Higher-Order

STML Developer's Guide

Draft, Do not Copy or Distribute

Andreas Gawecki, Axel Wienberg

© Higher-Order GmbH, Hamburg.

All rights reserved.

October 15, 1997

Contents

1 Introduction **2**

2 Getting Started **3**

3 Overview of the STML Application Framework **4**

4 STML Elements **5**

4.1 Embedding Tycoon code 5

4.2 Sending Messages 6

4.3 Conditional Processing 6

4.4 STML Variables 6

4.5 Using Backquotes 7

4.6 STML Functions 8

5 Extending the STML Application Framework **9**

5.1 Specializing Processor Objects 10

5.2 Using Specialized Application Objects 10

5.3 Using Form Fields 11

6 STML and SGML **12**

6.1 STML Document Type Definition 13

6.2 Quieting the SGML Parser 13

6.3 Using Entities 14

6.3.1 Internal Entities 14

6.3.2 External Entities 14

A STML Element Reference **15**

A.1 `Define` 16

A.2 `Ref` 16

A.3 `Assign` 17

A.4 `If, ElseIf, Else` 17

A.5 `Fun` 17

A.6 `Apply` 18

A.7 `Send` 18

A.8 `Tycoon` 19

Higher-Order

1 Introduction

A common requirement in electronic publishing is the production of document variants based on information about a particular reader or audience. Electronic commerce applications with interactive and personalized HTML pages are widely known examples. The production of the document variant is usually done on-the-fly, i.e. dynamically at document request time.

Traditionally, such problems are solved with the common gateway interface (CGI) and some kind of scripting language. The same problem can be solved with the Structured Tycoon Markup Language (STML) application framework in a much more structured way. Unlike CGI programs, STML applications are closely integrated with the web pages that control them.

STML supports the dynamic generation of SGML documents in a flexible yet reliable manner. The framework is flexible because the generation process may include arbitrary complex computations written in the object-oriented Tycoon Language. The framework is reliable because the generation process is specified in a type-safe way. Before any output is generated, the Tycoon type checker is called to verify the Tycoon code, eliminating run-time type errors such as “message not understood”. The STML content type checker verifies that the resulting SGML document will conform to the underlying SGML document type definition (DTD). This ensures that the generated output will indeed be valid HTML.

The STML application programmer specifies the structure of the resulting document with the elements of the underlying document type definition, e.g. HTML. These elements may be grouped by STML elements (i.e. tags) which define certain transformations on the given document instance. STML provides tags for conditional processing, tags to declare and manipulate variables, tags for procedural abstraction within SGML, and tags for embedding arbitrary Tycoon code. Activation, processing and delivery of the resulting document is carried out by the Tycoon Web Server via the standard Hypertext Transfer Protocol (HTTP).

STML documents are valid SGML documents themselves, conforming to one of the available STML DTDs. This means that any SGML tool available on the market can be used for STML processing. Examples include SGML structure editors and SGML browsing tools.

2 Getting Started

Suppose you would like to publish a simple HTML page containing some markup and a nifty logo of your company stored as a JPEG image. But some browsers are still not able to display JPEG images inline. Of course, you could convert all your images to GIF, but JPEG often compresses much better, saving network bandwidth. What you'd like to do is this: check the browser the client uses (this is encoded within a standard HTTP header), and send a GIF image if you are not sure whether the browser can handle JPEGs. The STML code that produces the intended output is as follows:

```
<!doctype stml system>

<title>Web Publisher Inc. Homepage</title>

<h1>Welcome to the Web Publisher Inc. Homepage </h1>

<if true='request.hasUserAgent && {request.userAgent[0].product = "Mozilla"}'>
  <img src="logo.jpg">
<else>
  <img src="logo.gif">
</if>
```

This example STML document has to be made accessible through your Tycoon Web Server, somewhere below its document root directory. In addition to features offered by standard web servers like apache, the Tycoon Web Server knows how to compile and evaluate STML code. A typical Tycoon code sequence to launch the Tycoon Web Server would be:

```
server :HttpServer := HttpServer.new(8080);
server.bind;
root ::= HttpDirectoryResource.new(Directory.new("/local/www/htdocs"));
root["stop"] := HttpShutdownResource.new(s);
server.root := root;
server.serve;
```

where a document root directory path `/local/www/htdocs` is assumed. A special resource for shutting down the server is installed under “stop” in the server's root directory.

By convention, STML source is stored in files with the extension '*.stml'. If you have stored the above STML document in a file 'home.stml' directly within the document root directory, you may access the document from your Web browser with the URL `http://hostname:8080/home.stml`.

If you try to fetch this URL right now, the result would be disappointing: the above document will be displayed in plain ASCII. This is because your Tycoon Web Server must be told about STML as a new kind of web resource. This may be done with the following code sequence:

```
application ::= SimpleStmlApplication.new;
root.resourceFactory := HttpResourceFactory.new;
root.resourceFactory.generators["stml"] :=
  fun(file:File)
    StmlResource.new(application, file);
```

This code sequence installs a resource factory in the root directory, which creates `StmlResource` objects for files with the extension '*.stml'. It uses a simple STML application

object with no special functionality. You will learn more about `StmlResource` and application objects in the following section. Your Tycoon Web Server should now be properly configured for your first dynamically generated STML web page.

3 Overview of the STML Application Framework

STML applications are built within an object-oriented application framework for STML document processing, which is an extension of the Tycoon Web Server framework. The STML framework comes with several predefined classes. Some of these classes may be extended by subclassing according to the specific requirements of the application:

`StmlProcessor` objects define the context in which STML documents are evaluated. They contain data specific to the current HTTP request. An `StmlProcessor` object is created by the STML framework when a particular STML web page is requested by some client agent. `HtmlFormProcessor` is a subclass of `StmlProcessor` with added support form HTML forms (see section 5.3, “Using Form Fields”).

`StmlApplication` objects maintain data global to an entire STML application. An application object is created at Tycoon Web Server configuration time. An instance of `SimpleStmlApplication` may be used if no global data is required by the application. More interesting application objects will be introduced in section 5.2, “Using Specialized Application Objects”.

`StmlResource` objects define the behaviour of STML documents in terms of the Tycoon Web Server framework. Each `StmlResource` object is responsible for one '*.stml' file. It takes care of recompiling the file after modifications, and creates an `StmlProcessor` object for each HTTP GET request.

`StmlCompiler` objects translate STML documents into executable Tycoon code when first requested. The resulting code is cached by the resource object for faster document delivery on subsequent requests.

`StmlProcessor` objects are the most interesting objects for application programmers. When an STML web page is requested by the client browser, the corresponding `StmlResource` object delegates the request to a newly created `StmlProcessor` object. This process is transparent to the application. The important point here is that this new processor object defines the environment in which the STML document is being processed (hence the name 'processor').

In particular, the pseudo-variable `self` (which may occur within embedded Tycoon code) refers to this processor object. This has the important consequence that any free identifiers occurring within the Tycoon code are implicitly treated as messages to this `StmlProcessor` object, according to the usual Tycoon scoping rules. You can think of an STML document as defining a method specific to this particular `StmlProcessor` object[1]. Understanding these interrelations is the key to the extensibility of the STML application framework.

`StmlProcessor` objects have a couple of predefined slots. According to the previous statements, these are all directly accessible from Tycoon code embedded in the STML page:

[1] In fact, the STML document source is compiled into a function object which is stored within a private slot of `StmlProcessor`.

`request`: the HTTP request as recognized by the web server. This object contains all HTTP header values sent by the web browser.

`reply`: the HTTP reply being assembled. The application may add headers or modify the values of existing headers. The value of the `body` part of the reply should not be modified since this value is set to the generated output page by the `StmlProcessor` object.

`out`: the output stream of this request. The application may write any additional markup into this output stream. Roughly speaking, the output will be included in the resulting document at the place where the statement is embedded in the STML source document. For convenience, the output stream is an instance of `HtmlGenerator`, making it easy to generate properly nested HTML tags.

`application`: the application object, as created during the Tycoon Web Server configuration process.

The `application` slot is discussed in section 5.2, "Using Specialized Application Objects". The other slots will be explained in the following section.

4 STML Elements

This section gives an overview of the available STML elements and walks you through some sample code. STML elements may occur anywhere within STML pages. However, the STML content type checker verifies that the resulting SGML will conform to the underlying document type definition (see section 6, "STML and SGML"). Notable exceptions are the `send` and `tycoon` tags since they may produce arbitrary output computed at run time. Therefore, if this flexibility is not really needed, other STML tags should be used instead.

4.1 Embedding Tycoon code

The `tycoon` tag allows Tycoon statements to be inserted into STML pages. These statements may contribute to the generated output document by sending text to the `HtmlGenerator` object named by the identifier `out`:

```
<tycoon>
  out.p(fun() {
    out << "The current time is " << Date.now << ",\n"
        << "brought to you by the <b>Tycoon<" << "/b> tag.\n"
  })
</tycoon>
```

This might produce the following output:

```
<p>The current time is Wed May 07 14:13:46 1997,
brought to you by the <b>Tycoon</b> tag.
```

Note how, in the code, the bold end tag is split into two different string literals. This is necessary because the SGML content model of the `tycoon` tag is raw character data (RCDATA). When the SGML parser reads the content of the `tycoon` tag, it will simply scan for

something looking like an end tag, ignoring any start tags. So the first string which looks like an end tag **must** be the closing `tycoon` tag. See section 6, "STML and SGML", for more information on SGML parsing.

Other applications of the `tycoon` tag include state manipulation of the processor and application objects, and modifications of the `reply` header fields (see section 5, "Extending the STML Application Framework").

4.2 Sending Messages

STML provides a means of sending messages to Tycoon objects with arguments containing structured markup. This may be done using the `send` tag. Receiver and selector of the message are specified with `send` attributes of the same name, while arguments are passed inside the body of the `send` using the `arg` tag. The following example sends a string object as an argument to the `HtmlGenerator` `out`:

```
<send receiver='out' selector='<<'>
<arg>This markup is written by the <tt>HtmlGenerator</tt>.
</send>
```

This example works because the argument markup is passed to the `HtmlGenerator` as a Tycoon `String` value. This is exactly what the method `<<` of `HtmlGenerator` expects.

The `receiver` attribute is optional. The default receiver is `self`, which refers to the `StmlProcessor` object as explained in section 3, "Overview of the STML Application Framework". In section 5.1, `Specializing Processor Objects`, we will be ready to give some interesting examples which take advantage of this feature.

4.3 Conditional Processing

The tags `if`, `elseif`, and `else` provide a means of conditional processing in STML. Both `if` and `elseif` require an attribute, `true`, which must be a boolean Tycoon expression.

```
a is
<if true='a < b'>
  less than
<elseif true='a > b'>
  greater than
<else>
  equal to
</if>
b.
```

4.4 STML Variables

STML variables may be used to store and retrieve markup which should occur at more than one place within the resulting HTML page. Variables are defined with `define` and referenced with `ref`. For example, the code

```
<define name='x'><p>This is some <i>markup</i> which should
```

```
occur <b>more</b>than once on the output page.
</define>

<p><ref name='x'></ref>
<p><ref name='x'></ref>
<p><ref name='x'></ref>
```

will produce the output

```
<p>This is some <i>markup</i> which should
occur <b>more</b>than once on the output page.
<p>This is some <i>markup</i> which should
occur <b>more</b>than once on the output page.
<p>This is some <i>markup</i> which should
occur <b>more</b>than once on the output page.
```

The value of a variable may be modified with the `assign` tag. The code

```
<define name='y'>First value of <tt>y</tt>. </define>

<p><ref name='y'></ref>
<assign name='y'>Second value of <tt>y</tt>. </assign>
<p><ref name='y'></ref>
```

will produce the output

```
<p>First value of <tt>y</tt>.
<p>Second value of <tt>y</tt>.
```

4.5 Using Backquotes

Sometimes it is merely necessary to compute the value of some HTML attribute, rather than the element as a whole. Use backquotes (`) to enclose Tycoon expressions as substitutes for HTML attribute values:

```
<p>This page has been acessed
<img src='` "count" + application.counter.printString + ".gif" `'>
times.
```

Here, the outer quotes (') delimit the SGML attribute value. We use single quotes rather than double quotes (") because we need the double quotes as a part of the attribute value. The inner backquotes (`) tell STML to treat the enclosed string as an Tycoon expression that evaluates to a `String` object which is inserted in place of the attribute value. This feature can be used e.g. for computed link destinations, for initial form field values, for hidden fields, for user customized colors etc.

Some SGML document types have attribute values which cannot be explicitly specified, and thus cannot be computed using backquotes. In an SGML DTD, an attribute can be defined as having a *default* value, its presence can be *required*, or its value can be *implied*. In the latter case, the program analyzing the document is assumed to compute the value. Some DTDs abuse this by making the implied value different from any of the attribute's legal

values. This is a problem, for example, with the `selected` attribute of the HTML `option` tag.

The `selected` attribute may be specified by giving the attribute value of the same name, `selected`, which is the only one allowed:

```
<option value='xyz' selected>
```

This is the same as

```
<option value=xyz selected='selected'>
```

There is no equivalent way to express an implied, i.e. absent, `selected` attribute:

```
<option value='xyz'>
```

You cannot use backquotes here, since backquotes *always* produce an attribute value. Simply returning the empty string will not work since that value is not allowed by the HTML DTD: if there is a value, it **must** be the string `'selected'`. In such a case, use the `tycoon` tag. The following code sequence generates the above HTML, where `isSelected` is a boolean Tycoon variable controlling the HTML `selected` attribute:

```
<tycoon> out.option(value ::= "xyz", selected ::= isSelected); </tycoon>
```

We resisted the temptation to extend the STML syntax and parser in a way incompatible to SGML in order to overcome these restrictions. Such an approach would preclude using third-vendor SGML tools such as SGML structure editors.

4.6 STML Functions

The STML developers of Higher-Order have carried the concept of procedural abstraction into the world of SGML. STML programmers factor out common markup into STML *functions* the same way as programmers factor out common code into subroutines in more conventional programming languages. And, like subroutines, the markup may be parameterized by one or more formal parameters.

Formal parameters can be referenced inside the function body with the `ref` tag, just like variables introduced using `define`. Unlike variables, however, it is not possible to assign new values to a formal parameter.

An STML function can be defined with the `fun` tag inside a variable declaration:

```
<define name='list3'>
<fun param='x,y,z'>
<ol>
  <li> <ref name='x'></ref>
  <li> <ref name='y'></ref>
  <li> <ref name='z'></ref>
</ol>
</fun>
</define>
```

This STML function will produce an ordered list with three items. STML functions are called with the `apply` tag; actual arguments are supplied by the `arg` tag. The code

```
<apply name='list3'>
<arg>First item.
<arg>Second item.
<arg>Third item.
</apply>
```

will generate the output

```
<ol>
  <li> First item.
  <li> Second item.
  <li> Third item.
</ol>
```

Of course, the actual arguments may contain markup themselves, including function application and other STML tags.

The above example shows that STML variables may also hold STML functions. Therefore, the value of an STML variable may either be a string representing markup, or an STML function. Which kind of value an STML variable may hold is determined by the contents of the `define` element. Subsequent assignments using the `assign` tag must use the same kind of value. You cannot assign an STML function to a string-valued STML variable, and vice versa.

An STML function may also be passed directly as an argument to a Tycoon method or another STML function without storing it in an STML variable first. Such STML functions are called *anonymous* functions since they do not have a name. For example, passing an STML function to a Tycoon method can be used to apply the same formatting task to every element in an iteration. The following code sequence formats the header values of the HTTP request as a table with two columns:

```
<!doctype stml system>

<title>Http Headers</title>

<table>
<send receiver='request.headers.keys' selector='do'>
<arg><fun param='h'>
 <tr>
  <td><pre><ref name='h'></pre>
  <td><pre><tycoon>
      SgmlEntityOutput.new(out) << request.headers[h].string
    </tycoon></pre>
 </tr>
</send>
</table>
```

5 Extending the STML Application Framework

The STML framework may be extended by subclassing the `StmlProcessor`, `HtmlFormProcessor`, or `StmlApplication` classes.

5.1 Specializing Processor Objects

The functionality directly accessible to STML documents may be easily extended by subclassing the `StmlProcessor` class.

The `StmlProcessor` class defines a `process` method. This method is called by the `StmlResource` object to actually process the request and to produce the output page. Subclasses may extend this behaviour in order to perform additional initialization depending on the request. For example, class `HtmlFormProcessor` analyzes the form fields encoded in the query string (see section 5.3, "Using Form Fields"), and stores them in the processor object for access from within the STML page. Modifications to the reply's HTTP headers can also take place in the `process` method.

A `StmlProcessor` subclass can also be used to define utility methods to be called from within the STML document. As explained in section 3, "Overview of the STML Application Framework", these methods are directly accessible in Tycoon code, without having to specify a receiver. The current processor object is always implied as `self`.

The name of the actual `StmlProcessor` subclass must be specified as the `processor` attribute of the STML tag. The STML tag is implied at the top of every STML document. If ommitted, the Stml compiler uses the default `StmlProcessor`, which offers no special functionality.

The application object must be informed of all `StmlProcessor` subclasses it may encounter. This is best achieved by defining a subclass of `StmlApplication` that registers all necessary processors during initialization, as explained in the following section.

5.2 Using Specialized Application Objects

The default `StmlApplication` class may be sublcassed and extended to hold some data global to the application. A good example is a global counter for the number of page hits of the application. This counter may be implemented as a slot of the application object:

```
<class> MyApplication
<super> StmlApplication
<metaclass> SimpleConcreteClass(MyApplication)
<public>
<slot> counter :Int
<private>
<method>
_init
{ super._init;
  counter := 0;
}
```

STML code can access this counter via the `StmlProcessor`'s `application` slot.

```
This page was used <tycoon> out << application.count </tycoon> times.
```

The easiest way to count page hits for the application is to define a customized `StmlProcessor` subclass which extends the `process` method.

```
<class> MyProcessor
<super> StmlProcessor(MyApplication)
<metaclass> StmlProcessorClass(MyApplication, MyProcessor)
<public>
<method>
process
{
  application.counter := application.counter + 1;
  super.process;
}
```

By extending `MyApplication`'s `_init` method, the application's `StmlCompiler` is informed of the new processor class:

```
_init
{ super._init;
  counter := 0;
  compiler.registerClass(MyProcessor);
}
```

More than one STML Application may be installed on the Tycoon Web Server by registering seperate stml resource factories for different subdirectories of the document hierarchy. Assuming you have two `StmlApplication` objects, `storeApplication` and `newsApplication`, the following code installs them under "/services/store" and "/services/news", respectively:

```
storeDir := root.lookupDirectory(List.with2("services", "store"));
storeDir.resourceFactory := HttpResourceFactory.new;
storeDir.resourceFactory.generators["stml"] :=
  fun(file:File) {
    StmlResource.new(storeApplication, file)
  };
newsDir := root.lookupDirectory(List.with2("services", "news"));
newsDir.resourceFactory := HttpResourceFactory.new;
newsDir.resourceFactory.generators["stml"] :=
  fun(file:File) {
    StmlResource.new(newsApplication, file)
  };
```

As in section 2, "Getting Stared", this tells the Tycoon Web Server how to create resource objects from '*.stml' files. The behaviour specified for the respective subdirectories overrides that specified for the root directory.

5.3 Using Form Fields

When an HTML form is submitted, the browser appends an encoding of all form fields to the `ACTION` URL specified in the form. If this URL refers to an STML page, class `HtmlFormProcessor` can be used to decode the names and values of all form fields. They are translated into a dictionary mapping each field name to the given value. This dictionary is stored in the `HtmlFormProcessor` object's `formFields` slot.

Sometimes, there are multiple form fields of the same name. For these cases, there is another slot named `multiFormFields` which maps each field name to a *list* of all values given for that name.

`HtmlFormProcessor` is a subclass of `StmlProcessor`, and can be subclassed further if necessary. `HtmlFormProcessor` provides an example of extending the `process` method. Omitting details, its code looks as follows:

```
<class> HtmlFormProcessor(...)
<super> StmlProcessor(...)
<metaclass> StmlProcessorClass(...)
<public>
<method>
process
{
  parseQuery;
  super.process;
}
<private>
<slot> formFields :Dictionary(String, String)
<slot> multiFormFields :Dictionary(String, List(String))
<method>
parseQuery :Void
{ (* fill <formFields> and <multiFormFields> *)
  ...
}
```

A complete STML page using `HtmlFormProcessor` might look like this:

```
<!doctype stml system>
<stml processor=HtmlProcessor>
<TITLE>Query argument example</TITLE>

<H1>Query arguments:</H1>

<send receiver='formFields.keys' selector='do'>
<arg>
<fun param="name">
  <B><ref name=name></B>
  <I>is</I>
  <tycoon> out << multiFormFields[name] </tycoon>
<BR>
</send>
```

The page displays one line for each supplied form field name, with a list of all values given for that name. The looping construct is similar to that described in section 4.6, "STML Functions", where we iterated over the HTTP request headers.

6 STML and SGML

SGML is an ISO standard for defining document structures for the application of mark-up schemes [Gol90]. It provides a consistent and precise manner of applying mark-up for describing the component parts of a document.

6.1 STML Document Type Definition

A valid SGML document must include a Document Type Definition (DTD)[2]. A DTD describes each element of the document, and defines the relationships between the entities, elements (i.e. tags) and attributes that make up a document.

STML is basically a set of markup tags which is added to a given markup scheme described by some DTD. The resulting markup language is therefore described by an extended DTD. Higher-Order provides a number of such extended DTDs for some widely used HTML variants.

The default HTML variant is the current final W3C DTD for HTML version 3.2. This DTD is implied by the following `doctype` declaration:

```
<!doctype stml system>
```

which is the same as

```
<!doctype stml system "stml-html-3.2.dtd">
```

If a different dialect of HTML should be used, the system identifier[3] for the extended DTD must be specified.

Note that STML is not tied to HTML in any way. Virtually any DTD may be extented with the STML tag set in a fairly straightforward way, provided no name conflicts occur.

6.2 Quieting the SGML Parser

The Tycoon system parses STML pages with a full-blown verifying SGML parser, namely `nsgmls` by James Clark. `nsgmls` may sometimes produce confusing error messages if it cannot infer an omitted start tag which has been declared optional in the DTD. In HTML, this may happen with the `html`, `head` and `body` tags. For example, if you try to compute the title of an STML page:

```
<!doctype stml system>

<tycoon>out.title({ out << "News on " << Date.now })</tycoon>


<h1>Some more interesting content</h1>
```

you will get the following error message:

```
nsgmls:example.stml:6:3:E: document type does not allow element "H1" here
nsgmls:example.stml:6:39:E: "STML" not finished but document ended
```

Don't kill the messenger. `nsgmls` needs a little help in this case. Simply insert the optional `head` tag before generating the title, and `nsgmls` will be quiet:

[2] The DTD might be implied by the application context. This is usually the case with HTML.
[3] This is the SGML term for system-specific entity names such as file names.

```
<!doctype stml system>

<head>
<tycoon>out.title({ out << "News on " << Date.now })</tycoon>

<h1>Some more interesting content</h1>
```

However, a better solution is to move the `title` tag outside of the Tycoon code so that `nsgmls` can recognize it:

```
<!doctype stml system>

<title>
<tycoon> out << "News on " << Date.now </tycoon>
</title>

<h1>Some more interesting content</h1>
```

6.3 Using Entities

Since STML is built on top of the SGML standard, the usual SGML entity mechanisms can be used within STML documents. There are two different kinds of entities: internal entities and external entities.

6.3.1 Internal Entities

Internal entities are defined within the document itself. The only place where the SGML standard permits the declaration of entities is within the document prologue, i.e. inside the document type definition:

```
<!doctype stml system [
  <!entity WP "<b>Web Publisher</b>">
]>
```

The shortcut `&WP;` may now be used throughout the document. It will be expanded by the SGML parser to `<b>Web Publisher</b>` before being interpreted by STML. Note that entities may contain arbitrary markup.

6.3.2 External Entities

External entities are stored outside of the source document within separate files. The entity reference then refers to the contents of the entire file, similar to the `#include` mechanism of the C preprocessor. This is a convenient way for including the same header and footer within a couple of documents, for example:

```
<!doctype stml system [
  <!entity header system "header.stml">
  <!entity footer system "footer.stml">
]>

&header;  <!-- include header file here -->

<!-- some interesting markup -->

&footer;  <!-- and include footer file here -->
```

The external entity files do not have to be stored in the same directory as the document itself. The entity manager of the SGML parser uses two mechanisms to locate external entities:

Via search path: the environment variable `SGML_PATH` may contain a search path for relative path names of system identifiers. You may wish to include your web server's document root into the path.

Via catalog files: the environment variable `SGML_CATALOG_FILES` may contain a search path for catalog files. A catalog file contains a mapping of entity names to 'system identifiers' (i.e. filenames).

Catalog files have the advantage of handling public identifiers as well, but at the cost of an additional indirection. STML uses the catalog file `$HOX_HOME/lib/sgml/CATALOG`. See the `nsgmls` documentation for more information about catalog files.

A STML Element Reference

This appendix gives a detailed description of the STML DTD. The DTD is customized through the entity `stml.any`, which refers to arbitrary client content inside STML tags.

The description uses the following terms:

Tycoon identifier A valid identifier in Tycoon program source code, starting with a letter and consisting of alphanumeric characters including underscores.

STML type An STML type expression. Type expressions obey the following EBNF grammar:
Type `::= 'String' | 'Fun' '(' ` *Parameters* `')' ':' ` *Type* `;`
Parameters `::= [` *Parameter* `{ ',' ` *Parameter* `} ] ;`
Parameter `::=` *name* `|` `[`*name*`] ':' ` *Type*`;`

The *Type* in a Parameter defaults to `String`. In type expressions, parameter names serve documentation purposes only and can be omitted. They are only used to generate error messages. Parameter names are needed in the `fun` tag's `param` attribute, however.

For example, `Fun(x:String): String` is the type of functions mapping strings to strings. This is the same type as `Fun(y:String):String` or `Fun(:String):String` or `Fun(x):String`, but the parameter is `x`, `y`, anonymous or `x`, resp.

Markup from the client DTD always has type String. If executed, it will create output. So do some of the STML expressions. If markup is written in sequence, it will be executed and will produce output in that order.

Some STML expressions (notably `fun` expressions) evaluate to function closures and do not create output. It does not make sense to write two function closures next to each other, so this results in a compile time error.

It is also a compile time error to write an expression of function type in string (output) context, since the function can not be written to the output and would have to be ignored.

The distinction between string and function context is important for whitespace handling. In string context, whitespace is printed on the output literally, while in function context, it is completely ignored.

Finally, there are some STML expressions that do not evaluate to anything at all. These are `assign` and `define`, and can be used in both contexts.

A.1 Define

```
<!ELEMENT define - - %stml.any>
<!ATTLIST define                        -- variable declaration --
        name      CDATA   #REQUIRED     -- name of this variable --
        type      CDATA   #IMPLIED      -- type of this variable --
        >
```

Define a variable with name `name` and type `type` in the current block. It is initialized with the contents of the `define` tag. `name` must be a valid Tycoon identifier. `type` is an STML type. If omitted, it defaults to the content's type.

The variable can be referenced using the `ref` element or from Tycoon code later in the same or an inner block.

The contents of the `define` expression constitute a block of their own. The content's type must match the declared type.

`define` expressions can be used in any context.

A.2 Ref

```
<!ELEMENT ref - - ANY>
<!ATTLIST ref
        name      CDATA   #REQUIRED     -- name of referenced variable --
        >
```

Reference the variable, argument or slot with name `name`. `name` must be a valid Tycoon identifier. It may have been bound in a let expression in a `tycoon` code block, in a `define` expression or in a `fun` argument list, or it may be the name of a slot or method of the STML Processor object.

If `name` is a string variable, its value is inserted at the current position in the output. In function context, when `name` is a function variable, the `ref` expression evaluates to `name`'s current value.

The type of the `ref` expression is the type of the referenced identifier.

A.3 Assign

```
<!ELEMENT assign - O %stml.any>
<!ATTLIST assign
        name      CDATA   #REQUIRED    -- name of assigned variable --
        >
```

Assign a new value to the variable or slot with name `name`. `name` must be a valid Tycoon identifier. It may have been bound in a let expression in a `tycoon` code block or in a `define` expression, or it may be the name of a slot or method of the STML Processor object. `fun` arguments are not mutable, and cannot be assigned.

The contents of the `assign` expression form a block of their own, and are evaluated to the variable's new value.

`assign` expressions can be used in any context.

A.4 If, ElseIf, Else

```
<!ELEMENT if - - (%stml.any, elseif*, else?)>
<!ATTLIST if
        true      CDATA   #REQUIRED    -- predicate --
        >

<!ELEMENT elseif - O %stml.any> <!-- -(elseif | else) -->
<!ATTLIST elseif
        true      CDATA   #REQUIRED    -- predicate --
        >

<!ELEMENT else - O %stml.any> <!-- -(elseif | else)-->
```

This element facilitates conditional execution. The `if` expression's `true` attribute is a Tycoon expression evaluating to a Boolean. At runtime, if the expression evaluates to true, the contents up to the `elseif` or `else` tag is executed. Otherwise, the `elseif` conditions are tested in order, and the first branch evaluating to true is executed. If this fails, the `else` branch is executed.

The different branches constitute different blocks, so a `define` expression in one branch is only visible up to the end of that branch.

The type of the `if` expression is the common supertype of all branches. In practice, the branches either all have type string or some function type.

A.5 Fun

```
<!ELEMENT fun - O %stml.any>
<!ATTLIST fun
        param      CDATA   #IMPLIED    -- parameter list --
        type       CDATA   #IMPLIED    -- result type --
        >
```

Create a function closure. `param` gives the names and optionally the types of the function's parameters. The syntax is that of function type parameter lists:

Parameters ::= [*Parameter* { ',' *Parameter* }] ;
Parameter ::= *name* | [*name*] ':' *Type* ;

where *name* is a valid Tycoon identifier, and *Type* is an STML type expression as defined above. If the type is omitted, `String` is assumed. If the name is omitted, the argument is ignored.

The optional *type* attribute explicitly states the functions return type. If omitted, it defaults to the content's type. Otherwise, the content's type must match the declared return type.

The type of the `fun` expression is a function mapping the parameters to the return type.

The function body may refer to the parameters using `ref` elements or from Tycoon code. All variables defined in outer blocks are also accessible.

Functions can be executed using the `apply` tag or from Tycoon.

`fun` expressions imply function context.

The *type* attribute is currently ignored.

A.6 Apply

```
<!ELEMENT apply - - (arg*)>  <!-- warning: may include floating %stml -->
<!ATTLIST apply
        name      CDATA   #REQUIRED    -- name of function --
        >

<!ELEMENT arg - O (%stml.any)>
<!ATTLIST arg
        name      CDATA   #IMPLIED    -- name of assigned variable --
        >
```

Supplies actual arguments for a function closure and executes the function. `name` is the name of a variable, argument or slot, just like the `ref` element's `name` attribute. `name` must refer to a function.

Each `arg` tag encompasses a block of its own. The contents are arbitrary STML expressions which are evaluated to a string or a function closure. Each `arg` tag becomes one argument in the function call.

The types and number of the arguments must match the called function's parameters. The type of the `apply` expression is the function's return type.

Functions closures can be defined using the `fun` tag, or from Tycoon. They can be bound to a name as described in the `ref` section.

A.7 Send

```
<!ELEMENT send - - (arg*)>  <!-- warning: may include floating %stml -->
<!ATTLIST send
        selector      CDATA   #REQUIRED   -- method selector --
        receiver      CDATA   'self'      -- receiver expression, default: self --
        >
```

Send a message. The receiver is a Tycoon expression defaulting to `self`, the current processor object. The selector is a Tycoon method selector, an arbitrary string. The `arg` elements, as described in subsection `apply`, contain the STML expressions evaluating to the actual arguments.

Since the called method may produce text output using the processor's `out` slot or using STML function closures, the `send` expression always has type String.

A.8 `Tycoon`

```
<!ELEMENT tycoon - - RCDATA>
```

Execute arbitrary Tycoon code. Like the Tycoon code in `if`, `elseif` and `send` tags, this code may refer to all local variables and arguments defined earlier in this or outer blocks, including the current processor object `self`, and the current output `self.out`. STML function closures are directly accessible as Tycoon function closures taking string or function arguments.

Vice versa, variables bound at the first level in the Tycoon code using let expressions are accessible from STML.

Since the Tycoon code may produce text output, the `tycoon` expression always has type String.

In stml-html, Tycoon code can also be included using the `script` tag with attribute `type='text/tycoon'`.

References

[Gol90] Charles F. Goldfarb. *The SGML Handbook*. University Press, 1990.