The TRiX Template Resolution Framework

Anders Kristensen
Extended Enterprise Laboratory
HP Laboratories Bristol
HPL-98-04
January, 1998

templates, web applications, XML, HTML, macros, OO frameworks

This paper describes a framework for applying templates to applications and documents on the Web. The primary motivation is the need of Web application developers to separate program logic from presentation logic. A template is a prototypical document or part thereof. It consists of content in the target language, HTML, XML, or plain text, plus markup specifying variable parts of the document. The Template Markup Language (TML) is an application of XML which defines a generic and flexible set of template markup elements. TRiX (Template Resolution in XML/HTML) is a framework for processing TML. It excels in being highly extensible - both in the types of values variables can take, variables being URLs, and in the set of template elements recognized.

Internal Accession Date Only

The TRiX Template Resolution Framework

Abstract

This paper describes a framework for applying templates to applications and documents on the Web. The primary motivation is the need of Web application developers to separate program logic from presentation logic. A template is a prototypical document or part thereof. It consists of content in the target language, HTML, XML, or plain text, plus markup specifying variable parts of the document. The Template Markup Language (TML) is an application of XML which defines a generic and flexible set of template markup elements. TRiX (Template Resolution in XML/HTML) is a framework for processing TML. It excels in being highly extensible - both in the types of values variables can take, variables being URLs, and in the set of template elements recognized.

1. Introduction

Applications on the World-Wide Web use the Common Gateway Interface or Web server APIs in order to generate content dynamically in response to HTTP invocations. Typically program logic embeds the HTML document directly in the application source code but customizes it in small ways whenever the HTML is output in response to a request.

Experience with writing Web applications has demonstrated the importance of separation between program logic and presentation logic, the latter typically being in the form of HTML. This is especially true as applications get bigger and more mission critical. As the skills and tools required for writing Web application code and authoring the GUI are so clearly different there is a lot to gain from separating the two activities. First, the HTML code can be modified without access to the application source code and without needing to recompile and retest the application. Second, HTML and application code can be edited with whatever tools are most appropriate for each task. Third, localization is done on documents rather than on program code and is hence much easier and cheaper.

The approach of embedding application source code directly in the HTML page has been proposed but fails to address this problem.

1.1 Web Applications as FSMs

The request-response style of interactions between client-side user agents and server-side applications on the Web naturally leads to the application being structured as a finite state machine (FSM). When a client makes an HTTP request it triggers a state transition in the application which then returns a response in the form of a new HTML page. The FSM corresponding to a medium sized Web service could consist of, say, 5-10 nodes.

The notion of HTML templates relies on two observations. First, different invocations triggering a transition to the same "node" in the service FSM will receive roughly the same HTML page but with key bits differing, and second, HTML pages corresponding to different nodes will more often than not have markup such as headers, footers, and other structure in common. Defining such structures in one place guarantees consistency in the pages and simplifies maintenance.

1.2 Templates

The idea of templates is to separate the presentation logic from application logic. HTML documents are stored separately from program logic but contain special markup which places key parts of HTML code under application control. The typical scenario is that a Web server receives an HTTP request and passes the request on to the application. The application figures out which "node" (corresponding to an HTML template) to transition to next, computes the set of name-value parameters for this template, and asks the template processor to *resolve* the template in the context of these parameters. The result of this process, which contains no template markup, is what gets send back to the user-agent.

The template markup defined in this paper allows

- definition of variables - as literal values or content of Web resources
- variable substitution
- flow control directives used to specify conditional content

The most basic function of the template language is variable substitution. A variable is a binding between a name and a value in some context. Variable names are URLs and values are pieces of content - text strings which can themselves contain markup including variable substitution and flow control directives.

The template mechanism proposed here has the following properties:

- **Modularity**: separation of work of programmers and authors is highly beneficial.
- **Nesting**: template directives (definitions, substitutions, and conditionals) nest.
- **Consistency**: we use XML markup for template syntax and URLs for variable names.
- **Programming language independent**: although templates are envisioned as being particularly useful for generating content at the control of a program the interactions between the program and templates are very simple and not specific to any particular programming language.
- **Extensibility**: the template resolution framework allows applications to extend the set of known URLs and template elements via *handlers*.

It is useful to contrast templates with a time-proven technology. The analogy between our template processor and the C preprocessor, `cpp`, is quite good. Template definitions are like macro definitions and flow control directives are like `cpp` conditional compilation (but allows more powerful conditions). However there are some important differences. First, unlike `cpp` the TRiX template processor respects the target language syntax by doing transformations on *parse trees* rather than source text. Second, the equivalent of macro definitions can have a number of sources, only one of which is the source text itself. And third, the TRiX framework is extensible in ways `cpp` is not; TRiX handlers are pieces of code and can thus do arbitrary transformations on the parse tree.

Section 2 discusses the template "lifecycle". Section 3 presents the Template Markup Language (TML), and section 4 shows how this, and more, is realized in the TRiX framework. Section 4 also discusses how TRiX extensions are written and how they interact with each other, using a database access component for the purpose of illustration.

2. The Template Processing Model

A template consists of "static" portions in the target language (e.g. straight HTML) together with dynamic *template elements* which are resolved at template load or write time.

Templates are loaded in a template *context*. The context associates variable names with values and knows about template *handlers* configured with the processor.

Figure 1 shows the steps taken by the TRiX engine to load a template.

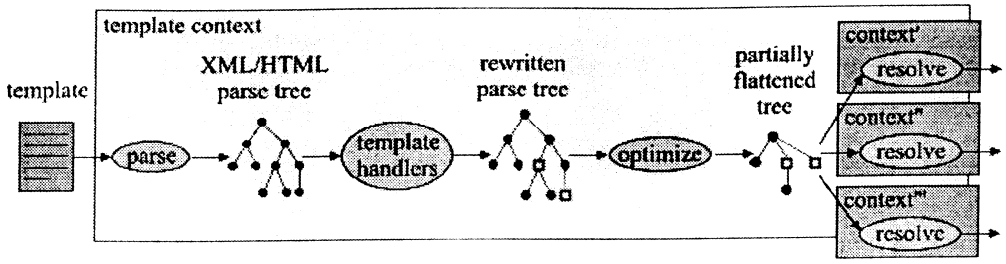

Fig. 1: Loading Templates

The steps are as follows:

- **Parsing**. The template document is loaded from a file and parsed. The resulting parse tree has support for tree navigation, attribute and content management, as well as for writing itself on an output stream in a specific context.
- **Template handlers**. The context in which the template is loaded recorded which nodes in the parse tree correspond to template elements. After completely constructing the parse tree the handler associated with each template element is invoked. These handlers have full access to the parse tree, and can rewrite the document in any way they like. They also have access to the context. In particular the `define` TML element stores variable bindings in the context during this phase.
- **Optimization**. In Web applications templates are loaded once but written all the time. The purpose of the optimization phase is to "flatten" the parse tree as much as possible prior to resolution. This step changes the template representation from an XML/HTML parse tree to a parse tree with fewer nodes and where there isn't generally a one-to-one correspondence between nodes and markup.
- **Resolution**. A template can be repeatedly resolved. The result of the resolution process is that the template document is written to an output stream with all template elements substituted for content in the target language. The substitution can be a simple text expansion or it can be anything computable.

Templates are resolved in a context but not necessarily the same one in which they were loaded. In Web applications we *wrap* the original context with a context which is specific to the HTTP request in question. Hence the internal representation of a template can refer to properties of HTTP requests yet to be received. We go into this in more detail in section 4.1.

The model doesn't assume a client-server content delivery model. Server-side template resolution is of interest to writers of server-side Web applications but templates could be interesting in other environments, e.g., as a client-side mechanism for dealing with variances in user-agent capabilities. Also, the distinction between template load-time and resolution-time is important for some applications, such as server-side Web applications, but for others a template may be resolved at the same time as it is loaded.

3. The Template Markup Language

The template framework defined in this document is intended to be usable with a variety of target languages. The primary motivator is the need for HTML templates in Web applications but it is important that the same framework applies to XML languages and plain text documents.

Template definitions are given as XML DTDs in the text. XML applications can use these elements without change to their DTD by using namespaces. Extending the HTML DTD to include the template DTD given here would be straightforward. The examples are given using HTML but sometimes with an XML syntax (without namespaces).

3.1 Variable Definition

Within an HTML or XML document content can be associated with a variable name using the `define` element:

```
<!ELEMENT define   (#PCDATA | subst | if)* >
<!ATTLIST define   id       ID              #REQUIRED
                   href     CDATA           #IMPLIED
                   delayed  (true|false)    "false" >
```

Attribute definitions:

`id` = *name*

The value of the `id` attribute is used to refer back to the element content later. The `id` attribute can be associated with almost any element in HTML 4.0 and hence it is natural to use it here as well. Note, however, that whereas in HTML 4.0 the namespace within which the `id` is unique is that of the containing document for the purposes of this specification the namespace may be a larger collection of documents loaded by the template processor.

`href` = *URL*

This attribute specifies the location of the data which is being associated with the *id*. If the href element isn't specified or if retrieving that resource fails the variable is set be the contents of this element, if any. This provides for a simple and robust error handling mechanism.

`delayed` = `true|false`

This attribute specifies whether to evaluate the `define` element at template load time (the default) or at resolution time.

Upon encountering a `define` element the template processor associates the definition with the `id`. The contents of the `define` element isn't sent to a client or written to an output stream until this is explicitly requested by the `subst` element. The following example associates the variable *brown-addr* with some HTML `address` markup:

```
<define id="brown-addr">
  <address>
    J. R. Brown<br>
    8723 Buena Vista, Smallville, CT 01234<br>
    Tel: +1 (123) 456 7890
  </address>
</define>
```

Unlike other TML elements the `define` element is typically interpreted and resolved at the time the template in which it occurs is loaded. Setting the `delayed` attribute to `true` changes this behaviour.

3.2 The `var` URL Scheme

Before proceeding to present the `subst` element we need to discuss the nature of template variables and in particular how they are referenced in more depth.

We define the `var` URL scheme to denote TML variables. By denoting variables using a URL syntax we can extend the semantics of template elements to have a useful function for URLs in general - in particular anything in URL space can be assigned to variables.

The `var` form of URLs is one of:

```
var:<variable-name>
var:<subscheme>:<variable-name>
```

where *variable-name* is an identifier taken from the URL alphabet as defined in [RFC1738].

3.2.1 Relative URLs and Default Protocols

Within template documents we define the default protocol for relative URLs to be `var` [RFC1808]. This means that the `var:` part of URLs can be omitted. Hence the address variable defined above can be referred to either as `var:brown-addr` or simply `brown-addr`.

It also means we can refer to template variables using relative URLs and fragment identifiers as in "`../defs.tml#brown-addr`". Such a reference causes the template processor to load the resource *defs.tml* relatively to the template itself (typically from a file system) and search for an element with the specified name within that resource.

Note that this scheme for variable substitution is readily generalized to content defined using ordinary HTML/XML elements using the `id` attribute or the `name` attribute of the HTML `a` element. Assuming that the following markup is part of file "foo.html":

```
<H1 id="title">My Beautiful Document</H1>
```

then a TML element may refer to that definition as "`foo.html#title`" and the template processor would evaluate this to "My Beautiful Document".

Of course the URL in this statement needn't be relative - it can refer to remote resources using absolute URLs. A template resolution engine may be more or less clever about which URLs it recognizes - it will obviously only know about a finite number of protocols and this number may be 0. All implementations should recognize local relative URLs though, as this is quite useful.

3.2.2 `var` Subschemes

The *subscheme* variation of `var` URLs can be used to allow access to an open-ended set of variables. We have defined and implemented the following:

```
var:http:<variable-name>
var:form:<variable-name>
var:query-string:<variable-name>
var:cookie:<variable-name>
```

```
var:sys:time;format=d+m+y+H:M
```

The first four `var` subschemes correspond to typical sources of parameters to Web applications. The `var:http:` URL scheme, for example, defines variables corresponding to HTTP headers. A server-side Web application can read HTTP request headers by referencing, for example, `var:http:user-agent` and can set HTTP response headers such as `var:http:server`. Some subschemes, such as `var:cookie:`, might allow assignment to variables belonging to it while others, such as `var:query-string:` might not.

The TRiX template resolution engine recognizes all of the above URLs and can be extended to understand more. Handlers implementing subschemes may define additional structure in the *variable-name* part of the URL, e.g. allowing the specification of a set of named parameters in the URL.

The `var` scheme is actually stretching the notion of URLs since variables denoted in this way are not *Universal* at all, but are quite specific to a particular context of template resolution. Hence one cannot give `var` URL to someone else end expect them to get the same result from accessing it.

3.3 Variable Substitution Using the `subst` Element

Variables are substituted into documents in two ways depending on the context in which they're substituted. Ordinarily variables are resolved using the `subst` element, but within attribute values variables are dereferenced using the `$` (dollar) syntax known from various shell programming languages. This section presents the `subst` element and section 3.4 discuss substitution within attribute values.

The `subst` element is defined as a *simple* XLL link [XLL]. Attributes other than *href* and *cond* are defined simply for conformance with XLL and all have fixed values.

```
<!ELEMENT subst    (#PCDATA | define | subst | if)* >
<!ATTLIST subst    href            CDATA                #REQUIRED
                   cond            CDATA                #IMPLIED
                   xml-link        CDATA                #FIXED "SIMPLE"
                   inline          (true|false)         #FIXED "true"
                   show            (embed|replace|new)  #FIXED "embed"
                   actuate         (auto|user)          #FIXED "auto" >
```

An HTML DTD for `subst` would allow arbitrary HTML markup as element content. The intention is that if the `subst` operation fails, e.g. because the variable isn't defined, then the contents of the element is displayed. This is like the behaviour of the HTML 4.0 `object` element and again provides for a more robust protocol by including content for error messages.

Attribute definitions:

`href` = *locator*
: Specifies the variable whose value is to be written to the output stream. The value is an XLL *locator*.

`cond` = *condition*
: An expression in the condition language defined in section 3.6. If the condition evaluates to false or the variable defined by *href* is undefined then the contents of this element is written to the output stream. Otherwise the value of *href* is written. If this attribute isn't explicitly

provided the condition is defined as being satisfied if the variable defined by *href* is defined.

The following examples demonstrate different use of variable substitution.

```
<subst href="brown-addr"/>                  <!-- var defined in same doc -->
<subst href="defs.tml#brown-addr"/>         <!-- different doc; rel. URL -->
<subst href="http://foo.net/scripts.js"/> <!-- entire remote resource -->
```

An XLL locator is a string which can be used to locate a resource. Locators are URLs with a (very) generalized notion of '#'-fragments. Locator "fragments" (*XPointers*) allow addressing part of a document in a number of ways based on the structure of the document. This allows us to address Web resources in a very powerful manner. The following is a simple example which expands into the title of a remote Web document (assuming it has one):

```
<subst href="http://www.acme.org/index.html#DESCENDANT(1,TITLE)"></subst>
```

Since TML elements operate on XLL locators it is possible to do quite sophisticated processing with remote Web applications. A related approach would be to address using paths as defined by the Document Object Model [DOM] (this is used in webObjects Web Interface Definition Language [Allen]).

3.4 Variable Substitution Within Attribute Values

Within attribute definitions in the target document variables are dereferenced using syntax like "`$var:name`". The variable name may be delimited by curly braces, as in "`${var:name}`", to avoid ambiguities. Curly braces are considered unsafe in URLs so can safely be used as URL delimiters [RFC1738].

What appears within the braces can be any URL, not just ones belonging to the `var` scheme (which is the default scheme). When a template document is loaded all attribute values are scanned for embedded variable references. The template is stored as a tree structure which supports efficient resolution.

An example of attribute-embedded variable references:

```
<a href="${var:http:path}?map=${map}&long=${longitude}&lat=${latitude}">
```

The result of resolving this HTML code against the set of variable bindings { `var:http:path`="/servlets/maps", `map`="uk", `longitude`="2-33", `latitude`="54-30"} would be

```
<a href="/servlets/maps?map=uk&long=2-33&lat=54-30">
```

If a literal dollar sign should appear in the resolved attribute value it must be written using the character reference "`&#36;`".

The ability to dereference template variables within attribute values is important in many applications. It has a special role in this specification as the conditional inclusion directives encode conditions in attribute values and need to refer to variables within these.

3.4.1 Computed Variable Names

Another use of substitution in attribute values is that variable names in the `subst` element needn't

be known "statically", i.e. at template load time. The effect of writing something like

```
<subst href="$addr"></subst>
```

is that "$addr" is first substituted to, say, "brown-addr" which is then dereferenced to substitute in the value that will actually appear on the output stream. Basically both the `subst` element and the `$variable` syntax provides a level of indirection and they can be combined to achieve a double indirection.

3.5 Conditional Inclusion

The conditional inclusion elements in this proposal are modelled over the flow control features of server-side includes in the Apache Web server.

The general format of the `if` element is:

```
<!ELEMENT if       (#PCDATA | define | subst | if | elif | else)* >
<!ATTLIST if       cond    CDATA   #REQUIRED >

<!ELEMENT elif     (#PCDATA | define | subst | if)* >
<!ATTLIST elif     cond    CDATA   #REQUIRED >

<!ELEMENT else     (#PCDATA | define | subst | if)* >
```

Attribute definitions:

`cond` = *condition*
: An expression in the condition language. If the condition is satisfied the content of the element is recursively resolved and written to the output stream.

An example of the `if` element in action:

```
<if cond="$tel-work">
  Work telephone number: <subst href="tel-work"/>.
<elif cond="$tel-home">
  Home telephone number: <subst href="tel-home"/>.
</elif>
<else>
  No phone number available.
</else>
</if>
```

Notes:

- For readability it is preferred that the content emitted by the template processor in case of the `if` condition being satisfied should occur immediately after the `if` start tag - before any embedded `if`, `elif`, or `else` elements.
- The sequencing rules of the `if` elements are those commonly found in programming languages. Any number of `elif` elements (possibly none) can follow the `if` element after which follows an optional `else` element. The conditions are evaluated in order and the content associated with the first condition which evaluates to true gets emitted by the template processor.
- `if` elements may be nested to any depth.

3.6 The Condition Language

A *condition* is of one of the following forms (same as Apaches flow control expressions):

string
: true if *string* is not empty

string1 op string2
: Compares *string1* with *string2* using one of the relational operators =, !=, <, <=, >, >=. If *string2* is of the form */string/* then *string1* is matched against it as a regular expression.

(*condition*)
: grouping of conditions using parentheses.

!*condition*, *condition1* && *condition2*, *condition1* || *condition2*
: boolean negation, conjunction, and disjunction repsectively.

Strings can be either literal text or the result of variable substitution. Literal strings may be delimited by single-quotes. This may be necessary e.g. if the string contains white space characters.

Example:

```
<if cond="${var:sys:time;format=H} < 12">Never drink before lunch.</if>
```

The conditional language could itself be expressed in XML but this quickly becomes quite bulky and less readable for a small gain. A possible extension would be to allow more data types in conditions variables hodling integer values coupled with teh ability to do simple arithmetic operations. However, it is not clear that there is a need for this and it was deemed preferable to keep the language minimal.

4. The TRiX Framework

TML is recognized in TRiX (Template Resolution in XML/HTML). TRiX consists of an XML parser with hooks for handling HTML, a parser for the TML condition language, and a set of interfaces and classes representing parse trees, `var` URLs, contexts, etc.

The framework is extensible in two ways: by adding handlers for `var` URL subschemes and template elements. The framework is mostly independent of TML. TML is implemented simply as a particular set of template element handlers - one for each of the `define`, `subst`, and `if` elements.

Applications can load handlers explicitly by registering them with the `TemplateContext` or implicitly by adding them to the `trix.handlers` Java property. This property is expected to hold a colon separated list of class names. Class `TemplateContext` loads each class on the TRiX handler list at startup. These classes will typically have some static code which registers the handler with the TemplateContext. This is the same mechanism as is used by JDBC (Java Database Connectivity) to load database drivers.

4.1 Template Processing Environments

The framework has been used to create three incarnations of a template processor: a standalone processor, a Web server filter which resolves any file of a particular suffix (".tml") before sending it to the client, and an API which can be used from Web applications written to the standard *servlet API* [Servlet API].

We'll take a closer look at the latter two.

4.1.1 Web Applications Using Templates

The TRiX API allows any Java application use of its template model. What is special for Web applications is that `var` URLs may reference information not available at the time the template is loaded (typically at servlet initialization time). Hence templates are loaded in a `TemplateContext` but resolved in an `HttpContext`, see figure 2.

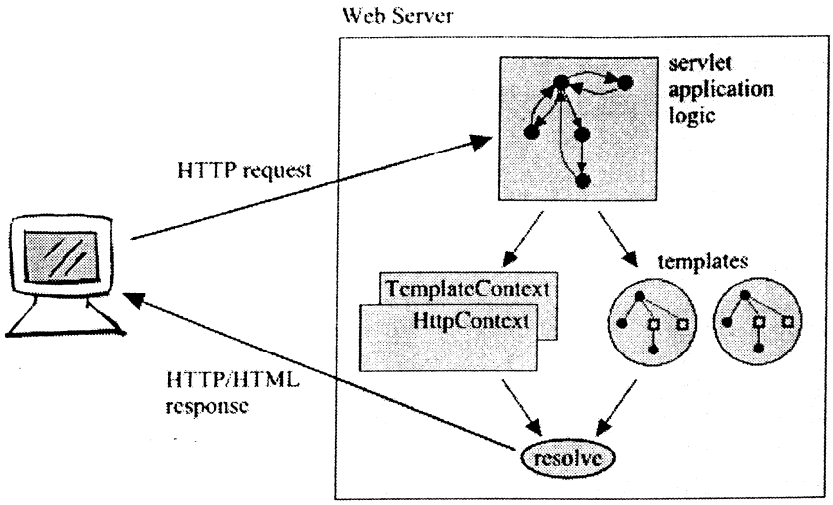

Fig. 2: Servlet Template API

The `TemplateContext` is shared between all servlet invocations but a different HttpContext, parameterized with the HTTP request/response object, is constructed for each invocation. Variables which are `defined` in TML alters the `TemplateContext` and hence such definitions are shared amongst all requests to this servlet. All `HttpContexts` has a reference to the shared `TemplateContext` and during resolution references to variables which are undefined in the `HttpContext` are dereferenced by the `TemplateContext`. This mechanism allows for sharing of variables across servlet invocations.

The following code skeleton shows what it takes to use TRiX templates from within servlets.

```
import hplb.trix.Template;
import hplb.trix.TemplateContext;
import hplb.trix.servlet.HttpContext;
// other imports...

public class MyServlet extends HttpServlet {
  static TemplateContext ctxt = new TemplateContext();
  static Template page1, page2;

  public void init(ServletConfig config) throws ServletException {
    super.init(config);
    try {
      page1 = ctxt.loadTemplate("page1.tml");
      page2 = ctxt.loadTemplate("page2.tml");
    } catch (IOException ex) {
      throw new ServletException("Template not loaded: " + ex.getMessage());
    }
  }
```

```
  public void service(HttpServletRequest req, HttpServletResponse res)
    throws IOException, ServletException
  {
    Hashtable args = new Hashtable();  // contains variable def's
    Template page;                     // set to page1 or page2

    // service implementation...
    args.put("addr", "brown-addr");

    // write template to servlet output stream in new HttpContext:
    new HttpContext(ctxt, args, req, res).write(page);
  }
}
```

4.1.2 Serving Static Files with Templates

It's convenient to be able to include TML markup in Web pages without having to write, install, and manage any service logic. Although TML wasn't intended to replace Web application logic entirely - just separate logic from presentation - it's actually possible to do simple services without writing any code at all (apart from the template markup). Depending on the Web server, this can be accomplished by registering a piece of code, the template processing *filter*, to handle all HTTP requests for files with a particular suffix, e.g. ".tml".

We've written a `TPServlet` that does just this in the Nexus Web server. For performance it is important that TML files not be parsed for each request, that is it is necessary to be able to cache loaded templates.

4.2 Writing a `var` URL subscheme handler

A handler for a new `var` subscheme is simply a factory for representations of URLs knowing how to set and get values for that scheme. The following is the simplest possible example of a `var` subscheme. We define a `var:sys:time` URL whose value is the current time in some fixed format defined by the Java `Date` class. (The `var:sys` scheme defined in TRiX actually does a lot more than this.) Error checking is left out for simplicity.

```
// VarScheme handlers map URL strings of the form var:sys:xxx
// to instances of classes implementing the VarURL interface
public class SysScheme implements VarScheme {
  static {
    // register the var:sys URL subscheme with the template context:
    TemplateContext.regVarScheme(new SysScheme(), "sys");
  }

  // invoked as part of the template loading process.
  public VarURL getVarURL(String url) throws IllegalArgumentException {
    // check url matches expected values...
    return new SysURL();
  }
}

class SysURL implements VarURL {
  // return the current time as a string
  public String getValue(TemplateContext ctxt) {
    return new java.util.Date().toString();
  }

  // empty; we don't allow templates to set the system clock:
  public void setValue(Object value, TemplateContext ctxt) { }
```

```
  public boolean eval(TemplateContext ctxt) {
    return true;  // as the time is always well-defined
  }
}
```

Having done this we can now use `var:sys:time` in `subst` template elements, in conditional expressions, and in other contexts expecting a `var` URL.

4.3 Writing a Template Element Handler

The TML elements are adequate for most applications, but the ability to add handlers for new template elements is quite powerful. It is fairly easy to implement new elements which mix well with existing ones, e.g. elements which nest arbitrarily with other elements and which can take computed arguments.

The steps required for implementing a new template element are analogous to those for implementing a `var` URL handler: a method is invoked on the handler during template loading. The handler has access to the template node and the rest of the XML/HTML parse tree. The handler method returns a tree node which replaces the original node.

An example use of this extension mechanism is our database-to-Web connectivity markup as a TML extension. The idea is to allow content to be generated from a database by including a `query` and a `iter` element in the HTML page (several commercial products for generating database-driven HTML work this way). The `query` associates a name with an SQL query, and also precompiles the query, while the `iter` element causes the query to be executed and then iterates over all rows in the result set.

The following shows a full, working example:

```
<html><body>
<h1>JDBC Access from HTML Templates</h1>

<query id="books" datasource="jdbc:odbc:books-db">
select author,title,year from <subst href="var:form:database"/>
order by <subst href="var:form:sort-field"/>
</query>

<table>
  <tr bgcolor="d0d0d0"><th>Author</th><th>Title</th><th>Year</th></tr>
  <iter id="books">
  <tr>
    <td><subst href="author"/></td>
    <td><subst href="title"/></td>
    <td><subst href="year"/></td>
  </tr>
  </iter>
</table>

</body></html>
```

This retrieves a set of records from a database and displays the result as an HTML table without requiring additional code to run. There are several things to note:

- The query in this example is composed "dynamically" using TML elements to retrieve information from a just-submitted form. The `query` element handler must be written so as to allow such "late binding" (this is exactly what the `define` element does with the `delayed` attribute).

- All code runs in a single Java virtual machine and hence the connection to the database can be shared amongst all requests for this page. Combined with query precompilation this potentially makes this type of database access very fast.
- Within the `iter` element unqualified `subst` elements (without the "`var:`" bit) are interpreted as naming fields in the database result set. The implementation of the `iter` template element accomplishes this by wrapping the context in which it's being resolved in its own context *during resolution of its own content*. The variables themselves are represented the same inside the `iter` element as outside.

This way of displaying the result of a database query is quite natural and a very common thing to do. However one might certainly want to display the result set in a different way. An example might be a set of reservations stored using one record per reservation. One might want to display the result as a table with a row per time-unit, that is in displaying the table we really want to iterate over time rather than reservations. Another example is where one wants to combine two or more related queries in the same table.

There are (at least) two ways of accommodating such "alternative" styles. One is either to write custom template elements or extend existing ones to do what is needed. The other possibility is to use a client-side scripting language, such as JavaScript, to assign the result of the database query to an array and then use the scripting language to do perform special-purpose layout in the client. The client-side code can itself be auto-generated from a GUI development environment but that is outside the scope of this paper.

4.4 The XML/HTML Parser

The XML parser currently used in the TRiX implementation is a fairly straight-forward non-validating parser. That it's non-validating means that it doesn't understand XML DTDs, i.e. it has to reconstruct the structure of XML documents without a priori knowing anything about this structure. Interested as we are in applying TML to HTML and Web applications it is also necessary to be able to correctly parse HTML.

HTML use features of SGML which have been eliminated in XML. This includes optional start and end tags, and SGML empty elements. It looks likely that XML embedded in HTML will become a popular way of exchanging machine-readable data in a disciplined way on the Web in the future. We discuss a few simple measures which, although not dealing with all SGML features, increases the ability of XML parsers to deal correctly with HTML without "understanding" SGML DTDs.

HTML use features of SGML which have been eliminated in XML. This includes optional start and end tags, and SGML empty elements.

The TRiX XML parser interface has methods for, programmatically, providing it with some of the information that is included in an XML or SGML DTD. This includes the ability to tell the parser

- that an element is *empty*, i.e. not to expect an end tag. This is used for HTML `BR`, `IMG`, and `HR` elements amongst others. This is not really necessary for correct parsing of XML documents because of the special `<name/>` XML empty elements syntax.
- that an end tag is *optional* (HTML only). Within unordered lists (`UL` element), for example, the end tag for the `LI` elements delimiting list items is optional. Hence in order to correctly parse HTML unordered lists the parser must know that an `LI` element is terminated by either an `LI` end tag, another `LI` start tag, or the `UL` end tag.

The case where a start tag is implicitly ended by an end tag for its ancestor is taken care of by letting an end tag close parsing of the "nearest" element of the same type still open. This mechanism also catches accidentally missed end tags.

Another piece of information which a more clever parser can derive from a DTD concerns how to handle whitespace. It is not immediately obvious where in a markup document whitespace is significant. In XML whitespace is significant in *mixed* content and insignificant in *element* content. This information is in the DTD but could be provided programmatically.

The template processor implementation has some other special requirements on the XML/HTML parser. We're generally interested in reproducing the static parts of template documents exactly as they appear in the original document. In some ways this means that the stupider the parser is the better. For example, all XML processing instructions and DTDs must pass through on the output stream. A simple way of achieving this is not to recognize such markup except as part of PCDATA. Another special requirement is that all comments should reappear on the output stream and that template markup within comments should be recognized and expanded appropriately. One use of this is to make client-side scripting of database query results work (see section 3.3), as scripts are usually enclosed in comments to hide them from ignorant browsers.

5. Related Work

XML has some support for macros and conditional inclusion through general entities, parameter entities and marked conditional sections. It is possible to share common elements between large document collections using only features build into XML. However this really only works for static documents, not for documents generated on-the-fly by Web applications. There are some specific problems with trying to extend XML notions of entities to the kind of work described in this paper: the entity character set doesn't include common URL characters, hence URLs would need encoding; every document needs to fiddle with its DTD - this isn't something most people want to do; tools have to parse entity declarations which adds a lot to the amount of work needed on a parser; marked sections are not an appropriate (as they stand) basis for doing flow control - flows and tests are too simple.

It seems that an approach based on an XML language and namespaces is neater as it will be more readily approachable by most people and it would seem to be exactly the kind of application XML was designed to address.

Another important body of related work is that of commercially available database-to-Web connectivity tools, such as Bluestone's Sapphire Web, Allaire's Cold Fusion, Oracle's Developer/2000, etc. These tools provide functionality comparable to the database template elements presented in section 4.3. However they don't typically provide such a high degree of openness and integration as is attainable in TRiX.

Mawl is a domain-specific language for programming form-based services [Atkins]. Like TRiX it attempts to solve the problem of separating application logic from presentation logic but in very different way. Being a special-purpose language Mawl has built-in support for setting and retrieving variables from *forms*, where forms is an abstraction covering, for example, HTML pages and IVR systems. A Mawl template contains GUI details and is specific to the medium on which the form is rendered. TRiX differs in allowing Web applications access to details of the request and can thus be highly protocol and media dependent. In our experience such low-level control is actually needed when writing Web applications.

6. Summary

Writing numerous Web applications has shown to us that TRiX does indeed solve the problem of entangled application and presentation logic. TML combined with the notion of variables as URLs provides for a powerful and general language for the construction of documents from templates on-the-fly. We applied it to server-side Web applications but it could equally well be applied on the client-side as an alternative to using scripting languages. A completely different use has been in automatic generation of email messages from plain text templates.

The major benefit of the TRiX framework lies in its extensibility, both in the number of `var` URL subschemes and the set of template elements it knows about, and in the high level of integration that is readily achievable between template elements. Modelling variables as URLs has proven itself very useful. The URL has the same unifying role in the template processor as it has on the Web at large in making TML elements independent of the sources of data they operate on.

XML and XLL has made it possible to define languages which extend HTML in various ways. We firmly believe it would be worthwhile standardizing TML and `var` URL schemes pertaining to different environments such as Web servers and browsers. Later more specific extensions for vertical domains, such as server-side database access markup, could be defined. This would be of obvious advantage to Web site maintainers as they now have to live with vendor-specific methods.

7. Acknowledgements

The TML language and the notion of the template processor was first proposed as a standard on the servlet API mailing list. The work described in this paper evolved partly from feedback from people on that list. Particularly, thanks goes to Cimarron Taylor for his interesting ideas on arrays and iteration and to Dave Hollander for numerous helpful comments on this paper.

References

[Allen]
"WIDL - Application Integration with XML", Charles Allen, In "XML: Principles, Tools, and Techniques", World-Wide Web Journal, Winter 1997, Vol. 2, No. 4.

[Atkins]
"Experience with a Domain Specific Language for Form-based Services", D. Atkins et al., Proceedings of the Conference on Domain-Specific Languages, Oct. 1997.
http://www.usenix.org/publications/library/proceedings/dsl97/atkins.html

[DOM]
Document Object Model Specification.
http://www.w3.org/TR/WD-DOM/

[HTML 4.0]
HTML 4.0 Specification, Working Draft, Sep 1997.
http://www.w3.org/TR/WD-html40/cover.html

[RFC1738]
"Uniform Resource Locators", T. Berners-Lee, L. Masinter, and M. McCahill, December 1994.
ftp://ds.internic.net/rfc/rfc1738.txt

[RFC1808]
"Relative Uniform Resource Locators", R. Fielding, June 1995.

ftp://ds.internic.net/rfc/rfc1808.txt

[Servlet API]

The Servlet API, Sun Microsystems.
http://jserv.javasoft.com/products/java-server/servlets/

[XLL]

"Extensible Markup Language (XML): Part 2. Linking", Tim Bray, Steve DeRose.
http://www.w3.org/TR/WD-xml-link

[XML]

"Extensible Markup Language (XML): Part 1. Syntax", Tim Bray, Jean Paoli, C. M. Sperberg-McQueen (eds.).
http://www.w3.org/TR/WD-xml-lang

URLs

Apache SSI	http://www.apache.org/docs/mod/mod_include.html
The Nexus Web Server	http://www-uk.hpl.hp.com/people/ak/java/nexus/
Bluestone Sapphire/Web	http://www.bluestone.com/
Allaire Cold Fusion	http://www.allaire.com/
Oracle Developer/2000	http://www.oracle.com/products/tools/dev2k/

Appendix A. TML DTD

```
<!ELEMENT define   (#PCDATA | subst | if)* >
<!ATTLIST define   id       ID              #REQUIRED
                   href     CDATA           #IMPLIED
                   delayed (true|false)     "false" >

<!ELEMENT subst    (#PCDATA | define | subst | if)* >
<!ATTLIST subst    href            CDATA               #REQUIRED
                   cond            CDATA               #IMPLIED
                   xml-link        CDATA               #FIXED "SIMPLE"
                   inline          (true|false)        #FIXED "true"
                   show            (embed|replace|new) #FIXED "embed"
                   actuate         (auto|user)         #FIXED "auto" >

<!ELEMENT if       (#PCDATA | define | subst | if | elif | else)* >
<!ATTLIST if       cond     CDATA    #REQUIRED >

<!ELEMENT elif     (#PCDATA | define | subst | if)* >
<!ATTLIST elif     cond     CDATA    #REQUIRED >

<!ELEMENT else     (#PCDATA | define | subst | if)* >
```