Tichy, "Design, implementation, and evaluation of a revision control system", in *6th International Conference on Software Engineering, Tokyo*, 1982, pp. 58-67.

[13] Walter F. Tichy, "RCS - A system for version control", *Software - Practice and Experience*, vol. 15, no. 7, 1985, pp. 637-654.

[14] P. Neuhaus, "Revision Control System Primer", Web site at http://www.coling.uni-freiburg.de/~neuhaus/manuals/rcs/rcs.html, October 1995.

[15] J. A. Plaice and W. W. Wadge, "A new approach to version control", *IEEE Transactions on Software Engineering*, March 1993, pp. 268-276.

[16] J. A. Plaice and W.W. Wadge, "Reducing the complexity of software configuration", in M.Tchuente, Ed,, Proc. *1st African Colloquium on Research in Computer Science*, Yaounde, Cameroon, October 1992, pp. 85-96.

[17] J. A. Plaice and S. Ben Lamine, "Eduction: a general model for computing", in E. A. Ashcroft, editor, *Intensional Programming II*, Singapore: World Scientific, in press, 1997.

[18] J. A. Plaice and W. W. Wadge, "A UNIX tool for managing reusable software components", *Software Practice and Experience*, vol. 23(9), September 1993, pp. 933-948.

[19] P. Rondogiannis and W. W. Wadge, "A dataflow implementation technique for lazy typed functional languages", Technical Report-LACIR 93-01, Department of Computer Science, University of Victoria, March 1993, pp. 7-8.

Glossary

attribute A property of an HTML element; specified in the start tag of the element.

browser A software program for observing the Web; synonym for a Web client.

CERN Centre Europeen pour la Recherche Nucleaire. The European laboratory for particle physics, where the Web originated in 1989.

CGI Common Gate Interface, a standard for interfacing external applications with information servers such as HTTP.

client A software program that requests information or services from another software application, a server, and displays this information in a form required by its hardware platform.

dimension An identifier, defined by IHTML authors, used to specify a distinct coordinate in which versions of a site may vary. The actual value of the coordinate is specified by a version expression.

dimension modifier An assignment of the form: *dimension_identifier = version_expression,* which sets the value of the dimension specified by *dimension_identifier* to version_expression. Dimension modifiers are contained in trans-

version elements of IHTML.

flock() Applies or removes an advisory lock on an open file.

form HTML element that allows users to fill in information and submit it for processing.

GIF GIF, for Graphics Interchange Format, is a format for storing image files. It is one of only three formats that can appear in-line in an HTML document, the other two being X-Bitmaps and X-Pixelmaps.

hash-consing A technique to convert lists into natural numbers. The idea is to store a list in a hash table as a pair (head, position of tail). The list is then represented by the position of the tuple in the table.

hotspot The region of displayed hypertext that, when selected, links the user to another point of the hypertext or another source.

HTML Hyper Text Mark-up Language.

HTTP HyperText Transfer Protocol, or HTTP, is the native protocol of the Web, used to transfer hypertext documents.

HTTPD The daemon that WWW servers run to talk with clients using HTTP.

hypermedia Hypertext that may include multimedia; text, graphics, images, sound, and video.

hypertext Text that is not constrained to a single sequence for observation; Web-based hypertext is not constrained to a single server for creating meaning.

intensional include An intensional variant of a server-side include which inserts the contents of the corresponding version of the target IHTML file (after transforming into conventional HTML) into the virtual HTML document under construction.

internet The cooperatively run, globally distributed collection of computer networks that exchange information via the TCP/IP protocol suite.

internet resources The collection of data, documents, and databases available on the Internet.

JPEG From Joint Photographic Experts Group, also known as JPG, JPEG is another image format. In general, JPEG allows for higher-quality images than GIF.

MIME Mime, or Multipurpose Internet Mail Extensions, is a scheme for allowing electronic mail messages to contain mixed media (sound, video, image, text). The World Wide Web uses the MIME content-type to specify the type of data contained in a file or being sent from an HTTP server or client.

NFS The Network File System, or NFS, allows a client workstation to perform transparent file access over the network.

NCSA National Center for Supercomputing Applications. At the University of Illinois at Urbana-Champaign; developers and distributors of NCSA Mosaic.

page A single file of hypertext mark-up language.

server A software application that provides information or services based on request from client programs.

site A file section of a computer on which Web documents (or other documents served in another protocol) reside; for example, a Web site, a Gopher site, an FTP site.

SSI Server Side Include, a directive placed in HTML documents to execute programs, insert files into current document or output environment variables and file statistics.

TCP/IP Transmission Control Protocol/Internet Protocol. TCP/IP is the basic communication protocol that is the foundation of the internet. All the other protocols, such as HTTP, FTP, and Gopher, are built on top of TCP/IP.

transversion link An intensional link that modifies (as specified by dimension modifiers of the link) the current version of the site by creating a link to a different version of the target page.

version A variant of a Web document which is specified by an arbitrary number of dimensions.

version code The hash-cons value (see hash-consing) of the list that represents the version.

version expression An expression which specifies the value of one of the many dimensions that make up a version.

Web (World Wide Web) A hypertext information and communication system popularly used on the Internet computer network with data communications operating according to a client/server model. Web clients (browsers) can access multiple protocol and hypermedia information (where appropriate multimedia helper applications are available for the browser) using an addressing scheme.

web server Software that provides the services to web clients.

Appendix A

Major Data Structures

Version Expression:

```
typedef struct vstruct {
    int vtype;
    union {
        struct vstruct *sub_expression;
        char *label;
    } node1;
struct vstruct *node2;
int numeric;
} *expression
```

Dimension Multiplier Sum:

```
struct dim {
    char* name;
    char* value;
};
```

Hash Table Entry:

```
struct entry {
    struct dim dimension;
    int tailcode;
    int head;
};
```

Appendix B

Version Comparison Function

This function combines LESS_EQ, MORE_EQ, EQUAL and INCOMPARABLE predicates and returns one of the four possibilities; VS_LESSSPECIFIC, VS_MORESPECIFIC, VS_EQUAL, VS_INCOMPARABLE. The first two parameters c1 and c2, are the integer codes that represent the versions to be compared. The last parameter, status, keeps track of the current status of the comparison, and should be set to VS_EQUAL for the first call.

```
int ve_compare(c1, c2, status)
    int c1, c2, status;
{
    v_expr v1, v2;
    int diff;
    if (!ve_validcode(c1) || !ve_validcode(c2))
        er_die("ve_compare", "Code out of range", 0);
    if (c1 == c2)
        return(status);
    if (c1 == 0 && c2) {
        if (status == VS_MORESPECIFIC)
            return(VS_INCOMPARABLE);
        else
            return(VS_LESSSPECIFIC);
    }
```

```
if (c1 && c2 == 0) {
    if (status == VS_LESSSPECIFIC)
        return(VS_INCOMPARABLE);
    else
        return(VS_MORESPECIFIC);
}
if (c1 && c2)
    if (strcasecmp(table[c1].dimension.name,
        table[c2].dimension.name) < 0) {
        if (status == VS_LESSSPECIFIC)
            return(VS_INCOMPARABLE);
        else
            return(ve_compare(table[c1].tailcode, c2,
                VS_MORESPECIFIC));
    }
    else if (strcasecmp(table[c1].dimension.name,
            table[c2].dimension.name) > 0) {
            if (status == VS_MORESPECIFIC)
                return(VS_INCOMPARABLE);
            else
                return(ve_compare(c1, table[c2].tailcode,
                    VS_LESSSPECIFIC));
    }
    else {
        /* get the values of the dimensions and compare them */
        v1 = (v_expr) vp_parse(table[c1].dimension.value);
        if (vp_check(v1))
            v1 = (v_expr) vp_canon(v1);
        else
            er_die("ve_compare", "Invalid version expression",0);
        v2 = (v_expr) vp_parse(table[c2].dimension.value);
        if (vp_check(v2))
            v2 = (v_expr) vp_canon(v2);
        else
            er_die("ve_compare", "Invalid version expression",0);
        diff = vu_compare(v1, v2);
```

```
            if (diff == VS_EQUAL)
                return(ve_compare(table[c1].tailcode,
                    table[c2].tailcode, tatus));
            else if (diff == VS_MORESPECIFIC)
                if (status == VS_LESSSPECIFIC)
                    return(VS_INCOMPARABLE);
                else
                    return(ve_compare(table[c1].tailcode,
                        table[c2].tailcode, VS_MORESPECIFIC));
            else if (diff == VS_LESSSPECIFIC)
                if (status == VS_MORESPECIFIC)
                    return(VS_INCOMPARABLE);
                else
                    return(ve_compare(table[c1].tailcode,
                        table[c2].tailcode, VS_LESSSPECIFIC));
            else
                return(VS_INCOMPARABLE);
        }
    /* to avoid compiler warnings */
    return(status);
}
```

Appendix C

A Perl Script for Aggregation

The following script aggregates the relevant files by creating an unordered IHTML list. The items of the list are the contents of the files it receives (in receiving order). To change the order of aggregation, the array that contains the names of the files, @files, can be sorted by "sort @files" for ascending or "reverse sort @files" for descending order.

```
#! /public/bin/perl
#
if (@ARGV) {
        @files = split(/\+/, $ARGV[0]);
        $path = $ARGV[1];
        $version = $ARGV[2];
        $script = $ARGV[3];
        $from = $ARGV[4];
}
else {
        die "Parameters required\n";
}
$| = 1;
# use process identifier $$ in temporary file's name
$tempfile = "temp.$$.ihtml";
```

```
open(TEMP, ">>$tempfile");
# turn the buffering off to avoid data loss
select((select(TEMP), $| = 1)[0]);
print TEMP "\n<ul>\n";
# append all the relevant files in the order received
foreach $file (@files) {
        # append file contents into the temporary file
        print TEMP "<li>";
        open(F, $path . $file) ||
              die "Can't open ", $path . $file , "\n";
        while (<F>) {
              print TEMP;
        }
        print TEMP "\n";
        close(F);
}
print TEMP "</ul>\n";
# process the aggregate file by the CGI script
system($script, $tempfile, $version, $from);
#remove the temporary file
unlink($tempfile);
```

Appendix D

A Sample Intensional Page

The following IHTML code together with the code included supports 87,480 (30 backgrounds, 2 languages, 2 types of display, 27 different text and link colors) virtual HTML pages, four of which are shown below:

```
<ihtml>
        <head>
                <include file = /homepage/work/head/ >
        </head>
        <body
                <include file = /homepage/backgrnd/>
                <include file = /homepage/text/>
                <include file = /homepage/link/> >
                <include file = /homepage/work/body/>
                <include file = /homepage/footer/>
                <include file = /homepage/update/ date="1997.01.09">
        </body>
</ihtml>
```

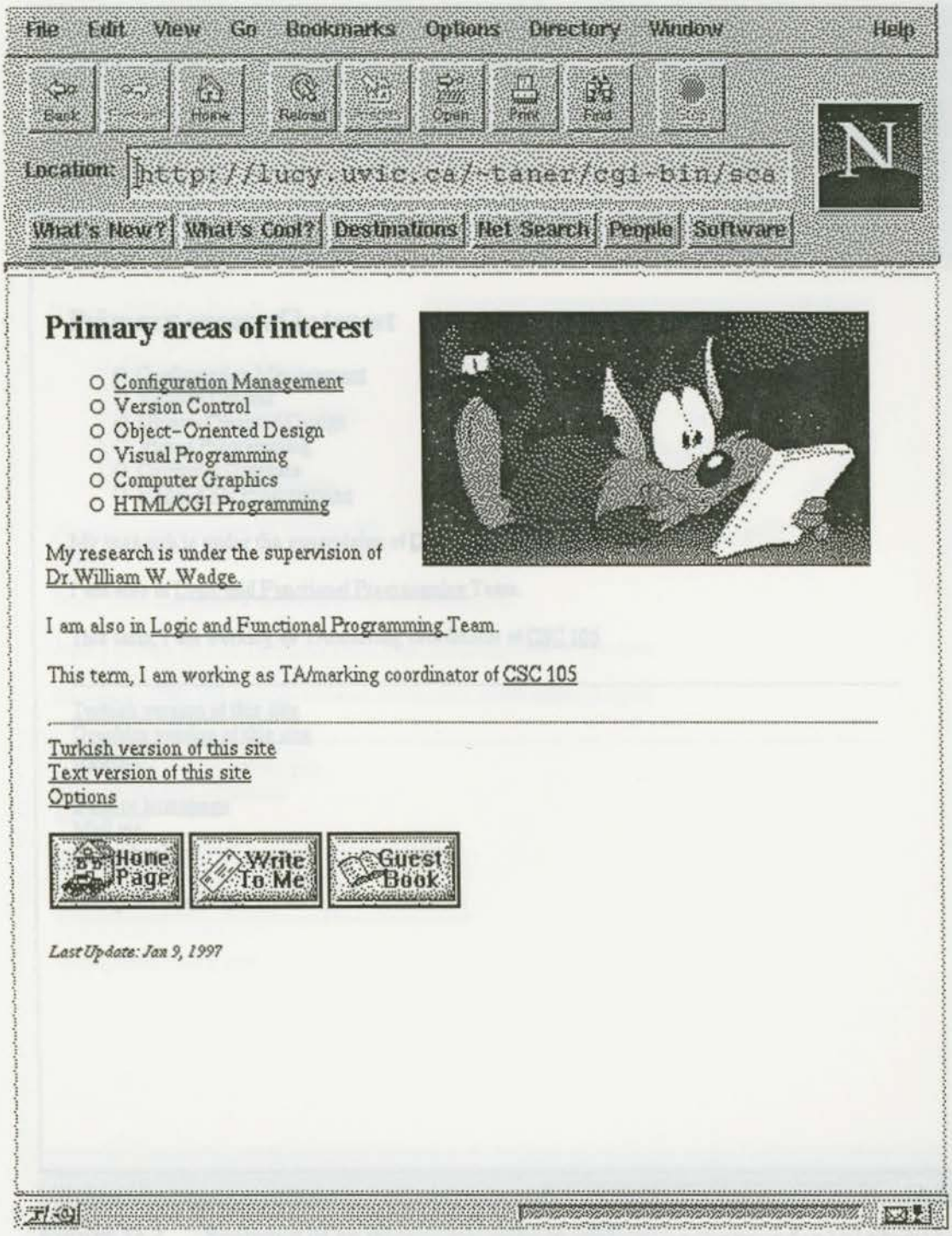

Figure D.1 Snapshot of an intensional page in version: `background:plain%white + date:1997.01.09 + display:graphics + language:english`.

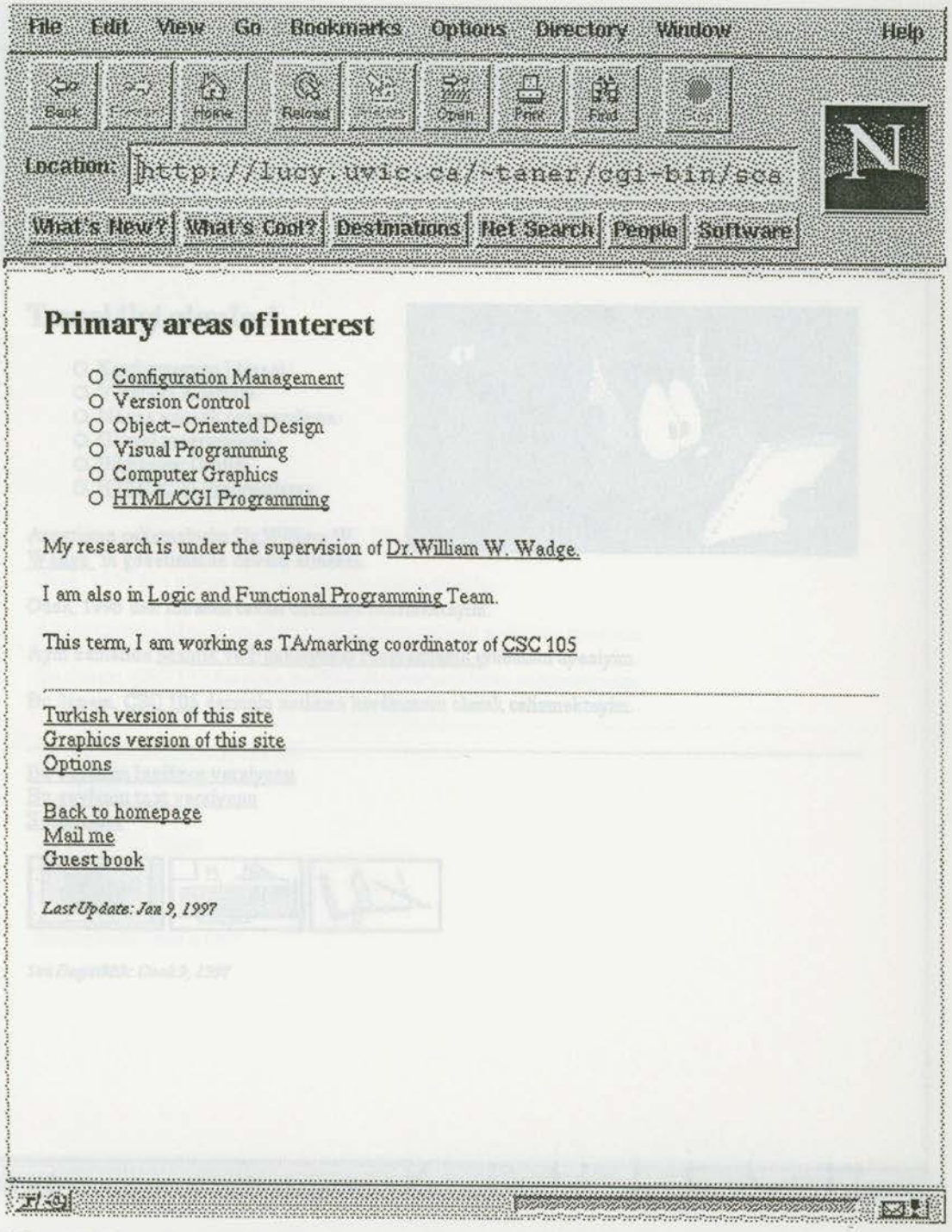

Figure D.2 Snapshot of an intensional page in version: `background:plain%white + date:1997.01.09 + display:text + language:english`.

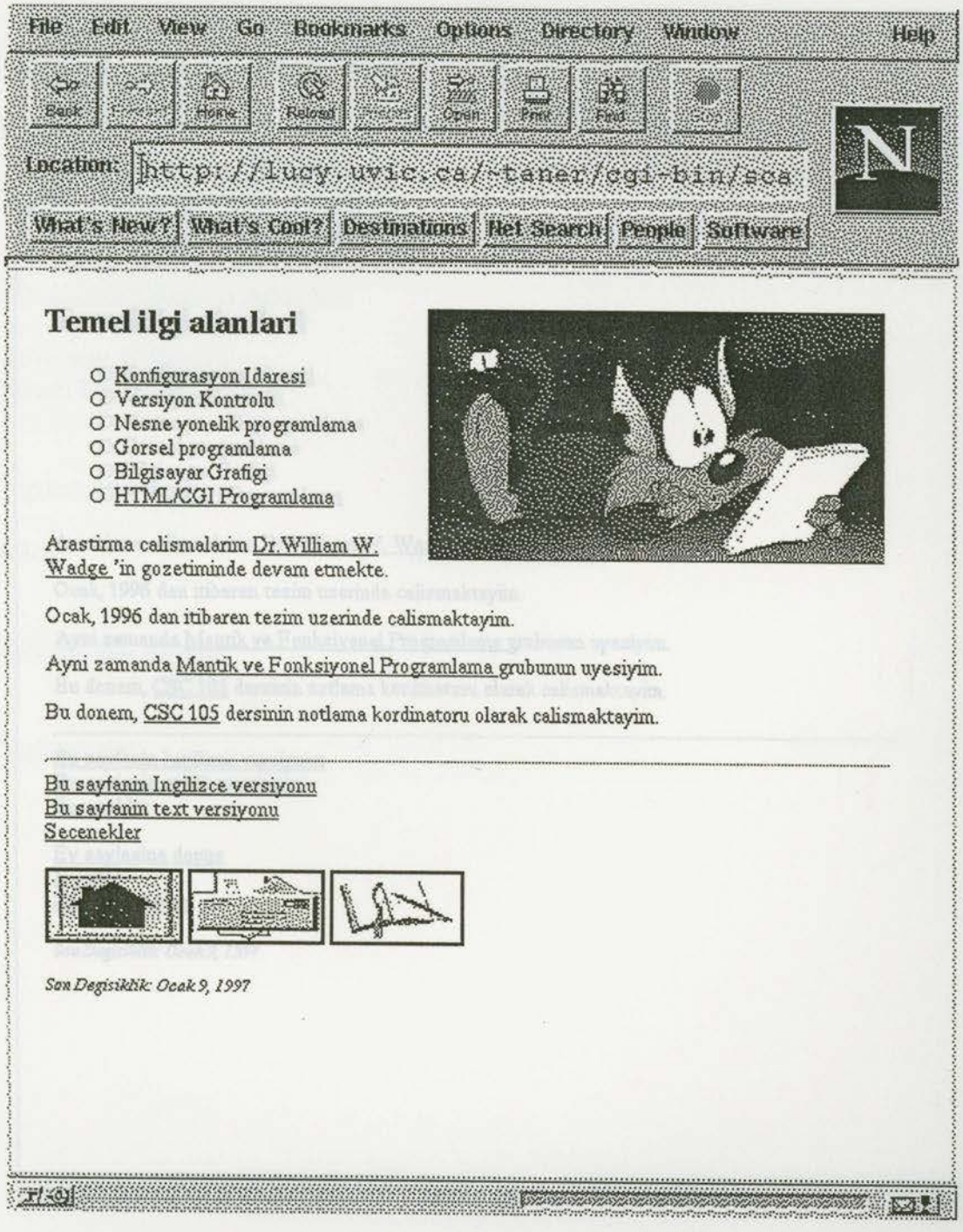

Figure D.3 Snapshot of an intensional page in version: `background:plain%white + date:1997.01.09 + display:graphics + language:turkish`.

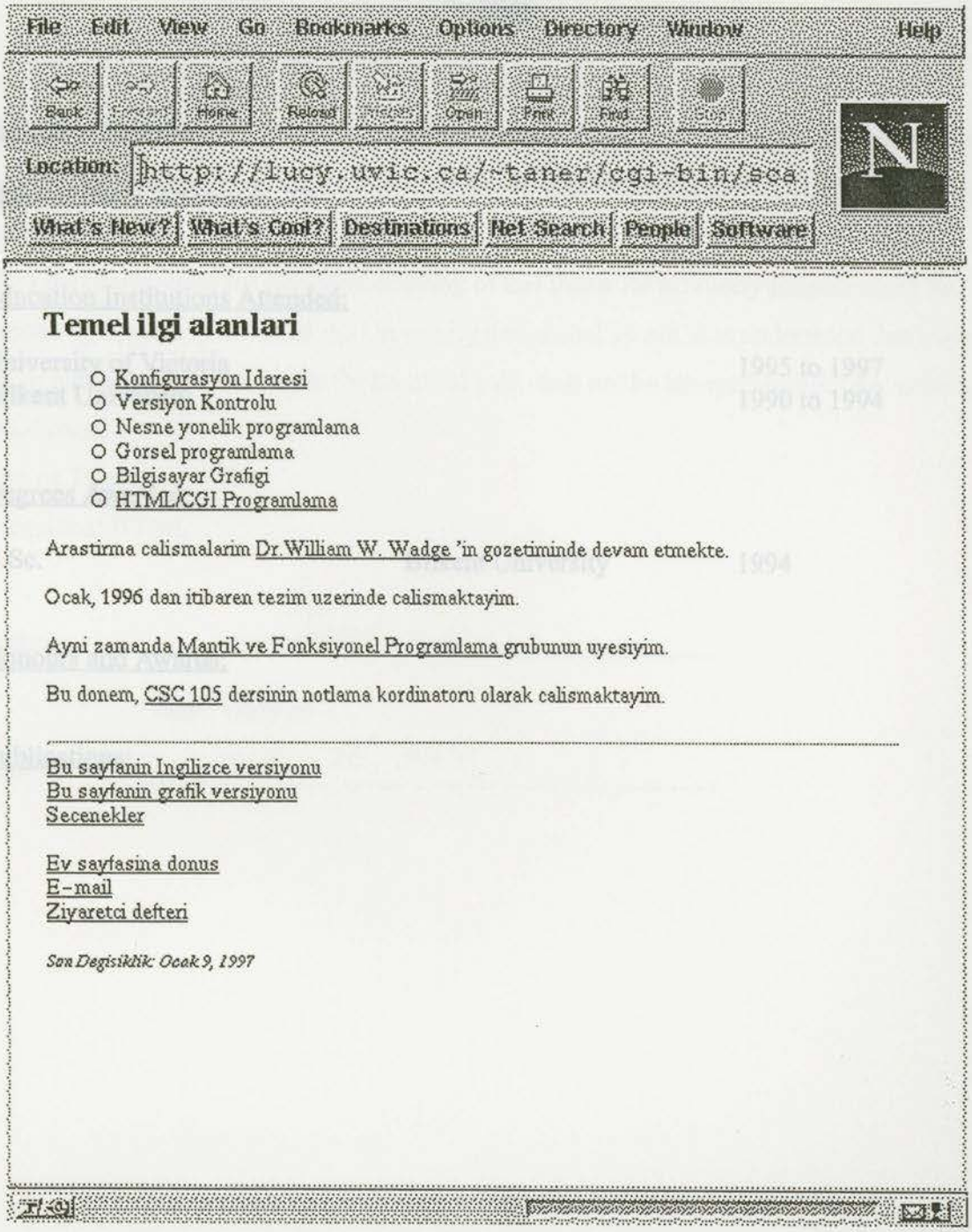

Figure D.4 Snapshot of an intensional page in version: `background:plain%white + date:1997.01.09 + display:text + language:turkish`.

Vita

Surname: Yildirim Given Names: Taner

Place of Birth: Adiyaman, TR

Education Institutions Attended:

University of Victoria	1995 to 1997
Bilkent University	1990 to 1994

Degrees Awarded:

B.Sc.	Bilkent University	1994

Honours and Awards:

Publications:

Partial Copyright Licence

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library from any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

Intensional HTML

Author 

Taner Yildirim

Date June 26, 1997