The Extensible Templating Language: An XML-based Restricted Markup-Generating Language

Greg J. Badros
InfoSpace, Inc., 601 108th Ave. NE, Suite 1200, Bellevue, WA 98004, USA
greg.badros@infospace.com

ABSTRACT

Popular web templating languages embed general-purpose programming languages. The Extensible Templating Language was born out of questioning the fundamental assumption that the front-end markup-generating engine of a multi-tier web application requires all the power and expressiveness implied by that design. ETL restricts the set of language features to a useful subset that provide the necessary functionality without compromising the simplicity and understandability of templates. By forcibly limiting what can be done by templates, we ensure a better separation of presentation details from business logic. Furthermore, ETL improves the analyzability of source templates by using an XML-based representation where markup is intermingled with XML elements corresponding to programming constructs. This approach reduces the possibility of generating improper markup and facilitates tool-building including semantically-aware editors and debuggers. ETL runs inside of the Extensible Templating Language Server which is currently employed by InfoSpace to serve millions of requests per day using over sixty thousand ETL templates.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.3 [**Software**]: Programming Languages; C.2 [**Computer Systems Organization**]: Networks

Keywords

Restricted domain-specific programming language, static analysis, web server, templates, XML, XSLT, markup, WML, HTML.

1. INTRODUCTION

The loosely-coupled client/server model implied by the current breed of applications deployed via the World Wide Web presents substantial new complexities for application developers. A variety of new programming languages and paradigms attempt to address these novel problems. One important characteristic of this new world of software development is the need to generate HTML, WML, and other markup languages to describe client-side presentation and behaviour.

The original server-side dynamic markup-generation capabilities of the Web were defined by the Common Gateway Interface [23]. For each request of a CGI, the web server maps HTTP request details into environment variables, command-line arguments, and the standard file descriptors. In particular, the standard output written by that process was sent by the Web server as the response to the client browser (instead of simply responding with an unchanging file from disk). Over time, various techniques arose to reduce the cost of each request: extension mechanisms such as the Apache module system [2] and scripting languages hosted by the Web server have increased performance by eliminating process creation overhead.

WWW2003, May 20–24, 2003, Budapest, Hungary.
ACM xxx.

```
1  <? $books = GetBooks(...); ?>
2  <html>
3   <table>
4    <? for ($i=0; $i < count($books); ++$i) { ?>
5     <tr>
6      <td><? print($books[$i][author]); ?> </td>
7      <td><? print($books[$i][title]); ?> </td>
8     </tr>
9    <? } ?>
10  </table>
11 </htm>
```

Figure 1: Cleanly-written PHP templates require only the simplest programming language constructs. Unfortunately, over the evolution of the presentation layer, PHP developers sometimes engage in far broader interactions because the power of the language allows them to do so.

Today, server-side markup generation is dominated by expressive and flexible general-purpose programming languages such as Java [3], Perl [41], and PHP [11]. Ironically, the languages are then mostly used in fairly restricted ways, often being tied to a templating syntax (e.g., JSP [28] for Java or HTML::Template [39] for Perl). For example, consider the PHP template in Fig. 1. The only constructs required by the template are a) some means of populating a data model from the back-end business logic (line 1); b) data-driven iteration (line 4); and c) simple expression evaluation to access parts of the data model (lines 6 and 7). The extra power provided by typical templating languages not only goes unused, but it also complicates certain important analyses of the code. For example, the obviously critical property that a template's execution terminates is undecidable for these general purpose languages.

For most web application development frameworks, the restrictions desired of the front-end templating layer are imposed only by process and convention. Template authors are asked to limit themselves to a subset of the available features of a language as a means of facilitating scalable, secure, well-behaved systems. Importantly, however, there is nothing inherent that prevents presentation code from, for example, making individual database connections (which would impact performance) or maintaining undesirable server-side state (which would impose additional requirements on the load-balancing mechanism and reduce scalability). Languages such as PHP actually encourage this undesirable mixing of front-end pre-

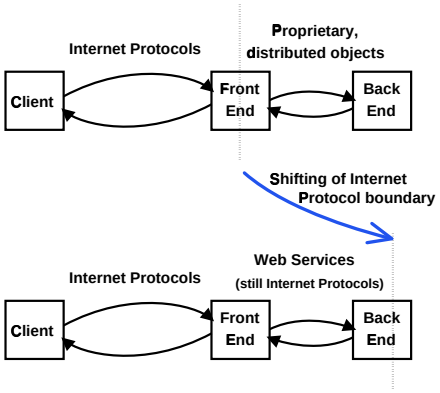

Figure 2: Web services enable the front-end templating tier to communicate with back-end applications via standard Internet Protocols such as XML and SOAP; they no longer need to be aware of distributed object protocols or data models.

sentation with back-end business logic [15]. That fact further blurs the common organizational line between front-end presentation developers and back-end application engineers.

Another substantial problem with existing templating technologies is that they often involve the lexical mixing of two separate programming paradigms. JSP, for example, is implemented in terms of a pre-processing rewrite of the template into a Java servlet [14]. Pre-processing approaches are convenient for programmer expressiveness but have a huge hidden cost in complicating software engineering analyses and tools that could otherwise help understand, maintain, and evolve the complex systems [5, 16][36, p. 424]. The precise analysis of an arbitrary JSP template necessarily requires full knowledge of both the rewrite rules and the semantics of Java.

1.1 Restricting the language

Almost all popular markup templating languages integrate with a general-purpose programming language. This approach offers some advantages. For example, undoubtedly the language is sufficiently powerful and expressive for the needs of markup generation. Additionally, numerous language-level tools such as compilers and debuggers, are already available for the templating language to leverage. For platform providers that have a large investment already made in a general-purpose language, it is natural to extend that investment to a templating language.[1]

Despite the broad assumption that a markup language requires the use of a general-purpose programming language, an obvious alternative exists: use a domain-specific language specifically tailored to the needs of generating markup. Such an approach can mitigate several of the problems mentioned previously. In particular, using a restricted programming language at the front-end to generate markup can enforce better separation of front-end presentation from back-end application logic and may simplify the templates such that they are easier to write, read, and evolve.

If we limit the functionality of a markup templating language, one capability that remains important is accessing back-end applications. The architecture of the classic three-tier application appears at the top of Fig. 2. Historically, the communication between the front-end and back-end servers uses an object-based data model and involves remote procedure call mechanisms such as DCOM or Corba. As the web service architecture depicted in the bottom of Fig. 2 increases in popularity, the need for the front end templating language to reason about language-level objects is reduced. Instead, the XML data model implied by web services can be the primary communication format and the front end need only use Internet standards such as HTTP and XML in interacting with other applications. This observation suggests that markup languages need only concern themselves with a single data model and can thus be made simpler to learn and use.

[1] XSLT can be used in a templating style [12, 2.3], but popular use of XSLT also leverages scripting extensions to deal with the far too severe restrictions of the language.

1.2 Analysis of markup templates

Given the wide variety of browsers, crawlers, and other user agents, it is important that the markup created by the presentation layer conforms to appropriate Internet standards. For example, WML markup needs to first be well-formed XML and second be valid with respect to the appropriate schema or DTD [17]. Although HTML browsers are more forgiving of variations in syntax, best-practices still dictate that the markup returned comply with the HTML specification [30]. Popular templating languages including PHP, JSP, ASP, and many others process arbitrary text and are ignorant of the rules of the markup being generated: the mistaken close tag on line 11 of Fig. 1 goes unnoticed by PHP, yet will result in a user-perceived complete failure under an XHTML browser.

Tools such as HTMLTidy [31] are available to check the responses from servers for conformance, but often testing involves simply using various browsers to exercise the application while looking for bugs. Both of these approaches require exhaustively visiting every page of interest. For quality assurance to scale, we require a way to statically analyze the source templates themselves to gain confidence that they will, in fact, generate proper markup. If static analysis of templates can rule out the possibility of certain kinds of errors, testing time and costs can be dramatically reduced.

Another important requirement for software development systems is to support ad-hoc queries of the source code. For small and simple dynamic web applications, the automated analysis of templates is often unnecessary: a web developer or two understands all of the code. Over time, however, these simple applications grow in size and complexity such that tools analyzing the templates become an essential part of maintaining and evolving the system. For example, a site may wish to add an extra input field to all of its registration forms. If a generic form is not already factored out into a common location, the first step in approaching the task is finding all the forms. Or suppose a new handheld device has a limitation in its handling of cookies: we need a means of narrowing the scope for careful review to only places where that cookie is manipulated.

Typically, web developers approach to the above scenarios with an arsenal of imprecise lexical tools. While these tools may satisfy some simple queries, when the desired query has additional structure, lexical approaches fall short. Suppose we wish to find dead code to simplify our application. No regular expression will let us identify all unreachable templates to satisfy that query. To analyze more deeply, we need to uncover the semantic structure of the templating language. Typically, that requires a language-specific parser and much greater effort in building a valuable tool or an extensible integrated development environment [35, 27].

1.3 XML supports analyses and tools

An increasingly popular way to perform structured queries of programming language source code is to use standard XML tools such as XSLT [12] and XQuery [7] to operate over a complementary XML-based representation [4, 33, 18, 29]. With a carefully-chosen XML representation, valuable semantics of the code are immediately available to XPath expressions, thus enabling a broad

class of source code tools. However, for a general-purpose programming language, XML is unwieldy for use as the primary representation. Instead, the conventional grammar-based language is edited by humans and is only converted into XML for the tools to use.

This research applies the benefits of an XML representation of source code to the domain of restricted markup templating languages. Because web developers are already writing markup directly, it is natural for them to express the limited logic of their dynamic templates directly in markup as well. Importantly, this lets us use a single representation for both developers and tools. For example, we can use schema-aware XML editors to easily provide syntax coloring and completion capabilities.

In this paper, I introduce the Extensible Templating Language. From one perspective, ETL is an XML-based templating language that embeds programming constructs directly in the XML representation, intermingled with literal target-language markup. Alternatively, we can view ETL as an imperative domain-specific programming language that uses XML as its surface syntax and simplifies the writing of programs that generate markup languages.

ETL leverages well-formedness and local-validity checks of the source template to support those same properties in the generated response markup. By limiting the programming-language constructs to include only the features that are appropriate at the front-end of a scalable web application architecture, we ease numerous analyses and encourage proper separation of the presentation details from the backend application. Finally, because of its use of XML as a representation, ETL simplifies ad-hoc software engineering analyses to better support evolution and maintenance of large dynamic web applications. We have built an ETL runtime inside of a production-quality web server, the Extensible Templating Language Server. InfoSpace runs ETLS to serve millions of requests per day using over sixty thousand ETL templates.

1.4 Outline of paper

The rest of this paper is organized as follows. Section 2 explains the Extensible Templating Language and its benefits. Section 3 details our implementation and describes our experiences in using ETL in production systems over the last year. Section 4 describes and contrasts related work and Section 5 concludes.

2. EXTENSIBLE TEMPLATING LANGUAGE

The Extensible Templating Language is an imperative markup templating language that allows web developers to intermingle literal XML markup with XML-based programming language constructs. Our primary design goals for ETL were to:[2]

1. Allow easy analyses of source templates to permit development of supporting tools;
2. support only the essential programming language constructs required by a templating layer;
3. focus on an imperative programming model; and
4. utilize an XML data model pervasively to integrate seamlessly with Internet and Web standards.

An ETL template must be well-formed XML that is valid with respect to the language's XML Schema Definitions [38, 6]. By enforcing those requirements, we catch a large class of syntactic and semantic programming errors very early in the development process without need for the ETL compiler at all. Those schemas are also used by our development environment (see Fig. 3) to support syntax coloring and completion of element and attribute names. Additionally, because of how easy it is to analyze XML trees, it is straightforward to build numerous ad-hoc software engineering tools that are language-aware. For example, it was trivial to build a call-graph extractor (see Fig. 4).

ETL intentionally restricts the set of programming constructs to limit the amount of logic performed in the front-end of a multi-tier web-delivered application. As a result, we achieve better separation of presentation and application, thus permitting their independent evolution. ETL is already used by dozens of production applications, demonstrating that the set of available constructs is sufficient for real-world use.

An example ETL template appears in Fig. 5 and is analogous to the PHP example from Fig. 1. The source code is simply an XML document with root element `bl:template`[3] The template contains other elements in the `bl:*` namespace which we refer to as ETL

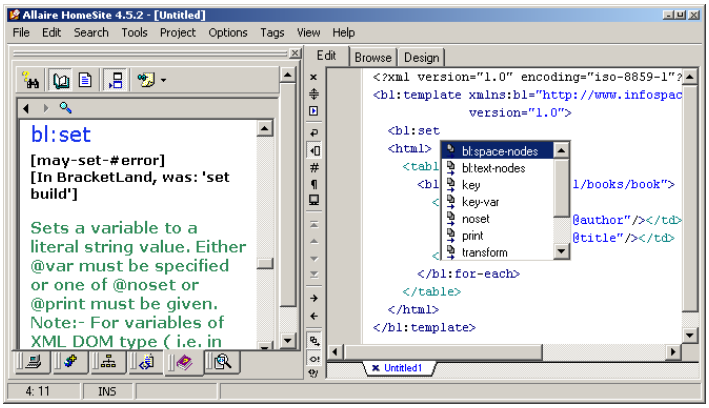

Figure 3: This integrated development environment based on Homesite was easy to build using custom tag information generated automatically from the XSD schema for ETL. The environment supports automatic completion of names of elements and their valid attributes. Help information extracted from the server codebase is also readily available in the IDE.

```
1  <?xml version="1.0"?>
2  <bl:template xmlns:bl="http://www....">
3   <bl:set var="#xml/books">...</bl:set>
4   <html>
5    <table>
6     <bl:for-each var="#xml/books/book">
7      <tr>
8       <td><bl:get var="@author"/></td>
9       <td><bl:get var="@title"/></td>
10     </tr>
11    </bl:for-each>
12   </table>
13  </html>
14 </bl:template>
```

Figure 5: The ETL template corresponding to Fig. 1's PHP code is required to be well-formed XML. Thus, the developer avoids the mistake from line 11 of the PHP example.

[2] A fifth design goal was to enable automatic translation of our legacy language into ETL. This paper focuses only on the essential language, and Section 3 mentions our migration to ETL a bit more.

[3] The "bl" stands for "base language." For brevity, we use namespace prefixes consistently to refer to elements in distinguished namespace URIs.

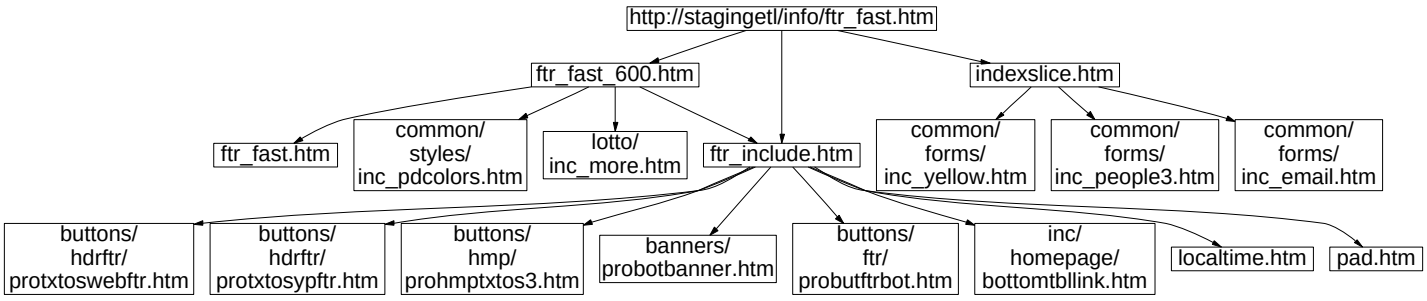

Figure 4: This call-graph of templates possibly used in generating an HTML page footer was extracted using server-side XPath expressions and a simple Perl script.

primitives (see Section 2.1). Additionally, the template can contain markup from arbitrary other namespaces (or unqualified elements) which we call *literal result markup* (see Section 2.5).

ETL templates are processed within the context of an HTTP request. The path specified in the URL of the request selects a template where processing starts. The primary goal of executing a template is to generate an HTTP response including the markup document along with the various HTTP headers such as the content-type, cookie settings, and other meta-data. Generally, an ETL template is similar to a method in an object-oriented programming language: it can invoke other templates as subroutines and can also perform subsidiary HTTP requests to other servers (see Section 2.4).

2.1 Primitives, attributes, and slots

An ETL primitive is an element in the `bl:*` namespace that has a specific well-defined behaviour inside of an ETL template. Primitives either output text to the current destination stream (initially the response document) or perform some side-effect (or both). For example the `bl:set` primitive assigns a variable a value without outputting any text, while the `bl:http` primitive outputs either `http` or `https` based on whether the current request arrived over an ordinary or secure connection.

The behaviour of a primitive is controlled by its attributes. Some primitives, such as `http` mentioned above, are always empty—they never are allowed to contain child elements. Other primitives are allowed to be non-empty, using the markup generated by their contained elements as an extra implicit argument. For example, in:

```
<bl:if var="showcopyright"> (C) 2002 </bl:if>
```

the contents of the `bl:if` are output if and only if the variable's value is non-empty.

Many primitives derive their arguments not from individual attributes but from pairs of attributes that are together called a *slot*. For example, the `bl:cr` directive outputs linefeeds, and the number of linefeeds it generates is determined by a slot called `count`. That slot is specified using a pair of attributes: `count` and `count-var`. The attributes are mutually-exclusive: it is a statically-checked error to specify both on the same `bl:cr` element. The `count` attribute has type `xsd:integer` and is used to provide a literal integer argument to the primitive. In contrast, the `count-var` attribute has type `bl:identifierType` (a restriction of `xsd:string`) and names a variable to evaluate at runtime. In both cases, the resulting value is used as the number of linefeeds to generate, but for `@count` that number is set statically while for `@count-var` it is determined dynamically (at run-time).

Using a pair of attributes to represent a logical field is a novel mechanism to improve type-safety and ease analysis. The common alternative is using a small domain-specific language inside the value of an attribute. For example, we could have used `count="2"` or `count="$num"` to represent the literal number 2 or the desire to use the current value of the variable "num", respectively. (XSLT's `value-of` element uses this basic approach.) Unfortunately, when using a single attribute, the type for the attribute's value must be broadened and is hence less precise. That lack of precision can allow more developer errors to go uncaught.

Primitives can, of course, have numerous arguments, some of which are controlled by slots and others by individual attributes. When an argument is allowed to be set only via an attribute instead of a slot, that parameter of the directive cannot be influenced at runtime. Often this restriction is imposed to permit optimizations or to improve our ability to support static analyses. For example, `bl:get` writes out the value of a variable and has a `@transform` attribute that supports various encodings or decodings of the value (e.g., `url-encode` or `xml-decode`—see Section 2.3). We disallow the use of `@transform-var` because we want to always know from static inspection what transformations may occur, thus facilitating some analyses and optimizations.

2.2 Variables and values

As with all imperative programming languages, ETL has the notion of variables that store values. Variables can be declared inside a single `bl:header` element (which is required to be the first element in a template) and are assigned values using the `bl:set` primitive. The target variable is named via the `@var` attribute and the value is given by `@value`, `@value-var`, or by executing the children of the `bl:set` element. For example:

```
<bl:set var="baseuri"><bl:http/>://<bl:get
 var="hostname"/>/</bl:set>
```

might set the variable named "baseuri" to the value "http://www.infospace.com/". While executing its contained elements, `bl:set` redirects the default output stream to be used to construct a value to assign to the variable; the output stream is restored after the child elements are processed (e.g., allowing nesting of `bl:set` primitives).[4]

After assignment, values are retrieved by referencing them by name in a slot or by copying them directly to the output stream using `bl:get`. Undeclared variables have dynamic scope and live until the end of processing the current request, while declared variables have static scope and may only be referenced in the same template.

[4]Several other primitives follow this same pattern. For example `bl:set-mime-type` evaluates its contained elements to generate a value to be used for the MIME content-type of the response.

ETL has the notion of special reserved variables that may have side-effects and are used internally for conveying request parameters or other system-level details. These reserved variables always start with the octothorpe ("#") character and may be read-only (e.g., `#browsertype`) or may alter the behaviour of primitives based on the value they are assigned. Unlike ordinary variables, there is no guarantee that the value retrieved from a reserved variable equals what was last stored.

Various other data is accessible to ETL templates via *buckets*. Buckets can be thought of as top-level elements in a virtual (lazily-evaluated) XML data model that is implicitly available to ETL templates. They replace field references and accessor methods of distinguished objects in languages such as Java. For example, where a JSP developer might write:

```
<%= request.getHeader("User-Agent") %>
```

the ETL developer accesses the `#http` bucket instead:

```
<bl:get var="#http/User-Agent"/>
```

Numerous buckets exist in the XML data model providing ways to access URL parameters, HTTP headers, user settings, configuration data for applications and brands, and cookies.

One additional bucket, `#xml`, is unique in that the values its variables bind to are XML trees instead of just strings. When the variable is evaluated in a context that requires a string, the subtree is serialized in its XML syntax. Additionally, a trailing XPath expression is allowed after the variable name when referencing the `#xml` bucket. This feature allows easy querying of XML values and integrates XPath cleanly into ETL.

2.3 Transformers & formatters

HTTP and XML standards have various different encoding formats to support arbitrary data over channels that allow limited representations. For example, when passing a GET parameter to an HTTP request, the value is inserted into the URL using an URL-encoding format that involves (among other things) converting each `SPACE` character to a `+` character. To support such encoding formats in a general way, ETL defines a set of "transformers."

A *transformer* is a streaming converter from one byte sequence to another. For example, the `url-encode` transformer converts "hello world" into "hello+world". Many transformers have an inverse that is also supported: the `url-decode` transformer converts "hello+world" into "hello world". Transformers can be used by ETL code whenever a variable is being set (via `bl:set`) or accessed (via `bl:get`) using the `transform` attribute which specifies an ordered whitespace-separated list of transformers to apply. For example:

```
<bl:get var="a" transform="trim urlencode"/>
```

results in writing out the value of the variable `a` after first eliminating leading and trailing whitespace and then URL-encoding the resulting trimmed string. The order of transformations does matter. Using

```
<bl:get var="a" transform="urlencode trim"/>
```

results in a different output string. If `a` contains " `Hello World` ", the former example produces "`Hello+World`" while the latter variant generates "`+Hello+World+`" because the trimming occurs on the URL-encoded string which already has spaces replaced by "+" characters.

Formatters are similar to transformers, but their input is a streaming virtual XML document (similar to SAX, the Simple API for XML [21]) and their output is a text stream. Numerous primitives that generate XML as their output have a `formatter` slot that names the formatter to use. Several built-in formatters exist or XSLT stylesheets can be used. When using XSLT as a formatter, the events first construct a DOM that is then transformed via the named stylesheet, letting the stylesheet generate the formatter's response string.

2.4 Control flow

There are four primary means of control flow in ETL: template selection, conditionals and loops, exceptions, and remote invocations.

2.4.1 Template selection

Requests made of the ETL server always have an associated abstract *site* that can be explicitly specified as part of the URL, or can be implicit based on the `Hostname` header of the HTTP request (e.g., when multiple hostnames refer to the same server). That site name is used to select which template is invoked for the initial processing of a request and whenever the `bl:call` primitive is used to execute another template as a subroutine. Template selection is analogous to method dispatch in object-oriented programming languages. Importantly, ETL only permits bounded recursion, thus ensuring termination.

For example, an `index.htm` page may `bl:call` a `header.htm` template to output a banner for the top of a page, then do some work to generate the meat of the page, and finally invoke `footer.htm` to generate another banner for the bottom. The default `header.htm` at the top-level of the hierarchy may provide some basic functionality, while an overriding `header.htm` implementation could specialize the banner.

2.4.2 Conditionals and loops

ETL supports both `bl:choose`/`bl:when`/`bl:otherwise` and `bl:if` constructs. They behave much as they do in XSLT. The primary difference is that ETL does not permit the generality of a domain specific expression language, and instead uses two slots to represent the guard conditions. For example:

```
<bl:if var="pagenum" less-than-var="max-page">
   <a href="...">Next</a>
</bl:if>
```

Although this is slightly more verbose than a domain-specific language in a `test` attribute, it does make it easier to use XPath to find occurrences of specific operations being used in conditional guards. Additionally, it allows us to factor out the left-hand-side value of a comparison guard into the `bl:choose` parent element to approximate the convenience of a `switch` statement:

```
<bl:choose var="pagenum">
  <bl:when less-than="1">..</bl:when>
  <bl:when equals="1">..</bl:when>
  <bl:when greater-than-var="max-page">..</bl:when>
  <bl:otherwise>..</bl:otherwise>
</bl:choose>
```

where each of the `bl:when` clauses compare the same variable (without redundant specification of its name).

For looping, only data-driven loops are supported. Because only finite data structures exist, this ensures that loops terminate. A `bl:for-each` primitive iterates over nodes in a node-set of an XML variable or splits strings at a specific delimiter and operates on the substrings.

2.4.3 *Exceptions*

Some primitives throw run-time exceptions. For example, assigning a string that contains ill-formed XML to an XML-typed variable results in a parse exception. To recover from those exceptions, `bl:try` blocks that include `bl:catch` and `bl:finally` children blocks are supported. Additionally, templates (especially subroutines) can use `bl:throw` to generate their own exceptions to be handled by higher stack frames.

2.4.4 *Remote invocation*

Two mechanisms are provided to allow templates to interact with back-end applications and other hosts. Simple HTTP requests, both GETs and POSTs, are supported using a `bl:http-include` primitive that specifies the URL along with POST data, as required. Alternatively, a `bl:ws-call` primitive invokes a web service given its end-point and the arguments to the web service.[5] In both cases, the output of the primitive is the document returned by the server.

2.5 Literal markup

While typical templating languages are text-based, ETL is based instead on the higher level abstraction of markup. It is this distinction that enables ETL to favor generating well-formed output documents. For example, an ASP template might read:

```
<a href="<%= target %>"><%= link_name %></a>
```

which has semantics that are strictly textual—it concerns itself with outputting arbitrary character sequences. Unfortunately, this snippet:

```
<a hre=<%= target %>><%= link_name %></b>
```

is essentially the same as the preceding code, despite the fact that this second example contains several errors.

As a markup-based templating language, ETL favors writing templates that will generate well-formed and valid markup. For example, to output a literal XML fragment such as "`<b>Good</b>`", that fragment can be written directly in the ETL template. However, to generate the ill-formed XML fragment "`<a>Bad</b>`", you must fall back on outputting individual characters and write:

```
&lt;a&gt;Bad&lt;/b&gt;
```

Note that ETL does not forbid outputting ill-formed markup, it only encourages proper markup by making that common case far more natural to write.

Of course, the markup we need to output often includes dynamically computed attributes and content (as with our initial ASP example). To output an attribute with its value derived from a variable, you can write:

```
<a href="?target">...</a>
```

where the `?target` syntax means to use the value of variable "target" rather than the literal characters `?target`[6] Alternatively, one can write:

```
<a><at:href><bl:get var="target"></at:href>...</a>
```

Here, the `href` attribute of the `a` element is given the value computed by evaluating the elements contained by `at:href`. The special namespace `at` is used to make this format more concise than the long-winded `xsl:attribute` variant that XSLT uses. The local name of the `at:*` element is used as the name of the attribute, and the contents can be arbitrary ETL code (e.g., including conditionals and subroutines). No matter which form is used to specify dynamic attribute values, the language ensures that proper quotations and encoding are used. Furthermore, by using the DTD or XSD of the output document's desired markup language, we can confirm that attribute names (and sometimes their values) are correct.

Another common need is to conditionally include a tag. For example, consider this ill-formed template fragment:

```
<bl:if var="#pro/usebold">
  <b>
</bl:if>
    Possibly bolded text
<bl:if var="#pro/usebold">
  </b>
</bl:if>
```

Although the intent is reasonable, XML requires proper nesting of elements and thus disallows this code. Instead, ETL supports that behaviour using a special `bl:if-var` attribute on an arbitrary literal result element:

```
<b bl:if-var="#pro/wantbold">
  Possibly bolded text
</b>
```

where the output of *both* the open and close `b` tags is controlled by the test of the single variable (and the contents of the `b` element are always executed). This technique also lets us factor out the guard, thus eliminating the possibility of a mismatch between the test used for the start tag and the one used for the end tag.

Comments are another subtle aspect of markup templating languages. They fall into two categories: 1) traditional comments intended for the developers never to be part of the generated output; and 2) target-language markup comments that are intended to be sent over the wire as part of the response document. Similar to XSLT, the former are written as simple XML comments using the `<!-- ... -->` syntax, while comments intended for the response are constructed using the `bl:comment` primitive.

Finally, to support internationalization of templates, developers can choose to use the `bl:tt` (token translation) primitive in all places where literal text or markup would appear. For example, instead of writing `Hi <bl:get var="name"/>`, a properly internationalized template might read:

```
<bl:tt id="greet1">Hi</bl:tt><bl:get var="name"/>
```

Each template has a corresponding dictionary of token-translation definitions for the various locales that have been defined. Based on the request's locale selection (either from a stylized URL or the `Accepts-Language` header), the appropriate dictionary is used to replace each `bl:tt` element with the XML fragment to which the given identifier maps.

2.6 Controlling whitespace

Whitespace characters in a source template have mixed uses.[7] In some cases, the developer wishes to improve the readability of

[5] Web service invocation is a part of an upcoming version of ETLS.

[6] A leading backslash is ignored and suppresses the special meaning of the question mark in case you really do want the attribute to be set to those seven characters.

[7] Whitespace that is not a part of the XML Infoset [13], such as whitespace between attributes inside element tags, is discarded by the XML processor before the application sees it and is therefore not controlled by the rules described in this section.

```
<foo>
1 <bar>
2   Hello3 World
4   <baz>5 </baz>
6 </bar>7
</foo>

1 6 7  whitespace-only
5      whitespace-only content
2 4    leading/trailing whitespace
3      internal whitespace
```

Figure 6: XML has various kinds of whitespace nodes. ETL permits careful control over which nodes are considered significant.

the program by using indentation or blank lines. In other cases, the whitespace is intended to be copied into the output document or a variable's value. The `xml:space` attribute provided by the XML standard [10, 2.10] is not sufficiently expressive to support the diverse needs of the template author. Careful control over which whitespace is deemed meaningful is critical to generating the right markup. It also can improve performance and reduce memory and bandwidth consumption by eliminating unnecessary extra characters from going over the wire.

ETL permits fine-grained control over the whitespace that will be generated via two orthogonal special attributes that are allowed on every XML element in source ETL documents: `@bl:space-nodes` and `@bl:text-nodes`. These attributes are inherited by contained (i.e., children) elements and their contents unless overridden. They control the various classes of whitespace illustrated in Fig. 6: the `@bl:space-nodes` attribute controls whitespace-only (and whitespace-only content) nodes (1, 5, 6, and 7 in the figure), while `@bl:text-nodes` controls leading/internal/trailing whitespace of text nodes (2, 3, and 4 in the figure). Importantly, these whitespace rules are statically-scoped: a called template is not impacted by a calling template's attribute annotations.

The allowed settings for `@bl:space-nodes` are: a) **`preserve`**, to leave the whitespace unchanged; b) **`compact`**, to replace sequences of 1 or more whitespace characters with a single space (0x20); c) **`strip`**, to eliminate the text node entirely; or d) **`normalize`** to strip whitespace nodes that contain only newline (0x0A) characters entirely and replace sequences of 1 or more whitespace characters that include a space or a tab with a single space (0x20). For `@bl:text-nodes`, the above settings are extended by **`trim`** which trims all leading and trailing whitespace.

2.7 Custom tags

ETL allows custom tags to define new abstractions only in terms of built-in primitives. The mechanism ETL provides is based on abstract syntax tree (AST) rewriting similar to Scheme's hygienic macros and other syntactic macro systems [32, 1]. Given that the AST of an ETL template is simply the XML source code itself, we simply use XSLT to describe the rewrite rules (see Fig. 7). Those rules are then applied to the source template thus reducing that code to contain only the core language. That reduced template is then byte-compiled for execution. This approach allows a great deal of flexibility in new abstractions without compromising our tight control over the limited behaviour we permit the presentation layer.

3. IMPLEMENTATION & EXPERIENCE

A byte-code compiler and runtime for the Extensible Templating Language is an integral part of InfoSpace's ETL Server 2.0.

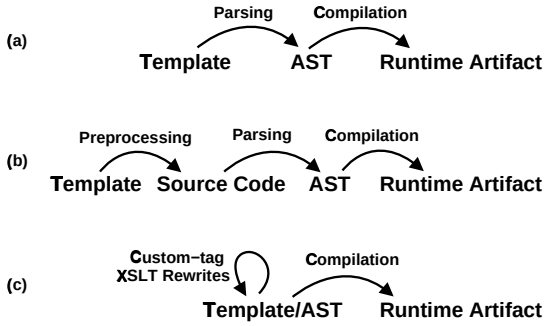

Figure 7: The pre-execution processing of various templating languages can be fairly involved. The pipeline (a) illustrates a mod_perl-like system where the template is parsed and then byte-compiled into the runtime artifact, while (b) shows a JSP-like preprocessing system where the template is first converted into textual source code which then is parsed and compiled. ETL's use of XML (c) allows the web developer to manipulate and analyze the AST thus simplifying the system.

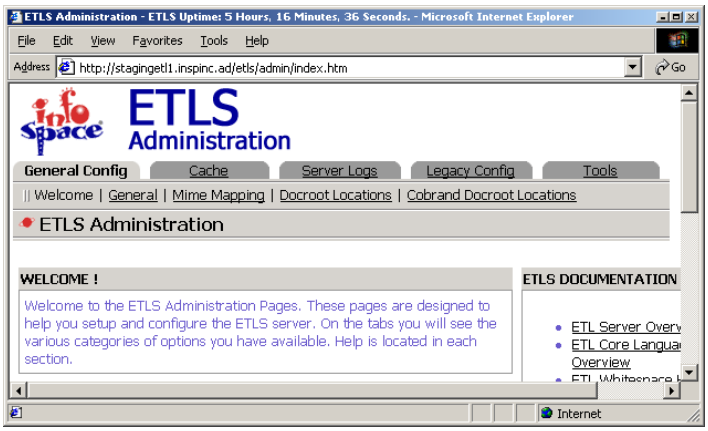

Figure 8: The ETL Server has a web-based administration interface to support configuration and management.

Byte-compilation happens transparently and results in up to a 100% performance improvement over the fully interpreted language ETL replaces. The ETL Server is architected as an ISAPI extension to Microsoft's IIS and could be easily ported to any web server that exposes a similar Extension Control Block interface. IIS is used to manage the raw socket connections and many HTTP protocol details, but ETLS generates the response for all incoming requests. The implementation of ETLS is about sixty thousand non-comment, non-blank lines of C++ code.

In addition to supporting the ETL language, our secondary design goals in building ETLS were: 1) to be an appliance network box, so that all interactions with the server could occur via simple network protocols such as HTTP and SMB fileshares; 2) support easy management via HTTP and platform specific management interfaces such as the Microsoft Management Console (MMC) and the Service Control Manager; and 3) to provide support for secure debugging and run-time reflection on template execution.

To support administrative management of the server, we implemented special primitives in a separate `admin` namespace that expose reflection and configuration capabilities. For example, with these primitives you can retrieve an XML dump of the state of the

```
Template 'books.html.etl' has 0 errors.
0 template('books.html.etl') {
1  set_simple(#xml/books := '...')
2  #element('<html>','</html>') {
3   #element('<table>','</table>') {
4    for-each(#xml/books/book) {
5     #element('<tr>','</tr>') {
6      #element('<td>','</td>') {
7       get(@author)
       } // #element('<td>','</td>')
8      #element('<td>','</td>') {
9       get(@title)
       } // #element('<td>','</td>')
      } // #element('<tr>','</tr>')
     } // for-each(#xml/books/book)
    } // #element('<table>','</table>')
   } // #element('<html>','</html>')
  } // template('books.html.etl')
```

Figure 9: To support debugging, the ETL server permits decompilation of the internal representation of the ETL template from Fig. 5. The nesting of elements is preserved, but the byte-codes have been linearized for performance.

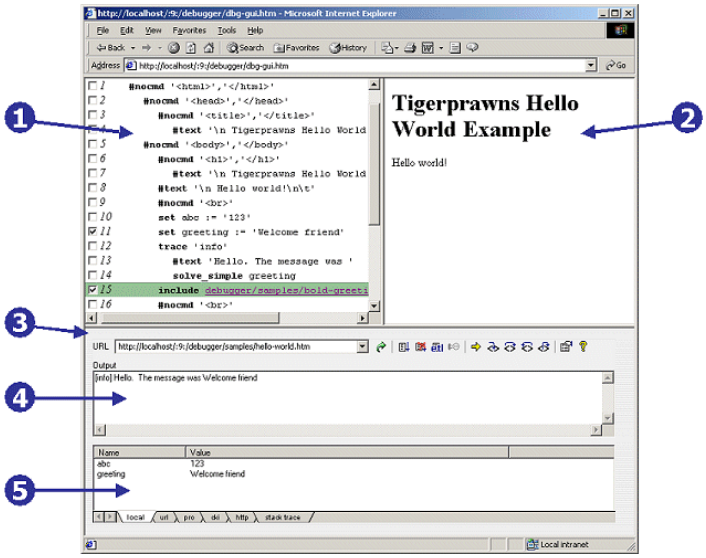

Figure 10: The ETL debugger has five main components: 1) the byte-codes being executed; 2) the presentation of the response thus far generated; 3) the address and tool bars; 4) the warning and trace message console; and 5) the variable inspection pane.

cached URL-to-template mapping or reset that cache. These special capabilities are only available when the request is properly authenticated and authorized. To facilitate easy management of the server, we implemented a web application (in ETL, of course) that organizes and exposes these underlying primitives and integrates the documentation for the server (see Fig. 8). All server parameters are settable via `admin:*` primitives, and the server is configured by executing a distinguished `etl-server-conf.etl` template at startup.

To facilitate debugging, ETLS supports numerous special URL parameters, each prefixed with `etl-`, that enable special processing of the request. As with the special primitives, these extra behaviours function only when the request is authenticated and authorized. For example, `?etl-show-resolution` suppresses the normal response document and instead generates an XML document that describes the dynamic call graph used in servicing the request. Another parameter, `?etl-decompile` can be added to a URL to request that the server dump the internal byte-compiled representation as the response (see Fig. 9). Other parameters, such as `?etl-no-random` and `?etl-dummy-ads`, simplify testing by eliminating some major points of non-determinism in template execution. These capabilities enormously simplify regression testing.

Although the special `etl-*` parameters are very valuable, often single-stepping through code or using breakpoints still is the best way to track down specific problems. To that end, we built an ActiveX-based debugger (Fig. 10) that embeds itself in Internet Explorer and communicates with the ETL server via a separate persistent TCP connection. It supports single stepping (both forwards and backwards) as well as the setting of breakpoints and the inspection of variables and request parameters. The debugger also provides a convenient display mechanism for trace messages (output using `bl:trace`) and other warning messages that would normally be sent to the standard-error output.

InfoSpace's production web tree contains over sixty thousand ETL templates and the clusters of ETL Servers serve millions of requests per day. Many of the current templates were generated by a complex multi-step automatic migration process from a lower level proprietary templating language that pre-dates ETL. The automatic migration was made possible by the fact that the preceding language, too, was very restrictive on the language level constructs it afforded. However, that language, like ASP and JSP, was text-based, not markup-based, and used a proprietary and obscure syntax that made writing tools and analyzing the source code especially troublesome.

As part of migrating from our legacy language to ETL, we wrote a lint-style checker to uncover ambiguities in the original source templates. That script uncovered about forty-five thousand total major warnings spanning about fourteen thousand templates that required a human to resolve an ambiguity or an error. Fortunately, most of the corrections were straightforward and the entire process only took a couple of months of background work by our web development teams. The quantity of bugs found in the templates of the legacy language suggest the substantial additional value that the static checking of ETL provides.

Currently, only a couple hundred templates have been authored directly in ETL outside of the server development team. Most notably the administration interface was written using the tools described above, and the feedback from those web developers has been extraordinarily positive.

4. RELATED WORK

Earlier work, including JavaML, argues that XML adds much value as a complementary representation for source code [4, 29, 33, 18]. ETL takes the next step and uses XML as the primary representation for a markup templating language—an approach that proves especially worthwhile because template developers are already writing markup directly.

XSLT [12] is the only popular templating language that uses a restricted language. Unfortunately, it suffers from three fatal weaknesses for our intended usage scenario: 1) its declarative model confuses developers used to procedural programming; 2) it is a bit too limited, often causing the need to use it in conjunction with a server-side scripting language such as JavaScript or Visual Basic (in particular, XSLT does not support building its data model by doing web service or other HTTP requests); and 3) it lacks a

streaming output mechanism, thus limiting its applicability to large incrementally-processable documents or situations where the desired user experience makes it important to send some HTML over the wire while the remainder of processing occurs. XExpr [24] and XInclude [20] are two other languages that use an XML representation but focus on domains even more restricted than the markup-generating template problem that ETL solves.

LAML [26, 25] has similar goals to ETL, but takes an opposite approach: LAML embeds the simpler markup languages in Scheme, an especially powerful and expressive language. Each markup element has a corresponding mirror procedure that outputs the appropriate tags, does validity checking, and otherwise reflects the semantics of the markup language. Thus, LAML gets many of the same benefits of ETL with respect to increasing the probability of generating well-formed markup. However, it uses the syntax of Scheme's S-expressions rather than XML markup, thus forgoing the ability to leverage XML-based tools for analysis. Additionally, the extra power of Scheme is in direct contradiction of our goal of reducing the expressiveness of the language. A similar approach has been applied by embedding markup in Haskell and Curry, two other functional languages [37, 22, 19].

`<bigwig>` is another significant research project [9] that has similar goals. That work adds a first class markup-fragment type to a conventional programming language. That system is more aggressive in its verification of the well-formedness of markup values: an iterative data-flow analysis bounds the possible values which can then be confirmed to be valid [8]. As with the other approaches that use a general-purpose language, the benefits ETL provides in separating the presentation details from back-end applications are lost.

5. CONCLUSIONS & FUTURE WORK

The Extensible Templating Language questions a fundamental assumption: that the front-end markup-generating engine of a multi-tier web application requires a general-purpose programming language. We designed ETL by carefully choosing the minimal required language constructs, giving those features an XML syntax, and permitting them to be intermingled directly with literal markup. Limiting the power of the templating language forces improved separation of front-end presentation details from back-end application logic. Additionally, ETL is simpler than other templating languages, exposing a unified XML data model and integrating cleanly with modern Internet and web standards.

By leveraging a 100% XML syntax, we make analyses and tools especially easy. In particular, we show how the structure of the source program can be used to reduce the probability of generating ill-formed or invalid response documents. We introduced a novel use of pairs of attributes, called slots, to improve the type-safety of the XML syntax. ETL also contributes a taxonomy of XML whitespace along with rules for finer-grained control of which whitespace is significant. To support extensible abstractions, ETL uses XSLT as a rewrite engine for macro-like custom-tags. We have over a year of experience running ETLS in production and it serves millions of requests per day.

While developing ETL, our novel uses of XML exposed a couple of limitations in important web standards. First, ETL's slot syntax would be more precisely validated by the XML Schema Definition language if XSD supported better control over sets of attributes. There is currently no way to express the notion that exactly one of a pair of attributes must be given, thus our system must check this constraint procedurally. Second, we have found limitations in using XSLT for source to source transformations of ETL templates. A template developer may care about non-infoset syntactic artifacts such as newlines between attributes, but XSLT discards those details [40].

Thus far, we have been satisfied with increasing our ability to catch markup errors statically. Enabling provably well-formed response documents is a noble goal and time will tell whether our needs will justify applying `<bigwig>`'s lessons. Another interesting area for future work is using Perl-like taint checking on input variables to improve the security of the server. Combined with server-integrated support for input parameter integrity and checking, these features could defeat many attacks [34].

Over the coming months, we expect to learn a great deal from the increased usage of the Extensible Templating Language and its server, but already ETL and ETLS have been an enormous success in revitalizing the InfoSpace platform.

6. ACKNOWLEDGMENTS

This research is supported by InfoSpace and its advanced server development group. I thank Russ Arun and Steve Newman for their early recognition of the value of this approach and support. ETL itself was designed in collaboration with Abhishek Parmar and is influenced by the legacy language of its predecessor which was implemented by Jean-Remy Facq. The ETL Server was built by the author along with Abhishek Parmar, Venkatesh Juryala, Sridhar Koneru, Mark Sandori, Michael Harrison, Kris Bradley, Angela Plyler, Sunil Thomas, and Howard Zhao. Testing of ETLS was ably performed by Zine Rif, Sriram Krishnan, Michael Schaffer, Shavkat Azimov, Jeff Wells, Ilian Georgeiw, and Russell Ashmun, and credit goes to Antonio Casacuberta for continued support of the server group. I also thank Jeff Torgerson, Matthew Benedict, Vasanth Cattamanchi and all of the InfoSpace web developers for their contributions. ETLS is a trademark of InfoSpace, Inc.

7. REFERENCES

[1] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. Transformation in intentional programming, September 1997. `http://research.microsoft.com/ip/overview/TrafoInIP.pdf`.

[2] Apache Foundation. Apache HTTP server version 2.0 documentation. Web page, 2002. `http://httpd.apache.org/docs-2.0/`.

[3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1998.

[4] G. J. Badros. JavaML: A markup language for java source code. In *Proc. of the Ninth Intl. Conf. on the World Wide Web*, Amsterdam, May 2000. Elsevier Science B. V. `http://www.cs.washington.edu/homes/gjb/JavaML`.

[5] G. J. Badros and D. Notkin. A framework for preprocessor-aware C source code analyses. *Software—Practice and Experience*, 2000.

[6] P. V. Biron and A. Malhotra. XML schema part 2: Datatypes. W3C Working Draft, November 1999. `http://www.w3.org/TR/xmlschema-2`.

[7] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. W3C Working Draft, August 2002. `http://www.w3.org/TR/xquery/`.

[8] C. Brabrand, A. Møller, and M. I. Schwartzbach. Static validation of dynamically generated HTML. In *Workshop on Program Analysis for Software Tools and Engineering*, Snowbirrd, Utah, June 2001. `http://www.brics.dk/bigwig/publications/valid.pdf`.

[9] C. Brabrand, A. Møller, and M. I. Schwartzbach. The `<bigwig>` project. *Transactions on Internet Technology*,

2002. http://www.brics.dk/bigwig/publications/bigwig.pdf.

[10] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation, February 1998. http://www.w3.org/TR/REC-xml.

[11] J. Castagnetto, H. Rawat, S. Schumann, C. Scollo, and D. Veliath. *Professional PHP Programming*. Wrox Press, Birmingham, United Kingdom, 1999.

[12] J. Clark. XSL transformations. W3C Recommendation, November 1999. http://www.w3.org/TR/xslt/.

[13] J. Cowan and R. Tobin. XML information set, October 2001. http://www.w3.org/TR/xml-infoset/.

[14] D. Coward. Java servlet 2.3 specification. http://www.jcp.org/aboutJava/communityprocess/final/jsr053/.

[15] A. Crane. Experiences of using PHP in large websites. In *UKUUG Linux Developers' Conf.*, Bristol, July 2002. http://www.ukuug.org/events/linux2002/papers/html/php/.

[16] M. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 2002. To appear.

[17] W. Forum. Wireless markup language specification, November 1999. http://www1.wapforum.org/tech/terms.asp?doc=SPEC-WML-19991104.pdf.

[18] K. Gondow and H. Kawashima. Towards ANSI C program slicing using XML. In *Electronic Notes in Theoretical Computer Science*, volume 65(3), 2002. http://www.cwi.nl/ftp/markvdb/entcs65.3/65.3.005.pdf.

[19] M. Hanus. High-level server side web scripting in Curry. In *Proceedings of the Third Intl. Symposium on Practical Aspects of Declarative Languages*, 2001. http://www.informatik.uni-kiel.de/~mh/publications/papers/PADL01.ps.gz.

[20] J. Marsh and D. Orchard. XML inclusions 1.0. W3C Candidate Recommendation, September 2002. http://www.w3.org/TR/2002/CR-xinclude-20020917/.

[21] Megginson Technologies. SAX 1.0: The simple API for XML. Web document, 1999. http://www.megginson.com/SAX.

[22] E. Meijer. Server side web scripting in Haskell. *Journal of Functional Programming*, 10(1), January 2000.

[23] NCSA Software Dev. Group. The CGI 1.1 specification. Web page. http://hoohoo.ncsa.uiuc.edu/cgi/interface.html.

[24] G. T. Nicol. XEXPR – a scripting language for XML. W3C Note, November 2000. http://www.w3.org/TR/xexpr/.

[25] K. Nørmark. Programmatic WWW authoring using scheme and LAML. Web Engineering track at WWW2002 conference, May 2002. http://www.cs.auc.dk/~normark/laml/papers/www2002/p296-normark.html.

[26] K. Nørmark. Web programming in scheme - the LAML approach, 2002. http://www.cs.auc.dk/~normark/laml/papers/jfp.pdf.

[27] Obj. Tech. Intl. Eclipse platform technical overview, July 2001. http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[28] E. Pelegri-Llopart. Java server pages 1.2 specification. http://www.jcp.org/aboutJava/communityprocess/final/jsr053/.

[29] J. F. Power and B. A. Malloy. Program annotation in XML: a parse-tree based approach. In *Proceedings of 9th Working Conf. on Reverse Engineering*, Richmond, VA, November 2002. http://www.cs.may.ie/~jpower/Research/Papers/2002/wcre02.pdf.

[30] D. Raggett, A. Le Hors, and I. Jacobs. HTML 4.01 specification, December 1999.

[31] D. Raggett and C. Reitzel. HTML tidy library project. http://tidy.sourceforge.net/.

[32] W. C. Richard Kelsey and J. Rees. *Revised 5 Report on the Algorithmic Language Scheme*, February 1998.

[33] S. Schonger, E. Pulvermüller, and S. Sarstedt. Aspect-oriented programming and component weaving: Using XML representations of abstract syntax trees. In *Second German AOSD Workshop*, Bonn, Germany, February 2002.

[34] D. Scott and R. Sharp. Abstracting application-level web security. In *Proc. of the 11th Intl. Conf. on the World Wide Web*, May 2002.

[35] D. Soroker, M. Karasick, J. Barton, and D. Streeter. Extension mechanisms in montana. In *Proceedings of 8th Israeli Conf. on Computer-Based Systems and Software Engineering*, June 1997.

[36] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, 1994.

[37] P. Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(4&5):435–468, July 2002. http://www.informatik.uni-freiburg.de/~thiemann/papers/modeling.ps.gz.

[38] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema part 1: Structures. W3C Working Draft, November 1999. http://www.w3.org/TR/xmlschema-1.

[39] S. Tregar. HTML::Template documentation. http://www.perldoc.com/cpan/HTML/Template.html.

[40] M. L. V. D. Vanter. Preserving the documentary structure of source code in language-based transformation tools. In *IEEE Intl. Workshop on Source Code Analysis and Manipulation*, November 2001. http://research.sun.com/people/mlvdv/COM.sun.mlvdv.doc.scam_nov01.paper_pdf.pdf.

[41] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, California, 3rd edition, 2000.