Intensional XML

By

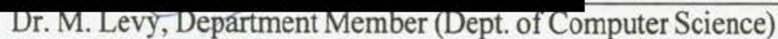
Vu Thanh Phong
B.Sc., University of Victoria, 1998

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

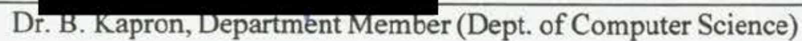
MASTER OF SCIENCE
in the Department of Computer Science

We accept this thesis as conforming
to the required standard

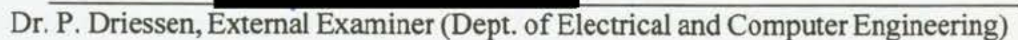
Dr. W. W. Wadge, Supervisor (Dept. of Computer Science)



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



Dr. B. Kapron, Department Member (Dept. of Computer Science)



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

© Phong Vu, 2000
University of Victoria

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

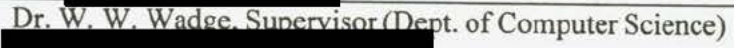
Supervisor: Dr. W. W. Wadge

Abstract

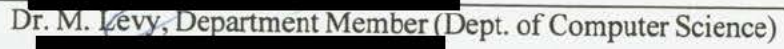
This thesis presents and describes Intensional XML (IXML), an implementation of Intensional HTML (IHTML) created by the author. The object of IXML is to retain the strengths of previous implementations of IHTML such as stability, performance, and programming flexibility while addressing the shortcomings of these implementations. In particular, IXML stresses that the framework is easily deployable across multiple platforms and Web servers and is easily useable by the majority of people, especially Web page designers who have little or no programming experience.

The key concept that allows IXML to achieve these goals is that it only uses common and existing technologies such as XML, XSL, Java, Java Servlets, and Cocoon. IXML is built as an application of Cocoon, which is built as an application of Java Servlets, and so wherever Java Servlets can be deployed, IXML can be used. IXML uses XML to easily express multi-version content that abstracts the versioning logic and other programming complexities. And IXML only focuses on the multi-versioning infrastructure of a Web infrastructure and relies on the underlying Cocoon Web publishing framework and Java Servlets to handle all the other common aspects of a Web framework such as performance and stability.

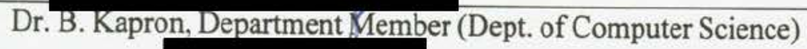
Examiners:



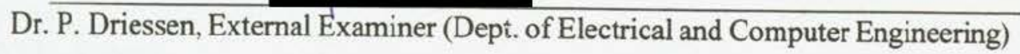
Dr. W. W. Wadge, Supervisor (Dept. of Computer Science)



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



Dr. B. Kapron, Department Member (Dept. of Computer Science)



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

Contents

Contents ... iv
List of Figures ... vi
Acknowledgments ... vii
1. Introduction ... 1
2. Technology Background ... 4
XML ... 4
Applications of XML ... 4
MathML ... 5
CML ... 5
MusicML ... 5
SSML ... 5
FpML ... 5
XML Syntax ... 6
XSL ... 7
XSLT Overview ... 8
How XSLT works ... 9
Template Rules ... 10
CGI ... 11
Overview ... 11
Lifecycle CGI ... 11
Advantages CGI ... 12
Disadvantages CGI ... 12
Java Servlets ... 14
Overview ... 14
Applications of Java Servlet ... 14
Advantages of Java Servlets ... 15
Architecture of Java Servlets ... 18
Lifecycle of a Java Servlet ... 19
Disadvantages Java Servlets ... 20
Cocoon ... 22
Overview ... 22
Cocoon Model ... 23
Cocoon Internal ... 24
Advantages of Cocoon ... 25
3. IHTML ... 26
Background ... 26
Limitations of IHTML ... 28
4. IXML Design ... 31
5. IXML Architecture ... 33
IXML Version Parameters ... 34
IXML Tag Library ... 35
Overview ... 35
IXML Tags ... 36

IXML Class Library 45
ParameterHandler 46
SelectHandler 46
TreeHandler 47
IXML Style-Sheet 48
Overview 48
Default HTML presentation of IXML tags 48
6. IXML in Practice 51
Example 51
How the multi-version example is constructed 51
7. Comparison of IXML to IML 58
Case Study One: A simple multi-parameter page 58
Case Study Two: Pop-text 64
Comparison 66
8. Wrap-up 68
Future Work 68
Conclusion 70
Bibliography 71
Appendix A: XML Markup 73
Elements 73
Entity References 74
Comments 74
Processing Instruction (PIs) 74
CDATA 75
Document Type Definitions 75
Element type declarations 76
Attribute List Declarations 77
Entity Declarations 79
Appendix B: List of XSL instructions 81
XSLT Tag Summary 81
Appendix C: How to setup IXML 94
Install Java 94
Install Cocoon and JServ 95
Install the core IXML files 95
Compile the IXML Java class library 96
Download and uncompress sample.zip 96
The sample greeting example 97
Start the Apache Web Server 97

List of Figures

Number *Page*

Figure 1. A sample XML file describing an inbox for emails. 6

Figure 2. An XSL template and its corresponding output 11

Figure 3. Request Handling for Java Servlets, Regular CGI and Fast-CGI. 18

Figure 4. Cocoon internal workings 24

Figure 5. IXML Architecture 33

Figure 6. IXML Semantic Notation. 34

Figure 7. Sample Version URLs 35

Figure 8. Example of the <iSelect> tag. 38

Figure 9. Example of <iParameterSelects> tag 41

Figure 10. Corresponding HTML representation of <iParameterSelects> tag 41

Figure 11. Example iTree 44

Figure 12. Corresponding HTML representation of iTree code above. 44

Figure 13. IXML Java Processing 46

Figure 14. Listing of clean-page.xml 52

Figure 15. Listing of page-xsp.xsl. 55

Figure 16. Listing of page-html.xsl 56

Figure 17. A simple multi-parameter page in IML. 58

Figure 18. A simple multi-parameter page in IXML. 59

Figure 19. A pop-text section 65

Acknowledgments

First of all, I would like to thank my supervisor, Dr. Wadge, for accepting me as a grad student and for the financial assistance he provided during my studies. I would also like to thank him for giving me the freedom to work out my thesis.

And I would like to thank Dr. Trussell for supporting most of my under-grad and grad studies through his scholarship. But most of all I would like to thank him for caring enough to take the time and effort to get to know, advise and help me and his other Trussell scholars.

I would also like to thank my friends especially "Ca doan Teresa" of Vancouver who I met during my grad studies. Other students make mention of cola or powerbars to keep them going through their studies -- you guys are the ones that keep me going.

And I would especially like to thank my family who set the best example, providing me with the environment to realize the important things of life and to do well at all the things I do.

And most of all, I would like to thank the One who gives me the gift of learning and also the gift of working hard and who also gives me everything else that I ever or will ever need.

abstracted away so that a typical Web page designer who has little or no programming experience can effectively use IXML.

The key concept that allows IXML to achieve all these goals is that it only uses common and existing technologies such as XML, XSL, Java, Java Servlets, and Cocoon. Since IXML is an application of Cocoon which itself is built as an application of Java Servlets, IXML will run on most operating systems and Web Servers. Multi-version content is easily assembled using an XML tag library that abstracts the versioning and programming logic from the end Web page designer. As a result common multi-version Web pages that have been implemented in previous implementations of IHTML such as multi-language pages and pop-text are not only possible in IXML, but are much more organized.

The thesis is organized as follows:

- Chapter 2 is a background on the key technologies that IXML is built on such as XML, XSL, Java Servlets and Cocoon.
- Chapter 3 reviews the previous implementations of IHTML
- Chapter 4 describes the design goals of IXML.
- Chapter 5 describes the architecture of IXML.
- Chapter 6 describes, by example, a high level view of IXML and how it is used in practice by a typical Web page designer.
- Chapter 7 is a comparison of IXML to IML from a Web page designer point of view
- Chapter 8 is a wrap-up with future work and conclusion

- Appendix A describes XML markup in more details
- Appendix B describes XSL instructions in more details
- Appendix C lists the source file for an example pop-text in IML and IXML
- Appendix D shows how to install and run IXML

2. Technology Background

XML

The Extensible Markup Language (XML), a World Wide Web Consortium (W3C) recommendation, is a simplified subset of the Standard Generalized Markup Language (SGML) specifically designed for Web applications to define document types [1]. A particular document type is defined by its DTD (Document Type Definition). DTDs describe the syntax of a language implemented in XML.

XML was conceived by the W3C to improve HTML Web publishing. Since HTML is a language for describing graphics, behavior and hyperlinks on Web pages, it was not designed for publishing and processing large quantities of data and complex dynamic information. [4]

With XML, the following properties are possible

1) Extensibility
2) Structure
3) Validation

Extensibility allows new tags to be defined as needed. Structure allows data to be modeled to any level of complexity. Validation allows the data to be checked for structural correctness. [5]

Applications of XML

When XML was first created, it was thought mainly as a language for describing meta-content, information about a document's contents, such as its title, author, file size, creation date, revision history, keywords, etc [6]. But as XML has developed, the

extensibility and flexibility of XML has showed that it can be used for many other Web applications. Some applications where XML is utilized:

MathML

The Mathematical Markup Language (MathML) is a W3C proposal for a markup language to specify mathematical equations. MathML solves the problem of expressing equations in a common format for different applications. Equations expressed in MathML can be displayed in a browser, or imported into a spreadsheet to be used in calculations or graphs.

CML

The Chemical Markup Language is an extensive markup language for managing molecular information. It has the capability to specify a wide scope of information including Macromolecular Sequence, Macromolecular Structure, Spectra, Organic Molecules, Publishing, Quantum Chemistry, Inorganic Crystallography.

MusicML

MusicML provides all the information necessary to display a musical score.

SSML

Speech Synthesis Markup Language (SSML) is a markup language for text-to-speech synthesis. SSML allows text documents to be annotated with tags that instruct a text-to-speech synthesiser how to read the text. SSML contains tags that define phrasing, emphasis and pronunciation.

FpML

The Financial Product Markup Language (FpML) has recently been announced as a standard for wholesale transactions in the financial derivatives market to allow trades between distributed heterogeneous computer systems.

XML Syntax

The figure below shows a sample XML file marking up an email inbox.

```
<?xml version="1.0"?>
<!DOCTYPE inbox [
  <!ELEMENT inbox (email+)>
  <!ELEMENT email (to, from, date, subject, body)>
  <!ELEMENT to (#PCDATA)>
  <!ELEMENT from (#PCDATA)>
  <!ELEMENT date (#PCDATA)>
  <!ELEMENT subject (#PCDATA)>
  <!ELEMENT body (#PCDATA)>
]>
<inbox>
  <email>
    <to>ptv@uvic.ca</to>
    <from>vupt@yahoo.com</from>
    <date>8:30 Jan 2. 2000</date>
    <subject>hello</subject>
    <body> hello</body>
  </email>
</inbox>
```

Figure 1. A sample XML file describing an inbox for emails.

The document begins with a processing instruction: <?xml ...?>. While it is not required, its presence explicitly identifies the document as an XML document and indicates the version of XML to which it was authored. The next section, <!DOCTYPE …]>, is the document type definition (DTD) which can be optionally supplied or linked to understand a document unambiguously. The DTD in the example specifies the structure of an inbox. It indicates that an inbox consists of one or more email tags. And an email tag must have the following tags that are text data: to, from, date, subject, body. The last section, <inbox>…</inbox>, shows the data content of the XML file. It also conforms to the specified DTD.

XML documents are composed of markup and content. There are six kinds of markup that can occur in an XML document: Elements, Entity References, Comments, Processing Instruction, Marked sections, and Document type declaration.

Elements are the most common form of markup. They are delimited by angle brackets and they usually identify the nature of the content they surround. Elements may be empty, eg/ <applause/>. If an element is not empty, it begins with a start tag, eg/ <to>, and ends with an end tag, eg/ </to>.

Elements may have attributes associated with them. Attributes are name-value pairs occurring inside start-tags after the element name. For example, <parameter language ="ENGLISH">, is an element with name parameter and with attribute language having value ENGLISH.

Document Type Definitions allow a document to communicate meta-information about its content. Meta-information includes the allowed sequence and nesting of tags, attribute values and their types and defaults, the names of external files that may be referenced and whether or not they contain XML, the formats of some external (non-XML) data that may be referenced, and the entities that may be encountered [7]. For a more detailed description of these and other XML markup, please refer to Appendix A.

XSL

Since it is possible to define custom tags in XML, there are no predefined semantics in XML. Consequently it is necessary to have a standard mechanism for

describing how XML tags are to be presented. Extensible Stylesheet Language (XSL) is the standard that defines XML semantics.

XSL consists of two parts:

1. a language for transforming XML documents [XSLT]
2. a XML vocabulary for specifying formatting semantics.[XSLFO] [8]

XSL provides presentation information for documents conforming to XML DTDS. The model used by XSL for rendering documents on the screen builds upon many years of work on Document Style Semantics and Specification Language (DSSSL), a complex ISO-standard style language.

XSLFO is an XML DTD for describing 2D layout of text in both printed and digital media. It is mostly used as a final DTD, meaning that a transformation is used to generate a formatting object description of a document starting from a general XML file [4]. XSLT is a language for transforming one well-formed XML document into something else (which may not necessarily be another XML document, although it most often will be).

XSLT Overview

Although XSL is made up of two parts, the transformation and formatting halves of XSL can function independently of each other [9]. For Web applications where the final presentation format is HTML, XSLT can be used exclusively to style XML content into XHTML, a format of HTML that is well-formed. Cocoon and Internet Explorer 5.x uses this approach to style XML content into HTML.

XSLT transforms one XML tree into another XML tree. Although it can transform an XML tree into plain text or mixture of plain text and elements, it was not intended for these types of transformations [9]. XSLT was specifically designed for XML-to-XML transformations. XSLT cannot output text that is malformed XML. And the input must be an XML document. Therefore XSLT cannot be used to transform from non-XML formats like PDF or TeX.

How XSLT works

An XSL processor processes both an XML document and an XSL style sheet and from instructions in the XSL style sheet, it outputs a new XML document. The XSLT processor first represents the XML document as a tree. The processor assumes a XML tree contains seven possible types of nodes: Root, Elements, Text, Attributes, Namespaces, Processing instructions, and Comments. The DTD and document type declaration are specifically not included in this tree.

The XSLT processor formats an XML document using the template rules specified in the XSL style sheet. A template rule has a pattern specifying the trees it applies to and a template to be output when the pattern is matched. The XSLT processor scans the XML document tree looking through each sub-tree in turn. As each tree in the XML document is read, the processor compares it with the pattern of each template rule in the style sheet. When the processor finds a tree that matches a template rule's pattern, it outputs the rule's template. A template generally includes some markup, some new data, and some data copied out of the tree from the original XML document.

Template Rules

XSL uses XML to describe template rules and patterns. Each template rule is an xsl:template element which associate a particular output with a particular input. Each xsl:template element has a match attribute that specifies which nodes of the input document the template is instantiated for. The output template is the content of the xsl:template element. All instructions in the template for doing things like selecting parts of the input tree to include in the output tree are performed by one or more XSL element. These XSL elements are identified by the xsl: prefix on the element names. Elements that do not have an xsl: prefix are part of the result tree.

The figure below shows an XSL snippet with a simple template rule that is applied to the root node of the input tree. When the XSL processor reads the input document, the first node it sees is the root. This rule matches that root node, and tells the XSL processor to emit the HTML inside the template.

```
<xsl:template
match="/">
  <html>
    <head>
    </head>
    <body>
    </body>
  </html>
</xsl:template>
```

a. XSL template rule

```
<html>
  <head>
  </head>
  <body>
  </body>
</html>
```

b. Template output

Figure 2. An XSL template and its corresponding output

CGI

Overview

The Common Gateway Interface (CGI) is a standard for writing programs that can interact through a Web server with a client running a Web browser. These programs allow a Web developer to deliver dynamic information (usually in the form of HTML) via the browser. Although the CGI standard allows any language to be used, Perl is chosen the majority of the time [11]. CGI programs are commonly used to add search engines, guest-book applications, database-query engines, interactive-user forums, and other interactive applications to Web sites [11].

Lifecycle CGI

A CGI program must interpret the information sent to it, process the information in some way, and generate a response that will be sent back to the client [11].

The basic CGI request processing proceeds as follows:

1. For each request, the server creates a new process and the process initializes itself.
2. The Web server passes the request information (such as remote host, username, HTTP headers, etc.) to the CGI program in environment variables.
3. The Web server sends any client input (such as user-entered field values from an HTML form) to the CGI program's standard input.
4. The CGI program writes any output to be returned to the client on standard output.
5. When the CGI process exits, the request is complete. [12]

Advantages CGI

CGI have many strong qualities, attested by their popularity and long time use. The advantages of CGI are as follows:

- Simplicity
- Language independence
- Process isolation
- Open standard
- Architecture independence

[12]

CGI programs are easy to understand and easy to develop. CGI programs can be written in nearly any language. Since applications run in separate processes, buggy applications cannot crash the Web server or access the server's private internal state. Since it is an open standard, some form of CGI has been implemented on every Web server. And CGI is not tied to any particular server architecture (single threaded, multi-threaded, etc.).

Disadvantages CGI

But CGI programs are burden with the following problems:

- Cannot interact with a Web server or take advantage of the server's abilities
- Problems of state management
- Security

- Performance

Since CGI programs run as a separate process, CGI programs can't link into other stages of Web server request processing, such as authorization and logging. Therefore it cannot write to the server's log file or take advantage of other server's abilities.

And since a new process is created for every request, there is no obvious method to manage state between requests. If libraries are not already written or available, they have to be written to manage all the state management mechanisms [13].

Security can be a problem when developing CGIs. The largest area of concern is processing user input. This could be from forms or from data within the URL. Many CGIs written in Perl are vulnerable to attacks where the end user tricks the CGI into executing a command on the server.

But the biggest problem with CGI is performance [14]. CGI programs are a drain on system resources. In a typical CGI environment, each request creates a new process that then loads the Perl interpreter (typically over 500 KB in size). The Perl interpreter then loads the CGI script, compiles it, and executes it. Additionally, if the application requires communication with an external program such as a database, a new connection needs to be established for each CGI (a rather expensive process for some databases). Creating a process for every such request requires time and significant server resources. And efficiency is poor since the process is thrown away when the request is done. [12]

Java Servlets

Overview

The Java Servlets API was created by Sun Microsystem as the first standard extension to the Java language. And so while it is not part of the core Java framework which must always be part of all products bearing the Java brand, it will be made available with such products by their vendors as an add-on package [14]. Servlets are protocol and platform independent server side components, written in Java, which dynamically extend Java enabled servers. Through generic server extension, Java classes can be loaded dynamically to expand the functionality of a server. And so Servlets are modules that run inside request/response-oriented services such as Web servers and extend them in some manner.

Applications of Java Servlet

Although Servlets can extend any response/request oriented service in general, they are most commonly used, however, to extend Web servers, performing tasks traditionally handled by CGI programs [13].

Some possible applications for Servlets are:

1. Processing data POSTed over HTTPS using an HTML form, and passing data such as a purchase order (with credit card data). This would be part of an order entry and processing system, working with product and inventory databases and perhaps an on-line payment system.

2. Allowing collaboration between people. A Servlet can handle multiple requests concurrently; they can synchronize requests to support systems such as on-line conferencing.
3. Forwarding requests. Servlets can forward requests to other services and Servlets. This allows them to be used to balance load among several servers that mirror the same content. It also allows them to be used to partition a single logical service over several servers, according to task type or organizational boundaries.
4. One could define a community of active agents, which share work among each other. The code for each agent would be loaded as a Servlet, and the agents would pass data to each other.

Advantages of Java Servlets

The advantages of using Java Servlets have made Java Server programming very popular. In a poll conducted by Javaworld, an online Java magazine, the top three reasons why Servlet programmers prefer working with Servlets are:

1) platform and server independence
2) power of Java Servlets
3) performance

Since Servlets are based on Java, they are portable between operating systems and between servers. For various reasons, many sites are looking for a portable solution. Perhaps they want developers to use Linux or Windows NT while their production server runs Solaris. They may want to sell their Web application to as many customers

on as many platforms as possible. Java Servlets run on the majority of popular operating system and popular web server [15].

And Servlets have full access to the various Java APIs and third-party component classes, making them ideal for use in communicating with applets, databases, and RMI servers. And unlike any other current server extension API, Java Servlets provide strong security policy support because all Java environments provide a Security Manager which can be used to finely control access. By default, all Servlets loaded over the network are untrusted and are not allowed to perform operations such as accessing network services or local files. Local Servlets controlled by the server administrator are fully trusted and granted all privileges. And Servlets which have been digitally signed as they were put into Java Archive files, can be trusted and granted more permission

Also Java Servlets were designed with performance at a very high priority. Since there are many implementations of the Java Virtual Machine ranging from entry-level servers to mainframe multiprocessors, applications are able to scale easily. In addition, Servlets automatically take advantage of additional processors.

But the biggest performance feature of Servlets is that they do not require the creation of a new process for each request. As a result the cost of the initial process creation is spread over many methods. And client requests have an opportunity to share data and communications resources, benefiting more strongly from system caches. Also context-switching is more efficient. Most Web server environments consist of a single-process, multithreaded Web server written in another language other than Java. In such an enviroment, Servlets can run in parallel within the same process as the server and so

the JVM can often be embedded inside the server process. Having the JVM as part of the server process maximizes performance because a Servlet becomes, in a sense, just another low-level server API extension. Such a server can invoke a Servlet with a lightweight context switch and can provide information about requests through direct method invocations. When used in such environments with HTTP, Servlets provide compelling performance advantages over both the CGI approach and the Fast-CGI approach. The figure below summarizes the request handling for Java Servlets, Fast-CGI and regular CGI. Fast-CGI involves heavy weight process context switching on each request. And regular CGI requires even heavier weight process startup and initialization code on each request.

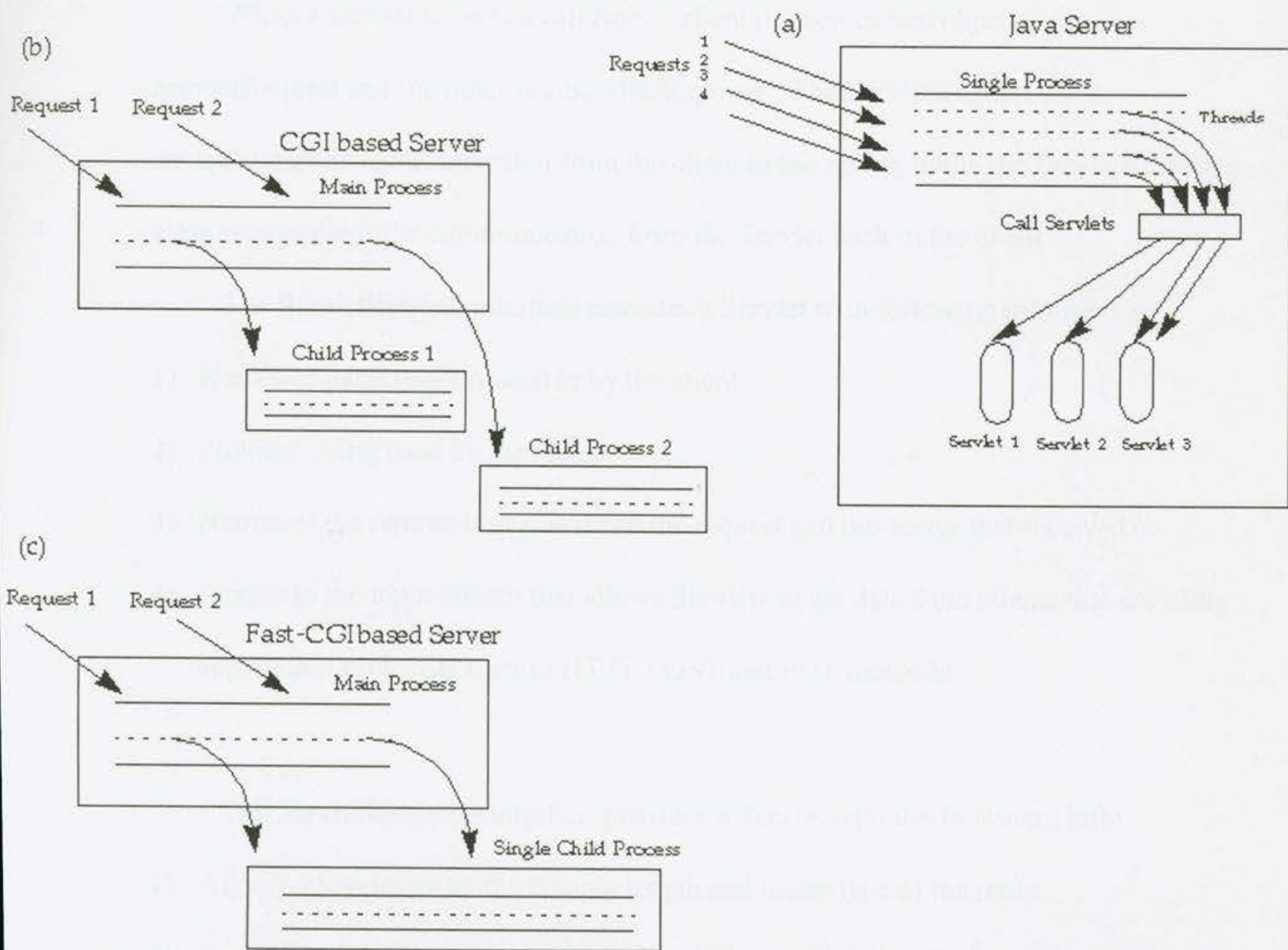


Figure 3. Request Handling for Java Servlets, Regular CGI and Fast-CGI.

Architecture of Java Servlets

The central abstraction of Servlets is the Servlet interface. All Servlets implement this interface, either directly or, more commonly, by extending a class that implements it such as HttpServlet. The Servlet interface provides for methods that manage the Servlet and its communications with clients

When a Servlet accepts a call from a client it receives two objects, one is a ServletRequest and the other is a ServletResponse. The ServletRequest class encapsulates the communication from the client to the server, while the ServletResponse class encapsulates the communication from the Servlet back to the client.

The ServletRequest interface provides a Servlet with following information:

1) Names of parameters passed in by the client
2) Protocol being used by the client
3) Names of the remote host that made the request and the server that received it
4) Access to the input stream that allows Servlets to get data from clients that are using application protocols such as HTTP POST and PUT methods.

The ServletResponse interface provides a Servlet with the following info:

1) Allows a Servlet to set the content length and mime type of the reply
2) Provides an output stream and a Writer through which the Servlet can send the reply date

HTTP Servlets have additional objects that provide session-tracking capabilities that can be used to maintain state between the Servlet and the client that persists across multiple connections during some time period.

Lifecycle of a Java Servlet

A Servlet engine must conform to the following Servlet life cycle contract: [15]

1. Create and initialize the Servlet.
2. Handle zero or more service calls from clients.
3. Destroy the Servlet and then garbage collect it.

After the Servlet engine instantiates and creates the Servlet, the Servlet engine calls the Servlet's init method. The init method is only called once and will not call it again unless it is reloaded. And the Servlet engine cannot reload a Servlet until after it has removed the Servlet by calling the destroy method. A Servlet can be initialized at these time:

1) When the server starts
2) When the Servlet is first requested, just before the service() method is invoked
3) At the request of the server administrator

After a Servlet is created and initialized it can handle zero or more service calls from clients. A Servlet handles client requests in its service method and sends back its response.

A Servlet will run until they are removed from service at request of a system administrator by calling the Servlet's destroy method. Afterwards, the Servlet object will be garbage-collected by the JVM.

Disadvantages Java Servlets

Although Java Servlets are extremely powerful and useful as attested by their popularity, there are disadvantages of the Java Servlets. The easiest and most commonly

used technique to generate output is to use the println() output stream to generate HTML content. As a result, HTML content has to be created within code, which is a onerous and time consuming task. And content creators have to ask developers to make all content changes which is a hassle because these two Web publishing contexts rarely overlap and usually are overseen by different people.

Another disadvantage is that at the present moment, the latest Java Servlet API (2.2) does not allow Servlet chaining, where output from one Servlet becomes input for another Servlet. This is not because Servlet piping is bad or disadvantageous, but because unfortunately the need for a componentized request handler was not taken into serious consideration in the design phase but later at the implementation phase [4].

Servlet chaining would allow Servlet design to be modular, where a single Servlet can be designed to accomplish a well-defined task that can be used as a service by other Servlets. For example, one Servlet can process a request and output XML content. And a second Servlet can post-process the XML. For example an XSL transformation Servlet can translate the content to HTML.

The implementation workaround provided by Java Servlet is Servlet nesting, where a Servlet is able to include the output of another Servlet in its own output transparently [4]. This allows programmers to separate different logic on different Servlets, thus removing the need for Servlet chaining. But Servlet nesting had to be implemented in the confines of the Servlet API which do not allow Servlet piping. And so the common design pattern is that no Servlet is allowed to post process the output of

another Servlet. As a consequence, the example above where a XSLT Servlet processes the XML output of another Servlet is not possible.

Cocoon

Overview

The Apache Cocoon project is a 100% Java publishing framework for web sites of medium to high complexity [4]. Stefano Mazzochi, the main Cocoon architect and project creator, uses Java Servlets, XML, and XSL technologies to create a Web publishing architecture that allows complete separation of document content, document processing and document formatting.

Document content containing no style information (such as text formatting or graphics) is represented as XML. Document processing transforms and evaluates the XML content. For example, mixing static content and dynamic logic can create a dynamic web page. Document formatting converts the XML content to a format such as HTML that is readable by a client.

Separating document content, document processing and document presentation allows Web sites to be highly structured and well designed. These three Web development tasks are usually created and maintained by different individuals or working groups and so separating them allows each group to concentrate on their own job. It reduces duplication of efforts and site management costs by allowing different presentations of the same data depending on the requesting client. And so it is possible for Cocoon to apply a different layout presentation to the same content page depending

on the particular client making the request such as Microsoft Internet Explorer, Netscape, a PDA device or cell-phone.

Cocoon Model

Cocoon is a Servlet that serves all the user XML requests. Since Cocoon is a Servlet it needs a Servlet engine to run. When a request comes in requesting an XML file, the request is passed to the Servlet engine. The Servlet engine, which is configured to map all XML user requests files to Cocoon, then sends the request to Cocoon to serve the request.

The Cocoon processing model is based on the separation of

- Production - where XML content is generated based on Request parameters
- Processing - where the produced XML content is transformed/evaluated
- Formatting - where the XML content is finally formatted into the wanted output format for client use.

Producers initiate the request-handling phase and are responsible for evaluating the HttpServletRequest parameters provided and create XML content that is fed into the processing reactor. Processors transform a given XML document into something else, driven both by the input document and by the request object that is also available. The Cocoon distribution includes a number of processors that implement common tasks. The XSLT and SQL are common processors included with Cocoon. The XSLT processor applies XSLT transformations to the input document. And the SQL processor evaluates simple tags describing SQL queries to JDBC drivers and formats their result-set in XML.

Cocoon Internal

The figure below shows that the Cocoon publishing system has an engine based on the reactor design pattern. The Request component wraps around the client's request and contains all the information needed by the processing engine. The request must indicate what client generated the request, what URI is being requested and what producer should handle the request. The Producer component handles the requested URI and produces an XML document. The Reactor component is responsible for evaluating what processor should work on the document. The Formatter component transforms the memory representation of the XML document into a stream that may be interpreted by the requesting client. The Response component encapsulates the formatted document along with its properties (such as length, MIME type, etc..). And the Loader is responsible for loading the formatted document when this is executable code.

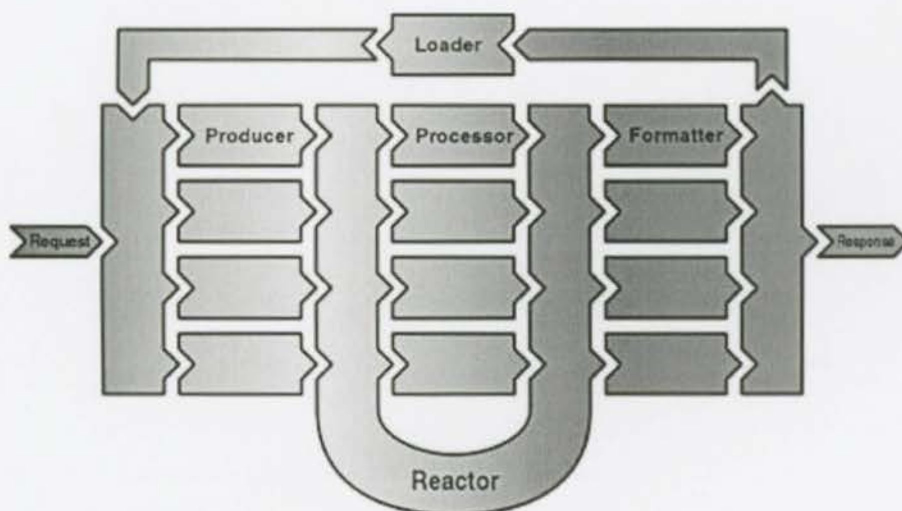


Figure 4. Cocoon internal workings

Advantages of Cocoon

Cocoon seamlessly integrates XML and XSL with Java Servlets to allow complete separation of content, processing and presentation. As a result Cocoon addresses the two shortcomings of Java Servlets where content is produced inside Java code using the println() stream and the lack of Servlet chaining. Now HTML content creators can work independently of Java programmers using XML and XSL. And since content, processing and presentation context are so completely separated, the internal Cocoon design allows the equivalent componetized handlers that Servlet chaining would have provided.

3. IHTML

Background

For several years now, Dr. Wadge and his team of collaborators have developed and refined IHTML. IHTML reduces the effort required to develop multi-versioned Web sites by allowing site developers to ensure that only those parts of a page that actually vary from one version to version are distinct [3].

IHTML is based on an intensional approach to versioning where the different versions of a document are specified by expressions in a simple algebraic version language [3]. A request for an IHTML page consists of two parts, a conventional URL indicating the name of the page requested, and a version expression indicating which version of the page is requested. The simplest version expressions consist of a version parameter and a corresponding value, separated by a colon. For example, the expression lg:fr specifies that the value of the lg version parameter (which might denote the document language) is fr (which might denote French). More generally, sums of such expressions can be formed, specifying that each of the given version parameters has its respective value. For example,

lg:fr + size:a4 + mod:ml4 + lv:expert

might stand for the French language and a4 manual for the ML4 model, written at the expert level. [3]

The first implementation of IHTML (IHTML 1.0) was a research prototype implemented as a Perl CGI program. Although it demonstrated that the theory behind IHTML was viable, it was difficult to use. Installation and use required detailed

knowledge of the implementation [1]. Also performance was a real concern because it had all the limitations of a Perl CGI program.

The second implementation of IHTML (IHTML 2.0) was implemented as a server-module for the Apache Web Server. Although IHTML 2.0 provided a stable, responsive, usable, clean syntax and versatile implementation of IHTML, practical experience with IHTML 2.0 revealed some serious limitations. The first is that the extra markup required for even simple effects (like pop-text) can be impracticably complex when faced with a document with a large number of pop-text sections [3]. The reason for this is that IHTML was built as a super-set of HTML 3.2, HTML 3.2 with some new tags added. And so it is not possible to build more complex tags from the basic tags. As a result each time a high level construct such as a pop-text section is created, the complete markup for a pop-text has to be constructed.

The second problem is that IHTML lacks programming control structures. There are no provisions for evaluating expressions – this rules out constructs (like a slide show) because the author cannot generically specify a link in which a parameter is incremented. There is no iterative construct, so that sequences of similar marked-up copy have to be (re)produced by hand. There are no functions, and this means (among other things) that copy must appear on the rendered page in the same order it appears in the marked up source. [3]

To overcome the lack of programming control features in IHTML 2.0 the Intensional Sequential Evaluator (ISE), a full-featured Perl-like scripting language that incorporates a (run-time) parametric versioning system was developed. ISE allows all

entities in the language (variables, arrays, functions etc) to be versioned. And so ISE is used in conjunction with CGI to create multi-versioned Web sites.

In the same way that the programming versatility of Perl made it the most popular CGI programming language [16], the intensional-imperative programming versatility of ISE/CGI allows it to do intensional CGI Web programming task. Although ISE overcomes the inflexibility of previous versions of IHTML, the complicated nature of using a versioned language to create a Web site inspired the creation of the Intensional Markup Language (IML), a set of Groff macro definitions which translate text with high-level (intensional) markup into ISE source. IML was designed to make the advantages of ISE available to authors who want only to mark up copy, and not program [3].

Limitations of IHTML

From experience with ISE/IML and previous implementations of IHTML and other Web publishing technology, the author note the following shortcomings of ISE/IML, the latest implementations of IHTML:

1. The multi-versioning architecture does not run on multiple platforms and Web Servers
2. The multi-versioning architecture is forked from an existing standard
3. The multi-versioning architecture uses CGI as the Web publishing framework
4. The multi-versioning architecture does not organize content, versioning logic and HTML presentation very well

Aside from IHTML 1.0, all previous implementations of IHTML only ran under a specific operating system and specific Web server. IHTML 2.0 was implemented as an Apache module and thus only works with the Apache Web Server. And ISE/CGI only runs under Linux and the Apache Web Server.

Although it is possible to port the ISE or IHTML 2.0 to other platforms, but in both cases the framework has to be installed and configured on the Web server before it can used. Since these IHTML implementations are custom-made and not in widespread use, it will be very difficult for most people to take advantage of these multi-versioning frameworks.

Both IHTML 2.0 and ISE were developed by taking an existing framework and forking it to have versioning logic capabilities. In the case of IHTML 2.0, HTML 3.2 was supplemented with additional IHTML tags and in the case of ISE, Perl was taken and intensional logic was built into it. And so when there are modifications to the base standards, these frameworks have to be modified to keep up.

As discussed previously there are many disadvantages to CGI such as performance, security, state management and interaction with the Web server. It is suggested that ISE should be coded into an Apache module to be a truly effective server-side versioning tool [16]. But doing this would limit its portability to other Web servers.

All of the previous implementations of IHTML mix together content, versioning logic and html presentation. A Web page, especially a multi-versioned page can become

extremely complicated. And having the user manipulate versioning logic means they would need programming experience to be comfortable in that environment. Although IML helps to make advantages of ISE available to authors who want only to mark up copy, and not program, by shielding the user from the programming complexities of ISE, the Groff macro implementation of IML is not very ideal for a markup language.

4. IXML Design

Considering the limitations of ISE/IML, the latest implementation of IHTML, and the maturity of technologies such as Servlets, XML and XSL, the author felt that a new implementation of IHTML could be advantageous. In designing a new implementation, the advantages and disadvantages of previous implementations were taken into account and in particular the following points were stressed:

1. The multi-versioning architecture needs to easily run on multiple platforms and Web servers
2. The multi-versioning architecture needs to be better organize and abstract the multi-version content and logic
3. The multi-versioning architecture has to be fast and reliable
4. The multi-versioning architecture must support programming capabilities

Considering the difficulties in getting the current IHTML implementations to run on multiple platforms and multiple servers, the only easy and widespread method that would make this possible is to build the architecture from common and existing technology that most platforms and Web servers support.

And every implementation since IHTML 1.0 claimed to be faster and more reliable than the previous implementation and so the next version cannot take a step back. Also from experience with previous implementations of IHTML, having programming capabilities is a definite requirement.

Building a multi-versioning architecture using Java Servlets would satisfy these constraints. It is fast and reliable and easy to use. And Java Servlets allow all the capabilities of the Java programming language. Also it is platform and server independant and runs on virtually every operating system and web server.

And using the Cocoon Web publishing framework would allow the multi-version content and logic to be better organized and abstracted. The existing XML/XSL framework already allows for the content, programming logic and HTML presentation to be separated. Although the original purpose of IML was used to give a higher level abstraction from the underlying ISE multi-versioning architecture, the idea can be further developed to entirely separate the multi-versioning content from the multi-versioning logic.

And so a tag library can be built in XML to help Web page designers construct multi-version web pages. These tags would be like components that a web page designer can put together to form a multi-version page. As a result, a Web page designer only needs to know what each tag does and how it can be used but not how it is implemented. The programming complexities that implement the multi-version tags are abstracted away from the end user.

IHTML 2.0 does a good job of clearly defining multi-versioning tags such as ISELECT and ICOLLECT that are very useful when building multi-versioning content. These tags re-implemented in XML can serve as the basic tags for a multi-version tag library.

5. IXML Architecture

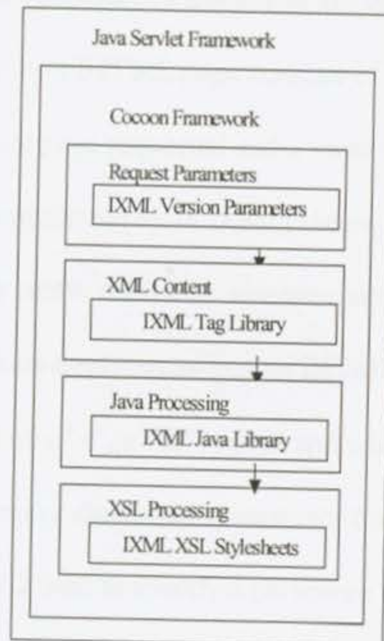


Figure 5. IXML Architecture

The figure above shows that IXML is built as a Cocoon application. The IXML version parameter framework is built within the context of the Java Servlets request parameters. It is used to specify a version of a Web page in a URL. The IXML tag library is built using XML and is used in conjunction with user defined XML to build the content of a multi-version Web page. An IXML Java class library is built in the context of the Java Servlets framework and is used to process the IXML tags. And finally IXML style-sheets are used in conjunction with other user defined style-sheets to create the HTML presentation.

IXML Version Parameters

In IHTML, the different versions of a document are specified in a simple algebraic version language. A request for an IHTML page consists of two parts, a conventional URL indicating the name of the page requested and a version expression indicating which version of the page is requested [3]. In IXML, the exact same approach to versioning is used. The figure below shows the semantic notation used in IXML. The only major change with previous implementations of IHTML is that the separating symbols are different. Instead of a "+", a "&" is used and instead of a ":", a "=" is used. These changes were made to make the version parameter framework consistent with the Java Servlet model. Also the ID used to specify a parameter variable is more restrictive. The version parameters must begin with "param". Since the Cocoon framework allow other user parameters to be received by the Web Server, putting "param" is just a convenient way to differentiate version parameters from other parameters.

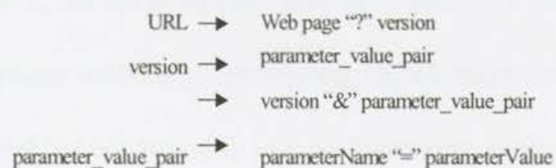


Figure 6. IXML Semantic Notation.

The figure below shows some sample version URLs. The first part of the URL indicates the Web page requested. The "?" separates the Web page from the version requested. In the example there are two version parameters, paramlanguage and paramdisplayMode.

```
http://localhost/greeting/page.xml?paramlanguage=ENGLISH&paramdisplayMode=GRAPHIC
http://localhost/greeting/page.xml?paramlanguage=FRENCH&paramdisplayMode=GRAPHIC
http://localhost/greeting/page.xml?paramlanguage=ENGLISH&paramdisplayMode=TEXT
http://localhost/greeting/page.xml?paramlanguage=ENGLISH&paramdisplayMode=TEXT
```

Figure 7. Sample Version URLs

IXML Tag Library

Overview

In IHTML 2.0, the ISELECT and ICOLLECT tags greatly improved the syntax of writing multi-version Web pages because they allow multi-version content to be organized. And in IML, the set of macros give Web authors a simple way to create multi-version Web pages without programming. IXML takes these ideas to create an XML tag library to aid the construction of a multi-version Web site.

IXML Tags

IXML tags are used in conjunction with other XML tags to construct multi-version Web pages. The convention used is that IXML tags are distinguished with names beginning with an "i". The following lists the main IXML tags:

1. <iParameters> & <iParameter>
2. <iSelect> and <iCase>
3. <iParameterSelects> and <iParameterSelect>
4. <iTree>
5. <iHandlers>

iParameters & iParameter

DTD

```
<!ELEMENT iParameters (iParameter+)>
<!ELEMENT iParameter EMPTY>
<!ATTLIST iParameter
    name  CDATA #REQUIRED
    value CDATA #REQUIRED
>
```

Description

The <iParameters> tag list the version parameters used in the page. The <iParameter> tag identifies each parameter and its default value. It is also used to declare the parameter into the system.

Example

In the URL below, there are two version parameters. The "language" parameter starts out with the default value of "ENGLISH" and "displayMode" starts out with the

default value of "GRAPHIC". As a user interact with the page, the parameter values change accordingly. For example if the user request the URL, http://localhost/greeting/page.xml?iParamlanguage=FRENCH&iParamdisplayMode=GRAPHIC], the language parameter would be updated to "FRENCH" and the displayMode parameter would remain as "GRAPHIC".

<iSelect> and <iCase>

DTD

```
<!ELEMENT iSelect (iCase*, defaultICase)>
<!ELEMENT iCase (*)>
<!ATTLIST iCase
    parameterAttribute+
    isVisible CDATA #REQUIRED
>
```

Description

An <iSelect> tag consists of zero or more user defined <iCase> tags and one default <iCase> tag. User defined <iCase> tags contain one or more attributes that specify a specific version of a version parameter and one isVisible attribute that is either "TRUE" or "FALSE". The last <iCase> is the default <iCase> that is selected if no other <iCase> is chosen. It only contains the isVisible attribute.

The <iSelect> and <iCase> tags, following the behaviour in IHTML 2.0, is used to provide a multi-version selective feature very much like a switch statement in the C programming language. The first <iCase> that refines to the requested version is selected. If none of the user defined <iCase>s matches the requested version then the default <iCase> is selected.

Example

In the example below, the <greeting> tag is composed of an <iSelect> tag. This indicates that the greeting tag is multi-versioned, whose value depends on the current value of the version parameters. If the displayMode parameter is "GRAPHIC", then the greeting would be an image. If the displayMode parameter is "TEXT" and the language parameter is "ENGLISH", the greeting would be "Hello!". If the displayMode parameter is "TEXT" and the language parameter is "FRENCH", the greeting would be "Bonjour!". Finally if there are no versions of greeting that satisfy the version parameters, then the greeting would have default value of "E (DEFAULT)".

```
<page>
 <iParameters>
   <iParameter name="language" value="ENGLISH"/>
   <iParameter name="displayMode" value="GRAPHIC"/>
 </iParameters>
 <greeting>
 <iSelect>
  <iCase displayMode="GRAPHIC" isVisible="FALSE">
    <image id="imageScolo">
      ../greeting.gif
    </image>
  </iCase>
  <iCase displayMode="TEXT" language="ENGLISH" isVisible="FALSE">Hello!</iCase>
  <iCase displayMode="TEXT" language="FRENCH" isVisible="FALSE">Bonjour!</iCase>
  <iCase isVisible="TRUE">E (DEFAULT)</iCase>
 </iSelect>
 </greeting>
</page>
```

Figure 8. Example of the <iSelect> tag.

<parameterSelects> and <parameterSelect>

DTD

```
<!ELEMENT iParameterSelects (iActionLink, iParameterSelect+, iSubmitHeading)>
<!ELEMENT iActionLink (#PCDATA)>
<!ELEMENT iParameterSelect (iHeading, iOption+)>
<!ELEMENT iHeading (#PCDATA)>
<!ELEMENT iSubmitHeading (#PCDATA)>
<!ELEMENT iOption (iOptionLink, iOptionHeading)>
<!ELEMENT iOptionLink (#PCDATA)>
<!ELEMENT iOptionHeading (#PCDATA)>
<!ATTLIST iParameterSelect
   name  CDATA #REQUIRED
>
```

Description

This tag specifies an interactive control that allows a user to update a Web page with different version parameter values. It consists of one or more <iParameterSelect> tags.

The following lists the tags that compose the <iParameterSelects> tag.

<iParameterSelect>

The <iParameterSelect> tag consists of an <iHeading> tag, one or more <iOption> tags and an <iSubmitHeading> tag. The name attribute of each <iParameterSelect> tag gives the name of the version parameter to be manipulated.

<iHeading>

The <iHeading> tag specifies what will be displayed for that version parameter. In the example, for the "paramlanguage" <iParameterSelect> tag, since the <iHeading> value is "language", this is the value that is displayed on the Web page.

<iOption>

The <iOption> tags list all the possible values that a version parameter can have. The tag consists of one <iOptionLink> tag and one <iOptionHeading> tag. The <iOptionHeading> tag shows what is displayed for that option and the <iOptionLink> shows the actual value that version parameter will have when the control is updated. In the example, the displayMode version parameter has two possible values, "TEXT" and "GRAPHIC". Since the <iOptionHeading> tags for this option is lowercase, this is what is displayed on the Web page. For example, if a user chose the displayMode to be "text" and the language version parameter to be "english", and updated the control, then the URL that will sent to the Web server will be,

http://localhost:8080/workMay/greeting/greeting/clean-page.xml?paramlanguage=ENGLISH¶mdisplayMode=TEXT

<iSubmitHeading>

The <iSubmitHeading> tag specifies what will be displayed on the submit button of the control. In the example, since the <iSubmitHeading> value is "update", this is the value that is displayed on Web page.

Example

The figures below show the <iParameterSelects> tag for the greeting example and also what the tag looks like when translated to HTML. Using this tag, users can change the language and displayMode version parameter values.

```
<iParameterSelects>
    <iActionLink>./clean-page.xml</iActionLink>
    <iParameterSelect name="language">
        <iHeading>language</iHeading>
        <iOption><iOptionLink>ENGLISH</iOptionLink><iOptionHeading>english</iOptionHeading></iOption>
        <iOption><iOptionLink>FRENCH</iOptionLink><iOptionHeading>french</iOptionHeading></iOption>
    </iParameterSelect>
        <iParameterSelect name="displayMode">
        <iHeading>displayMode</iHeading>
        <iOption><iOptionLink>TEXT</iOptionLink><iOptionHeading>text</iOptionHeading></iOption>
        <iOption><iOptionLink>GRAPHIC</iOptionLink><iOptionHeading>graphic</iOptionHeading></iOption>
    </iParameterSelect>
    <iSubmitHeading>Update</iSubmitHeading>
</iParameterSelects>
```

Figure 9. Example of <iParameterSelects> tag

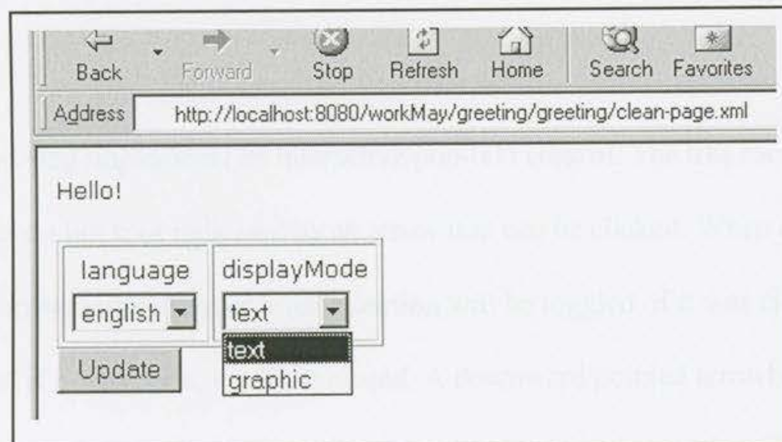


Figure 10. Corresponding HTML representation of <iParameterSelects> tag

<iTree>

DTD

<!ELEMENT iTree (iTreeHeading, (iBranch|iLeaf)*)>

```
<!ATTLIST iTree
    id CDATA #REQUIRED
    isOpen CDATA #REQUIRED
>
<!ELEMENT iTreeHeading (#PCDATA)>
<!ELEMENT iLeaf (*)>
<!ELEMENT iBranch (iTreeHeading, (iBranch|iLeaf)*)>
<!ATTLIST iBranch
    isOpen CDATA #REQUIRED
>
```

Description

The <iTree> tag implements an interactive pop-text control. The tree root and all tree branches have a hot spot indicated by an arrow that can be clicked. When a hot spot is clicked, the corresponding root or branch section will be toggled. If it was closed, it will be open, and if it was open, it will be closed. A downward pointed arrowhead means that section is open and a right pointed arrowhead means that section is closed.

The following are the tags that make up an iTree.

<iTree>

An <iTree> is made up of an <iHeading> and zero or more <iLeaf> or <iBranch> tags. The tree is uniquely identified by an "id" attribute. The "isOpen" attribute, which

can either be "TRUE" or "FALSE", indicates if the root is either initially opened or closed.

<iTreeHeading>

The <iTreeHeading> tag is used as the displayed label for an <iTree> or <iBranch> tag. In the example, the tree root has an <iTreeHeading> of "IHTML Implementations" and so this is what is displayed for the root on the Web page.

<iBranch>

The <iBranch> is almost exactly the same as an <iTree> tag. It is made up of an <iTreeHeading> and zero or more <iLeaf> or <iBranch> tags. The attribute "isOpen" and the < iTreeHeading > tag function the same way as in the case of <iTree>.

<iLeaf>

The <iLeaf> tag can contain any other tag or text.

Example

The figures below lists the IXML fragment that implements an <iTree> and also shows what the corresponding HTML page looks like. In the first picture, since the arrowhead is pointed to the right, the tree root is closed. In the second picture, the tree root is open but one of the root branches is closed.

```
<iTree id="tree1" isOpen="TRUE">
    <iTreeHeading>IHTML Implementations</iTreeHeading>
    <iLeaf>IHTML 1.0</iLeaf>
    <iLeaf>IHTML 2.0</iLeaf>
    <iLeaf>ISE/IML</iLeaf>
    <iBranch isOpen="FALSE">
      <iTreeHeading>IXML Technologies</iTreeHeading>
      <iLeaf>XML/XSL</iLeaf>
      <iLeaf>Java Servlets</iLeaf>
      <iLeaf>Cocoon</iLeaf>
    </iBranch>
</iTree>
```

Figure 11. Example iTree.

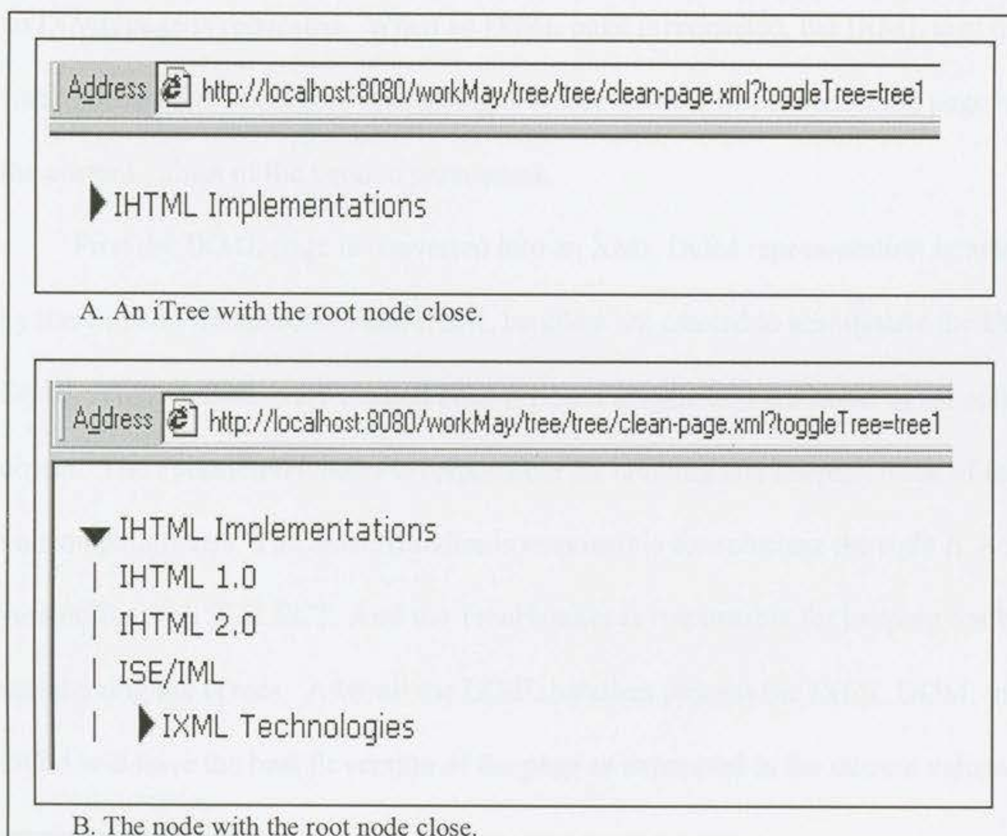


Figure 12. Corresponding HTML representation of iTree code above.

<iHandlers/>

DTD

<!ELEMENT iHandlers (NULL))>

Description

The <iHandlers/> tag is the last tag that goes before the root tag terminates. It is used to signal the IXML handlers to process the IXML page.

IXML Class Library

Th figure below shows the processing the IXML Java class library performs when an IXML page is requested. When an IXML page is requested, the IXML tags in the static content file undergoes processing to select the best fit version of the page based on the current values of the version parameters.

First the IXML page is converted into an XML DOM representation in memory by the Cocoon framework. Then IXML handlers are created to manipulate the IXML DOM. These handlers are created once for each session and are saved in the session object. The ParameterHandler is responsible for creating and keeping track of the version parameters. The SelectHandler is responsible for selecting the right ICASE version for each ISELECT. And the TreeHandler is responsible for keeping track and maintaining the ITrees. After all the IXML handlers process the IXML DOM, the DOM will have the best fit version of the page as expressed in the current values of the version parameters.

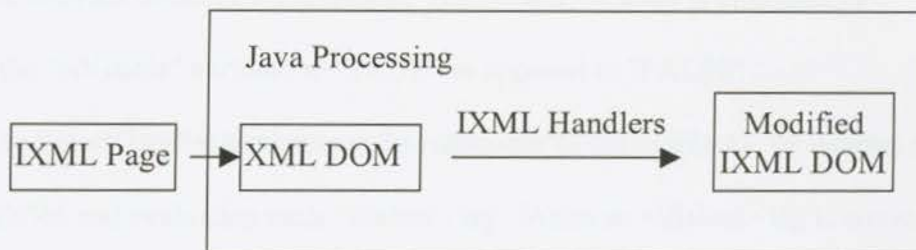


Figure 13. IXML Java Processing

ParameterHandler

The ParameterHandler is responsible for creating and keeping track of the version parameters. When it gets created, it looks at the <iParameters> tag and creates a variable for each <iParameter> and initializes it with the default value. Then for every request, it looks at the version portion of the URL and updates the version parameters accordingly. For example, receiving a URL like this,

http://localhost:8080/workMay/tan/tan/clean-page.xml?paramlanguage=TURKISH¶mdisplayMode=TEXT, the ParameterHandler will update the version parameter "language" with the value "TURKISH" and update the version parameter "displayMode" with the value of "TEXT".

SelectHandler

The SelectHandler is responsible for selecting the correct <iCase> statement based on the values of the version parameters. The behaviour of <iSelect> is that it should choose the first <iCase> that satisfies the request parameters and if no <iCase> is

satisfied, then the default case is chosen. The chosen <iCase> is indicated by setting the "isVisible" attribute to "TRUE" as opposed to "FALSE".

The SelectHandler implements the behaviour of the <iSelect> by looking at the IXML DOM and evaluating each <iSelect> tag. When an <iSelect> tag is encountered, the SelectHandler evaluates all the <iCase> tags of that <iSelect> with respect to the version parameters and it will set the "isVisible" attribute of one <iCase> statement to "TRUE" and all the others to "FALSE".

TreeHandler

The TreeHandler is responsible for keeping track and storing the state of all the trees in the Web page. As users interact with an iTree, they will open and close nodes. These changes are reflected in the "isOpen" attributes. If a node is open, the "isOpen" attribute is "TRUE", otherwise "FALSE".

When the TreeHandler gets created, it will parse the IXML DOM and store each iTree to keep track of the "isOpen" attribute of the root and branches. When users manipulate a iTree, which is indicated by the URL, the TreeHandler will update that iTree. For example a URL of,

http://localhost:8080/workFeb/show/tree/clean-page.xml?toggleTree=articles|article1, indicate the branch with id attribute "articles|article1" of tree id "articles" has been toggled. The TreeHandler will update that iTree by toggling the node that has been clicked. If the "isOpen" attribute is "TRUE", it will be set to "FALSE" and vice versa.

Finally the TreeHandler will modify the IXML DOM to reflect the latest version of the iTree in memory. And so all the "isOpen" attributes of the <iTree>s in the IXML DOM will get replaced with the current value in memory.

IXML Style-Sheet

Overview

The IXML style sheet helps transform an IXML page into an HTML page. It transforms IXML tags into HTML presentation. By the Cocoon model each XML page has a corresponding html style-sheet that defines HTML transformations for user defined XML tags. In the user defined XSL style-sheet, they can take advantage of the transformations in the IXML style-sheet by importing the IXML style-sheet. In a previous example, the IXML style-sheet is imported using the following line:

<xsl:import href="http://localhost:8080/workMay/ixml/page-ixml.xsl"></xsl:import>

Also the user can over-ride any tag in IXML style-sheet and render the IXML tags anyway they like by simply redefining the tag in the user style-sheet.

Default HTML presentation of IXML tags

There are four main IXML tags that the IXML style-sheet works on:

1. page
2. iSelect, iCase
3. iTree
4. iParameterSelects

<page>

The <page> tag represents the root of an IXML document. The IXML style-sheet simply transforms this tag into a generic HTML outline as follows:

```
<html>
   <head>
   </head>
   <body>
     ...
   </body>
</html>
```

If there is a need to put more details such as background, title, etc, the user can simply override the default behaviour of this tag.

<iSelect> and <iCase>

The <iSelect> and <iCase> tags are used to selectively include output in the final HTML document. After the IXML Java library processes the IXML DOM, every <iSelect> tag will contain one and only one <iCase> where its isVisible attribute is "TRUE". The IXML style-sheet will only process the content of the chosen <iCase> and ignore the others.

<iTree>

Figure 12 shows how the IXML style-sheet renders an <iTree> as an interactive control containing hotspots beside the root and all branches that can open and close.

<iParameterSelects>

Figure 10shows how the IXML style-sheet renders an <iParameterSelects> tag. As the figure indicates, <iParameterSelects> tag allows the user to easily switch between different values of the version parameters with dropboxes. In the example there are two version parameters, language and displayMode with the displayMode having two possible options, text and graphic.

6. IXML in Practice

Example

The figure below shows screenshots of a simple multi-version greeting page. The user uses the control at the bottom of the page to specify the 'language' and the 'displayMode' that the page should be viewed. Screenshot 'a' shows the greeting when the 'displayMode' is "graphic". Screenshot 'b' shows the greeting when the 'language' is "english" and the 'displayMode' is "text". And screenshot 'c' shows the greeting when the 'language' is "french" and the 'displayMode' is "text".

Error! Not a valid link.

How the multi-version example is constructed

There are three pages that make up the simple greeting, clean-page.xml, page-xsp.xsl, and page-html.xsl. The page 'clean-page.xml' specifies the content of the IXML page. The page 'page-xsp.xsl' allows the different IXML handlers such as ParameterHandler, SelectHandler and TreeHandler to process the IXML page. And the page 'page-html.xsl' translates the IXML page into HTML presentation. The figure below lists the content of 'clean-page.xml'.

```
<?xml version="1.0"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="page-xsp.xsl" type="text/xsl"?>

<page>
 <iParameters>
   <iParameter name="language" value="ENGLISH"/>
   <iParameter name="displayMode" value="GRAPHIC"/>
 </iParameters>

 <greeting>
 <iSelect>
   <iCase displayMode="GRAPHIC" isVisible="FALSE">
     <image id="imageScolo"> balloons.gif</image>
   </iCase>
   <iCase displayMode="TEXT" language="ENGLISH" isVisible="FALSE">
     Hello!
   </iCase>
   <iCase displayMode="TEXT" language="FRENCH" isVisible="FALSE">
     Bonjour
   </iCase>
   <iCase isVisible="TRUE">
     E (DEFAULT6)
   </iCase>
 </iSelect>
 </greeting>

 <iParameterSelects>
   <iActionLink>./clean-page.xml</iActionLink>
   <iParameterSelect name="language">
     <iHeading>language</iHeading>
     <iOption>
       <iOptionLink>ENGLISH</iOptionLink>
       <iOptionHeading>english</iOptionHeading>
     </iOption>
    <iOption>
      <iOptionLink>FRENCH</iOptionLink><iOptionHeading>french</iOptionHeading>
    </iOption>
   </iParameterSelect>
   <iParameterSelect name="displayMode">
     <iHeading>displayMode</iHeading>
     <iOption>
       <iOptionLink>TEXT</iOptionLink><iOptionHeading>text</iOptionHeading>
     </iOption>
     <iOption>
       <iOptionLink>GRAPHIC</iOptionLink>
       <iOptionHeading>graphic</iOptionHeading>
     </iOption>
   </iParameterSelect>
   <iSubmitHeading>Update</iSubmitHeading>
 </iParameterSelects>
 <iHandlers/>
</page>
```

Figure 14. Listing of clean-page.xml

The structure of an IXML content page must be as follows:

```
<?xml version="1.0"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="..." type="text/xsl"?>
<page>
  <iParameters>... </iParameters>
...
  <iHandlers/>
</page>
```

The first line indicates the XML version used to construct the page. The second line indicates that the XSLT processor is needed to parse the page. The third line indicates the next file that will process the page. For example, in the greeting example, 'page-xsp.xsl' will be the next file to process the page. Finally there must be one root tag <page>....</page>. Inside this tag will be IXML tags plus user defined tags that make up the content of the page. IXML tags can be easily distinguished from user defined tags by the convention that all IXML tags begin with an "i". For example <iParameter>.

For an IXML page the first tag inside the <page> tag must be the <iParameters> tag which specifies the version parameters that the IXML page will contain. If there are no version parameters then this tag can be omitted. Then any content can be added. Finally at the last line before the <page> tag is closed will be the <iHandlers/> tag that will allow the processing of the IXML tags in the next step.

In the greeting example, two version parameters are declared, 'language' with a default value of "ENGLISH" and 'displayMode' with a default value of "GRAPHIC".

Then there is a user defined <greeting> tag. The <greeting> tag is composed of the IXML <iSelect> tag which means that the <greeting> tag is multi-versioned whose value will depend on the current values of the version parameters. If the version parameter 'displayMode' is "GRAPHIC" then the greeting will be an image. If the version parameter 'language' is "ENGLISH" and the version parameter 'displayMode' is "TEXT" then the greeting will be "hello!!". And if the version parameter 'language' is "FRENCH" and the version parameter 'displayMode' is "TEXT" then the greeting will be "hello!!". Finally, in case there is no case that satisfies the version parameter, the default case will be selected.

Finally there is the IXML <iParameterSelects> tag. This tag defines the interactive control at the bottom of the page that allows the user to select between different values of the version parameters.

The page 'clean-page.xml', specifies that 'page-xsp.xsl' is the next file that processes the IXML DOM. The figure below lists the content of this file. This page initiates all the Java processing. As a result it handles all of the IXML Java processing such as the various IXML handlers. The way the 'page-xsp.xsl' file works is that it will parse the IXML DOM from the 'clean-page.xml' and as it encounters IXML tags it will initiate the corresponding IXML actions. For example, when the <iHandlers> tag is encountered, the various IXML handlers such as SelectHandler and TreeHandler are called to operate on the IXML page.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:import href="http://localhost:8080/workMay/ixml/ixml/ixml-html.xsl"></xsl:import>

 <xsl:template match="greeting">
   <strong>
    <xsl:apply-templates/>
   </strong>
 </xsl:template>

   <xsl:template match="image[@id ='imageScolo']">
    <IMG>
     <xsl:attribute name="src">
       <xsl:apply-templates/>
     </xsl:attribute>
    </IMG>
   </xsl:template>

 </xsl:stylesheet>
```

Figure 15. Listing of page-xsp.xsl.

The structure of 'page-xsp.xsl' must be as follows:

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsp="http://www.apache.org/1999/XSP/Core"
>
<xsl:import href="..ixml-xsp.xsl"></xsl:import>

<xsl:template match="/">
...
</xsl:template>
</xsl:stylesheet>
```

There are only a few things that have to be modified from the 'page-xsp.xsl' template given in the greeting example to work for the other examples. The first thing is that the value of the import statement should be changed to reflect the path to the ' ixml-xsp.xsl' file. The 'ixml-xsp.xsl' file contains most of the XSL template rules that handles the IXML processing. And the <xsl:template match="/"> template rule can be modified to specify an alternative file that will handle the next step of the processing. In the greeting example, it specifies that the 'page-html.xsl' should process the IXML DOM next. Also if the user has any need to do Java processing it can be done.

Finally the content of 'page-html.xsl' can be seen the figure below. This file contains all the template rules necessary to convert the IXML page into an HTML page.

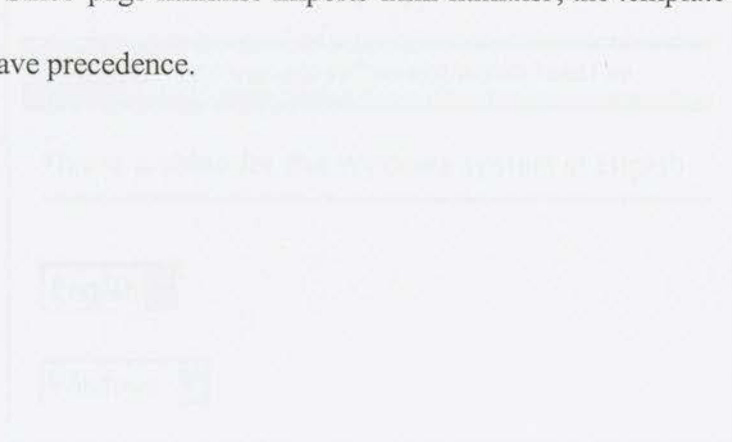
```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:import href="http://localhost:8080/workMay/ixml/ixml/ixml-html.xsl"></xsl:import>

 <xsl:template match="greeting">
   <strong>
     <xsl:apply-templates/>
   </strong>
 </xsl:template>

  <xsl:template match="image[@id ='imageScolo']">
   <IMG>
     <xsl:attribute name="src">
       <xsl:apply-templates/>
     </xsl:attribute>
   </IMG>
  </xsl:template>

</xsl:stylesheet>
```

Figure 16. Listing of page-html.xsl.

The structure of the file is as follows:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:import href="… ixml-html.xsl"></xsl:import>
…
</xsl:stylesheet>
```

All of the functionality needed to translate the IXML tags into HTML is imported from the 'ixml-html.xsl' style-sheet. For example, the HTML representation of the <iParameterSelect> tag is created in this way. The user can override how each IXML tag is translated into HTML by redefining that IXML template rule inside 'page-html.xsl'. Since 'page-html.xsl' imports 'ixml-html.xsl', the template rules defined in that file will have precedence.



7. Comparison of IXML to IML

Since both IXML and IML have a similar goal to make it easier for web designers with little or no programming knowledge to create multidimensional web site, it is worth comparing the usability of each system from a Web designer point of view.

Case Study One: A simple multi-parameter page

The figure below shows a simple multi-parameter page with two parameters "language" and "operating system" implemented in both IML and IXML. The language parameter can either be ENGLISH or FRENCH. And the operating system parameter can either be "WINDOWS" or "MACINTOSH".

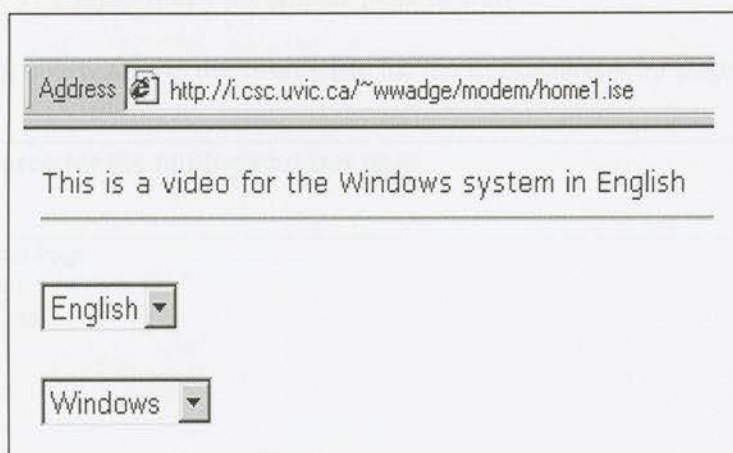


Figure 17. A simple multi-parameter page in IML.

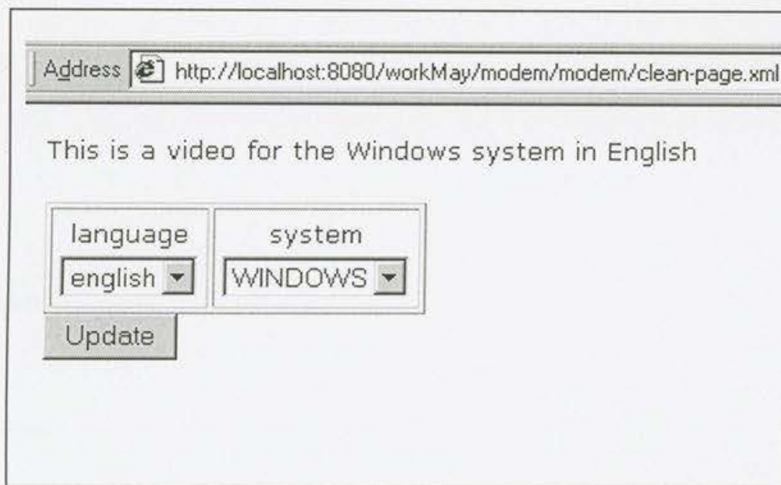


Figure 18. A simple multi-parameter page in IXML.

The listing below shows the source file for the multi-parameter page in IML & IXML.

a. IML source for the multi-parameter page

```
.bdoc Modem Page
.initdim lg en
.initdim os win
<body>
.val video
<hr>
.bmenu lg
.option en
.2ling English Anglais
.option fr
.2ling French Francais
.emenu

<p>

.bmenu os
.option mac
Macintosh
.option win
Windows
.emenu

.bdef video
.2ling "This is a video for the" "Voici un video pour le systeme"
.val sys
```

```
.2ling system ""
.2ling in en
.val language
.edef

</body>
.bdef sys
Weird OS
.edef

.bdef sys os:win
Windows
.edef

.bdef sys os:mac
Macintosh
.edef

.bdef language lg:fr
Francais
.edef

.bdef language lg:enEnglish
.edef

.edoc
```

b. IXML source for the multi-parameter page

```
<?xml version="1.0"?>
<?cocoon-process type="xslt"?>
<?xml-stylesheet href="page-xsp.xsl" type="text/xsl"?>
<page>
 <iParameters>
  <iParameter name="language" value="ENGLISH"/>
    <iParameter name="system" value="WINDOWS"/>
 </iParameters>

<sentence>
<beginning>
<iSelect>
 <iCase language="ENGLISH" isVisible="FALSE">
  This is a video for the
 </iCase>
 <iCase language="FRENCH" isVisible="FALSE">
  Voici un video pour le
 </iCase>
 <iCase isVisible="TRUE">E (DEFAULT7)</iCase>
</iSelect>
</beginning>
```

```
<system>
<iSelect>
 <iCase language="ENGLISH" system="WINDOWS" isVisible="FALSE">
  Windows system
 </iCase>
 <iCase language="ENGLISH" system="MAC" isVisible="FALSE">
  Macintosh system
 </iCase>
 <iCase language="FRENCH" system="WINDOWS" isVisible="FALSE">
  systeme Windows
 </iCase>
 <iCase language="FRENCH" system="MAC" isVisible="FALSE">
  systeme Macintosh
 </iCase>
 <iCase isVisible="TRUE">E (DEFAULT7)</iCase>
</iSelect>
</system>

<end>
 <iSelect>
  <iCase language="ENGLISH" isVisible="FALSE">
  in English
  </iCase>
  <iCase language="FRENCH" isVisible="FALSE">
  en Francaise
  </iCase>
  <iCase isVisible="TRUE">E (DEFAULT7)</iCase>
 </iSelect>
 </end>
 </sentence>

 <iParameterSelects>
  <iActionLink>./clean-page.xml</iActionLink>
   <iParameterSelect name="language">
     <iHeading>language</iHeading>
    <iOption>
      <iOptionLink>ENGLISH</iOptionLink>
      <iOptionHeading>
      <iSelect>
        <iCase language="ENGLISH" isVisible="FALSE">english</iCase>
        <iCase language="FRENCH" isVisible="FALSE">anglaise</iCase>
        <iCase isVisible="TRUE">E (DEFAULT7)</iCase>
       </iSelect>
      </iOptionHeading>
    </iOption>
      <iOption>
       <iOptionLink>FRENCH</iOptionLink>
      <iOptionHeading>
      <iSelect>
         <iCase language="ENGLISH" isVisible="FALSE">french</iCase>
         <iCase language="FRENCH" isVisible="FALSE">francaise</iCase>
```

```
    <iCase isVisible="TRUE">E (DEFAULT7)</iCase>
   </iSelect>
  </iOptionHeading>
 </iOption>
 </iParameterSelect>
    <iParameterSelect name="system">
      <iHeading>
    <iSelect>
       <iCase language="ENGLISH" isVisible="FALSE">system</iCase>
       <iCase language="FRENCH" isVisible="FALSE">systeme</iCase>
       <iCase isVisible="TRUE">E (DEFAULT7)</iCase>
      </iSelect>
    </iHeading>
     <iOption>
      <iOptionLink>WINDOWS</iOptionLink>
     <iOptionHeading>WINDOWS</iOptionHeading>
    </iOption>
      <iOption>
      <iOptionLink>MAC</iOptionLink>
     <iOptionHeading>MAC</iOptionHeading>
   </iOption>
    </iParameterSelect>
   <iSubmitHeading>Update</iSubmitHeading>
</iParameterSelects>
<iHandlers/>
</page>
```

There are three sections that compose the page:

- parameter initiation
- parameterSelect
- sentence output

In both IML and IXML the parameter initiation section is where the multi-version parameters are initialized. Both systems have tags to initialize multi-version variables. In IML, they are called dimensions.

In IML, you don't need to initialize your parameters but in IXML you do need to initialize all the multi-version parameters that you are using on the page. The reason for

this is that the tag used to initialize the multi-version parameter also serves to declare the variable to the system.

In IXML, since the multi-version parameters is based on JavaServlet parameters, the parameters are stored in the Session object. Whereas IML is based on CGI which has no state and so the entire multi-version parameter and value pair must be stored on the URL every time.

The parameterSelect section allows the person viewing the Web page to view the page with different parameter values. For example the person can use the control to change the language. The overall look and functionality between the systems here are very similar. The differences are that the IML syntax is very succinct compared to the IXML syntax. But the more complex markup for the IXML control allows a richer set of options. In IML you can an only update one parameter at a time whereas in IXML you can change the values of as many parameter as you want and then update the page all at once. Also in the IXML ParameterSelect control, the current values of the parameters are always displayed as the current values of the control.

The sentence output section is where an output is generated based on the values of the multi-version parameters. The output for both is the same but the methods used to generate the output are very different. Again, the IML source is more succinct. IML directly outputs the values of the multi-version parameters. Although IXML could have been implemented this way, IXML uses the iSelect tag to output multi-version content. Using iSelect to selectively output multi-version content is a little bit more powerful because the output does not necessarily have to be the same as the value of the multi-

version parameter as in the case of directly outputting the value of the parameter. Also the output can be based on the values of more than one multi-version parameter.

And although the IML source is more succinct, since the IXML source is based on XML, the data can be better organized and abstracted. For example the output is made up of the sentence tag as follows: <sentence><beginning/><system/></end></sentence>

This can be interpreted as follows: the <sentence> tag contains the sentence output and there are three parts that make up the sentence: <beginning/>, <system/>, and <end/>. And you can then systematically abstract in each tag to see a higher degree of complexity.

Case Study Two: Pop-text

The figure below shows a sample pop-text.

Figure 19. A pop-text section.

The look and functionality of both systems are almost identical. The only major difference is that the IXML implementation allows the page designer to control what areas of pop-text to be open by default. In IXML it is most natural to let the entire pop-text region to be initially all closed.

Although the look and functionality of both systems are almost identical, the implementation for each system is radically different. IXML abstracts pop-text as a tree control very much like a tree control found in most GUI toolkits. And so a pop-text region can be created with an iTree tag. And to put content in the tree, you can add

branches and leaves. Abstracting pop-text as a tree totally encapsulates a pop-text region. Everything related to a pop-text region is contained within its iTree. As a result it is easier to identify where a pop-text region starts and ends and also it is easier to manipulate the whole tree.

Another major difference is that the IXML implementation is mainly concerned about the data of the pop-text and the way the data is presented in HTML is handled later. As result, the IXML implementation is a little cleaner since it does not have HTML syntax scattered throughout the pop-text region.

Comparison

In addition to sharing a similar goal to make it easier for web designers with little or no programming knowledge to create multidimensional web site, both IXML and IML also provide equivalent functionality. For the Web designer, each system has its strengths.

The advantages of IML over IXML are:

- the tags are more succinct
- there is only one source file
- the system is more mature and so the implementation more polished

The advantages of IXML over IML are:

- there are three different source files that clearly separate the data, programming, and HTML presentation

- runs on multiple platforms and Web servers
- IXML markup is XML based and you are able to leverage all the existing XML tools

With the advantage of IML having more succinct markup and requiring only one source file, the system may be better at handling simpler multi-version pages. Also IML may be a better fit for legacy systems where you have lots of pages with existing HTML since with IML you can just simply insert IML markup to an existing page to convert it to a multi-dimension page.

With the advantages of XML being better structured by separating the data/programming/html presentation and using standard XML as markup, it may offer a more complete, scaleable and manageable system especially for complicated and big multi-version Web pages.

8. Wrap-up

Future Work

There are many things that still can be done to enhance IXML. It is noted that there are only three major IHTML elements that are of concern in efficiently converting legacy IHTML, versioned file-inclusion and ISELECT and ICOLLECT [16]. The <iSelect> tag has been converted but the <iCollect> tag and version file inclusion have not yet been implemented. Converting these two items should be as easy as the <iSelect> but the author wanted to concentrate on other things. Also XML should make version file-inclusion unnecessary because XML allows much more sophisticated levels of inclusion. It is possible to include a tag from one XML source file into another source file. And so it is possible to simulate a version file inclusion by importing the root of an XML document.

Also it is noted that the <iSelect> tag is not a best-fit mechanism. It simply finds the first version-case to refine to the requested version [16]. And so it might be useful to add a file-internal construct for best-fit blocks such as an <iBestFit> tag that has the similar syntax to <iSelect> but instead of choosing the first <iCase> that satisfies the version parameters, it chooses the <iCase> that best fits the version parameters.

And the behaviour of IXML tags can be enhanced in many ways. At the present moment clicking on a root or branch will toggle that node. It would be very easy to implement additional toggling behaviour. For example, clicking on a node, could open and close all the children of that node. Or clicking on a node can open or close all of its sibling nodes.

And at present <iParameters> tag must include all the parameters that the page requires. In a site it is very common that there are global parameters that are relevant to all pages. And so instead of including these global parameters manually for each page, with XML it should be possible to define the global parameters in one XML file and link that file to all the pages that require it.

And the IXML implementation can be made more efficient. At the moment, each IXML handler takes the entire DOM representation of an IXML page and has to scan the entire DOM looking for specific IXML tags. Since the 'page-xsp.xsl' already uses XSL template rules to scan the IXML DOM once, the process should not have to be repeated.

But the next great goal could be to make the system run client-side. At the present moment the system is server based and so each time a page is updated with new multi-version parameter values, the server must process the request and refresh the page. But since the system now is XML based, it is very conceivable that the entire processing can be done on the client-side. IXML consists of three parts, the IXML tag library and the IXML class library and the XSL templates to translate XML to HTML. If a Web browser had an XML and XSL parser, then it would be possible to process the XML markup on an IXML page to HTML on the client. But currently, no browser is capable of doing this very well but future browsers look very promising. The only other challenge would be to allow the IXML class library to run on the client-side. There are two possibilities for this. If the browser supports Java and allow it to run, then the IXML

class library may be able to be implemented as an applet. Or else it is also possible to implement the library in JavaScript since most browsers support it.

Conclusion

Although there are so many ways to improve IXML, the main goals are achieved. In addition to keeping the strong qualities of previous implementations of IHTML such as performance and programming versatility, IXML addresses the shortcomings of previous implementations of IHTML. In particular IXML stresses that the multi-versioning framework is easily portable between multiple operating systems and servers and IXML abstract the complexities of the framework so that a typical Web page designer who has little or no programming experience can effectively use IXML.

Since IXML is an application of Cocoon which itself is built as an application of Java Servlets, IXML will run on most operating systems and Web Servers. Multi-version content is easily assembled by using the IXML tag library that abstracts the versioning and programming logic behind the functionality of the multi-version tags. As a result common multi-version Web pages that have been implemented in previous implementations of IHTML such as multi-language pages and pop-text are not only possible in IXML, but are much more organized. And also IXML only focuses on the multi-versioning infrastructure of a Web infrastructure and relies on the underlying Cocoon Web publishing framework and Java Servlets to handle all the other common aspects of a Web framework such as performance and stability.

Bibliography

[1] G. D. Brown. Intensional HTML: A Practical Approach. Master's thesis, University of Victoria, Victoria, British Columbia, Canada, 1998.

[2] Taner Yildirim. Intensional HTML. Master's thesis, University of Victoria, Victoria, British Columbia, Canada, !997.

[3] Bill Wadge. Intensional Markup Language. University of Victoria, Victoria, British Columbia, Canada, 2000.

[4] Cocoon. The Cocoon Apache Project. http://cocoon.apache.org.

[5] John Bosak. Media-Independent Publishing:Four Myths about XML http://metalab.unc.edu/pub/sun-info/standards/xml/why/4myths.htm.

[6] H. Maruyama, K. Tamura, and N. Uramoto. XML and Java: Developing Web Applications. Addison Wesley Longman, Inc. One Jacob Way, Reading, Massachusett, 01867, USA, 1999.

[7] Norman Walsh. A Technical Introduction to XML. http://xml.com/xml/pub/98/10/guide2.html

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

[9] E.R. Harold. XML Bible. IDG Books Worldwide, Inc. 919 E. Hillsdale Blvd., Suite 400, Foster City, CA 94404-2112.

[10] Matt Kruse. Introduction to CGI. http://www.mattkruse.com/info/cgi/.

[11] Pat Durante. Write CGI programs in Java. http://www.javaworld.com/javaworld/jw-01-1997/jw-01-cgiscripts.html.

[12] Open Market. FastCGI: A High-Performance Web Server Interface. http://www.fastcgi.com/fcgi-devkit-2.1/doc/fastcgi-whitepaper/fastcgi.htm.

[13] Jason Hunter. Java Servlet Programming. O'reilly and Associate, Inc. 90 Sherman Street Cambridge, MA 02140.

[14] JavaServer Product Group. The Java Servlet API. http://jserv.javasoft.com/products/documentation/servlets/api.html.

[15] Oreilly Online Catalog. Java Servlet Programming. http://www.oreilly.com/catalog/jservlet/desc.html

[16] Paul Swoboda. Practical Languages for Intensional Programming. Master's thesis, University of Victoria, Victoria, British Columbia, Canada, !999.

[17] What is XML. http://xml.com/xml/pub/98/10/guide2.html

Appendix A: XML Markup

XML documents are composed of markup and content. There are six kinds of markup that can occur in an XML document:

1) Elements
2) Entity References
3) Comments
4) Processing Instruction
5) Marked sections
6) Document type declaration

Elements

Elements are the most common form of markup. They are delimited by angle brackets and they usually identify the nature of the content they surround. Elements may be empty, eg/ <applause/>. If an element is not empty, it begins with a start tag, <element>, and ends with an end tag, </element>.

Elements may have attributes associated with them. Attributes are name-value pairs that occur inside start-tags after the element name. For example, <parameter language ="ENGLISH">, is a parameter element with the attribute language having value ENGLISH.

Entity References

Some characters have been reserved by XML. For example the left angle bracket, <, is reserved to identify the beginning of an element start or end tag. Entities are used to insert reserved characters as content into your document. For example, the lt entity inserts a literal < into a document. So the string <element> can be represented in an XML document as < element>. Entities are also used to refer to often repeated or varying text and to include the content of external files.

Comments

Comments begin with <!-- and end with -->. Comments can contain any data except the literal string --. Comments are not part of the textual content of an XML document and the XML processor are not required to pass the string to an application.

Processing Instruction (PIs)

Processing instructions are used to provide information to an application. They are not textually part of the XML document but the XML processor is required to pass them to an application. PIs have the form <?name pidata ?>. The name, called the PI target, identifies the PI to the application. Applications should process only the target they recognize and ignore all other Pis. The pidata is optional data to be processed by the application.

CDATA

CDATA instructs the parser to ignore most marker characters. Here is an example CDATA section,

```
<![CDATA[
 *p = &q;
 b = (i <= 3);
]]>
```

Between the start section, <!CDATA[and the end of the section,]]>, all character data is passed directly to the application without interpretation. The only string that cannot occur in a CDATA section is]]>.

Document Type Definitions

A large percentage of the XML specification deals with various sorts of declarations that are allowed in XML. Declarations allow a document to communicate meta-information to the processor about its content. Meta-information includes the allowed sequence and nesting of tags, attribute values and their types and defaults, the names of external files that may be referenced and whether or not they contain XML, the formats of some external (non-XML) data that may be referenced, and the entities that may be encountered.

There are four kinds of declarations in XML: element type declarations, attribute list declaratations, entity declarations, and notation declarations.

Element type declarations

Element type declarations identify the names of elements and the nature of their content. An example of a typical element type declaration looks like this:

```
<!ELEMENT oldjoke (burns+, allen, applause?)>
```

This declaration identifies the element named oldjoke. Its content model, the definition of what an element may contain, follows the element name. In this case, an oldjoke must contain burns and allen and may contain applause. The commas between element names indicate that they must occur in succession. The plus after burns indicates that it may be repeated more than once but must occur at least once. The question mark after applause indicates that it is optional (it may be absent, or it may occur exactly once). A name with no punctuation, such as allen, must occur exactly once.

In addition to element names, the special symbol #PCDATA is reserved to indicate character data. For example, here is an element declaration

```
<!ELEMENT burns (#PCDATA | quote)*>
```

The vertical bar indicates an or relationship, the asterisk indicates that the content is optional (may occur zero or more times); therefore, by this definition, burns may contain zero or more characters and quote tags, mixed in any order. All mixed content models must have this form: #PCDATA must come first, vertical bars must separate all of the elements, and the entire group must be optional. Two other content models are

possible: EMPTY indicates that the element has no content (and consequently no end-tag), and ANY indicates that any content is allowed.

Attribute List Declarations

Attribute list declarations identify which elements may have attributes, what attributes they may have, what values the attributes may hold, and what value is the default. A typical attribute list declaration looks like this:

```
<!ATTLIST oldjoke
   name
ID
   label
CDATA
   status ( funny | notfunny ) 'funny'>
```

In this example, the oldjoke element has three attributes: name, which is an ID and is required; label, which is a string (character data) and is not required; and status, which must be either funny or notfunny and defaults to funny, if no value is specified.

Each attribute in a declaration has three parts: a name, a type, and a default value.

There are six possible attribute types:

- CDATA

CDATA attributes are strings, any text is allowed

- ID

The value of an ID attribute must be a name. All of the ID values used in a document must be different. IDs uniquely identify individual elements in a document. Elements can have only a single ID attribute.

- IDREF or IDREFS

An IDREF attribute's value must be the value of a single ID attribute on some element in the document. The value of an IDREFS attribute may contain multiple IDREF values separated by white space.

- ENTITY or ENTITIES

An ENTITY attribute's value must be the name of a single. The value of an ENTITIES attribute may contain multiple entity names separated by white space.

- NMTOKEN or NMTOKENS

Name token attributes are a restricted form of string attribute. In general, an NMTOKEN attribute must consist of a single word, but there are no additional constraints on the word, it doesn't have to match another attribute or declaration. The value of an NMTOKENS attribute may contain multiple NMTOKEN values separated by white space.

- A list of names

You can specify that the value of an attribute must be taken from a specific list of names. This is frequently called an enumerated type because each of the possible values is explicitly enumerated in the declaration. There are four possible default values:

- #REQUIRED

The attribute must have an explicitly specified value on every occurrence of the element in the document.

- #IMPLIED

The attribute value is not required, and no default value is provided. If a value is not specified, the XML processor must proceed without one. An attribute can be given any legal value as default.

- "value"

The attribute value is not required on each element in the document, and if it is not present, it will appear to be the specified default.

- #FIXED

An attribute declaration may specify that an attribute has a fixed value. In this case, the attribute is not required, but if it occurs, it must have the specified value. If it is not present, it will appear to be the specified default.

Entity Declarations

Entity declarations allow you to associate a name with some other fragment of content. That construct can be a chunk of regular text, a chunk of the document type declaration, or a reference to an external file containing either text or binary data.

A few typical entity declarations are shown below:

```
<!ENTITY
ATI
"ArborText, Inc.">

<!ENTITY boilerplate     SYSTEM
"/standard/legalnotice.xml">

<!ENTITY ATIlogo
SYSTEM "/standard/logo.gif" NDATA GIF87A>
```

There are three kinds of entities:

- Internal Entities

  Internal entities associate a name with a string of literal text. The first entity in fig 1is an internal entity. Using &ATI; anywhere in the document will insert ArborText, Inc. at that location. Internal entities allow you to define shortcuts for frequently typed text or text that is expected to change, such as the revision status of a document.

  Internal entities can include references to other internal entities, but it is an error for them to be recursive.

  The XML specification predefines five internal entities:

  - < produces the left angle bracket, <
  - > produces the right angle bracket, >
  - & produces the ampersand, &
  - ' produces a single quote character (an apostrophe), '
  - " produces a double quote character, "

- External Entities

External entities associate a name with the content of another file. External entities allow an XML document to refer to the contents of another file. External entities contain either text or binary data. If they contain text, the content of the external file is inserted at the point of reference and parsed as part of the referring document. Binary data is not parsed and may only be referenced in an attribute. Binary data is used to reference figures and other non-XML content in the document.

- Parameter Entities

Parameter entities can only occur in the document type declaration. A parameter entity declaration is identified by placing % (percent-space) in front of its name in the declaration. The percent sign is also used in references to parameter entities, instead of the ampersand. Parameter entity references are immediately expanded in the document type declaration and their replacement text is part of the declaration, whereas normal entity references are not expanded. Parameter entities are not recognized in the body of a document.

Notation Declarations

Notation declarations identify specific types of external binary data. This information is passed to the processing application, which may do whatever with the data. A typical notation declaration is:

```
<!NOTATION GIF87A SYSTEM "GIF">
```

Appendix B: List of XSL instructions

XSLT Tag Summary

XSLT Core	
<xsl:stylesheet>	The top-level element of an XSL stylesheet.
<xsl:template ... > ...	Establishes a pattern and replacement text.
<xsl:apply-templates ... > ...	Evaluates the children of the current node.
<xsl:text> ...	Writes the contents to the output.
<xsl:value-of .../>	Writes a calculated value output.
<xsl:for-each ...> ...	Loops over child select patterns.
<xsl:if ...> ...	Evaluates the containing content if an expression evaluates to true.
<xsl:import .../>	Imports a stylesheet.

XSLT	
<xsl:element>	Creates a new element.
<xsl:attribute>	Adds an attribute to the element.
<xsl:attribute-set>	Defines a named attribute set.
<xsl:processing-instruction>	Creates a new processing instruction.
<xsl:comment>	Creates a new comment.
<xsl:copy>	Copies the current node, but not children or

	attributes, to the output.
<xsl:copy-of .../>	Copies a sub-tree into the output.
<xsl:variable>	Assignes an XSL variable.
<xsl:call-template>	Calls a named template with the current node.
<xsl:param>	Declares an XSL parameter.
<xsl:apply-imports>	Like Java's super, calls the overridden template.
<xsl:sort>	Sorts nodes in xsl:apply-templates or xsl:for-each.
<xsl:choose ...> ...	Implements an if-elsif-else block.

Detailed description

<xsl:stylesheet>

The top-level element of an XSL stylesheet.

<xsl:template ... >

Establishes a pattern and replacement text. xsl:template registers its pattern with the XSL processing engine. When a node matches the pattern, XSL will process the contents of the template.

attribute	meaning
match	the XPath match pattern (required)
mode	string grouping templates into a special mode
name	Name for later use by xsl:call-template
priority	conflict-resolving priority, an integer

Example

In the following example, the template matches any 'box' tag. The contents of the box are placed in a centered table 80% of the current width.

XSL Snippet

```
<xsl:template match='box'>
<center><table width='80%'>
   <tr><td>
   <xsl:apply-templates/>
</table></center>
</xsl:template>
```

XML fragment

```
<box>
To be or not to be...
</box>
```

Output

```
<center>
  <table width='80%'>
    <tr><td>
      To be or not to be...
  </table>
```

</center>

<xsl:apply-templates ... >

Evaluates the children of the current node. xsl:apply-templates recursively processes the children. If a template has no xsl:apply-templates, then the children are ignored. The select pattern can restrict the children to evaluate. Stylesheets can use it to select elements and to reorder them.

Attribute	Meaning
select	An XPath select pattern selecting the nodes to evaluate next. (optional)
mode	only selects templates with the given mode

Example 1

This XSL snippet doubles the contents by calling xsl:apply-templates twice.

XSL snippet

```
<xsl:template match='double'>
<xsl:apply-templates/>
<xsl:apply-templates/>
</xsl:template>
```

XML snippet

```
<double>
Some <foo/> text.
</double>
```

Output

```
Some <foo/> text.
Some <foo/> text.
```

Example 2

The following example writes the 'a' nodes followed by the 'b' nodes and ignores everything else.

XSL snippet

```
<xsl:template match='a-b-test'>
<xsl:apply-templates select='a'/>
<xsl:apply-templates select='b'/>
</xsl:template>
```

XML snippet

```
<a-b-test>
Junk Text.
<b/>
<a>
Good text.
</a>
More Junk.
<b>
Some B text.
</b>
<a>
More Good text.
</a>
</a-b-test>
```

Output

```
<a>
Good text.
</a>
<a>
More Good text.
</a>
<b/>
<b>
Some B text.
<b>
```

<xsl:text>

Writes the contents to the output. xsl:text is useful when you need to force spacing or special text.

<xsl:value-of .../>

Writes a calculated value output. value-of is particularly useful for extracting attribute values.

Attribute	Meaning
select	An XPath expression to be printed.

Example

The following example creates a JSP tag which adds two numbers.

XSL Snippet

```
<xsl:template match='ct:sum'>
<jsp:expression>
<xsl:value-of select='@a'> + <xsl:value-of select='@b'>
</jsp:expression>
</xsl:template>
```

<xsl:for-each ...>

Loops over child select patterns. xsl:foreach gives stylesheets complete control over the actions for child nodes.

Attribute	Meaning
select	XPath select pattern

Example

```
<xsl:template match='contents'>
<ol>
<xsl:for-each select='section'>
<li><xsl:value-of select='@title'/>
</xsl:for-each>
</ol>
</xsl:template>
```

<xsl:if ...>

Evaluates the containing content if an expression evaluates to true.

Attribute	Meaning
test	XPath expression evaluating to a boolean.

<xsl:import .../>

Imports a stylesheet. xsl:import lets stylesheets borrow from each other.

Attribute	Meaning
href	Path to the imported stylesheet

<xsl:element>

Creates a new element. The name can be computed using an attribute value template.

Attribute	Meaning
name	Name of the new element.

Example

XSL Snippet

```
xsl:template match='a'>
<xsl:element name='b{@id}'>
<c/>
</xsl:element>
</xsl:template>
```

Ouput

```
<b3><c/></b3>
```

<xsl:attribute>

Adds an attribute to the element. The name can be computed using an attribute value template.

Attribute	Meaning
name	Name of the new attribute.

Example

XSL Snippet

```
<xsl:template match='a'>
<c>
<xsl:attribute name='b{@id}'>
<xsl:value-of select='c{@id}'/>
</xsl:attribute>
</c>
</xsl:template>
```

Output

```
<c b3='c3'/>
```

<xsl:attribute-set>

Defines a named attribute set. The attributes in the set are defined by xsl:attribute elements.

Attribute	Meaning
name	Name of the attribute set.

Example

XSL Snippet

```
<xsl:attribute-set name='font'>
<xsl:attribute name='font-size'>12pt</xsl:attribute>
<xsl:attribute name='font-weight'>bold</xsl:attribute>
</xsl:attribute-set>

<xsl:template match='a'>
<c xsl:use-attribute-sets='font'/>
</xsl:template>
```

Output

```
<c font-size='12pt' font-weight='bold'/>
```

<xsl:processing-instruction>

Creates a new processing instruction.

Attribute	Meaning
name	Processing instruction name.

Example

XSL Snippet

```
<xsl:template match='a'>
<xsl:processing-instruction name='foo'>
<xsl:text>Text for the PI</xsl:text>
</xsl:processing-instruction/>
</xsl:template>
```

Output

```
<?foo Text for the PI?>
```

<xsl:comment>

Creates a new comment. The contents of the xsl:comment element become the contents of the comment

Example

XSL Snippet

```
<xsl:template match='a'>
<xsl:comment>
<xsl:text>Text for the comment</xsl:text>
</xsl:processing-instruction/>
</xsl:template>
```

Output

```
<!--Text for the comment-->
```

<xsl:copy>

Copies the current node, but not children or attributes, to the output. To copy an element, a stylesheet must copy the attributes as well.

Example

The following example is the identity stylesheet. It copies input to the output including the attributes.

XSL Snippet

```
xsl:template match='@*|node()'>
<xsl:copy>
<xsl:apply-templates select='@*|node()'/>
</xsl:copy>
</xsl:template>
```

<xsl:copy-of .../>

Copies a sub-tree into the output. copy-of resembles value-of. value-of always converts the value to a string. copy-of will copy subtrees.

Attribute	Meaning
select	An XPath expression to be copied.

<xsl:variable>

Assignes an XSL variable. Variables can be retrieved using the XPath variable syntax.

Attribute	Meaning
name	variable name
select	variable value

Example

XSL Snippet

```
<xsl:variable name='foo' select='1+1'/>
<xsl:template match='a'>
<xsl:value-of select='$foo"/>
</xsl:template>
```

Output

2

<xsl:call-template>

Calls a named template with the current node. xsl:call-template lets stylesheets reuse common code, like functions. It works like xsl:apply-templates select='.' except that it calls based on a template name.

Attribute	Meaning
name	template name to call

mode	template mode

<xsl:param>

Declares an XSL parameter. xsl:param's select parameter as a default. If the variable has been assigned, it uses the old value.

Attribute	Meaning
name	variable name
select	variable value

Example

XSL Snippet

```
<xsl:template name='fun'>
<xsl:param name='foo' select='15'/>
<xsl:value-of select='$foo'/>
</xsl:template>

<xsl:template match='a'>
<xsl:call-template name='foo'>
<xsl:with-param name='foo' select='1+2'/>
</xsl:call-template>
</xsl:template>
```

Output

3

<xsl:apply-imports>

Like Java's super, calls the overridden template.

<xsl:sort>

Sorts nodes in xsl:apply-templates or xsl:for-each.

Attrib ute	Meaning
select	value to sort on (default = '.')
order	ascending or descending (default = ascending)
data-type	text or number (default = text)

<xsl:choose ...>

Implements an if-elsif-else block. The xsl:when statements are tested in order. The first matching one is executed. If none match, the xsl:otherwise block is executed.

Attribu te	Meaning
test	XPath expression evaluating to a boolean.

Example

XSL Snippet

```
<xsl:template match='a'>
<xsl:choose>
<xsl:when test='@color="red"'>
<xsl:text>stop</xsl:text>
</xsl:when>
<xsl:when test='@color="green"'>
<xsl:text>go</xsl:text>
</xsl:when>
<xsl:otherwise>
<xsl:text>yield</xsl:text>
</xsl:otherwise>
</xsl:choose>
</xsl:template>
```

Appendix C: How to setup IXML

The general steps to setup IXML on your system are as follows:

- Install Java
- Install Cocoon and Jserv
- Download and uncompress ixmlCore.zip
- Compile the IXML class library
- Add the IXML class library to Cocoon path
- Put the IXML Java processing style-sheet (ixml-xsp.xsl) and IXML XSL style-sheet (ixml-html.xsl)in a place accessible by other style-sheets.

This setup will walk through the setup for installing IXML and getting a sample application to run on system with the following configuration:

- Windows 98
- the Apache Web Server version 1.3.9
- Jserv Java Servlet engine

Install Java

If Java is not setup on your system, get it installed. There are lots of instructions at the java.sun.com site.

Install Cocoon and JServ

Go to xml.apache.org and download and install Cocoon and JServ. There are lots of instructions and sample applications on the Cocoon page to get the system up and running.

Note: The version of Cocoon that this is being used here is version 1.7.2.

To make things easier, I have set my apache document root to be E:\apache.

You can change the document root in the Apache httpd.conf file.

As a result the Web server will serve pages relative to that directory. For example if there was page whose path was E:\apache\index.html, then the URL of that page would be http://localhost:8080/index.html

Also I have configured Jserv to have access to the class files in the following directory: E:\apache\classes

You can make the configuration changes in jserv.conf with the addition of this line:

wrapper.classpath=E:\apache\classes

Install the core IXML files

Download and uncompress ixmlCore.zip.

This archive contains two items:

- xsp/ixml/*.java, which contains all the Java source files for the IXML class library
- ixml/StyleSheet, which contains the two IXML style-sheets ixml-xsp.xsl and ixml-html.xsl

Move the content of the directory xsp into a place on your hard-drive. I will move it to E:\apache\classes to make it accessible by Jserv.

Move the two style-sheets to a place on your hard-drive. I will move them to E:\apache\ixml.

Compile the IXML Java class library

Open a Dos prompt and go to the directory E:\apache\classes and type:

javac xsp/ixml/*.java

This will compile all the java files in the xsp.ixml package.

Download and uncompress sample.zip

This archive contains some IXML samples.

Each folder contains an example IXML application. They are as follows:

greeting: Demonstrates a simple greeting

itree: Demonstrates a simple iTree

tanner: Demonstrates a simple multi-language page

intra: Demonstrates a simple version of the IntraSpeak program

The sample greeting example

The following will guide you through getting one of the example application up and running. In the content of the uncompressed sample.zip, there is a folder labeled "greeting" that contains the files for the greeting sample which are ixml-xsp.xsl and ixml-html.xsl. Move the files to a place on your hard-drive. I will move them to E:\apache\greeting

The sample program should now run as it is. The only thing that might be needed to be modified is the import statements to the two IXML style-sheets. Open up page-xsp.xsl, and modify the import statement to the path of ixml-xsp.xsl.

```
<xsl:import href="http://localhost:8080/ixml/ixml-xsp.xsl"></xsl:import>
```

Also do the same for page-html.xsl:

```
<xsl:import href="http://localhost:8080/ixml/ixml-html.xsl"></xsl:import>
```

Start the Apache Web Server

Start the Apache Web Server if it has not yet been started. Go to the DOS promt and type: apache

Then access the greeting example with your Web browser:

Type in this URL:

http://localhost:8080/greeting/clean-page.xsl

You should see the greeting example.

Vita

Surname: Vu Given Names: Phong Thanh

Place of Birth: An Giang, Thot Not, Viet Nam

Education Institutions Attended:

College of the Rockies	1994
University of Victoria	1994-1998

Degrees Awarded:

B. Sc. with Co-op	University of Victoria	1998

Honours and Awards:

UVic Entrance Scholarship	1994
Canada Scholarship	1994-1995
Paul and Helen Trussell Science and Technology Scholarship	1995-1999

Publications:

Partial Copyright License

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

Title of Thesis:

Intensional XML

Author

Phong Vu

Date April 20/2001