Intensional HTML 2: A Practical Approach

By

Gordon David Brown
B.Sc., University of Victoria, 1990

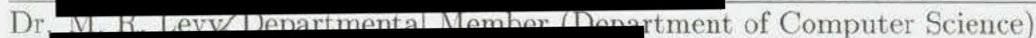
A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

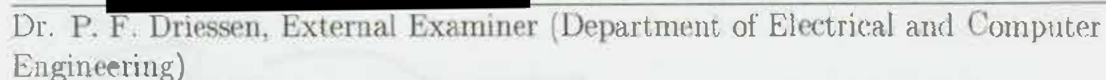
We accept this thesis as conforming
to the required standard

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



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

dr. m. c. shraefer, Departmental Member (Department of Computer Science)



Dr. P. F. Driessen, External Examiner (Department of Electrical and Computer Engineering)

© Gordon David Brown, 1998

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.

Supervisor: Dr. W. W. Wadge

ABSTRACT

Intensional HTML addresses the problem of providing a multi-versioned Web site, without the penalties usually associated with large-scale copying and modification of pages (the usual approach). IHTML uses two key ideas. First, the server uses a *best-fit* algorithm to find the appropriate file when some version of a page is requested. If the exact version requested is not present, the closest match is used. Second, anchors are interpreted as being generic (or *intensional*, as we will say): an anchor does not link to a particular page, but to a family of versions of a page.

The first version of IHTML demonstrated that the idea can work, but was difficult to use. IHTML 2 addresses most of the limitations, resulting in a system which ordinary users can use successfully. Reliability and performance are improved, the syntax is cleaned up, and several new features are added.

Examiners:

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

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

dr. m. c. shr[illegible], Departmental Member (Department of Computer Science)

Dr. P. F. Driessen, External Examiner (Department of Electrical and Computer Engineering)

CONTENTS

CONTENTS . . . iii

LIST OF TABLES . . . vi

LIST OF FIGURES . . . vii

ACKNOWLEDGMENTS . . . viii

1. Introduction . . . 1
   - 1.1 Terminology . . . 3
   - 1.2 Overview of Chapters . . . 4
2. Background . . . 6
   - 2.1 Intensional Logic . . . 6
   - 2.2 Intensional Programming . . . 7
   - 2.3 Version Control . . . 8
   - 2.4 The World Wide Web . . . 9
   - 2.5 Intensional HTML . . . 10
     - 2.5.1 Versions . . . 13
     - 2.5.2 The Best Fit Algorithm . . . 13
     - 2.5.3 Trans-version Links and Includes . . . 14
     - 2.5.4 Other Features . . . 15
     - 2.5.5 Limitations . . . 15
3. Practical IHTML . . . 18
   - 3.1 Versions and Version Modifiers . . . 18
     - 3.1.1 Version Comparison . . . 20
       - 3.1.1.1 Canonical Form . . . 21
       - 3.1.1.2 Comparison Algorithm . . . 22
   - 3.2 Version Representation in URLs and Files . . . 23
     - 3.2.1 Version Encoding . . . 24
     - 3.2.2 Versions in URLs . . . 25
     - 3.2.3 Storing Versioned Objects on Disk . . . 26
   - 3.3 Default URLs . . . 26

3.4 Absolute Version Specifications 27
3.5 Selective Inclusion of Text 28
3.6 Dimension Variables 29
3.7 Site Construction Utilities 30

4. Implementation 32
4.1 The Apache Module 32
4.2 URL to Filename Translation 34
4.3 IHTML to HTML Translation 37
4.4 Site Development 39

5. Related Web Technologies 42
5.1 The Common Gateway Interface 42
5.2 Apache's Server-Side Includes 43
5.3 Java and JavaScript 45
5.4 Cascading Style Sheets 46
5.5 The Extensible Markup Language and the Extensible Style Language 47
5.6 Dynamic HTML 49

6. Future Work 51
6.1 Applets 51
6.2 Enhanced Version Language 53
6.3 Aggregation 56
6.4 Version Manipulation 57
6.4.1 Javascript and IHTML 58
6.5 IHTML Editor 58
6.6 Intensional Browser 58
6.7 Non-IHTML Applications 59
6.8 Stand-alone IHTML-to-HTML Translation 59

7. Conclusions 61
7.1 User Experiences 61
7.2 Additional Advantages 62
7.3 Summary 63

Bibliography 65

A. IHTML Reference Manual 68

A.1 Overview 68

A.1.1 Best Fit 69

A.1.2 Intensional Links and Includes 69

A.1.2.1 Trans-version Links and Includes 70

A.1.3 Server Operation 70

A.2 Versions 71

A.3 IHTML Syntax 72

A.3.1 Modified HTML Tags 73

A.3.2 Text Substitution Tags 74

A.3.3 Document Structuring Tags 76

A.4 Site Construction Utilities 77

A.5 Common IHTML Techniques 78

A.5.1 Cascading Levels of Detail 78

A.5.2 Trans-version Link Files 80

A.5.3 IHTML and JavaScript 81

A.6 Installation Instructions 81

A.6.1 Unpacking and Preparation 82

A.6.2 The Apache Module 83

A.6.3 The IHTML Library 84

A.6.4 Installing the Site Construction Utilities 84

A.7 The Local Installation 84

B. Version Encoding 85

B.1 Arithmetic Coding 85

B.2 Binary to Text Conversion 87

C. Collected Version Algebra Axioms 88

LIST OF TABLES

B.1 Probabilities and ranges for a sample input alphabet 86

B.2 Translation table: 6-bit numbers to characters 87

LIST OF FIGURES

2.1 The World Wide Web and Intensional HTML's place in it 10

3.1 Finding the best fit for version `a:b+c:3` 22

4.1 Server-Side Operations 32

5.1 Versioned inclusion via the Apache SSI 44

B.1 Arithmetic coding example for the string *abceb0* 86

ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. W. W. Wadge, for his support, both financial and intellectual (as well as for struggling through the beta-testing of IHTML 2). Several others also provided valuable feedback on the software: dr. m. c. shraefel, Dr. P. F. Driessen, Paul Swoboda, and several members of Dr. Wadge's "Dataflow Programming" course.

I must also acknowledge the contributions of the Dr Pepper Company and PowerBar Foods Canada, Inc. Without the ongoing nutritional support of these companies' products, this work would never have been completed.

Lastly, I would like to thank my family and friends for their ongoing moral support and encouragement.

1. Introduction

Intensional HTML (IHTML) was developed to reduce the effort required to develop multi-versioned Web sites. Wadge, et al. argue convincingly in [1] that multi-versioned sites are useful: site developers may wish to present multiple versions of their information for many reasons: multiple languages, expertise levels, levels of subscription (for paid sites), geographic location, and so on. However, there are difficulties with maintaining several versions of a site. Different versions quickly diverge in more than the appropriate ways, as fixes are made to one version but not another. As new ways that a site can vary are required (what we will call *dimensions*), the number of versions which must be maintained may increase dramatically, making the maintenance problem that much worse. The consequence of these problems is that very few multi-versioned sites exist.

Intensional HTML solves the problem by allowing the site developer to ensure that only those parts of a page which actually vary from version to version are distinct. Content that is the same from version to version uses the same source, so changes can be made once to affect all versions to which they are applicable. Two key ideas make it all possible. First, the server uses a *best fit* algorithm to find the page requested by a client: if no version of the requested page is labelled with the exact version requested, the version which most closely matches is used. For this reason, the site developer need not provide separate sources for every possible version of a page, only for those which vary meaningfully. For example, suppose a page is purely graphical: clearly, there is no need for French and English versions. The server will pick the same source file in either case. Second, parts of a page may be included from separate files, and those inclusions use the same best fit algorithm. The idea is to include those parts of a page which vary in dimensions that the whole page need not vary in, thereby eliminating duplication of information. Content which really does vary from version to version, of course, must exist in all those versions. The usual technique is to

have *one* version of the main page, which includes, say, a header, body, and footer. The body may vary in the *language* dimension, but need not vary in the *background colour* dimension. The header might vary in the *background colour* dimension, but not *language*; it might also vary in other dimensions used to set page characteristics. Note that if the header is standardized from page to page, many pages can share (include) the same source file. The footer may vary in yet another dimension: say, the *link map* dimension, which controls which links are available at the bottom of the page. Included fragments may also include subfragments, of course, with no limit on nesting. The key point is that at each level of inclusion (from the main page down), the best fit algorithm is applied with respect to the *requested* version of the main page (possibly modified by the inclusion tag). Even if the page exists in one version only (as it frequently will), the server will look for included files with the requested version of the current file, not the actual one that was found. These two ideas, best fit and inclusion, make it possible to eliminate cloning.

The first implementation of IHTML was a research prototype. It demonstrated that the theory behind IHTML is viable, but proved difficult to use in practice, for several reasons. Installation and use required detailed knowledge of the implementation. Performance was a concern: since it was implemented as a Perl CGI program, every use required loading Perl, as well as the program itself. The penalty on a busy server would be severe. Because it was a CGI program, URLs for versioned pages all began with the same prefix: the path to the CGI program. For these reasons, as well as some minor implementation reasons, the first version was not widely used.

The objective of IHTML 2 is to provide a stable, usable implementation of IHTML, as a platform for further work: experimentation in how best to present multi-versioned information; exploration of what features IHTML should have to make the site implementer's job as easy as possible; and, of course, creation of multi-versioned sites for real-world use. The first two are already happening. The author and others have worked out some useful techniques for displaying versioned information; in the course of the above work, the need for new features became apparent. Use of the new

features led to new ideas for presentation, leading to new features, and so on. Thus far, IHTML 2 has not been used in permanent sites meant for public consumption, but it appears to have what is needed.

There appears to be very little work going on elsewhere related to multi-versioned Web sites. Any number of sites exist in different languages, but cloning is the order of the day, with all the problems mentioned above. Searches of the Web revealed one site in which Welsh and English versions of several pages were provided, with a very rudimentary best fit technique [2]. It uses two Internet hostnames for one machine, one "Welsh" name and one "English" name. The Apache server has different configuration files for the two hostnames, with instructions in each of them to fall back to the other if the page is not found on the requested server. Since the client still sees the original hostname, any relative URLs are still relative to the original server, thereby preserving the user's original choice of Welsh or English. This idea is conceptually very similar to IHTML, but of course extremely limited.

1.1 Terminology

Most of the terminology in this document is quite standard. A few terms are used in particular ways, though, and should be made explicit.

- A *site implementer* or *site developer* is a person who is creating a suite of WWW pages, but (typically) does not administer the HTTP server itself.
- The *site administrator* is the person responsible for maintaining the HTTP server.
- A *customer* is a person browsing a site. The word *user* refers to the site implementer, unless otherwise stated.
- A *client* is a Web browser (i.e. software, rather than a person).
- A *server* is an HTTP server program.

- A *component* or *element* is part of a Web page. It might be included text, an image, a script, or anything else which can be included in a page. IHTML pages tend to be made up of more components than normal HTML pages.

- The abbreviation *IHTML* itself is used in two ways. References to "IHTML" alone usually refer to the whole IHTML system, including server software, utility programs, and all. For example, "IHTML supports versioned CGI programs." "IHTML" may also refer specifically to HTML pages augmented with IHTML extensions: "IHTML pages (or components of pages) may include the `ISELECT` tag." This does not imply that, say, a versioned CGI program may include `ISELECT`. Which meaning is intended should be clear from context.

1.2 Overview of Chapters

Chapter 2 provides some background information which puts this work in context: intensional logic, intensional programming, versioning systems, the World Wide Web, and IHTML 1 are described briefly. Chapter 3 is a complete description of IHTML 2: what it does, why it does it, and how it is different from the original. Chapter 4 describes the design and implementation. Chapter 5 summarizes several other Web technologies and how IHTML relates to them: CGI programming, JavaScript (and ECMAScript), the Extensible Markup Language, the Extensible Style Language, Cascading Style Sheets, and the Document Object Model. Several avenues for future work are described in chapter 6; IHTML 2 is by no means the final word in intensional Web development. Some conclusions are presented in chapter 7, including some discussion of whether IHTML 2 is in fact usable by ordinary mortals.

Appendix A is a complete reference manual for IHTML 2. Note that there is some overlap between chapters 3 and 4 on the one hand, and Appendix A on the other, so that Appendix A will be useful without the entire body of this thesis accompanying it. Appendix B summarizes the arithmetic coding technique for text compression. Although it is not a central part of this work, it is sufficiently interesting

(and so little known) that it deserves inclusion. Appendix C collects the complete set of rules for the version algebra into one place, for easier reference.

2. Background

2.1 Intensional Logic

Intensional logic is the formal underpinning of the version system supported by Intensional HTML, so its origins and main ideas will be described briefly. Rudolph Carnap, in *Meaning and Necessity* [3] among other publications, addressed the problem of formalizing natural languages. He observed that many (perhaps most) natural language expressions are not *statements* in the logical sense: they do not have truth values in and of themselves. They require a context in order to have a truth value assigned. For example, the truth or falsity of the sentence "It is raining" depends on the time and location of the utterance. Prior attempts to formalize this kind of expression involved reformulating them systematically to expressions like "At time T and location X, it is raining", with specific values inserted for T and X. The difficulty with this approach is that it entails denying that many perfectly ordinary expressions mean something in and of themselves. Carnap preferred to recognize that although "It is raining" does not have a single truth value, there is a unifying concept underlying all statements of the form "At time T and location X, it is raining". He called that unifying concept an *intension*.

Others formalized Carnap's work. Bar-Hillel described *indexical expressions*, in the paper of that name [4]. Kripke, Montague, Scott and others went on to formalize these concepts further, moving somewhat away from the original notion of formalizing natural language in the process [5, 6]. Kripke, in particular, defined a formal semantics for modal and intensional logics in terms of an *accessibility* relation [7]; *Kripke semantics* have become the standard for describing the semantics of intensional logics ([8], pages 413–414). The result is a wide variety of formal systems, ranging from very the general dimensional logic with explicit indices defined by Scott to the very restricted modal logic consisting of first-order predicate logic extended with the *necessity* operator ([9], chapter 1).

The notion of *possible worlds*, attributed to Leibniz by Chellas ([9], page 3), Honderich ([10], page 707) and others, is a useful framework for giving semantics to intensional logic. Any state of affairs (i.e. assignment of truth values to statements) which is self-consistent is a possible world. Many variants of intensional logic have been described using possible-worlds semantics. Classical modal logic considers whether a statement is true in all possible worlds, some but not all, or none. Formalization of this classification of statements typically uses intensional operators such as "$\Box$": given an expression A, $\Box A$ means "A is true in all possible worlds". Chellas, in [9], part 1, gives details. Alternatively, a temporal logic may be defined by ordering the set of possible worlds by time and considering $\Box$ to be "true in all future worlds". Goldblatt describes several such logics in [11], chapter 8. More generally, we may consider a partial ordering of possible worlds by some arbitrary accessibility relation, with $\Box A$ meaning "true in all accessible worlds".

For Intensional HTML, the set of possible worlds is the set of possible versions of pages or parts of a page, for some particular site. The accessibility operator is "$\sqsubseteq$", which we read as "refines to": $A \sqsubseteq B$ means that version B is a refinement of A. The refinement operator defines a directed, acyclic graph of possible worlds, i.e. versions, which will be used to find the closest match to a version. The refinement operator will be described in detail in Sections 2.3, 2.5, and 3.1.

2.2 Intensional Programming

Intensional programming arose out of the recognition by Faustini and Wadge that the programming language Lucid could be given a formal semantics using intensional logic [12], although it was originally viewed as a dataflow language [13]. Lucid uses a very general intensional model, as opposed to the more restricted forms of modal logic. Its possible worlds are points in a programmer-defined multidimensional space, where each dimension is indexed by the non-negative integers. It has several intensional operators, allowing access to the value at the next index in a dimension, the previous index, all indices meeting some criteria, and so on. There is not a direct

evolutionary link from Lucid to IHTML, but they are based on the same formal system, and the same notion of demand-driven multidimensional processing. The next section describes in part the somewhat serendipitous path that led to IHTML.

2.3 Version Control

In the process of implementing a Lucid interpreter, Plaice and Wadge found that they needed a source code version control system with more flexibility than the existing tools such as SCCS [14] and RCS [15]. They needed variant control as well as revision control, and a convenient means for merging development streams. *Sloth*, *Lemur*[1], and *Marmoset* were developed to fill this need [16, 17]. They are successive refinements of the basic idea of combining versioned modules using a *best-fit* approach to software configuration (i.e. pick the version of each module which best approximates the requested version of the whole system).

It was only after these tools were in use that the developers realized that they, too, could be modeled as intensional systems, with the set of possible worlds being the set of possible versions. In [17], they define the following language for versions:

$$V ::= \varepsilon \mid N \mid x \mid V\%V \mid V\&V \mid V+V \tag{2.1}$$

$$N ::= n \mid N.n \tag{2.2}$$

where "ε" is the least element, or most generic version, n is a non-negative integer, "%" is a *subversion* operator, "&" is the greatest lower bound, "+" is the least upper bound, and "." indicates numeric subversions. For example, `mac+english+3.1` might describe one version of a particular software product; `unix+2.5` might describe another.

These tools incorporate a variety of ideas relevant to version control, but two in particular are used almost without change in IHTML: the notion of a set of versions of a file, and the use of a refinement ordering on versions in a best-fit approach to

[1] Lemur was implemented by the author.

select the appropriate version when constructing a complete system. The application of these ideas to IHTML is described in Section 2.5.

2.4 The World Wide Web

Although the World Wide Web really needs no introduction, a brief description here will help in placing the subsequent description of Intensional HTML in context. Broadly speaking, then, the Web may be considered a world-wide collection of documents, with globally unique names, a standardized protocol for access, and standard markup tags for rendering. More specifically, the Web includes the following components:

- Documents identified by globally unique addresses. They may be text, marked up with *hypertext markup language* (HTML) or not, graphics, sounds, programs (applets), or almost anything else which can be described with a MIME (*m*ultipurpose *i*nternet *m*ail *e*xtension) type. HTML documents in particular may include *anchors* which, upon command, direct the user to another (presumably related) document.
- A standard addressing scheme for documents: the *uniform resource locator* (URL). This scheme achieves global uniqueness by including the internet address of the machine providing the document, which is already globally unique.
- A standard protocol, the *hypertext transfer protocol* (HTTP), for retrieving the above documents. This works on a fairly ordinary client-server model, with server processes listening on well-known port numbers for connections from client processes.
- HTTP servers running on the computers which provide WWW documents, delivering documents to clients anywhere in the internet, on request.
- WWW browsers, which request documents from the servers, and render them according to the HTML tags they contain (if they are HTML text), or by other

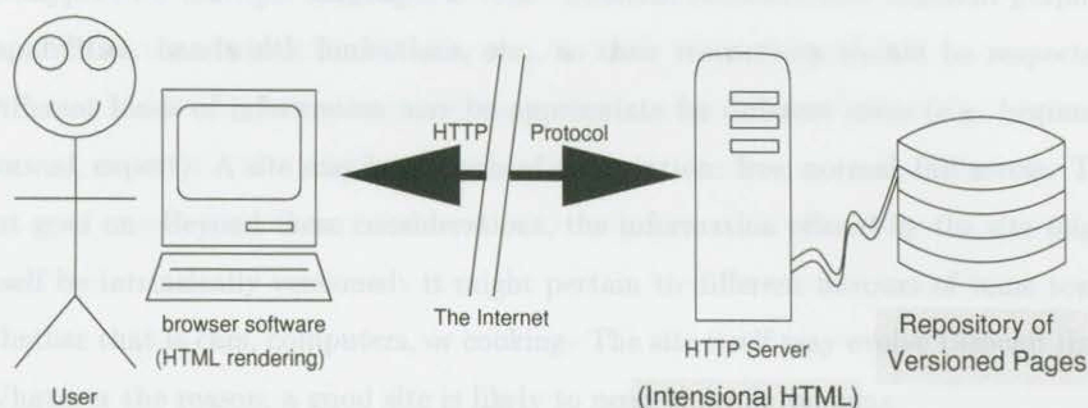


Figure 2.1: The World Wide Web and Intensional HTML's place in it

means, according to their MIME types.

Figure 2.1 shows the relationships among these components, and Intensional HTML's place in the Web. The important point is that it fits into the existing structure without changes to anything other than the versioned documents and the HTTP server itself. Consequently, Intensional HTML can be offered by a site without any change to anything outside of the site administrator's control.

Wadge and Yoder point out in [18] that the Web itself may be considered an intensional programming system. The set of all WWW documents is an intensional "uber-document", with URLs as the indices identifying the extensions (i.e. particular Web pages). HTML links, of course, are *trans-version* links to transport the user from one extension to another. Using this model, the version of an intensional page is a sub-version of the version denoted by the URL. From the immediate, practical perspective of providing multi-versioned sites, it may not make a difference. However, it is worth noting that the two notions of intension with respect to the Web can mesh sensibly.

2.5 Intensional HTML

That things frequently exist in many different versions is, the author hopes, uncontroversial. Wadge et. al. argue in [1] that the problem is particularly acute with respect to Web pages. People may browse one's site from anywhere in the world,

so support for multiple languages is vital. Different browsers have different graphics capabilities, bandwidth limitations, etc., so their restrictions should be respected. Different kinds of information may be appropriate for different users (e.g. beginner, normal, expert). A site may have levels of subscription: free, normal, full access. The list goes on. Beyond these considerations, the information offered by the site might itself be intrinsically versioned: it might pertain to different flavours of some topic, whether that is cars, computers, or cooking. The site itself may evolve through time. Whatever the reason, a good site is likely to need multiple versions.

The quick and easy way to present a multi-versioned site is to clone some original version, then make the necessary changes. Although it may seem to work in the short term, this approach quickly leads to trouble. It becomes impossible in practice to maintain the site in any consistent way. The maintenance problem has been documented extensively in the software development world, as can be seen from the number of publications addressing the problem ([14], [15], and many others). The same issues confront the Web site developer.

The solution is to ensure that there is only one copy of the parts that pages have in common, whether that is the overall format of a page, an image, a link, or some fragment of text. However, standard HTML does not support this approach. Consider a site with beginner, normal, and expert versions. Links from a particular version of each page should lead to the same version of other pages, and so on. Suppose a customer browses the site, choosing the expert level. Links she sees on a given page are then links to the expert version of other pages, and so on. Now consider a page which, by chance, presents exactly the same information for all three versions. The site implementer must provide three versions of this page, despite the identical content, so that links to other pages will lead to the correct versions of those pages. Similar analogies can be made for multi-language slide shows, and so on. Some savings may be made by careful use of the *server-side include* feature offered by some Web servers, but much duplication cannot be eliminated in this way.

Intensional HTML solves the problem with two innovations. The first is that

site implementers need not provide source files for every possible version of a page (or component of a page). If the requested version is not available, the server uses a *best fit* matching algorithm to find an appropriate page. With regard to the identical-content page described in the previous paragraph, if the site implementer provides a single, *vanilla* (i.e. default) version of the page, it will be chosen no matter what version is requested (beginner, expert, etc.). The second is that links, inclusions, and so on in pages are considered to be *intensional*: they are links to families of versions, not to particular source files. Which page is reached via a link (or which component is included) depends on the requested version (henceforth usually referred to as the *current* version) of the page being viewed, as well as the attributes in the tag. Again with respect to the previous example, a link such as

```
<A HREF="carburators.html">carburation</A>
```

on the vanilla-only page is a link to the same version of `carburators.html` as was requested for the current page. Our expert customer finds that this link reaches the expert's page on carburation. Other customers, browsing the site in beginner mode, find that it reaches the beginner's introduction to carburation, and so on. Note that they all see the same source file, not a clone.

In the example above, the benefit of IHTML may not be especially apparent. A single page is reduced from three versions to one. However, the benefits increase significantly as the number of dimensions (and variants within a dimension) increases. Suppose there were five levels of expertise, three possible languages, four subscription levels.... Reducing a page (or part of a page) to one or a few versions could save considerable effort.

As mentioned in section 2.4, Figure 2.1 shows that IHTML requires no change to the HTTP protocol, or the software on the client's side. Intensional HTML pages are translated to standard HTML before being sent to the browser. This is an essential feature; there are too many browsers, outside the control of site developers, to require specialized software. Although most browsers allow plug-in modules for

user-developed extensions, the plug-in interfaces are limited to handling new MIME types, and even then in relatively restricted ways [19].

2.5.1 Versions

IHTML 1 extends the version language given in Section 2.3 by adding explicit dimensions. Experimentation revealed that it might be convenient to distinguish, say, French cooking and the French language. A version is now a disjunction of dimension specifiers:

$$V' ::= \varepsilon \mid D{:}V + V' \tag{2.3}$$

where V is as in Section 2.3, and the D are dimension names.

2.5.2 The Best Fit Algorithm

If it were necessary to create IHTML source for every possible version, there would be little benefit to be had from IHTML. To avoid this problem, IHTML supports a notion of *best fit*. If the requested version of a component does not exist, the server software considers other versions, looking for one which could be refined to the requested version, and which is closer to the requested one than any other existing version (i.e. any other existing version which refines to the requested version also refines to this one). If such a version exists, it is used in place of the requested version. If not, it is an error.

The refinement ordering for versions in IHTML 1 is fully described in [16] and [20]. The IHTML 2 set is fully described in Section 3.1.1. The basic idea is that versions are ordered by the refinement operator $\sqsubseteq$, with rules such as

$$V \sqsubseteq V + V' \text{, and} \tag{2.4}$$

$$V \sqsubseteq W,\, V' \sqsubseteq W \;\rightarrow\; V + V' \sqsubseteq W \tag{2.5}$$

The complete set of rules defines a partial order for versions. Given a requested version R, and a set of existing versions, the best fit is the unique version V such that

$V \subseteq R$ and for all other existing versions W, $W \subseteq V$ or $W \not\subseteq R$. If there is no best fit, it is an error. This definition is exactly the best fit used in the software versioning systems described in Section 2.3.

Once the best fit version is found (supposing there is one), it is translated to HTML before being sent to the client. This step is essentially the same in IHTML 1 and IHTML 2; see Section 4.3 for details.

One key point is that, for purposes of translating IHTML to HTML, the component is treated as if it were the requested version. Intensional links and includes in particular are transformed to links to the requested version of the new page, not to the version of the component which was actually used. For example, if the requested version of a page is `foxtrot:oscar+alpha:delta`, but the best fit is vanilla, links in that page will still be converted to links to the `foxtrot:oscar+alpha:delta` version of the target pages. If it were done otherwise, then a page which existed only in the vanilla version would always include links to the vanilla versions of other pages, which is exactly what we would like to avoid.

2.5.3 Trans-version Links and Includes

The intensional behaviour of links and includes is described above. Although the default behaviour (using the current version) is usually appropriate, it is sometimes necessary to create links to other versions of a page or site (since there is no other way for customers to request arbitrary versions of a page). *Trans-version* links reach other versions of the same or different pages; trans-version includes include other versions of components of a page.

In IHTML 1, trans-version links and includes are specified by adding dimension-value pairs to tags. For example, the following anchor creates a link to the French version of a page:

```
<A HREF="clouseau.html" language=french>The Inspector</A>
```

Dimensions which are not mentioned are unchanged. See Section 2.5.5 for a discussion of this syntax.

2.5.4 Other Features

IHTML 1 provides other features, such as versioned inclusion of images, and aggregation. Images are treated much the way links are: the URL in the tag is modified to include the appropriate version code. Aggregation consists of including *all* relevant versions of a component, rather than just the best fit. It is used for creating lists of things in a flexible way, and so on. Its original motivation was to provide a mechanism for representing information flexibly: levels of detail, expertise level, and so on. Its form was suggested by schraefel (see [21, 22] for details).

2.5.5 Limitations

IHTML 1 demonstrates the power of Intensional HTML, but more work is needed for the system to be usable in real-world situations, by ordinary users. It is not possible to create versioned links to pages in separate version spaces, or links to particular versions of pages, ignoring the current version. Adding new versions requires changes to the configuration of the IHTML software. Aggregation provides no convenient means of ordering aggregated items. Construction of a versioned site requires detailed knowledge of the implementation of the software. Performance is sometimes an issue, since every page must be translated by a Perl CGI program. There are problems with simultaneous access by multiple users. The following few sections describe some of these difficulties in more detail.

Version Encoding IHTML 1 encodes versions as small integers, using an index file to map numbers to actual versions. The benefits of this approach are short version strings, and compact storage of versions (since each textual version is stored only once, and versions which share components can share space as well).

However, there are drawbacks. Since different sites will have different index files, there is no obvious way to create a link from one versioned site to another. This is a non-trivial obstacle to widespread use of IHTML. Also, the index file itself is a single point of failure for a site. Versioned files stored on disk are named with version

codes rather than explicit versions. If the index file is lost or corrupted, there is no way to reconstruct version information, except manual inspection of individual files and hand-reconstruction of the index file. Also, the index file is modified by the server software when a versioned file is served; mutual exclusion for several server processes is both necessary and difficult to implement correctly.

IHTML Syntax The syntax of IHTML 1 has some drawbacks. The most important is that the server-side software must know the set of dimension names in advance. Consider a typical trans-version link:

```
<a href="zork.html" language=french target=top>Zork!</a>
```

The software must treat the `language` attribute as specifying a dimension index, but `href` and `target` as standard HTML attributes. Since HTML-processing software is required to accept and ignore attributes and tags that are not recognized (for forward compatibility), there is no *a priori* way to distinguish between a dimension name and an unrecognized attribute. (The software cannot use a fixed list of tag and attribute names and assume everything else is a dimension, because it will treat unrecognized attributes as dimensions.) Aside from the necessity for the server administrator to configure the set of legal dimensions, this syntax makes it impossible to create a link to a different site, with a different set of dimensions (even if the version encoding allowed it). Even worse, suppose the next version of HTML, whatever it is, includes an optional `language` attribute in the `A` tag?

A less severe but still notable problem is that there is no way to create a link that disregards the current version completely, specifying the target version exactly. The only possibility is to specify the index of every dimension (which in turn is only possible because the set of dimensions is configured at the server level). However, suppose one wants to return to the completely vanilla version. There is no way in IHTML 1 to specify that a dimension's index should be set to vanilla.

Aggregates IHTML 1 provides a form of aggregation, by including every version of a component that can refine to the requested version. The drawback is that there is no ordering on the results. It is unlikely that an arbitrary ordering is acceptable, for most purposes. The quick solution is to provide a script which orders the elements. However, it is unfortunate to require ordinary site developers to write programs to implement aggregates.

Site Construction Construction of IHTML 1 pages requires knowledge of the implementation of the server-side software. The developer must create appropriately named directories, and create the versioned files within them. In particular, she or he must determine the encoding of each version, using software utilities to read and update the index file manually. For a prototype such techniques are fine, but they are not acceptable for general use.

URL Format All IHTML files in IHTML 1 are processed by a CGI script when they are requested by a client, to translate them from IHTML to HTML. It was unavoidable in IHTML 1, since one of the design criteria was that it should use unmodified HTTP server software. A side effect is that all anchors to IHTML files must include the path to the CGI script. The result is that IHTML is very visible to the browser of an intensional site: every URL starts with the same path to the IHTML-to-HTML translation program. Althogh not a serious flaw, it is a little unfortunate.

3. Practical IHTML

Several features of IHTML 2 improve the usability of IHTML. The approach to versioning allows relative and absolute trans-version links, includes, and so on. Links to pages using other version spaces are straightforward. There is no single point of failure for all version information in a site. There is no need to configure the server software with the set of allowable dimension names. In addition to changes in the version system, it provides a variety of new features: selective inclusion of text in a page, based on the requested version; dimensional variables which allow one dimension to use values from another dimension; the ability to create links to pages using different version spaces; and convenient tools for creating intensional WWW sites.

3.1 Versions and Version Modifiers

Each IHTML 2 site has a *version space* in which its pages exist. A version space is a set of dimensions, with a set of indices for each dimension. Each dimension's index set includes a unique *vanilla* value (usually written "ε"), which is treated specially when comparing versions. A version is a point in this dimensional space.

Versions are written as a disjunction (using the symbol "+") of dimension specifiers, where each dimension specifier names a dimension and (optionally) an index in that dimension. The following grammar describes the syntax of versions:

$$V ::= \varepsilon \mid D{:}I \mid D{:}\varepsilon \mid V + V \quad (3.1)$$
$$D ::= T \quad (3.2)$$
$$I ::= T \mid N \quad (3.3)$$

where N is a non-negative integer, and T is a token (a sequence of one or more characters from the set of letters, numbers, and "_"). Since it is not, generally, possible to type ε, an omitted dimension index will be interpreted as ε for that dimension.

An empty string (ε by itself in the above grammar) is the vanilla version, that is, the one that is vanilla in all dimensions. Examples of versions are:

```
language:french+graphics:hires
revision:3+platform:sparc+debugging:2
zork1:zork2+zork3:zork4+zork5:
```

and so on. Note the missing index values in a few places: their absence is interpreted as ε. Note also that dimensions which are not mentioned in a version string are assumed to be vanilla.

The + operator is idempotent, commutative, and has ε as its identity:

$$V + V \equiv V \quad (3.4)$$

$$V + W \equiv W + V \quad (3.5)$$

$$V + \varepsilon \equiv V \quad (3.6)$$

In some contexts, it is appropriate to describe, not a version, but a version *modifier*, that is, a modification of the current version (applicable only in contexts in which there is a current version). In particular, we will want to be able to override specific dimension values while leaving the remainder unchanged. The syntax for a version modifier is very similar to that of a version:

$$V ::= \varepsilon \mid D{:}I \mid D{:}\varepsilon \mid V{,}V \quad (3.7)$$

$$D ::= T \quad (3.8)$$

$$I ::= T \mid N \quad (3.9)$$

Note the comma, used to construct a list of dimension specifiers. In this case, each specified dimension will have the given value, but unspecified dimensions remain unchanged. For this reason, a dimension specification of $D : \varepsilon$ is not the same as omitting the dimension. $D : \varepsilon$ in a version modifier sets a particular dimension's index explicitly to vanilla; omitting it leaves its current value unchanged. This is the

key difference between a version and a version modifier. Version modfiers are only meaningful in a context with a current version (as with dimension variables), that is, in an IHTML page.

The version language described here differs from that of IHTML 1 in two significant ways:

- There is no notion of a subversion, as Yildirim has. There is no fundamental reason for omitting it; a later version of IHTML will probably include it. It was left out largely because it appeared well-understood and the effort involved in implementing it was better spent exploring new ideas.
- IHTML 1 has no explicit notion of the difference between a version and a version modifier. All version specifications are assumed to be relative to the current version. One consequence of this approach is that there is no way to refer to a specific version of a page or component of a page, without respect to the current version. Another (related) consequence is that it is impossible to include a link to a page in a different version space.

The IHTML syntax for versions and version modifiers in tags which support versioning (`A`, `IMG`, `!--#INCLUDE`, and so on) is

```
<tag ...VERSION=version ...>
```

for an absolute version,

```
<tag ...VMOD=version-modifier ...>
```

for a version modifier, or standard HTML tag syntax for a tag referring to the same version as the current page. Details on IHTML syntax are found in Appendix A.

3.1.1 Version Comparison

Finding the best fit for a requested version, given a set of existing versions of a component, implies the existence of some kind of ordering of versions. A partial

order is given by the *refinement* relation "$\subseteq$", read as "refines to" or "is more generic than", defined by the following rules:

$$\varepsilon \subseteq V \tag{3.10}$$

$$V \subseteq V + V' \tag{3.11}$$

$$V \subseteq W,\, V' \subseteq W \rightarrow V + V' \subseteq W \tag{3.12}$$

$$N_1 \leq N_2 \rightarrow D{:}N_1 \subseteq D{:}N_2, \text{ for integers } N_1, N_2 \tag{3.13}$$

Based on the above rules, the best-fit problem may be stated as follows: Given a requested version R of a particular IHTML component, and a set V of versions of the component which actually exist, the best-fit version is the unique $v \in V$ such that: $v \subseteq R$, and $\forall w \in V$ such that $w \neq v$, $w \not\subseteq R$ or $w \subseteq v$. There may be no best fit: there may be no $v \subseteq R$, or there may be multiple $v \subseteq R$, but $v_i \not\subseteq v_j$, and $v_j \not\subseteq v_i$. In other words, there may be no versions which refine to R, or the set of versions which refine to R may not have a unique maximum element.

For example, suppose the requested version were `a:b+c:3`. If that exact version existed, it would be used. If it did not, but `a:b+c:2` did, that would be used. If no version existed which had values for both the `a` and `c` dimensions, but one for `a:b` existed, it would be used. However, if versions `a:b` and `c:2` existed, there would be no unique best fit. Figure 3.1 shows two possible version spaces, with the maximum elements $\subseteq R$ circled. The left diagram, having a unique maximum element, has a solution to the best-fit problem. The right diagram has no solution, since `a:b` $\subseteq R$, and `c:2` $\subseteq R$, but `a:b` $\not\subseteq$ `c:2` and `c:2` $\not\subseteq$ `a:b`.

3.1.1.1 Canonical Form

To make comparison of versions straightforward, we define a canonical form. This is a simplified form of Yildirim's canonical form (simpler because the version space for IHTML 2 does not include subversions). The rules for conversion to canonical form are:

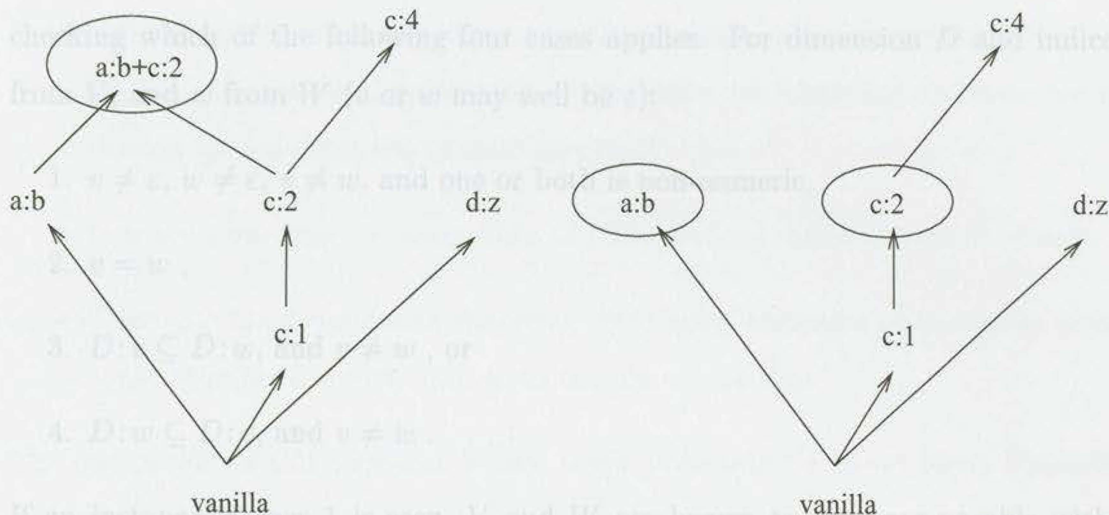


Figure 3.1: Finding the best fit for version `a:b+c:3`

- all dimensional variables must be resolved (see Section 3.6),
- all vanilla dimensions are removed, and
- non-vanilla dimensions are ordered alphabetically, by dimension name.

For example, the canonical form for version `zork:moria+thing:+alpha:bravo` is `alpha:bravo+zork:moria`.

3.1.1.2 Comparison Algorithm

Given the version ordering rules, the rule for comparison of versions is straightforward:

$$D_0 : V_0 + D_1 : V_1 + \ldots \sqsubseteq E_0 : W_0 + E_1 : W_1 + \ldots \tag{3.14}$$

if and only if for each D_i, either $V_i = \varepsilon$ or $D_i = E_j$ for some j, and in that case $D{:}V_i \sqsubseteq E{:}W_j$.

Algorithmically, we can take advantage of the canonical form to compare two versions in time proportional to the number of non-vanilla dimensions. For two versions V and W, the algorithm returns one of four values: `LESSTHAN`, `GREATERTHAN`, `EQUAL`, and `INCOMPARABLE`, indicating that $V \sqsubseteq W$, $W \sqsubseteq V$, $V = W$, or $V \not\sqsubseteq W$ and $W \not\sqsubseteq V$, respectively. The algorithm goes through the dimensions alphabetically,

checking which of the following four cases applies. For dimension D and indices v from V, and w from W (v or w may well be ε):

1. $v \neq \varepsilon$, $w \neq \varepsilon$, $v \neq w$, and one or both is non-numeric,
2. $v = w$,
3. $D{:}v \subseteq D{:}w$, and $v \neq w$, or
4. $D{:}w \subseteq D{:}v$, and $v \neq w$.

If an instance of case 1 is seen, V and W are known to be incomparable without further testing. If case 2 is seen, skip to the next dimension. If an instance of case 3 is seen (but no instances of case 4 have been seen), make a note of it and continue to the next dimension. Likewise, if an instance of case 4 is (but no instances of case 3 have been seen), make a note of it and continue. If cases 3 and 4 have been encountered (for different dimensions, obviously), stop immediately: V and W are incomparable. After all dimensions have been checked, if neither case 3 or case 4 has been seen, the versions are equal. If case 3 has been seen, then $V \subseteq W$. Otherwise, $W \subseteq V$. (We know that both have not been seen, because we would have halted immediately in that case.)

3.2 Version Representation in URLs and Files

One important design criterion for IHTML is that the versions in URLs be encoded, so that browsing users cannot request arbitrary versions, but only those allowed by the site implementer. IHTML 1 encoded versions as integers, with a central index file to map integers to versions. This approach has several limitations:

- It is slow, since the map file must be loaded and parsed on every HTTP request, and possibly modified and saved as well. (Caching the map would entail modifying the server software, which IHTML 1 was intended to avoid.)

- It is dangerous, since loss of the map file results in loss of all version information for the site (physical pages and components are identified by their encoded version integers, not the textual versions).

- It is not clear how to create links to sites that use different version spaces.

- There are synchronization concerns, since many instances of the server process could require read/write access to the file at one time.

The one benefit of this approach is that version codes in URLs are short (typically 2 or at most 3 digits).

The IHTML 2 approach is to encode the entire version in filenames, URLs, and anywhere else they are needed. The drawback is that URLs may include long, peculiar strings if the required version is complex. However, there are significant advantages:

- There is no single point of failure (no version file to be corrupted).

- There are no synchronization problems (no version file to get read/write access to).

- It is possible to encode versions for arbitrary version spaces, so one can include links to other versioned sites.

- There is little or no performance penalty.

- The implementation is considerably simpler.

The complexity of filenames and URLs is not a significant problem, since utility programs are provided which prevent the user from having to type the encodings of versions; all the user ever needs to see is the normal textual versions.

3.2.1 Version Encoding

Version encoding occurs in four steps: conversion to a canonical form, representation as a character string, compression, and conversion of the compressed (binary)

form to printable ASCII text (so it can be used in URLs, filenames, etc.). The canonical form is described in Section 3.1.1.1. Representation as a string is straightforward. The last two steps are described in Appendix B. Briefly, the compression step uses *arithmetic coding* to convert the version string to a compressed binary form, and the conversion to printable form is similar to traditional Unix *uuencoding*. The key feature is that the encoding contains all the information present in the original version expression.

Here are some examples of encoded versions:

Version	*Encoding*
lang:english+graphics:hires	`M1lDnCgO34qSj6vu8Jz-NgGa`
zork1:zork2+zork3:	`aXHTPmoM8b-JIBLNyRjDLIdc`
vanilla	`aai`

3.2.2 Versions in URLs

URLs of versioned objects (pages, images, or whatever) are like normal URLs, but with a version code inserted between the filename and the suffix of the object. For example, if `aXHTPmoM8b-JIeai` is the encoding of version `zork1:zork2` (which it is), then the `zork1:zork2` version of the page identified by URL

`http://www.moria.com/rogue.html`

is

`http://www.moria.com/rogue.aXHTPmoM8b-JIeai.html` .

Since all objects specifiable by URLs have a suffix identifying their type, this position seems reasonable. The IHTML server looks for URLs with version codes as described, and handles them as versioned objects. URLs without such version codes are handled as if they were requests for the vanilla version.

Note that it is not possible, via this scheme, to refer to versioned directories. The difficulty is where to put the version code. One possibility is to represent versioned directories as something like . . . */dirname.vcode*, but this has not been implemented. IHTML 2 does not support versioned directories in any way.

3.2.3 Storing Versioned Objects on Disk

Versioned objects such as pages, images, or parts of pages (components) are represented as directories, named as the object itself would be named. For example, the intensional page `zork.html` is represented as a directory named `.../zork.html`. The physical versions are files in the directory; the vanilla version of this page would be `.../zork.html/aai.html` (since *aai* is the encoded form of vanilla). Two points are important: first, the name of the file contains all information necessary to recover version information, so there is no single point of failure; second, the site implementer need never be aware of the directory structure, or the version encodings. This second point is made possible by the site implementation utilities described in Section 3.7.

3.3 Default URLs

Standard HTML provides anchor tags to create links to other pages, but it is rare (and peculiar) to provide a link to the current page (unless it is to another part of the page). In IHTML, trans-version links to the current page are commonly used to alter the version of the page being viewed. Since it is such a common usage, IHTML 2 does not require a `HREF` attribute in anchor tags. If omitted, it defaults to the current page. In this way, link tables which provide a standard set of links for many pages in a site may be created.

For example, suppose a site varied in the `background`, `graphics`, and `language` dimensions. The following fragment of IHTML could be included in all pages in the site (say, as the last element of the page), to provide a consistent set of links to other versions of the site. Since the anchors don't specify the URL, it will default to the current page, whatever that might be. If the set of links should change, it need only be changed in one place, even though it is used in many different pages. This fragment itself would (most likely) vary in the `language` dimension, but would not need to vary in the `background` or `graphics` dimensions.

```
<hr>
Background: <a vmod="background:blue">blue</a> |
```

```
            <a vmod="background:green">green</a> |
            <a vmod="background:red">red</a> |
            <a vmod="background:yellow">yellow</a> |
            <a vmod="background:black">black</a> |
            <a vmod="background:">default</a>
<br>
Graphics: <a vmod="graphics:none">none</a> |
          <a vmod="graphics:small">small</a> |
          <a vmod="graphics:large">large</a> |
          <a vmod="graphics:">default</a>
<br>
Language: <a vmod="language:english">English</a> |
          <a vmod="language:french">French</a> |
          <a vmod="language:turkish">Turkish</a> |
          <a vmod="language:swahili">Swahili</a> |
          <a vmod="language:">default</a>
```

When included at the end of an IHTML page, it would create a set of links something like the following:

Background: blue | green | red | yellow | black | default
Graphics: none | small | large | default
Language: English | French | Turkish | Swahili | default

If included on many pages, it would allow the user to change the version of the current page according to a standard set of choices.

3.4 Absolute Version Specifications

Under normal circumstances, a trans-version anchor or inclusion is interpreted as being relative to the current version of the page in which the link exists. Any dimensions specified in the link augment the current dimension, rather than replacing it. It may, however, occasionally be useful to provide a link to a specific version of a page, regardless of the current version of the referring page. One example is a link to the vanilla version of a page. Another is a link to some version of a page at another intensional site. IHTML 2 supports *absolute* (as opposed to relative) version specifiers, in addition to version modifiers, for these purposes.

For example, the following anchors lead to the vanilla version of the current page, and the `evil:empire` version of the off-site page, regardless of the version of the current page:

```
<a href="zork.html" version="">Back to Vanilla</a>
<a href="http://www.microsoft.com/"
      version="evil:empire">Abandon Hope, All Ye...</a>
```

3.5 Selective Inclusion of Text

Proliferation of small files is a frequent occurrence in IHTML 1. For example, one common dimension in which a site might vary is background colour. The approach used to implement this effect in IHTML 1 is to include at the top of each file a component which sets the background colour, which (naturally) varies in the `background` dimension. No other components need vary in this dimension. Once the user picks, say, the `background:green` version of the site, this dimension is propagated through links, includes, etc. in the usual way. The `background.html` component itself is composed of several small files, one per index (i.e. colour) in the background dimension.

It turns out that this general idea occurs frequently in IHTML sites. It works, but it is somewhat tedious to create the many small (frequently 1 line) files that are necessary. IHTML 2 provides a *selective inclusion* feature very much like a `switch` statement in the C programming language. Its syntax is as follows:

```
<ISELECT>
   <ICASE VERSION=v1>...text1...[</ICASE>]
   <ICASE VERSION=v2>...text2...[</ICASE>]
   <ICASE VERSION=v3>...text3...[</ICASE>]
   ...
</ISELECT>
```

In general, the output includes the $text_i$ corresponding with the first v_i such that $v_i \subseteq V$, where V is the current version. Note that the trailing `</ICASE>` tags are optional.

If they are omitted, the output includes the following case(s) until a `</ICASE>` is encountered, or the closing `</ISELECT>` is reached. This is analogous to the fall-through behaviour of a `case` without a closing `break` in a C `switch` statement. An `<ICASE>` with no version specified defaults to vanilla, and will therefore always be selected, if no previous case was chosen. Note that the first vanilla case makes all following cases inaccessible.

Consider the "background" example. Using the ISELECT approach, the `background.html` component, instead of consisting of many small files, would be a single (vanilla) file containing the following ISELECT statement:

```
<ISELECT>
    <ICASE VERSION="background:blue"><BODY BGCOLOR="blue"></ICASE>
    <ICASE VERSION="background:red"><BODY BGCOLOR="red"></ICASE>
    <ICASE VERSION="background:green"><BODY BGCOLOR="green"></ICASE>
    <ICASE VERSION="background:zork"><BODY BGCOLOR="zork"></ICASE>
    <ICASE><BODY BGCOLOR="black"></ICASE>
</ISELECT>
```

A similar structure called `ICOLLECT` includes *every* case which refines to the current version, i.e. all $text_i$ such that $v_i \subseteq V$. The syntax is essentially the same as that of ISELECT: just substitute COLLECT for SELECT. With ICOLLECT, though, a vanilla case does not block subsequent cases, since more than one may be chosen.

ISELECT and ISELECT originally arose as a replacement for the aggregation feature of IHTML 1. Wadge, schraefel, and Driessen came up with the notion in unpublished conversations; the author refined the notion somewhat and implemented it.

3.6 Dimension Variables

It is sometimes useful to use the current index of one dimension as a dimension name or index in a version or version modifier. For example, suppose one wanted

the ability to set the "visited link colour" of a page to the same as the current "unvisited link colour". A link such as the following would do the job, assuming the two dimensions were named `vlc` and `ulc`, and all the necessary includes were defined:

```
<A VMOD="vlc:$ulc">set link colour</A>
```

The following example shows the use of dimension variables to specify the dimension, rather than the index:

```
<A VMOD="$zork:rogue">do something...</A>
```

The current index of dimension `zork` will be treated as a dimension and the link will set that dimension to index `rogue`.

Dimension variables may also be embedded in strings. The complete syntax is:

$$
\begin{aligned}
T &::= L \\
&\quad | \quad [T]\$L \\
&\quad | \quad [T]\$\{L\}[T] \\
L &::= \text{1 or more of [a-zA-Z0-9_]}
\end{aligned}
$$

For example, all of the following are legal dimension variable expressions: `abc$def`, `a${bcd}ef`, `$abcdef`, `${abcdef}`, `$abc$def`. The dimensions which will be replaced are (in order): `def`, `bcd`, `abcdef`, `abcdef`, `abc`, and (also in the last expression) `def`. Note that if the dimension is unspecified, or vanilla, it is replaced with the empty string, without warning.

3.7 Site Construction Utilities

One serious limitation of IHTML 1 is the difficulty of creating intensional sites. Users are required to understand how versions are encoded, name files carefully to reflect the version encodings in the previously described map file, and so on. IHTML 2 addresses this difficulty by providing utility programs which insulate the user from

implementation details of IHTML. The objective is to allow the user to view an intensional page as a single entity, which happens to exist in multiple versions. Clearly, we must not expect users (either site implementors or browsers of intensional sites) to type or understand the encoded versions described in Section 3.1. Four utilities are provided to hide them from view. They are essentially front ends to standard Unix utilities, adding version encoding and decoding facilities.

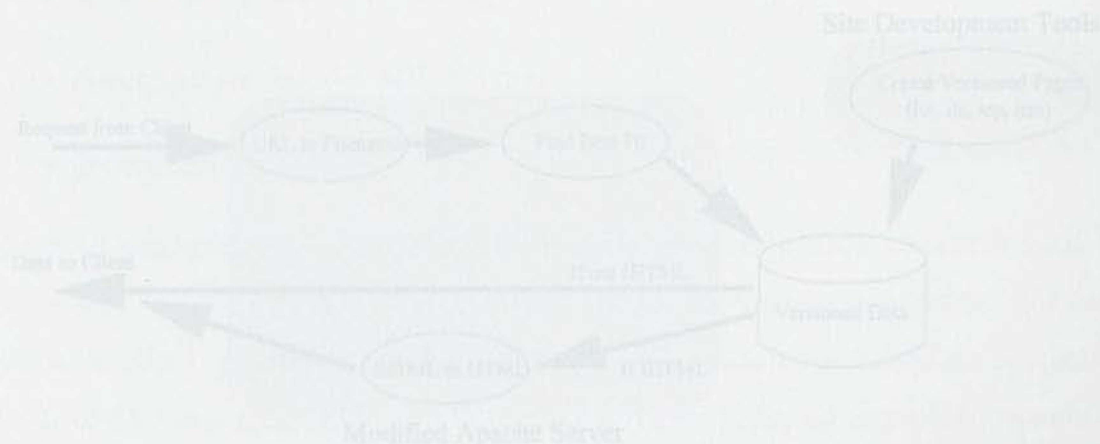
icp copies an ordinary file to a specified version of some versioned component or page, or copies a specific version of some component back to an ordinary file.

ivi invokes a user-specified editor on a given version of a versioned HTML page or component. The editor specified in the *EDITOR* environment variable is invoked. If none is specified, `/usr/ucb/vi` is used.

ils lists the files in a directory; for versioned things, also lists the versions in which they exist.

irm removes a particular version of a component.

These utilities are described in more detail in Appendix A.



4. Implementation

The operation of IHTML includes several phases, as shown in Figure 4.1. The site developer first creates a repository of versioned (and possibly unversioned) objects such as HTML pages, images, scripts, and so on. Once that exists, the server can accept requests for versioned objects. Processing a request includes four stages: translate the URL to a filename, find the best fit version of that object, translate it to HTML (if its suffix is "`.html`"), and send the result to the client. The last step uses output functions supplied by the Apache server. The remaining steps are described in the following sections. First, though, the interaction of IHTML with the existing Apache software is described, to put the rest of the section in context.

4.1 The Apache Module

IHTML is implemented as an Apache module, using the provided API (application programming interface). The API provides two basic forms of interaction with Apache: modules may define new server configuration commands, and they may interfere in the request-serving process. IHTML does both: new configuration com-

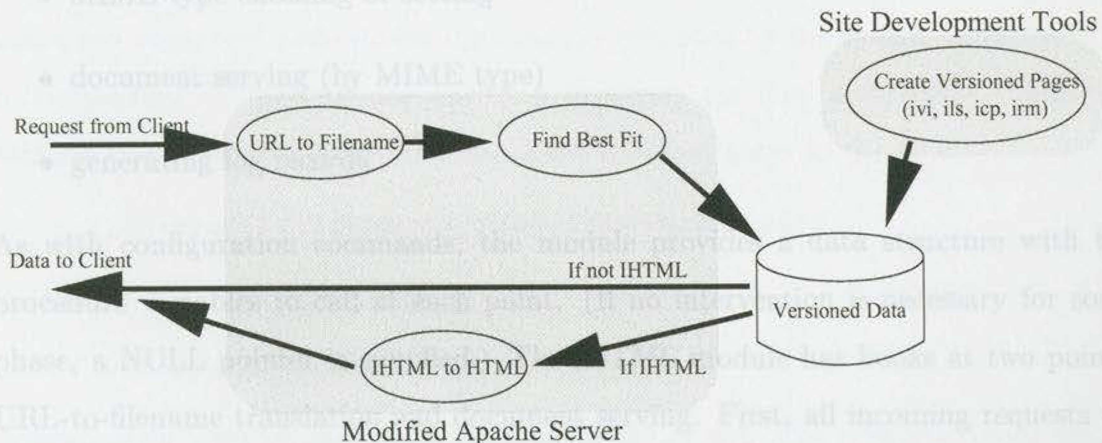


Figure 4.1: Server-Side Operations

mands allow the site maintainer to provide filenames for IHTML log files, and HTTP requests are interpreted as requests for versioned URLs. The API is described in detail in the Apache server documentation [23]. An overview, with IHTML-specific details, is given here.

IHTML adds two configuration commands to Apache, to define log files for IHTML log messages, and IHTML error messages. Their use is described in the Appendix, section A.6.2. The module binds new commands into the server by providing a data structure for each command, including the name of the command, the expected number and type of arguments, a procedure variable to call if this command is detected in a configuration file, and a brief textual summary of the purpose of the command. This last element allows the server to provide useful error messages if a command is misused.

Modules can insert hooks into the request serving process at several points:

- URL-to-filename translation
- HTTP header parsing
- access checking (based on client IP address, HTTP user authentication, and/or file/directory permission checking)
- MIME type checking or setting
- document serving (by MIME type)
- generating log records

As with configuration commands, the module provides a data structure with the procedure variables to call at each point. (If no intervention is necessary for some phase, a NULL pointer is supplied.) The IHTML module has hooks at two points: URL-to-filename translation and document serving. First, all incoming requests include version checking, to find the best fit version. This happens even if the files or requests themselves are not versioned: a URL not containing a version code is

treated as a request for the vanilla version; a file which is not versioned is treated as the vanilla version of that file. Second, when the document is to be sent to the client, if the MIME type of the request is "text/html", the file is filtered through the IHTML-to-HTML translator so that the client sees only standard HTML.

There are several benefits to implementing IHTML as an add-on to the Apache server (or any other server). Performance is good, since there is no overhead of invoking a script, opening files other than the actual requested file, and so on. The only files which need to be examined are the directory containing the versions of an object, and the chosen file itself. URLs are simplified, since there is no need to include the path to a script, or anything except a normal URL (possibly with a version code embedded in it). Since the server takes care of deciding whether to process a file through the IHTML-to-HTML filter, there is no way for a user to retrieve an untranslated IHTML file (which could be done easily with the script-based implementation). These benefits, as well as the ease of enhancing the Apache server, make this approach superior to the previous implementation.

4.2 URL to Filename Translation

URLs in HTTP "GET" requests are of the form

`...`*URLpath*`/`*filename*`.[`*vcode*`.]`*suff* `[/`*pathinfo*`]`

where the *vcode* and *pathinfo* are optional (as indicated by the square brackets). The corresponding file may or may not be a versioned file (i.e. a directory containing versions of a file). The server software must translate them to the form

`...`*unixpath*`/`*filename*`.`*suff*`/`*best-fit-vcode*`.`*suff*

or

`...`*unixpath*`/`*filename*`.`*suff*

depending on whether the target file is versioned or not. The *pathinfo* component is removed and stored separately (for the use of the CGI script, if this URL happens to refer to a script). There are several steps in the translation process.

1. Prepend the appropriate Unix path. It will be either the server's "document root", or a user's `.www` directory (if the path begins with `~userid` for some Unix userid). (Note that the standard server software has already removed the Internet address component of the URL.) This step is not changed in IHTML.

2. Remove any trailing "path info". For the standard Apache server, this step is straightforward: remove elements from the end of the path until an existing file or directory is found. (Note that trailing path info is only legal if the URL refers to a CGI script.)

   For IHTML, there is the added complication that each path component must be checked for the presence of a version code before the existence check is performed. For example, consider the URL path component

   `/path/zork.aai.html/part1/part2` .

   Once the last two components have been checked for and removed, the software must delete the `.aai` from the remaining path before checking, since `zork.aai.html` is not the Unix name of an actual file that the server is looking for.

3. Extract the version code from the URL. The software looks for a match to the pattern

   `.../`*`name.vcode.suffix`*

   at the end of the URL. (The *name* may contain periods, but not slashes. The *vcode* and *suffix* must not contain periods or slashes.) If enough periods are found (indicating that the *vcode* component might be a version code), it attempts to decode *vcode* into a version. If successful, it records the version, and reconstructs the filename as

   `.../`*`name.suffix`*

Note that it doesn't matter what the suffix is, only that it not contain periods (which it won't). Version codes are also designed so as not to contain periods. If either step fails (the pattern didn't match, or the *vcode* can't be decoded into a legal version), the software assumes that the request is for the vanilla version, and leaves the filename unchanged.

4. Find the best fit to the requested version for the named page or other object. There are a few special cases to consider:

   - If the requested object itself is unversioned (regardless of whether the request is for a non-vanilla version of it), treat the file as the vanilla version of a versioned object.
   - If the request does not contain a version code, treat it as a request for the vanilla version (as described in step three).

   Aside from these considerations, the software reads the set of available versions of the requested file, runs the best fit algorithm on them, and gets the best fit version. If there isn't one, the software returns a normal "404 Not Found" result to the client, and logs the failure.

5. Construct the final filename. At the best fit stage, the software noticed whether or not the requested file was a versioned object or not. If it was, the final filename will be

   .../name.suffix/bestfitcode.suffix

   where *bestfitcode* is the encoding of the best fit version found in the previous step. If the requested file is an ordinary file, the filename will simply be

   .../name.suffix

   Either way, the requested version is saved, for the IHTML-to-HTML translation step.

The result of this procedure is either a requested version and Unix path for the file to be sent to the client, or a result code indicating that no file was found.

4.3 IHTML to HTML Translation

Like ordinary Apache server-parsed documents, IHTML pages are translated to pure HTML before being sent to the client. The translation process follows these rules:

- Normal text is echoed unchanged.
- HTML tags which are not relevant to IHTML are echoed unchanged.
- HTML tags which include URLs are transformed as follows:

    1. Find the attribute which is to be altered: `HREF`, `SRC`, `ACTION`, or `VALUE`.
    2. If the tag includes a `VERSION` attribute, use that as the base version, otherwise use the current version for this page.
    3. If the tag includes a `VMOD` attribute, merge it into the base version, resulting in the requested version for this tag. To merge a `VMOD`, add each dimension/index pair in the `VMOD` to the base version. If a dimension to be added is already present in the base version, replace its index with the one from the `VMOD`.
    4. Construct the encoding for this version (see Section 3.2.1).
    5. Insert the encoding for this version between the filename and suffix of the URL.
    6. Output the modified tag (not including the `VERSION` or `VMOD` attributes).

    Tags transformed in this manner are: `A`, `IMG`, `OPTION`, `FORM`, and `FRAME` (i.e. all those which contain URLs).

- IHTML tags are replaced with the appropriate text:

include Process the included file as if it were a normal IHTML request, and copy the result to the output.

exec Run the named program and redirect its output to the client.

echo Echo the current index of the named dimension to the output.

iurl Construct a versioned URL from the HREF attribute, as described above, and copy it to the output.

dumpver Dump the entire current version to the output (for debugging use, mainly).

starttag Send "<" to the output.

endtag Send ">" to the output.

See Appendix A for details on the syntax and usage of these tags. Note: If the HREF attribute is missing from the "A" or "IURL" tags, the current page is used by default.

- An ISELECT tag such as

```
<ISELECT>
    <ICASE VERSION=v1>...text1...[</ICASE>]
    <ICASE VERSION=v2>...text2...[</ICASE>]
    <ICASE VERSION=v3>...text3...[</ICASE>]
    ...
</ISELECT>
```

is one case where non-tag text is not (or rather, might not be) echoed to the output. The output includes the first $text_i$ such that $v_i \subseteq V$ (V is the requested version), and possibly subsequent $text_j$ if the </ICASE> tag is missing. Output continues until a </ICASE> tag is encountered or the </ISELECT> is reached (naturally, any <ICASE> tags are not printed). Note that a missing VERSION attribute in an <ICASE> is interpreted as vanilla. Since $\varepsilon \subseteq V$ for any V, it acts as a default case, always chosen if nothing before it was.

- For ICOLLECT tags (same syntax as ISELECT, substituting COLLECT for SELECT), the output includes all $text_i$ such that $v_i \subseteq V$, but is otherwise the same as ISELECT.

The output of IHTML-to-HTML translation is standard HTML, with version codes embedded in URLs, to ensure that links and images refer to the appropriate versions of other Web objects.

4.4 Site Development

The objective of usability requires that as many implementation details as possible be hidden from the user (whether a site developer or a browsing customer). In particular, we would like to hide from site implementers the representation of versioned objects on disk. Two reasons are that we can then change the representation without disturbing the user, and that users must not be expected to type long, incomprehensible version codes (or even short, incomprehensible version codes, for that matter).

In pursuit of that objective, IHTML 2 provides some basic Unix utility programs which allow the user to view versioned objects as black boxes, knowing nothing of the implementation or version encoding. Since versioned objects are, in fact, represented as Unix directories containing files named with version codes, nothing is actually hidden. Future work includes a versioned repository system which could do a much better job of insulating the user from the harsh realities of version codes; see Section 6 for discussion. For now, IHTML 2 provides four basic utilities. They are essentially front ends to standard Unix utilities, providing version encoding and decoding facilities.

icp copies an ordinary file to a specified version of some versioned component or page, or copies a specific version of some component back to an ordinary file.

ivi invokes a user-specified editor on a given version of a versioned HTML page or component. The editor specified in the *EDITOR* environment variable is

invoked. If none is specified, `vi` is used.

ils lists the files in a directory; for versioned things, also lists the versions in which they exist.

irm removes a particular version of a component.

See Appendix A for details on the use of these programs.

One interesting feature of these programs is that they share the code for version manipulation and best fit matching with the IHTML code for the Apache server. It is interesting because the Apache programming rules require the use of non-standard functions for memory allocation, file manipulation, and so on. The objective is to prevent memory leaks: by using only, say, Apache's `palloc` to allocate memory, the Apache software can guarantee that memory is never lost irretrievably (an essential feature for a daemon-style process which may run indefinitely). A similar setup is used to ensure that all open files get closed, and so on. Sharing code, then, is complicated: when it's part of the Apache server, it must `palloc` memory; when it's part of, say, `ivi`, it must `malloc` memory. The usual C/Unix approach to this problem is the `#ifdef` compiler directive:

```
#ifdef APACHE
    x = palloc(...);
#else
    x = malloc(...);
#endif
```

It is unsatisfactory because it requires altering every usage of several functions, unnecessarily complicating the code. Perhaps more important, there is no easy way to tell for which version the code was last compiled. Standard tools like `make` consider only the timestamp of last compilation. If the Unix tool version is built, then the Apache server version is built, it will look to `make` as if the file does not need to be recompiled. The results will be annoying and ineffective.

A better solution, used in IHTML, is stubs. For each Apache special routine, a stub is created which does nothing but call the corresponding Unix system function(s). For example:

```
void *palloc(struct pool *p, int nbytes) {
  return malloc(nbytes);
}
```

(The `struct pool *p` is part of Apache's memory allocation tracking system.) If the code is being linked into the Apache server, the file containing the stubs is not included, so the real Apache `palloc` is used. If it is being linked into one of the Unix utilities, the stubs are included and the code calls the right Unix library function (via the stubs).

5. Related Web Technologies

The World Wide Web is in flux, with new technologies emerging and existing ones evolving. It is important to consider how any one technology fits with the others, to minimize duplication of effort as well as to explore ways in which they may enhance each other. The following sections summarize the major current and emerging technologies, and IHTML's connection (if any) with them.

It appears that the capabilities provided by IHTML are not duplicated by other Web technologies. It is certainly possible, in some sense, to achieve in other ways anything that can be done with IHTML; however, it does not appear to be straightforward. It is clear, for example, that CGI programs could achieve the same effect (particularly clear, since the first version of IHTML was a CGI program). It also seems likely that some combination of HTML, JavaScript, cascading style sheets, and the W3 Consortium's Document Object Model could duplicate most of the functionality provided by IHTML. The question is not whether it is possible, though, but whether it is convenient. After all, the fact that any C program, Perl script or other program could have been implemented in assembler does not persuade us that assembler is the best choice for all purposes.

On the other hand, it appears that IHTML's versions and best fit technique could be applied to the new technologies with minimal effort and some benefit. The following sections briefly describe various Web-related technologies, and how IHTML relates to them.

5.1 The Common Gateway Interface

The Common Gateway Interface (CGI) allows the execution of programs to be triggered by a request to a Web server for a particular URL, with the output of the program being sent back to the client. Depending on how a server has been configured, files in specific directories (usually named "`cgi-bin`") or with specific

suffixes (usually ".cgi") are recognized as programs, and are executed instead of being sent to the client verbatim. Such programs are usually called *CGI programs* or *CGI scripts.* They are typically used in conjunction with HTML pages containing forms: the forms provide the data on which the programs operate, while the programs return HTML pages showing the results of the form submission.

As mentioned, it is certainly possible to achieve the same effects as IHTML using CGI programs. However, as Wadge et al. point out in [1], CGI programs are an *escape* from HTML, rather than an integral part of it. A capability as basic as providing multiple versions of a page or site certainly deserves to be supported within HTML. Additionally, a site implementer wishing to provide interesting functionality via a CGI program must be a reasonably competent programmer. It seems a lot to be expected for such a basic capability. IHTML provides versioning without programming, and without going outside the HTML model. Understanding of a few new tags is all that is necessary.

IHTML supports versioned CGI programs, the same way it supports versioned HTML pages. In addition to invoking the appropriate version of a CGI program, IHTML makes the current version (i.e. the requested version of the CGI program) available as an environment variable for the script to use. Future work might include providing a library of version manipulation routines which would allow programs to manipulate versions in more sophisticated ways than is currently possible.

5.2 Apache's Server-Side Includes

The server-side include (SSI) feature of the Apache Web server provides facilities for including HTML files (or the output of programs) in pages, and selective inclusion of text based on conditions (as well as a few other odds and ends). A very primitive form of versioned pages could be implemented using these facilities, but it would not be convenient. In addition, encoding of versions would be difficult (without additional programming).

For example, Figure 5.1 shows a fragment of HTML (supplemented with SSI di-

```
<html>
<head>
  <title>Thing</title>
  <!--#set var="zork" value="y" -->
</head>
<body>
<hr>
<!--#if expr="$zork=x" -->
  <!--#include virtual="thing_x.shtml" -->
<!--#elif expr="$zork=y -->
  <!--#include virtual="thing_y.shtml" -->
<!--#else -->
  <!--#include virtual="thing_z.shtml" -->
<!--#endif -->
<hr>
<!--#exec virtual="rewrite.cgi thing2.html zork=$zork" -->Link</a>
</body>
</html>
```

Figure 5.1: Versioned inclusion via the Apache SSI

rectives) which will include different versions of a file, based on "version" information for dimension zork set in the first few lines. Note the #exec at the bottom: version information can be passed to other programs via URLs, as is done with IHTML, if the URLs are inserted by a program. The idea is that the rewrite.cgi program would output something along the lines of:

```
<A href="thing2.html,zork=y">
```

Note that in the exec case, all dimensions would have to be listed in the in #exec line: no automatic propagation of versions would occur. In addition, if new indices were added to the zork dimension, the #if statement would have to be expanded to include them. Added dimensions would cause combinatorial explosion of cases in the #if statement. Perhaps most importantly, it is not entirely clear how the script should rewrite URL's to pass version information along to subsequent pages. The only plausible solution seems to be to route the whole URL through yet another CGI program (as IHTML 1 does), which would parse the results and figure out which file

was the best fit. In other words, one would end up re-implementing most of IHTML anyways. A comparable fragment of IHTML to achieve the same objectives is:

```
<!--#include virtual="thing.html" vmod="zork:x" -->
<a href="thing2.html" vmod="zork:x">Link</a>
```

As the alert reader will have noticed, IHTML borrows much of its syntax from SSI. In fact, IHTML may, in some sense, be considered an enhancement of SSI. Inclusion and exec syntax, in particular, were borrowed almost verbatim; the only addition is version syntax (`VMOD` and `VERSION` attributes). The current implementation of IHTML precludes the use of full SSI and IHTML together, unfortunately; a better implementation would combine them cleanly.

5.3 Java and JavaScript

Java and JavaScript are programming languages used for making Web pages dynamic (among other things). JavaScript programs may be added directly to HTML pages; an interpreter is built into the major Web browsers (with some platform-specific variations). Java programs may also be executed in a Web page, but are typically restricted to running within a well-defined region of a page, inside of which they have complete control over what is displayed. See [24] or [25] for background information on JavaScript. See [26] for information on Java.

At present, both Java and JavaScript have limited access to the contents of the pages in which they occur. Appendix A of [25] gives a list of the elements of a page which are accessible; they include the fields of forms, and so on, but not the main text of the page. As with SSI, a crude form of versioning could be achieved through the use of JavaScript and a CGI program on the server to pass version information from one page to the next, but it would be a great deal of work.

Unfolding developments in Web technology may change this story soon, however. The W3 Consortium's Document Object Model [27] promises full access to all parts of a Web page from programs; combined with HTML 4.0's browser-side inclu-

sion features, it will make quite sophisticated page modifications possible. See Section 5.6 for more discussion of this development.

Since JavaScript is usually included directly in an HTML page, it takes advantage of the same version system that IHTML uses, without extra effort. If the user wishes to store JavaScript programs external to the page, the

```
<SCRIPT LANGUAGE="JavaScript" SRC="url">
```

tag may be used. IHTML will recognize versioning information in `SCRIPT` tags, doing the usual transformation on the `SRC` attribute; the usual `VMOD` and `VERSION` attributes are available. In this way, external scripts may also be versioned.

Java presents unusual difficulties with respect to IHTML. See 6.1 for discussion.

5.4 Cascading Style Sheets

Cascading style sheets (CSS) allow the author and/or reader of an HTML or XML document to control the rendering of the document on the page. For a given tag (possibly in a given context), a style sheet may specify various attributes such as font, size, colour, weight, and so on. The CSS Specification (version 2) [28] gives a complete list of the properties which may be specified. The "cascading" part of the name refers to the fact that style sheets may be supplied from several places; if multiple specifications are applicable, specificity and precedence rules control which one is applied.

Cascading style sheets are syntactically quite simple. For example:

```
P { color: red; font-style: italic; }
```

instructs the browser to display the contents of all `<P>` tags (i.e. the text in the paragraph) with red, italic letters. Providing multiple tag names indicates the context in which this rule should be applied:

```
h1 emph { color: yellow; font-family: helvetica; }
```

indicates that emphasized text in top-level headings should be yellow and use the Helvetica font. Without the leading `h1`, it would define the appearance of all emphasized text.

Style sheets may be included as part of a document, or may be imported from other files. In the latter case, it may be helpful to include the appropriate version of a style sheet.

```
<LINK REL=StyleSheet HREF="thing.css" TYPE="text/css">
```

will use style sheet `thing.css`; the usual IHTML rules for adding version information to the value of the `HREF` attribute and considering the `VMOD` and `VERSION` attributes are followed.

5.5 The Extensible Markup Language and the Extensible Style Language

The E*x*tensible *M*arkup *L*anguage (XML) and the E*x*tensible *S*tyle *L*anguage (XSL) are tools for the definition of document types, and for describing how they should be displayed and/or transformed, respectively. XML is a restricted form of the Standard Generalized Markup Language (SGML) [29], widely used to define the content of documents. The restrictions eliminate some of the more complex and less widely used features of SGML, while preserving its generality. XML is used to define *document types* in a formal way: what parts they must contain, in what order, and so on. For example, it might be used to define a design document style for a software company:

- A document includes an overview, a high level design, a detailed design, a user interface specification, and a report generation section.
- A high level design includes a functionality description, a risk assessment, a module breakdown, a data flow diagram, ...
- A functionality description includes a feature name, a feature summary, ...

and so on. The intent is to define a document type in terms of its semantic content: "this is a heading" rather than "this is bold, 18 point font". The XML version 1.0 specification [30] has complete details. XSL [31] is then used to provide presentation rules for the document. The rules may indicate the order in which elements of the document are to be displayed, how to display them, what other text to insert, how to transform the document to another format, and so on. Note that XSL is not yet a standard; it is still under development.

The relationships among the various technologies (XML, XSL, CSS, HTML, SGML, DSSSL) are a little complicated. SGML was first defined as a language for describing document types. It is concerned with semantic content, rather than presentation issues. DSSSL (Document Style Semantics and Specification Language) [32] was defined to supply presentation information for documents conforming to SGML document type definitions (DTD). It actually provides more than that: it may also be used to specify transformations from one SGML document type to another, and includes a subset of the Scheme language [33] as an expression evaluator. XML is a subset of SGML, with the same purpose as SGML: to specify formally the semantic content of documents. XSL is to XML as DSSSL is to SGML: it provides presentation information for documents conforming to XML DTDs. It is *not* a subset of DSSSL, but is (almost) a functional subset of it. It is actually an XML DTD describing a language for document transformation (in this case from HTML or XML to some appropriate presentation form). Efforts are under way to amend the DSSSL specification so that XSL is a subset of it ([31], section 1.3). XSL was also heavily influenced by CSS. It is a functional superset of CSS, although it is syntactically very different. Straightforward mechanical translation from CSS to XSL is one of the design goals of XSL. Where does HTML fit into all this? HTML 4.0 [34] is an SGML document type definition. It describes the set of tags which are legal in HTML, how they may be nested, and so on. It is *not* an XML document type definition, but work is in progress to amend it so that it is. (Regrettably, the necessary changes will not be compatible with existing HTML documents.)

It seems possible that once XML and XSL are fully implemented, that the functionality of IHTML might be subsumed by them (although the author can't quite see how). In the meantime, it is straightforward to add IHTML-style versioning to both XML and XSL specifications. One dimension of a document's version could specify the appropriate version of some XML DTD and XSL style sheet. In this way, the DTD, documents, and styles could all evolve in parallel. The syntax for providing a reference to an external DTD and external style sheet in XML are:

```
<!DOCTYPE document-type SYSTEM "urlname.xml">
<?xml-stylesheet href="urlname.xsl" type="text/xsl" ?>
```

(The second line is the current proposed syntax for including XSL style sheets; it should not be considered definitive.) Both cases could be versioned via IHTML in much the same way as other Web objects.

Another possibility is that IHTML could be re-defined in terms of XML and/or HTML 4.0. Features which are currently implemented by server-side transformation of HTML (such as inclusions and execs) are supported as part of the HTML 4.0 standard, making it possible to shift some of the load from the server to the client. Dynamic HTML (described in the following section) may allow even more of IHTML's capabilities to be shifted to the client. However, it seems likely that ultimately, some basic IHTML functionality will still be needed in the server, if only to implement the best fit matching algorithm when some version of a document (or component of a document) is requested.

5.6 Dynamic HTML

Dynamic HTML is a generic term for HTML pages with changing content. The actual implementation technique is (typically) a combination of HTML, scripts (typically ECMAScript [35], the European Computer Manufacturers Association's standardization of JavaScript), and style sheets (CSS or XSL). The glue that holds them together is the Document Object Model (DOM) [36], a standard interface allowing programs to access and modify the content of HTML or XML documents.

The DOM defines naming conventions for all parts of a document, as well as API procedures for examining and changing them. The objective is that any HTML or XML document should be representable as a sequence of DOM API calls. This feature guarantees that the API will be sufficiently flexible for arbitrary alterations to a document. Combined with style sheets (themselves XML documents, and therefore subject to the same kinds of alterations), and a sufficient set of trigger events (i.e. the ability to invoke scripts based on events such as mouse movements or selection of a region of text), scripts have complete control over the contents of a document.

Many of the purposes for which IHTML has been used can be replaced by this technology. Consider, for example, the classic example of setting the background colour of a page. Rather than using IHTML, the page designer could simply supply a link which, when activated, invokes a script to alter the style sheet specification for the page colour, or the HTML `BODY` tag itself to set the background colour. Including successive levels of detail is equally simple. Suppose the whole document was already present, but some elements (the "expert" ones) were marked with tags that were not displayed. To make a component visible, a script would simply change its tag to one that is displayed.

Other things are less easily replaced. Consider a site which exists in several languages. The page designer *could* include all the text for all languages in one document, and control their visibility via tag attributes. However, it would be very tedious to implement. IHTML provides a much simpler solution.

In the end, it seems that dynamic HTML might replace IHTML for many things which are not specifically content-related: colour, formatting, and so on. Presenting information which is, in some sense, fundamentally multi-versioned will probably continue to be simpler in IHTML.

6. Future Work

6.1 Applets

Versioned applets are not supported by IHTML 2. The difficulty is that IHTML depends on the ability to name files arbitrarily (i.e. with version codes). The Java class loader, however, is sensitive to filenames: compiled classes must be stored in files named for the class. The class loader locates Java classes and loads them into the Java virtual machine as they are needed by the running program. It constructs the expected name of the file containing a class from the name of the class, and searches for that file. Several problems arise. Suppose the applet needs version `a:b+c:d` of class `zork`. IHTML will translate the `APPLET` tag to

```
<APPLET CODE="zork.dAyj3_Eab.class" ...></APPLET>
```

to indicate the requested version (since `dAyj3_Eab` is the encoding of `a:b+c:d`). Periods are not acceptable in class names, but that can be solved easily enough by using some other unique symbol. Assume that has been done. The class loader will expect a class named `zork.dAyj3_Eab`, but the class in the file will be named `zork`. The class loader will complain. Suppose the user, attempting to fix this problem, names the class with its version code: each version of the class has a different name (e.g. `zork.dAyj3_Eab`). Ignoring the resulting programming nightmare, the problem still isn't solved, because of the best fit algorithm. The class loader asks for version `a:b+c:d`, but suppose the best fit is `a:b`. The server will send along version `a:b`. Again, the name will be wrong, and again the class loader will complain. Suppose that we get past this problem somehow; there is still the problem of how to tell the class loader that any classes used by the main class must also be loaded in the appropriate version. There does not appear to be a simple solution to this problem. The following paragraphs describe three possible (but not very simple) solutions.

One solution is to consider versions of the `codebase` attribute of the `APPLET` tag, instead of the `code` attribute. The programmer would create separate directories for each version of the applet. IHTML would be extended to understand versioned directories (at least in this one case). The drawback is that versioning would be done on whole applets, instead of classes. Suppose there were seven user-defined classes in the applet, but only the main one varied from version to version. All six others would also have to be duplicated in all `codebase` directories. This approach eliminates much of the benefit of the best fit technique, may require extensive duplication of code, and generally seems somewhat counter to the philosophy of IHTML.

A better solution is to modify the class loader to understand versions. The general idea is that the class loader would recognize `zork.dAyj3_Eab.class` as a versioned class file name, and expect to receive a class named `zork`. It would also store the version code, so that it could ask for version `dAyj3_Eab` of any other classes that were needed. It would not need to understand how version encoding works, or manipulate versions in any way; just recognize that it is a version, and act accordingly. The only real flaw with this solution is that it is a client-side change. Since one of the major objectives of IHTML is to be able to use it without any changes on the client side, it is a major flaw. However, if an intensional browser were developed (see Section 6.6), the modified class loader would be an appropriate solution.

Another (server-side) solution is to process Java class files somewhat the way IHTML files are processed, translating class names to the ones the class loader is expecting (both the name of the current class, and the names of classes used by this one). Using this technique, the programmer would name all the versions of a given class just with the name of the class, and leave it up to the translation software to add version codes. The format of a class file is very well defined (see [37], chapter 4), so all occurrences of class names could be identified and changed quite easily. Some care would have to be taken to ensure that new class names were compatible with Java syntax, but that is not a difficult issue. As well, the translator would have to leave standard Java class names unchanged. At first, this solution might seem quite

ugly, but actually it is not much different from what happens with IHTML files. The main advantages of this approach are that it is a server-side solution, and that the programmer is not required to name her classes strangely. The main disadvantage is that implementing the Java `.class` file translation software is a somewhat tedious task.

6.2 Enhanced Version Language

Several additions to the version language would be useful. Some have already been described, by Plaice and Wadge in [17] and Yildirim in [20], but have not been implemented in IHTML 2. Numeric subversions would allow major and minor revision tracking in a straightforward way. For example, treating version 3.3 as a refinement of version 3.2 is a natural and useful addition. Note the use of the period "." as the numeric subversion operator. A "greatest lower bound" operator (using "&"), for files which are appropriate for two different versions, would allow improved re-use of pages. For example, consider a page in English, Mandarin and Cantonese. Since written Mandarin and Cantonese are the same, a version such as `lang:mandarin&lang:cantonese` might be appropriate, indicating that this version refines to both `lang:mandarin` and `lang:cantonese`. Subversions (using "%" as the subversion operator) would allow the expression of natural notions such as `Mac%68k`, `Mac%powerpc`, and `Mac%G3`. All are refinements of plain `Mac`. A grammar including these additions, from [17], is:

$$V ::= \varepsilon \mid N \mid x \mid V\%V \mid V\&V \mid V+V \tag{6.1}$$

$$N ::= n \mid N.n \tag{6.2}$$

Refinement rules for the new elements include:

$$n \leq m \;\rightarrow\; n \sqsubseteq m \text{ , for integers } n, m \tag{6.3}$$

$$N \sqsubseteq N.m \tag{6.4}$$

$$N \subseteq M,\ N \neq M \ \rightarrow\ N.n \subseteq M \tag{6.5}$$

$$V \ \subseteq\ V\%V' \tag{6.6}$$

$$V_1 \subseteq V_2 \ \rightarrow\ V\%V_1 \subseteq V\%V_2 \tag{6.7}$$

$$V\&V' \ \subseteq\ V \tag{6.8}$$

Strictly speaking, equation 6.7 is not necessary. It follows from equation 3.11 and from the following rules from [16]:

$$V\%V' + V\%V'' \ =\ V\%(V' + V'') \tag{6.9}$$

$$V \subseteq V' \ \rightarrow\ V + V' = V' \tag{6.10}$$

It is included despite its redundancy because it greatly simplifies demonstrating some very obvious relationships like $x\%3 \subseteq x\%4$, which are otherwise somewhat tedious to prove.

Possible new enhancements include subdimensions, a notion of an "infinitesimal" dimension value (more specific than vanilla but less specific than any other value), and a "maximum" dimension value (more specific than any other dimension value). They were suggested by Wadge, in unpublished communications.

The "infinitesimal" idea appears straightforward. A new symbol τ is introduced as a minimal dimension index, with the following refinement rules:

$$D{:}\tau \ \subseteq\ D{:}v\ ,\ \text{if } v \neq \varepsilon \tag{6.11}$$

$$D{:}v \ \not\subseteq\ D{:}\tau\ ,\ \text{if } v \neq \tau \text{ and } v \neq \varepsilon \tag{6.12}$$

$$D{:}\tau \ \not\subseteq\ D{:}\varepsilon \tag{6.13}$$

The intent is that τ is the least refined value that is not vanilla. It would be used in situations where the user wants to indicate that a dimension must have *some* non-vanilla value, but it doesn't matter what it is. For example, suppose the user wants to include varying fragments of text depending on, not the value of some dimension,

but which dimensions have values. An ISELECT with cases such as

```
<ICASE VERSION="D:*">...text...</ICASE>
```

(assuming τ is typed as "*") will include the *text* associated with the first dimension D which is non-vanilla.

The "maximum" dimension value also appears straightforward. ω might be an appropriate symbol, with this refinement rule:

$$D{:}x \subseteq D{:}\omega\ , \text{ for any index } x. \tag{6.14}$$

One application is to include the absolute latest version of some numerically-versioned component:

```
<!--#include virtual="zork.html" vmod="revision:!" -->
```

(assuming ω were typed as "!") would include the version with the highest revision number. The site implementer would need to be careful: any set of versions without a unique maximum element will fail to have a best fit to ω.

Subdimensions are, as yet, poorly defined. Initial consideration suggests that they behave very much like subversions. For example, the equivalence rule

$$D{:}D'{:}x + D{:}D''{:}y = D{:}(D'{:}x + D''{:}y) \tag{6.15}$$

is remarkably similar to the distributive rule for "%" and "+":

$$V\%V_1 + V\%V_2 = V\%(V_1 + V_2)\ . \tag{6.16}$$

Similarly, the following refinement rules are very similar:

$$V_1 \subseteq V_2 \ \rightarrow\ D{:}V_1 \subseteq D{:}V_2 \tag{6.17}$$

$$V_1 \subseteq V_2 \ \rightarrow\ V\%V_1 \subseteq V\%V_2 \tag{6.18}$$

It seems that, with appropriate care, one or the other may be eliminated.

Since dimensions seem necessary (or at least useful), let us try eliminating subversions. At the same time, let us eliminate the privileged status of dimensions: at present, the top level of a version must be a disjunction (using "+") of dimension specifiers ($D : V$). I propose the following scheme: anything without an explicit dimension name is considered part of the "anonymous" dimension (which we might write as underscore, by analogy with Prolog's anonymous variables). Consider a version including subversions, such as

```
lang:english + hw:mac%G3 + food:french%(paris+rennes) + zork
```

and the corresponding version with subdimensions instead:

```
lang:english + hw:mac:G3 + food:french:(paris+rennes) + zork
```

Humans might read `lang` as a different kind of thing from `mac`, but there seems no reason to consider it differently from the point of view of version comparison. The trailing `zork` is treated as `_:zork`, for comparison purposes.

6.3 Aggregation

IHTML 1 supports a form of aggregation in which all versions of a component which refine to the current version are included in a document. IHTML 2 supports a quite different notion of aggretation, using `ICOLLECT` tags. Each has strengths and weaknesses, but some larger, unified notion of aggregation would be a good thing.

Several questions arise. How should the elements be ordered? Should the software include all relevant versions (V such that $V \sqsubseteq R$, where R is the requested version), or all *most* relevant versions (as defined by the best fit definition from Section 3.1.1)? How should the site implementer indicate on which dimensions aggregation may occur, or should it be allowed in all dimensions? Should a particular component of a page be "hard-coded" as an aggregate, or should aggregation be allowed on any object, specified in the tag somehow?

IHTML 1 answers these questions, but they may not be the right answers. In particular, the order in which items are listed is arbitrary, which is unlikely to be satisfactory. Also, aggregated components are identified by the names of the files, making it impossible to create a page component which is aggregated in some situations but not others.

IHTML 2's aggregation style answers these questions differently, but the result is a rather restrictive form of aggregation. As currently implemented, it includes all relevant versions, as opposed to all most relevant ones; the dimensions on which to aggregate are defined by the syntax of the `ICOLLECT` cases; aggregates are hard-coded into IHTML pages; and ordering is based on the order of the `ICOLLECT` cases. It works, but something more flexible would be better. For example, it would be nice to be able to add a new version of some component to the repository, and have it included into the aggregation.

One possibility is to allow the site implementer to specify aggregation in the `#include` tag:

```
<!--#include virtual="zork.html" aggregate="beer:!" -->
```

would indicate how component `zork.html` is to be aggregated. The `aggregate` attribute indicates that aggregation is requested; the "`="beer:!"`" operand indicates that it is acceptable in the `beer` dimension only (i.e. if both `lang:french` and `lang:english` match, it is an error, but if `beer:guinness` and `beer:newcastle` match, both are included).

6.4 Version Manipulation

The introduction of dimension variables (see Section 3.6) could be the beginning of a more sophisticated treatment of versions. It seems that it might be useful for IHTML pages to manipulate versions in a variety of ways: examine them, merge them, perform tests on them, construct them from scratch, and so on. Some of these things are already present, in limited forms, but it would be interesting to explore some more general ideas of a version manipulation language.

6.4.1 Javascript and IHTML

The idea of more flexible version manipulation opens the further question of interactions between JavaScript and IHTML. At the moment, interaction is extremely limited. The user may construct a versioned URL using the `#iurl` tag (see Section A.3.2), but once the page is generated, no further manipulation is possible. One useful capability which is currently impossible is to be able to select values for several dimensions in an HTML form, then go to the version thus described with a single HTTP transaction. A JavaScript program might also make use of the values of specific dimensions to choose courses of action, construct new versions in an arbitrary way, and so on.

6.5 IHTML Editor

The use of WYSIWYG HTML editors is increasingly widespread. Consequently, the development of an IHTML editor is a requirement for widespread use of IHTML. At the most basic level, an IHTML editor could be a small enhancement of a standard HTML editor, with the capability to save a file as a particular version of something, to mark lists or other components as included only if the current version meets some criteria (i.e. ISELECTs), and to mark an inclusion or anchor with a version or vmod. The editor itself would still be a simple single-version editor.

A more ambitious idea to explore is the idea of versioned editing. It might be possible to present a versioned document to the user as a single entity, with a "current version" not dramatically different from, say, the "current font" or "current text colour". One might mark a region of text as belonging to the "expert" version, another as part of the "normal" version, and yet another as part of both.

6.6 Intensional Browser

Two useful features for a future version of IHTML are versioned applets and an intensional "back" button. The first is discussed in Section 6.1; recall that one good solution required an intensional browser (or at least some modifications to the

browser). The second is a button which takes the user to the current version of the previous page, regardless of the version it was when the user was there. For example, suppose the customer is browsing page `zork`, links to page `rogue`, changes the current version to `level:expert`, then hits the "back" button. The standard behaviour would return her to the previous version of the current page. An intensional "back" would recognize that the last link didn't change the page, but only the version: it would go back to the `level:expert` version of page `zork`.

Both of these features require that the browser be aware of versions. Other features could also be added, once the browser became a candidate for enhancement. For example, IHTML-to-HTML translation could be moved to the browser side, greatly simplifying the server. Specific benefits include browser-side processing of `<ISELECT>` structures: one could change the version of the current page in a variety of ways, without having to go back to the server for new copies of the page.

On a somewhat higher level, the browser itself could be versioned: the "expert" version provides more flexibility in the user interface than the "beginner" version, and so on.

6.7 Non-IHTML Applications

There is nothing especially HTML-specific about a lot of this work. It could apply to other versioned text, such as software or non-HTML documents. In fact, many of the original ideas for this work arose from the software versioning systems Lemur and Marmoset, described in Section 2.3. Perhaps some of the ideas developed here could usefully be transferred back to versioned software development.

6.8 Stand-alone IHTML-to-HTML Translation

One relatively simple tool which would improve the usability of IHTML is a stand-alone IHTML-to-HTML translation utility (suggested by monica schraefel in unpublished communications). At present, there is no way to see the result of IHTML processing without an IHTML-enhanced HTTP server. At the simplest level, a stand-

alone translator would allow developers on machines without servers to find out what various versions of their pages will look like. A more sophisticated version could provide extensive consistency checking, reporting to the user a wide variety of errors which might be hard to diagnose when encountered via a server.

7. Conclusions

The objective of this work was to make IHTML usable, and in doing so find out what new features should be added to make it more useful, how to use it to advantage, and to make versioned sites available to the world. This chapter describes some user experiences and some advantages of IHTML, then summarizes the results of the project.

7.1 User Experiences

Although no formal study of users' reactions to IHTML has been done, the author has discussed their experiences with several users, with fairly uniform results. On the whole they find it a usable tool, but with several notable weaknesses.

The intensional model itself appears relatively straightforward for users to understand. A few examples are usually sufficient to give people the general idea. A collection of templates showing standard techniques would be a major asset, though, to save each user from having to re-invent standard techniques themselves.

With respect to specific features, conditional inclusion of text (`ISELECT`) is a great asset. Many effects are easily achieved which would be tedious using separate files. Dimension variables are also useful, particularly in some more complicated applications.

The major weaknesses are the diagnostic messages, and the site development tools. Users frequently find it difficult to locate errors in their IHTML files, since error messages are misleading or absent. An off-line tool to translate IHTML to HTML and verify pages for correctness would be a tremendous asset. The site implementation utilities also give users trouble. They are quite unsophisticated, and do not hide enough of the implementation from users. Some weaknesses are simple bugs, easily fixed. The more important one is that they allow users to see the storage mechanism of versioned entities: directories containing files named with versions. Some kind

of repository technology for storing versioned pages is called for. Paul Swoboda is currently developing such a utility; a future version of IHTML should certainly take advantage of it.

The major remaining weakness is that users must understand IHTML syntax in order to use it. An HTML editor which supports versioned editing is a strong requirement, particularly since relatively few creators of Web sites still write HTML by hand.

7.2 Additional Advantages

IHTML 2 can provide significant disk space savings over other approaches to providing multiple versions of a page. For example, one page developed by Bill Wadge has over four million versions. Given that a typical version of this page is about 15 kilobytes long, the copy-and-modify approach to versioning would require four million files, taking up roughly sixty gigabytes of disk space. The IHTML 2 version uses 38 files (split across 25 versioned page fragments), and a total of 27,707 bytes of IHTML source. The disk space saving is several orders of magnitude, but the true significance of the numbers is the simple fact that it is impossible (in any practical sense) to create so many versions of a page by hand. In this instance, IHTML is not only useful but essential.

A more typical example is a versioned home page created by Peter Driessen. A fairly normal-looking home page at first, it takes advantage of IHTML 2 to allow the user to choose background and text colours, the language of the page, and one of three areas of interest to focus on. In addition, selecting his name causes a picture of him to appear or disappear. Altogether, the site has 600 versions. At about 2500 bytes per version, the naive approach would require about 1.5 megabytes of disk space. The IHTML version uses 17,911 bytes, in 7 versioned page fragments and 22 individual files. Again, the saving in disk space is non-trivial.

There are other serendipitous benefits. Schraefel and Wadge noticed that it is possible to cache versioned pages intelligently, since each version has a different name

(as the browser sees it). A side effect is that it is possible to create bookmarks which identify the contents of frames, not just the name of the top-level page. Using normal HTML, a bookmark for a page which uses frames will load the initial contents of each frame, rather than the one present when the bookmark was created. If the content of the frames is determined by IHTML versions, the bookmark will retrieve the correct version of each frame, as well as that of the main page.

7.3 Summary

IHTML 2 seems to be reasonably usable. The installation step is straighforward and Unix-friendly (people other than the developer have been able to install it without difficulty). Several users (not directly involved in the development of the software) have successfully created non-trivial versioned sites. It has been used by undergraduate students for course projects in at least two University courses. The software is portable (at least among Unix platforms): it has been tested on the SunOS, Solaris, and Linux operating systems (all rather different implementations of Unix). Regarding performance, since the software is integrated into the Web server itself, there is virtually no penalty over, say, standard Apache server-side includes. The software has also reached a reasonable level of stability: aside from system shutdowns and the occasional bug introduced by a new feature, the server keeps on going and going.

A variety of new features have also been implemented, as a result of experiments with intensional sites. Conditional inclusion of text (`ISELECT` and `ICOLLECT`) was added as a replacement for IHTML 1's aggregation, and to allow the combining of versions of small files into one file. Dimension variables arose from a need to transfer information from one dimension to another. Treating unversioned HTML files as the vanilla version, if a request was received for some version of the file, also arose out of user experiences. Some things were also removed because experience showed that they were not the right answer. The first implementation of inclusion deliberately allowed conditional inclusion elements to begin in one file and end in another (possibly an included file, possibly a parent file). This "feature" proved annoying, so the current

version insists that inclusions nest properly with other IHTML tags.

With improved reliability and performance, and a variety of new features, IHTML 2 meets its objective of providing a usable, stable implementation of IHTML. However, it is by no means perfect. There are some key developments still to be made before IHTML is ready to be released on an unsuspecting world.

Bibliography

[1] W. W. Wadge, G. D. Brown, m. c. schraefel, and T. Yildirim. Intensional HTML. In E. Munson, C. Nicholas, and D. Wood, editors, *Proc. Principles of Digital Document Production 1998*, 1998.

[2] Andrew Cormack. *Multi-lingual Web Pages.* `http://www.cf.ac.uk/uwcc/comp/cormack/welsh/biling.html`.

[3] Rudolph Carnap. *Meaning and Necessity.* XXX, YYY.

[4] Yehoshua Bar-Hillel. Indexical expressions. *Mind*, LXIII:359–379, 1954.

[5] Richmond Thomason, editor. *Formal Philosophy: Selected Papers of Richard Montague.* Yale University Press, 1974.

[6] Dana Scott. Advice on modal logic. *XXX*, pages 143–173, YYY.

[7] Saul Kripke. Semantical considerations on modal logic. In *Proceedings of a Colloquium on Modal and Many-valued Logics*, volume XVI of *Acta Philosophica Fennica*, pages 83–94. Societas Philosophica Fennica, Helsinki, 1963.

[8] Robert Audi, editor. *Cambridge Dictionary of Philosophy.* Cambridge University Press, 1995.

[9] Brian F. Chellas. *Modal Logic: An Introduction.* Cambridge University Press, 1980.

[10] Ted Honderich, editor. *The Oxford Companion to Philosophy.* Oxford University Press, 1995.

[11] Robert Goldblatt. *Logics of Time and Computation.* Number 7 in CSLI Lecture Notes. Center for the Study of Language and Information, 1987.

[12] A.A. Faustini and W.W. Wadge. Intensional Programming. In J.C. Boudreaux, B.W. Hamill, and R. Jernigan, editors, *The Role of Languages in Problem Solving 2*, pages 119–132. Elsevier Science Publishers B.V. (North Holland), 1987.

[13] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language*, volume 22 of *A.P.I.C. Studies in Data Processing.* Academic Press, 1985.

[14] M. F. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[15] W. F. Tichy. Rcs—a system for version control. *Software—Practice and Experience*, 15(7):637–654, 1985.

[16] J. Plaice and W.W. Wadge. A New Approach to Version Control. *IEEE Transactions on Software Engineering*, pages 268–276, March 1993.

[17] J. Plaice and W. W. Wadge. Reducing the complexity of software configuration. In M. Tchuente, editor, *Proc. 1st African Conference on Research in Computer Science*, pages 85–96, Yaounde, Cameroon, October 1992.

[18] W.W. Wadge and Alan Yoder. The Possible-World Wide Web. In M.A. Orgun and E.A. Ashcroft, editors, *Intensional Programming I*, pages 207–213. World Scientific, 1996. (based on the papers at ISLIP '95).

[19] Netscape Communications Corporation, `http://home.netscape.com/comprod/development_partners/plugin_api/`. *Netscape Navigator LiveConnect/Plug-in Software Development Kit.*

[20] Taner Yildirim. Intensional HTML. Master's thesis, University of Victoria, Victoria, British Columbia, Canada, 1997.

[21] m. c. schraefel. *Talking with Antigone.* PhD dissertation, University of Victoria, Victoria, British Columbia, Canada, 1997.

[22] m. c. schraefel. A thousand papers for ISLIP97. In W. W. Wadge, editor, *Proc. Tenth International Symposium on Languages for Intensional Computing*, pages 41–45, Victoria, British Columbia, Canada, May 1997.

[23] The Apache Group. *The Apache HTTP Server Project.* `http://www.apache.org/`.

[24] Netscape Communications Corp. *JavaScript Guide.* `http://www.netscape.com/eng/mozilla/3.0/handbook/javascript/`.

[25] Ted Gesing and Jeremy Schneider. *JavaScript for the World Wide Web.* Peachpit Press, Berkeley, CA, 1997.

[26] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[27] *W3C — World Wide Web Consortium.* `http://www.w3.org`.

[28] The World Wide Web Consortium. *Cascading Style Sheets, level 2: CSS2 Specification.* `http://www.w3.org/TR/REC-CSS2/`.

[29] International Organization for Standardization, Geneva. *ISO 8879:1986(E). Information processing - Text and Office Systems - Standard Generalized Markup Language (SGML)*, 1986.

[30] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0*, 1998. `http://www.w3.org/TR/1998/REC-xml-19980210`.

[31] World Wide Web Consortium. *A Proposal for XSL*, 1997. `http://www.w3.org/TR/NOTE-XSL.html`.

[32] International Organization for Standardization. *ISO/IEC 10179: Document Style Semantics and Specification Language (DSSSL)*, 1996.

[33] *Revised(5) Report on the Algorithmic Language Scheme*. available at `http://www-swiss.ai.mit.edu/scheme-home.html`.

[34] World Wide Web Consortium. *HTML 4.0 Specification*, 1998. `http://www.w3.org/TR/REC-html40/`.

[35] European Computer Manufacturers Association. *ECMAScript: A general purpose, cross-platform programming language*, June 1997. `http://www.ecma.ch/stand/ecma-262.htm`.

[36] World Wide Web Consortium. *Document Object Model Specification*, 1998. `http://www.w3.org/TR/WD-DOM/`.

[37] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[38] Timothy Bell, John Cleary, and Ian Witten. *Text Compression*. Prentice-Hall, Inc., New Jersey, 1990.

APPENDIX A
IHTML Reference Manual

A.1 Overview

Intensional HTML (IHTML) is an extension of standard HTML, to support WWW pages which may exist in many different versions. An IHTML page is considered to be a family of actual pages, distinguished by versions. The implementer of an IHTML site decides in what versions the site will exist. Any particular page (or other WWW object: image, CGI script, or whatever) may exist in many versions, just a few selected versions, or only a single version. A page (object) which exists in no versions at all shall be deemed nonexistent, and is outside the scope of this document.

A versioned object is represented as a directory, named as the object would normally be named (e.g `thing.html`). In the directory are various versions of the object, named with a version code and the appropriate MIME suffix: `aai.html`, `zaslkwe0if.html`, and so on. The version codes are a compressed form of the version strings described below. The site implementer uses IHTML utilities to edit versioned pages, to copy files into versioned directories (objects), and to list the available versions of IHTML objects.

The intent is that the implementer of an IHTML site should view the versioned HTML pages (or other MIME types) as single objects, existing in multiple versions. The tools described below (`ivi`, `icp`, `ils`, `irm`) allow the implementer to construct and edit versioned objects without ever having to look inside the *`whatever.suff`* directories. A future generation of the tools might, in fact, not store things this way at all; it is purely an implementation detail. As such, it should be as transparent to the user as possible, and the user should not depend on the knowledge that these things are, in fact, directories.

A.1.1 Best Fit

It is not necessary to provide files for each version that the user might request. The server uses a *best fit* algorithm to choose the version that most closely approximates the requested one. If the requested version is available, it will be used. Otherwise, if there is a unique closest match, it is used. If there is not a unique closest fit (no match at all, or more than one equally good), an HTTP 404 error code is returned.

See Section A.2 for details on versions and the best-fit algorithm.

A.1.2 Intensional Links and Includes

The *best-fit* technique alone is not sufficient to make a useful system. Consider, for example, a page in which background colour could vary among a dozen possible colours, while the text could vary among three languages. The naive approach would require 36 versions. Of these versions, the three groups of twelve would each vary only in a single line: the one setting the background colour. Instead, we use *intensional includes* and *intensional links*.

An *include* is very much like the `#include` directive used with the C-language preprocessor: the complete contents of some file are included in another file, replacing the `#include` statement. With IHTML, though, includes are *intensional*: they refer to families of versions, rather than individual pages. Consider the example of the previous paragraph. Instead of 36 pages, there could be a single version of the main page, which includes the text of the page, and the line setting the background. The includes refer to the *current version* of the included file. Whatever version is requested, the vanilla version of the main page is used, since it is the only version. However, the includes in the file are evaluated with respect to the *requested* version, not the actual version used. The result is that if, say, the "blue, English" version of the page is requested, the "blue" version of the background file and the "English" version of the text file are used. Note that the background file exists in 12 versions, one for each colour. The text file exists in 3 versions, one for each language. Duplication

of data is minimized.

Links are treated in the same way. A link is considered to be a link to a family of versions, depending on the requested version of the current page. In this way, even if there is only one version of the current page, links to subsequent pages will still preserve the requested version.

A.1.2.1 Trans-version Links and Includes

By default, links and includes refer to the same version of the new page or included file as the page being viewed. However, it is also possible to create links to, or include, different versions of pages. These so-called *trans-version* links or includes may specify a modification to the current version, or a whole new version. For example, the anchor

```
<A HREF="zork.html" VMOD="language:french">Zork!</A>
```

creates a link to page "zork.html", with the same version as the current page, except that the *language* dimension is set to *french.* Similarly,

```
<A HREF="zork.html" VERSION="language:french">Zork!</A>
```

creates a link to page "zork.html" where the *language* dimension is *french*, and all other dimensions are vanilla.

A.1.3 Server Operation

The server behaves in most respects like a normal HTTP server. However, versioned files are treated specially. Requests of the form `.../name.suff`, or `.../name-.version-code.suff`, where `.../name.suff` is a directory, are interpreted as intensional requests. If `version-code` is included, it identifies the requested version of the page; if it is not, the vanilla version of the page is assumed. The request is transformed into `.../name.suff/bestfit-code.suff`, where `bestfit-code` is the encoding of the exact version requested, if it exsts, or the best fit to it, if not. If there is no unique best fit (i.e. the set of less specific versions does not have a maximum

element), the server returns the usual "404 - document not found" error (and optionally logs an error message to an IHTML log file). If the requested *.../name.suff* exists, but is not an IHTML directory, but just a regular file, it is considered the vanilla version of the file, and is used no matter what version was requested.

Once the appropriate file is chosen, the file (if it is IHTML, and not some other file type) is translated from IHTML to standard HTML. `include` directives are replaced with the contents of the named files (using the same best-fit approach as above). Included files may also include further files. Other IHTML-specific tags are modified so that the linking attributes have the versions encoded in their URL's, and the VERSION and VMOD attributes are removed. The result is standard HTML, with version information encoded in links.

A.2 Versions

IHTML currently supports a limited version space consisting of named dimensions and discrete index values in those dimensions. Dimension specifications are parsed according to the following grammar:

```
<version>     ::= <dimspec>
                |  <version> "+" <dimspec>
<dimspec>     ::= <dimension> ":" [ <index> ]
<dimension>   ::= <identifier>
<index>       ::= <identifier>
<identifier>  ::= 1 or more letters, digits, or "_"
                |  [ <identifier> ] "$"<label>
                |  [ <identifier> ] "${"<label>"}" [ <identifier> ]
<label>       ::= 1 or more letters, digits or "_"
```

Dollar signs indicate dimensional variables. The label following the dollar sign is the name of a dimension; it is replaced by the index of that dimension in the current version.

For example, `language:english+cuisine:french` describes a version with the value "english" in the "language" dimension, and "french" in the "cuisine" dimension. If the current version is `alpha:bravo+charlie:delta`, then the expression `x$alpha:y$charlie` evaluates to the version `xbravo:xdelta`.

Versions can be partially ordered in a refinement relationship, symbolized by "$\subseteq$": "$x \subseteq y$" reads "x is more generic than y", or "y refines x". In this version space, the rules for refinement are:

$$\varepsilon \subseteq V \tag{A.1}$$

$$V \subseteq W,\ W \subseteq X \ \rightarrow\ V \subseteq X \tag{A.2}$$

$$D : V \ \subseteq\ D : V + E : W \tag{A.3}$$

$$N < M \ \rightarrow\ D : N \subseteq D : M, \text{for integers N, M} \tag{A.4}$$

$$V \subseteq V', W \subseteq W' \ \rightarrow\ V + W \subseteq V' + W' \tag{A.5}$$

In English: refinement is transitive, a version which joins two simpler versions with the "+" operator is a refinement of each of them, and any version is a refinement of ε, the vanilla (or most generic) version.

Version specifications used in IHTML files or on the command line should not contain any embedded spaces, tabs, or other extraneous characters: just the dimension and index identifiers, and the "+" and ":" punctuation symbols.

A *version modifier* may be used in trans-version links or includes, with the `VMOD` attribute. The syntax of version modifiers is slightly different: the "+" operator is replaced with the "," operator. The reason is that version modifiers (as opposed to versions) are intended to override existing version specifications, as well as refine them. For example, if the current version of a page is `a:b+c:d`, and there is a link to the `c:e,f:g` version of another page, the correct version of the new page should be `a:b+c:e+f:g`. If the "+" operator were used, the result (following the rules of the "+" operator) would be `a:b+c:d+c:e+f:g`. Rather than have two different interpretations of "+", we use a different operator in the `VMOD` case.

A.3 IHTML Syntax

IHTML is a superset of HTML. Some standard tags support additional attributes, and some attributes are interpreted somewhat somewhat differently from

standard HTML. New tags are provided for including text in a page, and for selectively including parts of a page in the final page sent to the browser.

A.3.1 Modified HTML Tags

Affected tags are `A`, `IMG`, `FORM`, and `FRAME`: the tags with attributes that refer to other documents or files. These tags may include the following new attributes:

`VMOD=''version_mod_spec''`

This attribute specifies that the requested version of the linked object is the current version, plus the dimensions specified in the *version_mod_spec* (which may override some dimension values in the current version).

`VERSION=''version_spec''`

This attribute specifies that the requested version of the linked object is exactly *version_spec*, ignoring the current version of the present page. Any dimensions not explicitly mentioned in the *version_spec* are considered to be vanilla (i.e. the most generic, or least refined, value).

Specifying both `VMOD` and `VERSION` in one tag is a little strange, but will be interpreted as if the *version_spec* were the current version, with the *version_mod_spec* modifying it in the usual way.

For anchor tags, the `HREF` attribute is optional, and defaults to the current page. The purpose of this behaviour is to make it easy to provide links to different versions of the same page. For example, the tag

```
<A VMOD="language:french">
```

is a link to the current page, but with the language dimension set to "french". When used with includes, this behaviour makes it possible to create a file of version-changing links, and include it on many pages, so that, for example, every page has the same set of links to different versions of the site.

To make a link to an object served by a non-IHTML server, the link will need to contain a `VERSION=""` attribute, to prevent IHTML from adding the current version to the `HREF` attribute.

There is currently no way to specify a version if referring to a whole directory. For example, the URL `http://lucy.uvic.ca:8888/~gdbrown/` should find the file `index.html` in user gdbrown's `.www` directory. The problem is that there is no place to put the version of `index.html` that is required, without specifying the file exactly:

```
http://lucy.uvic.ca:8888/~gdbrown/index.vG8vccqmfeyce.html .
```

One possible solution is to allow URL's of the form `.../dirname.version_code/`. However, there would be no way to distinguish between a request for version *version_code* of directory *dirname*, and a request for a directory named *dirname.version_code*.

A.3.2 Text Substitution Tags

Several IHTML-specific tags are provided for insertion of text into pages. The syntax of these tags has been chosen to reflect the syntax of Apache server-side includes, and must be inside comment delimiters:

```
<!--#tag attributes -->
```

The motivation for this syntax is that these tags will not interfere with representation of the rest of the page, if the page is viewed with a browser before being translated to standard HTML.

`include virtual=URL [vmod=version_mod] [version=version]`

A standard Apache server-side include, except that the included item is versioned like any other IHTML page. Note that the version that will be requested for the included file is the *requested* version of the current page (in the absence of a `VERSION` attribute), not the best-fit version. For example, if the current page exists only in the vanilla version, but the request was for version "a:b+c:d+e:f",

the software will look for version "a:b+c:d+e:f" of the included file (subject to `version` and `vmod` attributes, naturally).

`exec virtual=`*URL* `[vmod=`*version_mod*`] [version=`*version*`]`

The program identified by *URL* is executed; its output is processed by IHTML and included in the page. This is identical to the Apache server-side exec feature, except that the program may be versioned, and the output is translated from IHTML to HTML.

`iurl href=`*URL* `[vmod=`*version_mod*`] [version=`*version*`]`

This tag is replaced in the output by the *URL*, with version information added, according to the current version and (optionally) `vmod` and `version` attributes. It is intended for use in JavaScript programs, or other places where a versioned URL is needed, but where IHTML does not normally construct them. This tag is useful in conjunction with `starttag` and `endtag`, to add version codes to non-standard tag attributes.

`echo dimension=`*dimension_name*

This tag echoes the value of the named dimension, according to the current version, to the HTML page.

`dumpver`

This tag echoes the complete current version to the HTML page. It is useful mainly for debugging.

`starttag`

This tag generates a literal "<" in the output. It is used to enclose arbitrary text in angle brackets, thereby turning it into a tag. It is necessary since nesting of tags is illegal, yet it is sometimes helpful to be able to use the `iurl` tag inside a tag which is not normally versioned, or inside JavaScript code.

`endtag`

This tag generates a literal ">", used to close a generated tag (used with

`starttag`, above).

A.3.3 Document Structuring Tags

Many common IHTML idioms involve versioning on short fragments of text. For example, the "background colour" example from section A.1.2 uses many versions of a single line of text. One approach to implementing this kind of thing is to `#include` a fragment of IHTML which varies in the background dimension, with a separate version for each possible background colour. However, these many small files are tedious to create and maintain. To avoid this problem, IHTML provides for selective inclusion of text, based on the current version.

The best analogy is a C-language *switch* statement, which selectively executes code fragments. The IHTML equivalent is the `ISELECT` tag, with the following syntax:

```
<ISELECT>
  <ICASE VERSION=v1>...text1...[</ICASE>]
  <ICASE VERSION=v2>...text2...[</ICASE>]
  <ICASE VERSION=v3>...text3...[</ICASE>]
  ...
  <ICASE>...default text...[</ICASE>]
</ISELECT>
```

Starting from the top, the translation software finds the first v_i such that $v_i \sqsubseteq V$, where V is the current version. The corresponding $text_i$ is included in the output. If the trailing `</ICASE>` is missing, the following text is included, until a `</ICASE>` is seen or the `</ISELECT>` is seen. In other words, it falls through in the same way that a case in a C *switch* falls through if there is no `break` statement at the end of the case. If no `VERSION` is supplied, it defaults to vanilla. Since $\epsilon \sqsubseteq V$ for any version V, `<ICASE>` behaves as a default case, matching whenever none of the earlier cases matched. Although there is nothing preventing other cases from following the default case, they will never be selected.

The `<ICOLLECT>` tag is similar, except that *every* matching case is included in the output, instead of the first one. In this situation, it may be appropriate to have default cases in the middle of the expression, for text which should always appear.

See section A.5 for usage examples.

A.4 Site Construction Utilities

Several Unix utilities are provided to aid the site implementer in creating versioned pages. `ivi` invokes an editor on some version of a text document; `ils` lists the versions in which things exist; `icp` copies files into and out of versions of pages; `irm` removes named versions of a document. Details for each command follow.

`ivi` — edit a particular version of an IHTML page

Syntax: `ivi [-v` ***`version`*** `[-b` ***`baseversion`***`]]` ***`filename`***

Invokes an editor on the named *version* of the IHTML page named *filename*. If the supplied version doesn't exist, and *baseversion* is supplied, it copies the *baseversion* version of the page to the *version* version, then edits that page. If *version* or *baseversion* doesn't exist, but a unique best fit to it does, that version is used as the *baseversion* instead.

`ivi` invokes the editor specified by the `$EDITOR` environment variable. Defaults to `/usr/ucb/vi` (the author's favourite editor) if `$EDITOR` is undefined.

`icp` — copy a file into or out of a versioned object

Syntax: `icp [-f] [-v` ***`version`***`]` ***`srcfile destfd`***

Copies *srcfile* to *destfd*, which may be a file or directory. If either *srcfile* or *destfd* is an IHTML directory, the supplied *version* of the file is used (to copy from, or to, or both). The `-f` option forces `icp` to overwrite existing files.

For example, the command "`icp -v a:b thing.html zork.html`", for a normal file `thing.html` and an IHTML page `zork.html`, will copy `thing.html` to `zork.html/dQNzGa.html`, where `dQNzGa` is the encoding of the version specification `a:b`.

`ils` — list the versions of one or more versioned objects

Syntax: `ils [fdname ...]`

This behaves similarly to the standard Unix `ls` command. If no arguments are specified, the current directory is listed. For each IHTML object in the directory, the versions in which it exists are also listed. Non-versioned files are also listed, but only their names are shown. If one or more arguments are specified, the named files and directories are listed. Again, for each IHTML file named (or in a directory that is named), its versions are listed.

`irm` — remove a version of a page

Syntax: `irm [-v version] filename`

Removes the named *version* of *filename*, or the vanilla version if no *version* is given.

A.5 Common IHTML Techniques

This section describes a few common techniques used in IHTML documents. Note that this section is in no way exhaustive; it merely shows a few of the possibilities.

A.5.1 Cascading Levels of Detail

One common pattern on WWW pages is a list of links leading to other pages. A useful addition to this pattern is a convenient way of displaying a short summary for a link, in the current page. The following fragment of IHTML uses a dimension named "summary" to display a summary for any one of the links.

```
...
<ul>
  <li><a href="link1.html">First Link</a>
        (<a vmod="summary:link1">summary</a>)
      <iselect>
        <icase version="summary:link1">
           <quote>
             A brief description of the first link
           </quote>
        </icase>
```

```
      </iselect>
   <li><a href="link2.html">Second Link</a>
        (<a vmod="summary:link2">summary</a>)
      <iselect>
        <icase version="summary:link2">
           <quote>
             A brief description of the second link
           </quote>
        </icase>
      </iselect>
   <li><a href="link3.html">Third Link</a>
        (<a vmod="summary:link3">summary</a>)
      <iselect>
        <icase version="summary:link3">
           <quote>
             A brief description of the third link
           </quote>
        </icase>
      </iselect>
 </ul>
 ...
```

Note the use of default HREF attributes in the links, to link to a new version of the current page. With this example, the value of the summary dimension determines which fragment of text is displayed. Picking a different summary causes the current one to be removed, and the new one displayed.

If the current dimension included "summary:link2", the browser would display something like:

...

- First Link (summary)
- Second Link (summary)

  A brief description of the second link
- Third Link (summary)

...

The previous example works if only one level of detail is appropriate, or if only one summary should be visible at a time. Taking advantage of ICOLLECT and numeric dimension values, we can have multiple levels of detail.

```
...
<icollect>
  <icase version="expertise:0">
    This is the text that everybody should see.
  </icase>
  <icase version="expertise:1">
    Here is some more detail, for those in the know.
  </icase>
  <icase version="expertise:2">
    Here are the real details, for the experts.
  </icase>
  <icase version="expertise:0">
    Here's a little summary, that everybod should see.
  </icase>
</icollect>
...
Expertise level: <a vmod="expertise:0">novice</a>
                 <a vmod="expertise:1">normal</a>
                 <a vmod="expertise:2">expert</a>
```

Using this technique, a single page can provide several levels of detail, with access to different levels via links. Two features of IHTML are important for this example: `ICOLLECT` includes *all* matching cases; higher numeric dimension values are considered refinements of lower numeric values.

A.5.2 Trans-version Link Files

Often, an entire site will vary within a standard set of dimensions. In such a situation, it is useful for each page to include a standard set of links to other versions of the site. The site developer can take advantage of default URL's in anchors to create a single IHTML fragment containing such a table, and include it in all pages.

```
<table>
  <tr><th>Colour<th>Expertise<th>Language<th>Graphics
  <tr><td><a vmod="colour:blue">Blue</a>
      <td><a vmod="expertise:novice">Novice</a>
      <td><a vmod="language:english">English</a>
      <td><a vmod="graphics:lores">Low Res</a>
  <tr><td><a vmod="colour:green">Green</a>
      <td><a vmod="expertise:normal">Normal</a>
```

```
      <td><a vmod="language:french">French</a>
      <td><a vmod="graphics:medium">Normal</a>
    <tr><td><a vmod="colour:red">Red</a>
      <td><a vmod="expertise:high">Expert</a>
      <td><a vmod="language:turkish">Turkish</a>
      <td><a vmod="graphics:hires">High Res</a>
  ....
</table>
```

The site implementer would keep this table in a separate IHTML file, and `#include` it in all files. In this way, the user would see a standard set of trans-version links on all pages, while the site implementer would be able to extend or modify the link table once for all pages.

A.5.3 IHTML and JavaScript

One might want to use JavaScript to trigger an HTTP GET request for a versioned object. Since there will be no way beforehand to know what the version code should be, the `iurl` command is used to construct a versioned filename:

```
...
window.location =
  "<!--#iurl href="zork.html" vmod="language:french" -->"
...
```

This expression will be replaced by `zork.`*`version-code`*`.html`, where *version-code* is the encoding of the current version, modified with `language:french`.

A.6 Installation Instructions

IHTML has three components: an Apache module, Unix utilities for managing an intensional site, and a library containing the common code used by the first two. The module is a standard Apache module; the other parts are generic C programs (for Unix). These instructions assume that the user already knows how to install the Apache server; for more information, see the INSTALL file which is part of the Apache source distribution.

A.6.1 Unpacking and Preparation

IHTML is distributed as a compressed `tar` file. When extracted, the contents of the file will be put into a directory called "ihtml". The following files should be present:

- Makefile — makefile for the ihtml library and utility programs
- Makefile.Mod — makefile for the Apache IHTML module
- README — brief installation instructions
- arith_coding.c — text compression and decompression utilities
- arith_coding.h — public interface for arith_coding.c
- decode.c — utility program for decoding encoded versions
- encode.c — utility program for encoding clear-text versions
- icp.c — utility for copying versions into and out of IHTML objects
- ils.c — utility for listing the versions of IHTML objects
- irm.c — utility for removing specific versions of IHTML objects
- ivi.c — utility for invoking an editor on specific versions of IHTML objects
- mod_ihtml.c — Apache IHTML module
- popts.c — command line argument parsing utilities
- popts.h — public interface for popts.c
- translate.c — main IHTML-to-HTML translation code
- translate.h — public interface for translate.c
- uri2fn.c — main URL-to-filename translation code
- uri2fn.h — public interface for uri2fn.c
- util.c — stubs for Apache utilities (used when compiling `ivi`, etc.)
- util.h — public interface for util.c
- version.c — version handling code (parsing, comparison, and so on)
- version.h — public interface for version.c

A.6.2 The Apache Module

The module component is installed as a normal Apache module. Aside from that, the configuration must be altered to include the IHTML library. For complete information, see the Apache reference documentation; the steps are summarized here. Note that these instructions are compatible with Apache version 1.2.5. Later versions may require some changes.

1. Create an `ihtml` directory in `.../src/modules/` in the root of the Apache source tree. Move `mod_ihtml.c` and `Makefile.Mod` to it. Move `Makefile.Mod` to `Makefile`.

2. Add the following line to the `Configuration` file in the Apache `src` directory, as the second-last module of the file:

   ```
   Module ihtml_module modules/ihtml/mod_ihtml.o
   ```

   If it is not the second-last module, there is a non-zero chance that some other module will trap HTML requests, preventing the IHTML module from handling them. (This is the part which is likely to change in later versions of Apache.)

3. Add the following items to the `Configuration` file (adding to, or replacing, the current definitions of these variables):

   ```
   EXTRA_LFLAGS = -L...path.../ihtml
   EXTRA_LIBS = -lihtml
   EXTRA_INCLUDES = -I...path.../ihtml
   ```

   where *...path...* is the absolute path containing the IHTML source directory (here called `ihtml`). Run the `Configure` script in the Apache source directory.

4. Run `make` in the Apache source directory. The result will be an `httpd` executable. Install it as usual for Apache.

5. Optionally, add the following lines to the `httpd.conf` file for your server, to enable IHTML logging and error logging. Replace the paths with your preferred location for the log files.

   ```
   IhtmlLogFile logs/ihtml_log
   IhtmlErrorFile logs/ihtml_err_log
   ```

   The given paths are relative to the server's root directory, unless they are absolute paths. The particular names in this example are the traditional ones, but the server maintainer is free to choose any names.

A.6.3 The IHTML Library

Relatively little configuration should be necessary for this part of the system. The Makefile will need to be adjusted to reflect the location of the Apache include files. If *gcc* is not the C compiler you wish to use, the `CC` variable will need to be changed.

After these changes, type `make` to build `libihtml.a`.

A.6.4 Installing the Site Construction Utilities

Use `make all` in the `ihtml` source directory to build `ivi`, `ils`, `icp`, and `irm`. These programs should be available to all users of IHTML, so copy them to some convenient, publicly visible location.

A.7 The Local Installation

There are two servers running: `lucy.uvic.ca:8888`, `valdes.uvic.ca:8534`. Typical URL's for information in a user's home WWW pages will be of the form:

`http://lucy.uvic.ca:8888/~`*`username/path/file.suff`*

where *username* is the user's userid, *path* is the (possibly empty) path within the user's ".www" directory to the versioned object, and *file.suff* is the versioned object (i.e. a directory containing things named with *version.suff*). As usual, all files and directories will need to be world-readable (and executable, for the directories).

Log files are found in

`/home/gdbrown/httpd/logs/`

on the two machines. Messages regarding IHTML-specific issues are found in the `ihtml_log` file.

APPENDIX B
Version Encoding

Version encoding consists of four steps: transformation to canonical form, representation as a string, compression (using arithmetic coding), and conversion of the compressed string to ASCII text. Step one is described in Section 3.1.1.1. Step two is straightforward. Step three compresses the text string as much as possible. Step four uses a simple transformation to represent the compressed string as printable ASCII characters. This last step is vital, since the encoded versions are used in URL's and as part of filenames. Unfortunately, it also counteracts the benefits of compression, leading to encoded versions which are about as long as the unencoded ones. However, the security benefit of an encoded form is significant even if the "compressed" form is as long as the uncompressed one.

B.1 Arithmetic Coding

Arithmetic coding provides optimal compression for text, given a correct model of the relative frequency of the characters in the input alphabet. (See [38], page 101, for a complete description.) The members of the alphabet are put in some arbitrary order (well-known to encoder and decoder), and are assigned ranges between 0 and 1, with the range for each character proportional to its probability. A string is then represented as a number between 0 and 1, with each successive character narrowing the range of possible numbers representing that string, according to the character's probability.

For example, consider the characters in Table B.1, and the string *abceb0*. (The character *0* will be used as a terminating character, marking the end of a string.) Since the first character is an a, the number representing the string must be in the interval $[0.0, 0.1)$. Considering the following b, and subdividing the new interval according to the same probabilities, the interval which will contain the final number is reduced to

Symbol	Probability	Range
a	0.1	$[0.0, 0.1)$
b	0.3	$[0.1, 0.4)$
c	0.2	$[0.4, 0.6)$
d	0.1	$[0.6, 0.7)$
e	0.1	$[0.7, 0.8)$
0	0.2	$[0.8, 1.0)$

Table B.1: Probabilities and ranges for a sample input alphabet

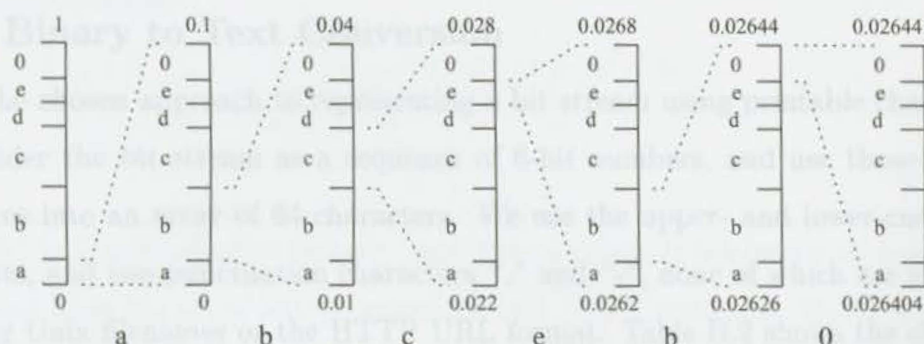


Figure B.1: Arithmetic coding example for the string *abceb0*

$[0.01, 0.04)$. Figure B.1 shows that the final range for the string is $[0.026404, 0.02644)$; any number in that range is a correct encoding of *abceb0*.

Note that the presence of a unique terminating character is essential. Were it missing, the number 0.0 could represent *a*, *aa*, *aaa*, and so on.

With respect to Intensional HTML, the input alphabet is the set of characters which may be found in versions: the letters and numbers, and the punctuation characters "+", ":", and "_". Note that to improve compression, upper- and lower-case letters are treated as equivalent in version strings.

Bell, Cleary, and Witten, in [38], chapter 5, describe arithmetic coding extensively; this description is based theirs. In particular, they provide a purely integer-math-based implementation, to avoid the performance penalty of arbitrary-precision floating-point math.

0	a	8	i	16	q	24	y	32	G	40	O	48	W	56	4
1	b	9	j	17	r	25	z	33	H	41	P	49	X	57	5
2	c	10	k	18	s	26	A	34	I	42	Q	50	Y	58	6
3	d	11	l	19	t	27	B	35	J	43	R	51	Z	59	7
4	e	12	m	20	u	28	C	36	K	44	S	52	0	60	8
5	f	13	n	21	v	29	D	37	L	45	T	53	1	61	9
6	g	14	o	22	w	30	E	38	M	46	U	54	2	62	_
7	h	15	p	23	x	31	F	39	N	47	V	55	3	63	-

Table B.2: Translation table: 6-bit numbers to characters

B.2 Binary to Text Conversion

The chosen approach to representing a bit stream using printable characters is to consider the bit stream as a sequence of 6-bit numbers, and use those numbers as indices into an array of 64 characters. We use the upper- and lower-case letters, the digits, and two punctuation characters "_" and "-", none of which are significant in either Unix filenames or the HTTP URL format. Table B.2 shows the characters used to represent each six-bit number.

If a bit stream is not an integral multiple of six in length, the string is padded with 0's. These padding bits do not cause problems when decoding the resulting string, since, as described in the previous section, a unique end-marker character is necessary for the arithmetic encoding phase anyway; once that character is found in the bit stream, any trailing bits are ignored.

Consider the following sample bit stream:

```
010100 010100 101011 110101 010110 10
```

For implementation reasons, the leftmost bit in a group of six is the least significant, so the first group, `010100`, is 10 rather than 20. Given that, and the mapping from Table B.2, the above sequence translates to the character string `kk1RAb`. When transformed back to a bit stream, the result will be:

```
010100 010100 101011 110101 010110 100000
```

but, as mentioned previously, the trailing bits will be ignored when the bit stream is decoded into a version string.

APPENDIX C

Collected Version Algebra Axioms

This appendix collects the grammar, equivalence rules, and refinement rules for IHTML 2's version algebra into one location.

Version Grammar

$$V ::= \varepsilon \mid D{:}I \mid D{:}\varepsilon \mid V+V \tag{C.1}$$
$$D ::= T \tag{C.2}$$
$$I ::= T \mid N \tag{C.3}$$

where N is a non-negative integer, and T is a token (a sequence of one or more characters from the set of letters, numbers, and "_"). Constraint: $D_1{:}V_1 + D_2{:}V_2 \rightarrow D_1 \neq D_2$ (in other words, you may not duplicate dimension names in a version).

Equivalence Rules for "+"

$$V+V \equiv V \tag{C.4}$$
$$V+W \equiv W+V \tag{C.5}$$
$$V+\varepsilon \equiv V \tag{C.6}$$

Refinement Rules

$$\varepsilon \subseteq V \tag{C.7}$$
$$V \subseteq V+V' \tag{C.8}$$
$$V \subseteq W,\ V' \subseteq W \rightarrow V+V' \subseteq W \tag{C.9}$$
$$N_1 \leq N_2 \rightarrow D{:}N_1 \subseteq D{:}N_2, \text{ for integers } N_1, N_2 \tag{C.10}$$
$$V \subseteq V',\ W \subseteq W' \rightarrow V+W \subseteq V'+W' \tag{C.11}$$

Canonical Form

$$D{:}\varepsilon \rightarrow \varepsilon \tag{C.12}$$
$$D_1{:}I_1 + D_2{:}I_2 \rightarrow D_2{:}I_2 + D_1{:}I_1\ , \text{ if } D_2 < D_1 \text{ alphabetically} \tag{C.13}$$
$$V+\varepsilon \rightarrow V \tag{C.14}$$

Comparison Rule

$$\forall D_i(V_i = \varepsilon \text{ or } \exists E_j(D_i = E_j \text{ and } D_i\!:\!V_i \subseteq E_j\!:\!W_j)) \tag{C.15}$$

$$\downarrow$$

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

Vita

Surname: Brown Given Names: Gordon David

Place of Birth: Burnaby, British Columbia, Canada

Educational Institutions Attended:

Simon Fraser University	1983-1988
University of Victoria	1988-1990, 1996-1998

Degrees Awarded:

B.Sc.	University of Victoria	1990

Honours and Awards:

SFU Open Scholarship	1984-1988
Woods Trust Scholarship	1990

Publications:

"Improved Dimensionality Analysis for Lucid", *Proc. 10th International Symposium on Languages for Intensional Programming*, University of Victoria, 1997.

"Intensional HTML" (with Wadge, schraefel, and Yildirim), *Proc. Principles of Digital Document Production 1998*, 1998.

"IHTML 2: Design and Implementation", *Proc. 11th International Symposium on Languages for Intensional Programming*, Palo Alto, California, 1998.

Partial Copyright License

I hereby grand the right to lend my theiss 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 of 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 2: A Practical Approach

Author:

Gordon David Brown
August 24, 1998