This is a chapter from the book

System Design, Modeling, and Simulation using Ptolemy II

First Edition, Version 1.0

Please cite this book as:

Claudius Ptolemaeus, Editor,
System Design, Modeling, and Simulation using Ptolemy II, Ptolemy.org, 2014.
http://ptolemy.org/books/Systems.

16 Web Interfaces

Christopher Brooks, Edward A. Lee, Elizabeth A. Latronico, Baobing Wang, and Roxana Gheorghui

Contents

Ptolemy II includes a flexible mechanism for creating web pages from models and for building web services. The more basic mechanism is the **export to web**, which simply makes a model available as a web page for browsing using a web browser. Such a web page provides easy access and documentation for models that archives both the structure of the models and the results of executing the models. It can be used to share information about models or their execution without requiring installation of any software, since an ordinary web browser is sufficient. More interestingly, the mechanism is extensible and customizable, allowing for creation of fairly sophisticated web pages. You can associate

hyperlinks or actions defined in JavaScript* with icons in a model. The customization can be done for individual icons in a model or for sets of icons in a model.

The more advanced mechanism described in this chapter turns a model into a web service. The machine on which the model executes becomes a web server, and the model defines how the server reacts to HTTP requests that come in over the Internet. A web service can be created that does anything that can be done in a Ptolemy II model. Some care is required, of course, to ensure that such a web service does not create unacceptable security vulnerabilities for the web server machine.

16.1 Export to Web

To export a model to the web, select [`File→Export→Export to Web`], as shown in Figure 16.1. This will open a dialog that enables you to select a directory (or create new directory). That directory will be populated with a file called index.html, some image files, and some subdirectories. One image file shows whatever portion of the model is visible when you perform the export. In addition, there will be an image file for each open plot window. Moreover, there will be one subdirectory for each composite actor that is open at the time of export.

The export dialog offers a number of options, as follows.

- *directoryToExportTo*: The directory into which to put the web files. If no directory is given, then a new directory is created in the same directory that stores the MoML file for the model. The new directory will have the same name as the model, with any special characters replaced so that the name is a legal file name.
- *backgroundColor*: The background color to use for the image model. By default, this is blank, which means that the image will use whatever background color the model has (typically gray). But white is a good option for web pages, as shown in Figure 16.1.
- *openCompositesBeforeExport*: If this is true, then composite actors in the model are opened before exporting. Each composite actor will also be exported into its own web page, and hyperlinks will be created in the top-level image to allow navigation to those web pages in the browser. If you want only some of the composite actors to be included in the export, then you can manually open the ones you want. Only open windows will be included in the export.

*By default, the export to web facility uses JavaScript to display the parameters of actors. JavaScript may be disabled in your web browser. To enable JavaScript. See http://support.microsoft.com/gp/howtoscript.

- *runBeforeExport*: If this is true, then the model is run before exporting. This has the side effect of opening plot windows, which will therefore be included in the export. If you want only some of the plot windows to be included in the export, then you can run the model and close the ones you don't want. Only open plot windows will be included in the export.
- *showInBrowser*: If this is true, then once the export is complete, the resulting web page will be displayed in your default browser.
- *copyJavaScriptFiles*: If this is true, then additional files will be included in the exported page so that the page does not depend on any files from the internet. The files include

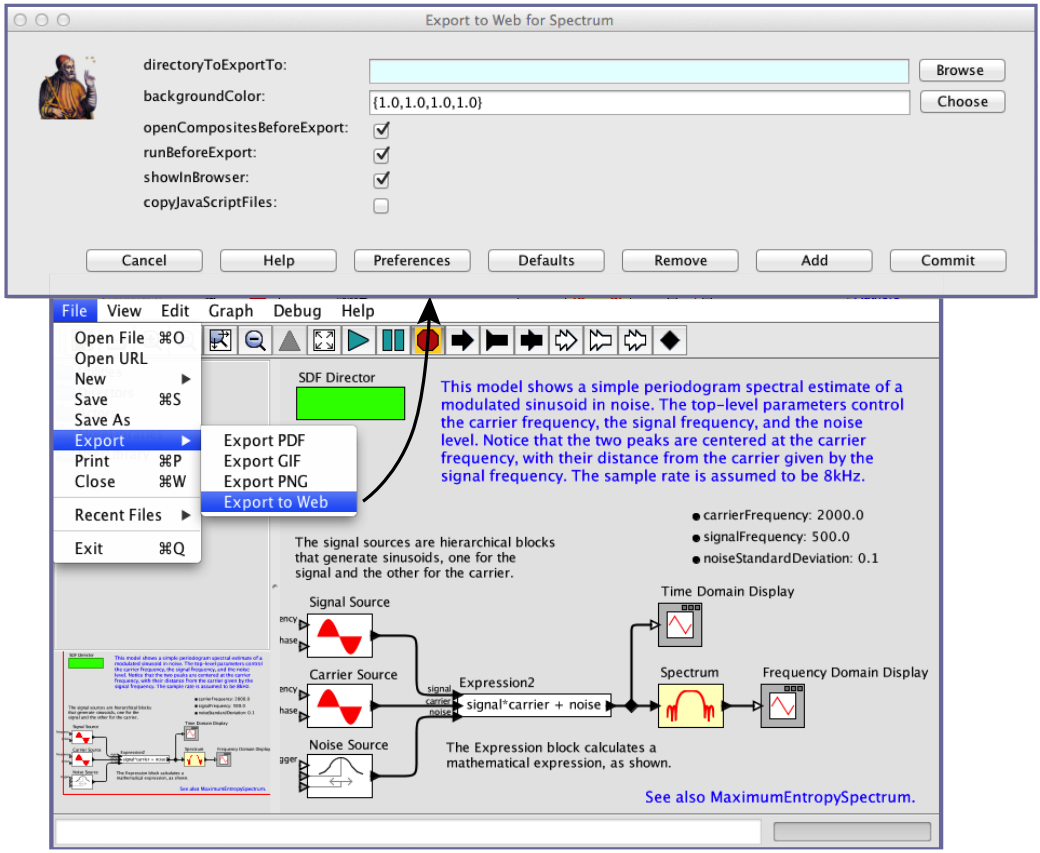


Figure 16.1: Menu command to export a model to the web.

JavaScript code and image files that affect the interactivity and look-and-feel of the web page. By default, these files are not included and are instead retrieved by the web page from `http://ptolemy.org`.

For the example shown in Figure 16.1, the resulting web page is displayed by the Safari web browser as shown in Figure 16.2. This page exhibits some of the default behavior of export to web. A title for the page is shown at the top; this is, by default, the name of the model. Moreover, in the image shown in Figure 16.2, the mouse is hovering over the Signal Source actor, which is outlined; when the mouse hovers over an actor, then by default, a table with the parameter values of the actor is displayed at the bottom of the page, as shown in Figure 16.2.

The generated web page shows the portion of the model visible in the viewing pane. Therefore, parts of the model can be hidden by resizing the viewing pane. For example, one might wish to hide a long list of parameters or attributes. Simply resize the pane, then perform the export.

In Figure 16.1, *openCompositesBeforeExport* and *runBeforeExport* are both set to true (the default is false). Hence, the model is executed before the export, opening plot windows. Hyperlinks to the plot windows are created, and clicking on a plot actor on the web page image will display the plot, as shown in Figure 16.3. In addition, the composite actors in the model, Signal Source, Carrier Source, and Spectrum, all have hyperlinks to a page showing the inner structure of the composite.

All these functions can be customized, as we will explain next.

16.1.1 Customizing the Export

As shown in Figure 16.4, the `Utilities→WebExport` library provides attributes that, when dragged into a model, customize the exported web page. This section explains each of the items in this library, shown on the left in the figure. In each case, you can right click (or control click on a Mac) and select `Get Documentation` to view documentation about the attribute. The attributes are related to one another as shown in the UML class diagram in Figure 16.5.

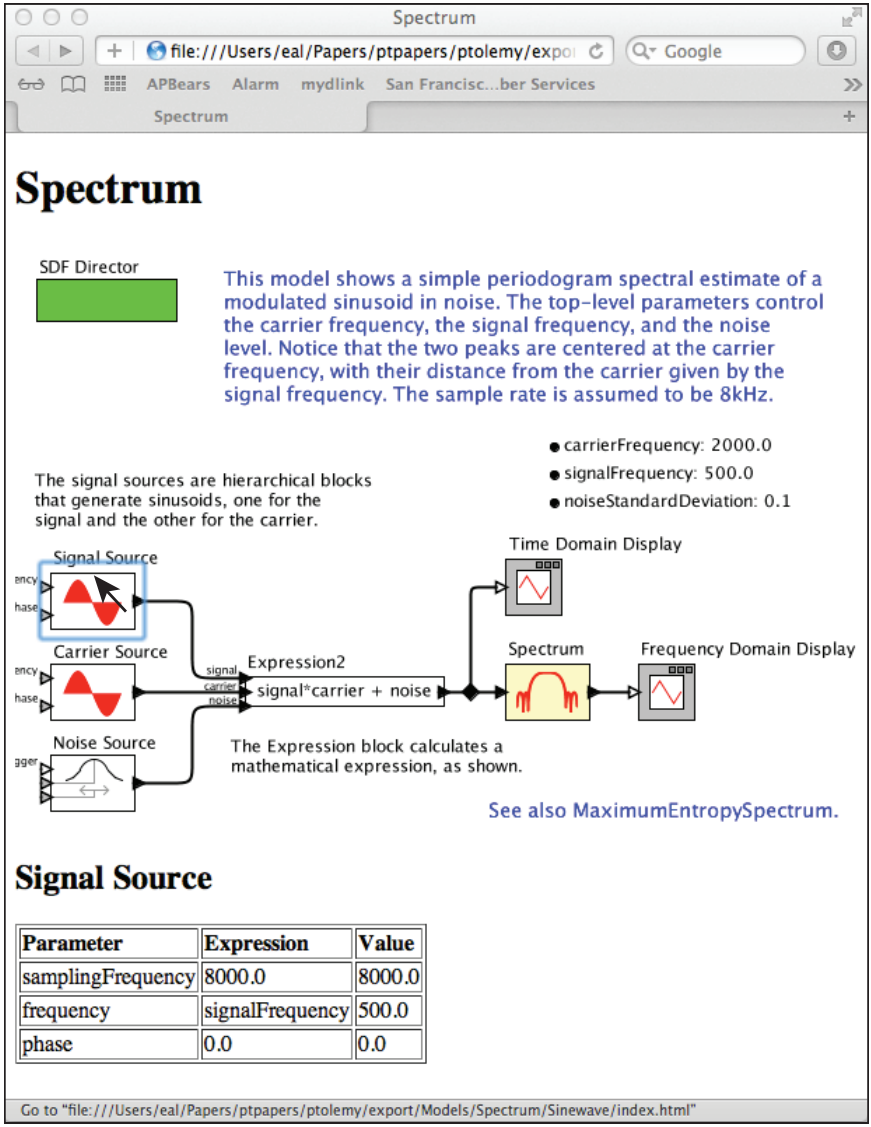


Figure 16.2: Web page exported from the model shown in Figure 16.1.

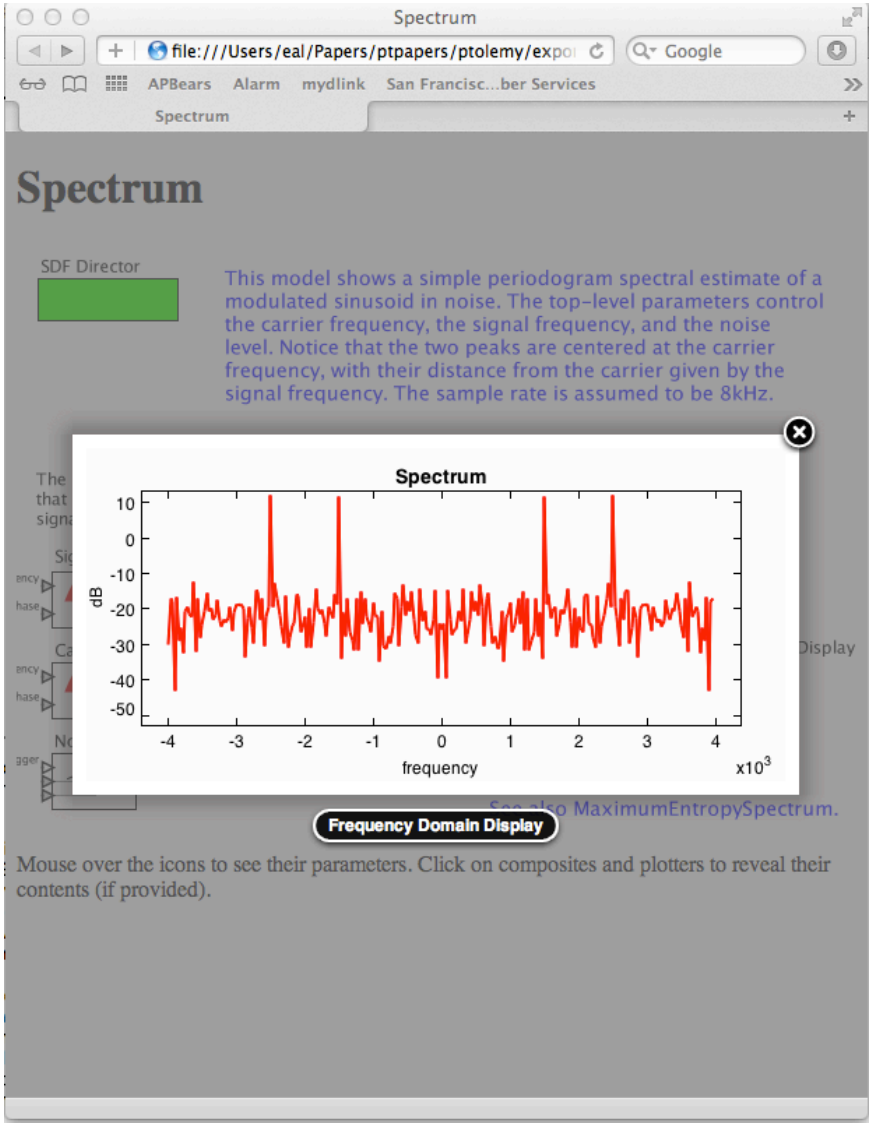


Figure 16.3: Clicking on the Frequency Domain Display actor in Figure 16.2 displays the plot generated by running the model.

HTMLText: Adding Text to Web Pages

The *HTMLText* attribute inserts HTML text into the page exported by Export to Web. Drag the attribute onto the background of a model, as shown in Figure 16.4, and double click on its icon to specify the HTML text to export. To specify the text to include in the HTML page, double click on the icon for the *HTMLText* attribute (which by default is a textual icon reading "Content for Export to Web"), as shown in Figure 16.6. You can type in the text to export, including any HTML content you like such as hyperlinks and

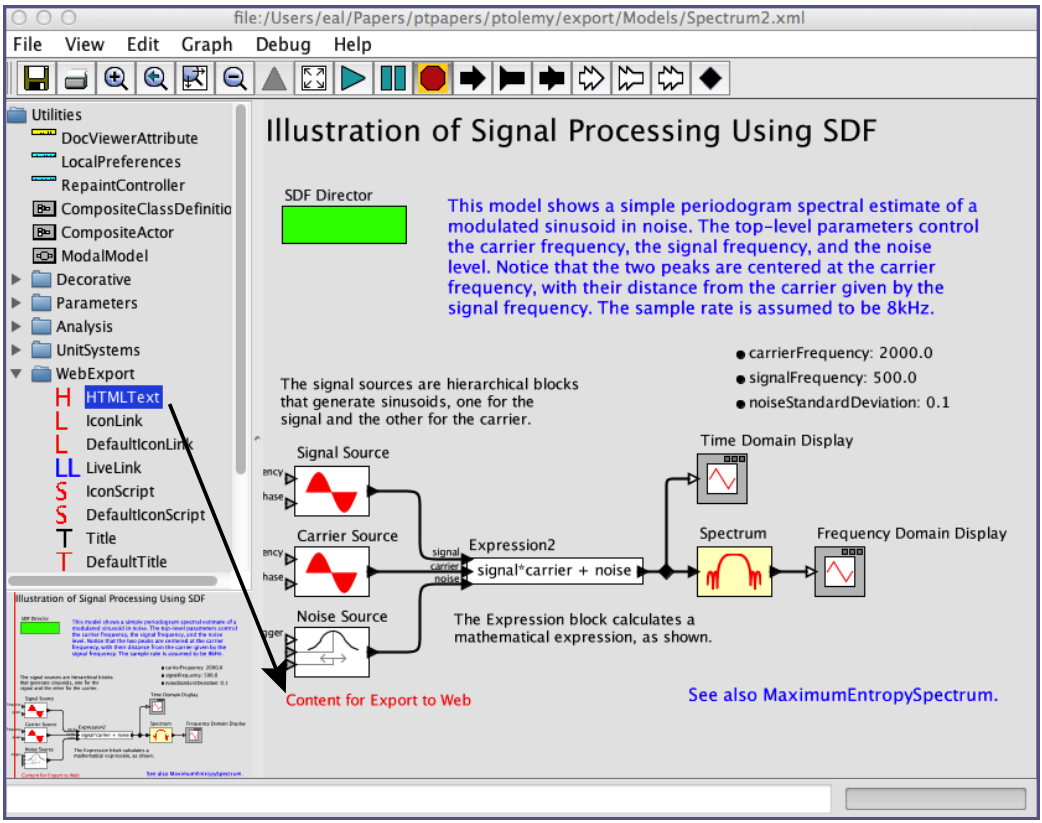


Figure 16.4: The `Utilities→Web Export` library provides attributes that, when dragged into a model, customize the exported web page. [online]

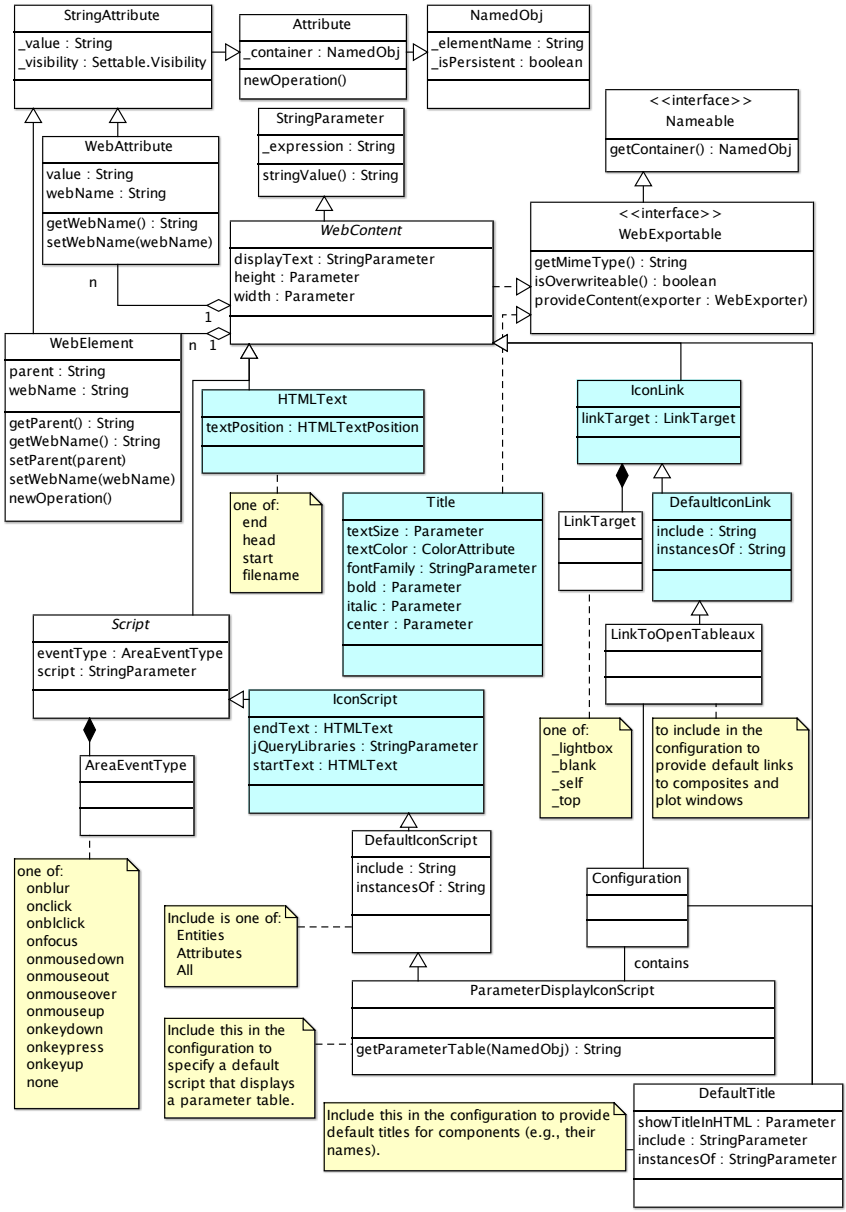


Figure 16.5: UML class diagram for the attributes for customization of exported web pages. The shaded attributes are the most commonly used in models.

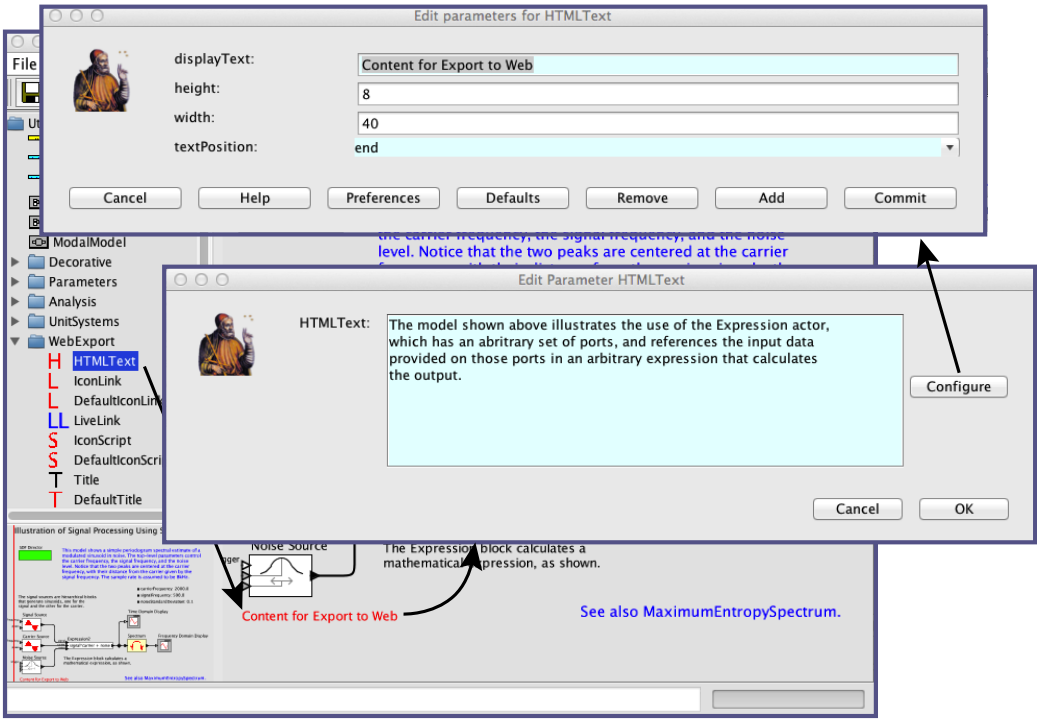


Figure 16.6: Dialog for customizing HTML text to include in an exported web page.

formatting directives. The web page including the text specified in Figure 16.6 is shown in Figure 16.7.

By default, this text will be placed before the image for the model, but you can change the position by setting the *textPosition* parameter, as shown in Figure 16.6. In that figure, you can see that the *HTMLText* attribute is configured to put the text at the end of the HTML file, which explains why that text appears at the bottom of the page in Figure 16.7.

The *HTMLText* attribute has several options for customizing it:

- *displayText*: This parameter determines what shows up in the model itself. By default, this is the text "Content for Export to Web." Notice that this text also appears in the exported web page in Figure 16.7, which is a bit odd. This text is not an interesting part

of the model; it is simply a placeholder for an attribute that customizes the exported web page. If you do not want this attribute to show up in an exported web page, you can simply move the attribute out of the field of view before doing the export. Alternatively, you can set *displayText* to an empty string, but this technique has the disadvantage of making it slightly more difficult to find the attribute to edit or customize the exported text. In Figure 16.8, the *displayText* has been set to the empty string. The *HTMLText* parameter is still present and can be selected (the small yellow box that is barely visible at the lower left in the figure is the *HTMLText* parameter), but since there is no visible icon, it is hard to find. An easier way to edit the *HTMLText* parameter is to right click on the background of the model, as shown in Figure 16.8. The *HTMLText* parameter appears as a parameter of the model, along with whatever other parameters have been defined in the model.

- *height*: The height of the editing box for specifying the text to export. If you change this value, close and re-open the dialog for the change to take effect.
- *width*: The width of the editing box for specifying the text to export. If you change this value, close and re-open the dialog to see the change.
- *textPosition*: As mentioned above, this parameter determines the position of the exported text. The built-in options are end, start, and head. Choosing "end" puts the text after the exported model image. Choosing "start" puts the text before the exported model image. Choosing "head" puts the text in the header section of the HTML page. If you specify any other value for *textPosition*, then that value is assumed to be the name of a file, and a file with that name is created in the same directory as the export. The specified text is then exported to that file.

IconLink: Specifying Hyperlinks for Icons

The *IconLink* parameter shown in the `Utilities→WebExport` library can be used to specify a hyperlink for an icon in the model. To use it, drag it from the library onto the icon that you would like to have the link. In the example of Figure 16.9, we have done such a drag onto the text annotation shown at the lower right that reads "See also MaximumEntropySpectrum." Double clicking on the text annotation reveals an *IconLink* parameter that can be set to a URL. The exported web page will include a hyperlink from the text annotation to that specified page.

The *IconLink* parameter can be customized (click on the Configure button at the lower right of the dialog in Figure 16.9). The parameters *displayText*, *width*, and *height* are the

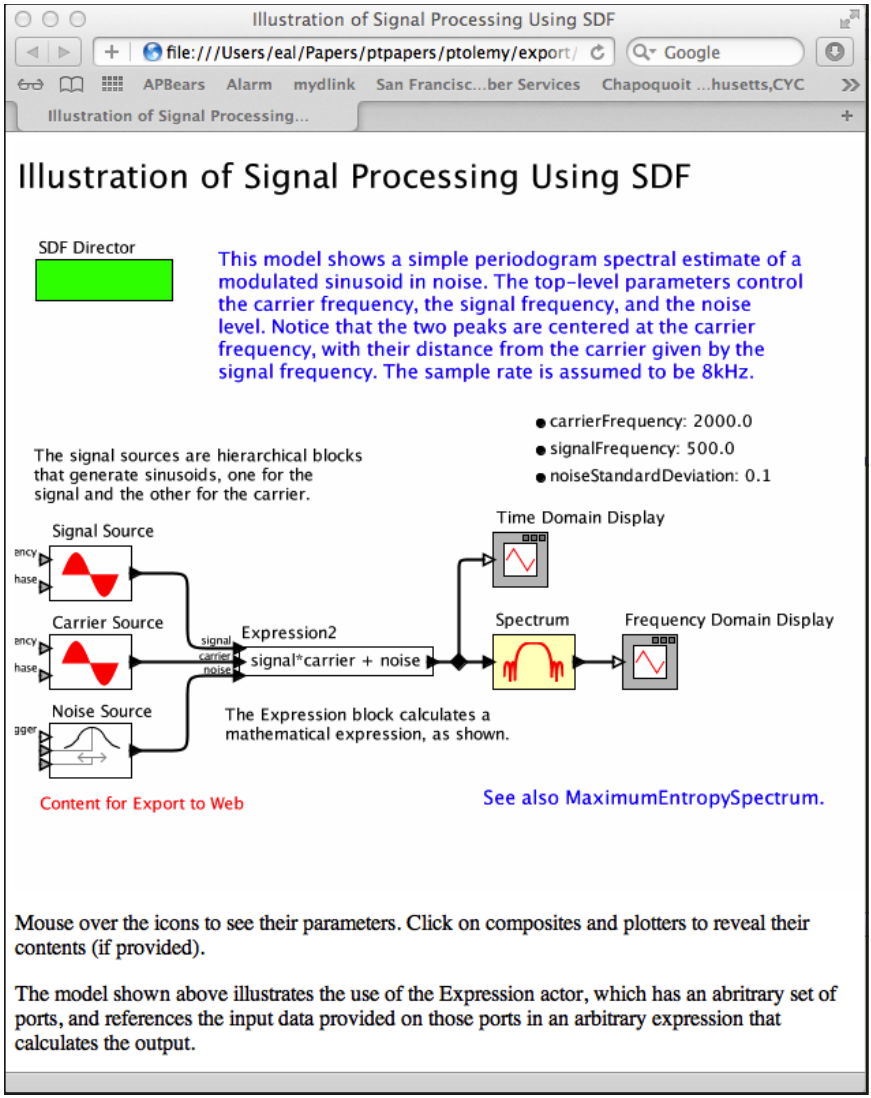


Figure 16.7: Page resulting from inserting an HTMLText attribute into the example of Figure 16.4 and configuring it as shown in Figure 16.6.

same as those for *HTMLText*, described above. A new parameter is *linkTarget*. This has four allowed values:

- `_lightbox`: Display the link in a pop-up lightbox.
- `_blank` (the default): Display the link in new blank window of the browser.
- `_self`: Display the link in the same window, replacing the current page or frame.
- `_top`: Display the link in the same window, replacing the current page.

An example of the lightbox display is the plot shown in Figure 16.3.

In addition, if the *linkTarget* parameter is given any other value, then that value is assumed to be the name of a frame in the web page, and that frame becomes the target.

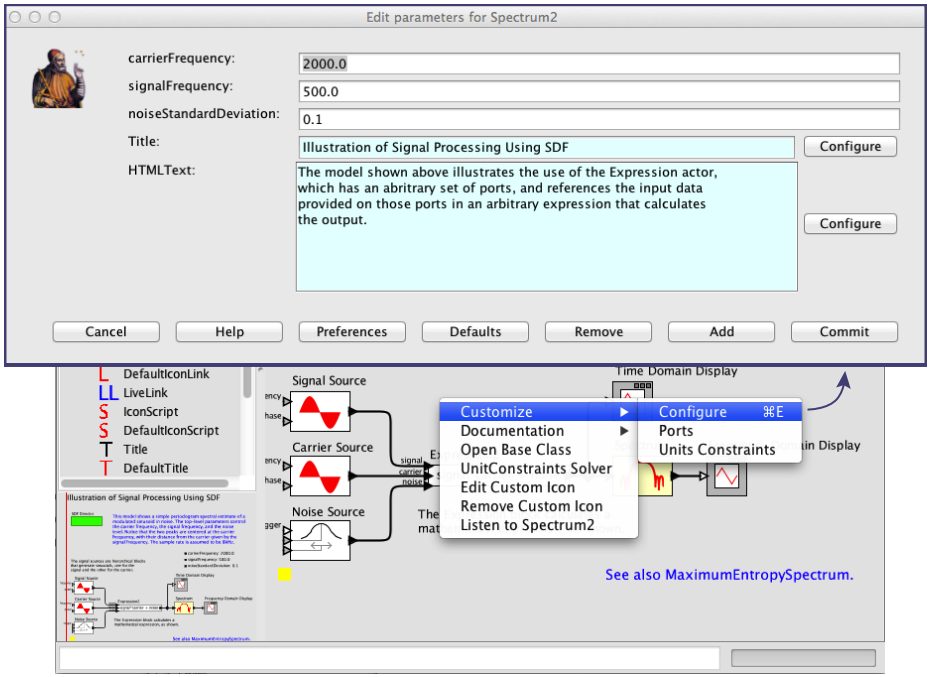


Figure 16.8: The *HTMLText* attribute can be hidden by setting its *displayText* parameter to the empty string. It can still be edited by right clicking on the background of the model. Notice that *HTMLText* appears in the list of model parameters.

DefaultIconLink: Default Hyperlinks for Icons

The *DefaultIconLink* parameter shown in the `Utilities→WebExport` library on the left in Figure 16.4 can be used to specify a default hyperlink for any icon in a model that does not contain an *IconLink*. In addition to the parameters of *IconLink*, *DefaultIconLink* has two additional parameters:

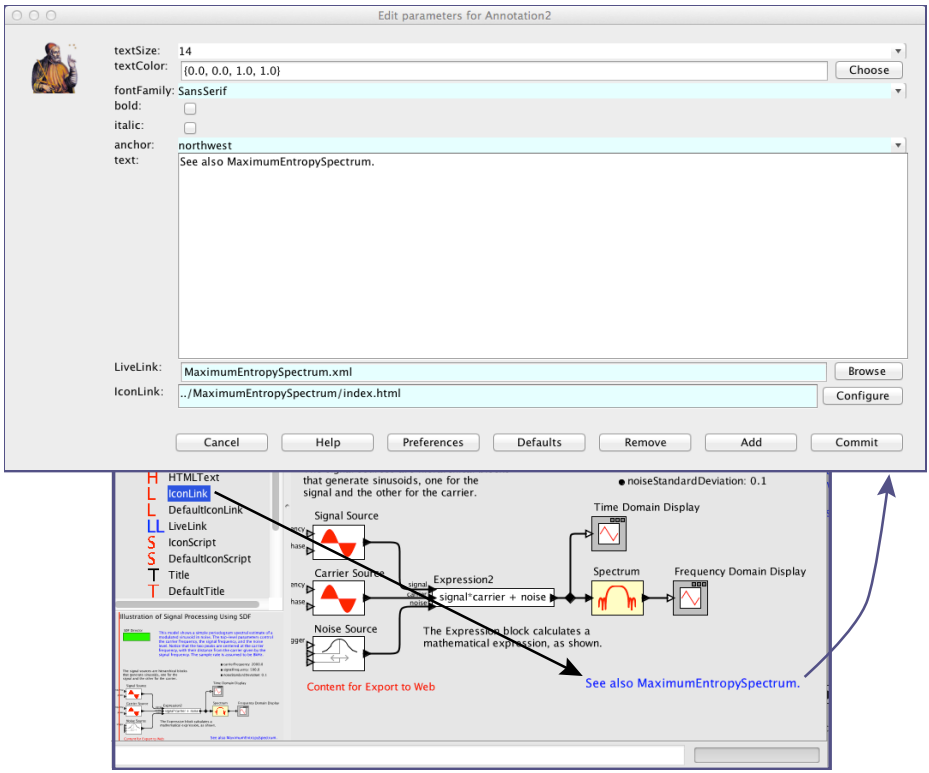


Figure 16.9: The *IconLink* attribute can be dragged onto an icon. The object onto which it is dragged acquires a parameter that can be used to specify a web page to link to from that icon when the model is exported to the web. Here, the exported web page will have a link on this icon to "../../MaximumEntropySpectrum/index.html".

- *include*: This parameter can be used to restrict icons to which the default applies. Specifically, the defaults may be specified for icons for attributes, entities, or both.
- *instancesOf*: If non-empty, this attribute specifies a class name. Only entities or attributes (depending on the *include* parameter) implementing the specified class will be assigned the default link.

LiveLink: Hyperlinks in Vergil

Although not directly related to web page exporting, the *LiveLink* parameter is included in the library because it works particularly well with *IconLink*. In particular, if you drop an instance of *LiveLink* onto an icon, then you can specify a file or URL to be opened when a user double clicks on the icon in Vergil (vs. clicking on an icon in a browser showing the exported web page). This does not automatically result in a hyperlink in an exported web page because typically a model will want to specify a different file or URL to be opened by Vergil than what would be opened by a browser. Vergil can open and display MoML files, for example, whereas a browser will simply display the XML content.

Example 16.1: Notice that in Figure 16.9, the annotation that reads "See also MaximumEntropySpectrum" contains both an instance of *IconLink* and an instance of *LiveLink*. The *LiveLink* references a MoML file, MaximumEntropySpectrum.xml, assumed to be stored in the same directory as the Spectrum model. The *IconLink* parameter, however, references an HTML file. That reference assumes that both Spectrum and MaximumEntropySpectrum will have exported web pages, and that the relative locations of these pages on a server are such that the specified path will provide a link to the HTML file for the MaximumEntropySpectrum.

Assuming all files are arranged appropriately in the file system, the Vergil hyperlink and the web page hyperlink will do essentially the same thing. They will each open the referenced model, MaximumEntropySpectrum. But Vergil will open it in Vergil, whereas a browser will open its exported web page in the browser.

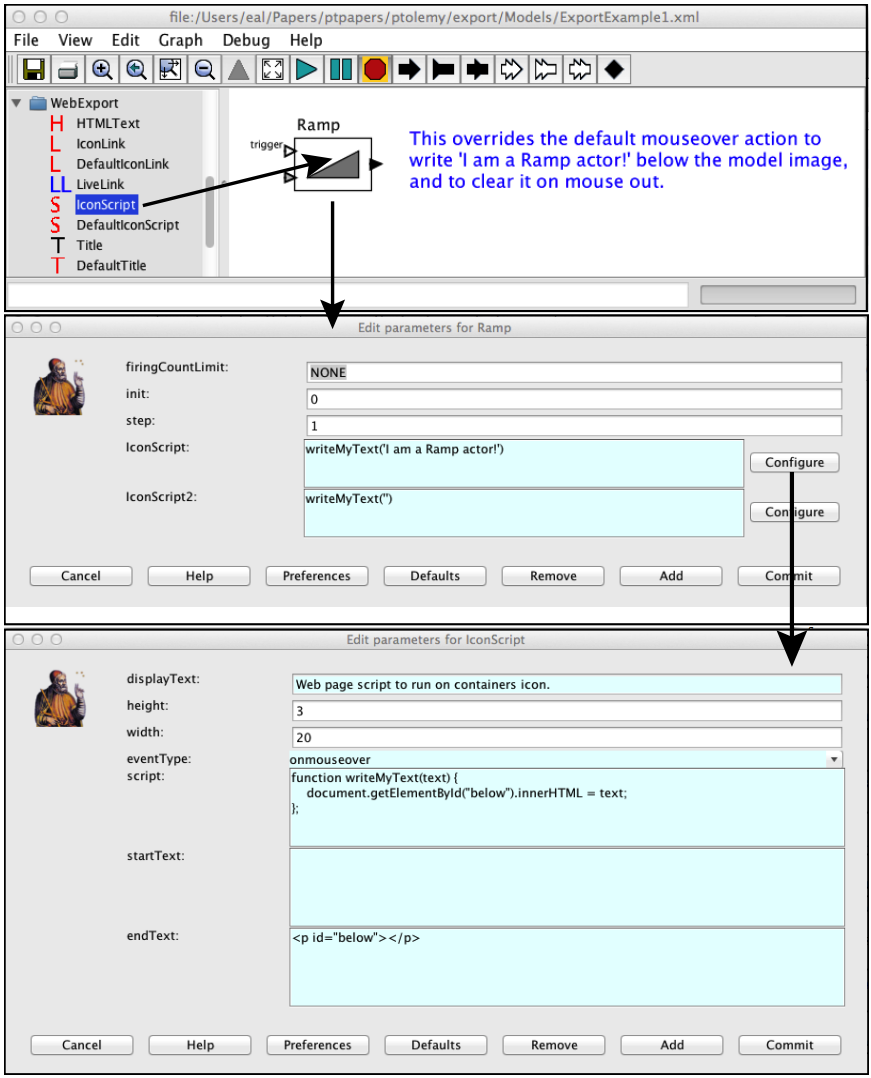


Figure 16.10: Here, two instances of the *IconScript* parameter have been dragged onto the icon for a Ramp actor. These parameters have been customized to display “I am a Ramp actor!” when the mouse enters the icon on the exported web page, and to clear the display when the mouse leaves the icon, as shown in Figure 16.11.

IconScript: Scripted Actions for Icons

The *IconScript* parameter is used to provide a scripted action associated with an icon in a model. Specifically, an action can be associated with mouse movement over the icon, mouse clicks, or keyboard actions. The action is specified as a JavaScript script.

Example 16.2: An example using *IconScript* is shown in Figures 16.10 and 16.11. In this example, two instances of the *IconScript* parameter have been dragged onto the icon for a Ramp actor. These parameters have been customized to display "I am a Ramp actor!" when the mouse enters the icon on the exported web page, and to clear the display when the mouse leaves the icon, as shown in Figure 16.11.

The way that this works is that the value of the first *IconScript* parameter is the JavaScript code:

```
writeMyText('I am a Ramp actor!')
```

This invokes a JavaScript procedure `writeMyText`, which is defined in the *script* parameter of the *IconScript* parameter to be:

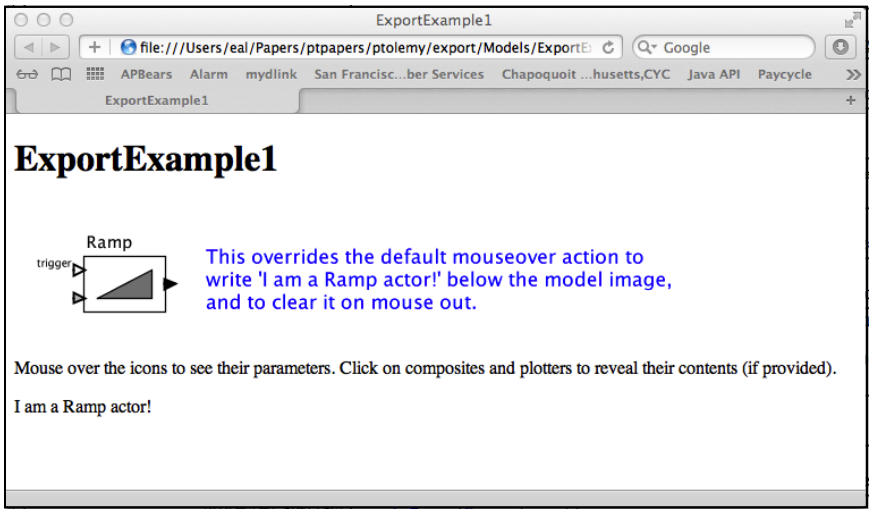


Figure 16.11: Web page exported by the model in Figure 16.11, shown with the mouse lingering over the Ramp icon.

```
function writeMyText(text) {
    document.getElementById("below").innerHTML = text;
};
```

This procedure takes one argument, `text`, and writes the value of this argument into the `innerHTML` field of the element with ID `below`. That element is defined in the *endText* parameter of the *IconScript* parameter as follows:

```
<p id="below"></p>
```

This is an HTML paragraph with ID `below`. This paragraph will be inserted into the exported web page below the model image. Finally, the *eventType* parameter of the *IconScript* is set to `onmouseover`, which results in the script being invoked when the mouse enters the area of the web page displaying the Ramp icon, as shown in Figure 16.11.

The second instance of *IconScript*, named IconScript2, specifies the following script:

```
writeMyText('')
```

This uses the same JavaScript procedure to clear the display when mouse exits the Ramp icon. The *eventType* parameter of this second *IconScript* is set to `onmouseout`.

If multiple instances of *IconScript* have exactly the same *script* parameter, then the value of that parameter will be included only once in the head section of the exported HTML page. Hence, the value of the *script* parameter is required JavaScript definitions. The web page exporter is smart enough to include those definitions only once if they are required at least once in the model.

DefaultIconScript: Default Scripted Actions for Icons

DefaultIconScript is similar to *IconScript*, except that it gets dragged onto the background of a model rather than onto an icon, and it specifies actions for many icons instead of just one. It has the same parameters as *IconScript*, but like *DefaultIconLink* described above,

it also has *include* and *instancesOf* parameters, which have the same meaning described above in Section 16.1.1.

DefaultIconScript can be used, for example, to override the default behavior that causes parameters to be displayed on mouse over, as shown in Figure 16.2.

Title: Title for Icons

The *Title* parameter is used to customize the title displayed in a web page. This parameter also appears as a title in the Vergil window. The title in Figure 16.7 is actually given by an instance of *Title* inserted into the model, with the default title changed to read "Illustration of Signal Processing Using SDF." This replaces the default title provided by the web export, which is the name of the model. This title also becomes the title defined in the header of the exported HTML file.

The default value of the *Title* parameter is the expression

```
$(this.getName())
```

which is an expression in the Ptolemy II expression language for string parameters (see Chapter 13). This expression invokes the `getName` method on the container object, so the default title that is displayed is the name of the model.

DefaultTitle: DefaultTitle for Icons

The *DefaultTitle* parameter is used to customize the title associated with each icon in a model. This title is what shows up on the exported web page as a tooltip when the mouse lingers over an icon. Like *DefaultIconLink* described above, it also has *include* and *instancesOf* parameters, which have the same meaning described above in Section 16.1.1. These can be used to specify default titles for subsets of icons.

16.2 Web Services

Ptolemy allows models to be run as web services. A **web service** runs on a server and is accessible on the Internet via a uniform resource locator (**URL**). Typically, a web service responds to requests by providing either a web page (typically formatted in **HTML**, the

hypertext markup language) or by providing data in some other standard Internet format such as **XML** (the extensible markup language) or **JSON** (the JavaScript object notation). The standard Ptolemy II library includes an attribute that turns a model into a web server, an actor to respond to HTTP requests, actors that facilitate constructing an HTML response, and actors for a model to access and use a web service.

16.2.1 Architecture of a Web Server

Figure 16.12 illustrates the operation of a web server. The URL for accessing the web server consists of the protocol, host name and port number (if other than the default, 80). For example, the URL `http://localhost:8078/` sends an HTTP request to the web server running on the local machine at port 8078.

A web server hosts one or more web applications (or web services). In our case, each application will be realized by one Ptolemy II model containing an instance of the WebServer attribute (see box on page 584). Each application registers an **application path** with the server. The application will handle an HTTP request for URLs that include the application path immediately after the hostname and port. For example, if application 2 registers the application path `/app2`, then the URL `http://localhost:8078/app2` will be handled by application 2. The application path can be the empty string, in which case all HTTP requests to this host on this port will be delegated to the

Sidebar: Command-Line Export

Given a MoML file for a model, you can generate a web page using a command-line program called `ptweb`. The command should have the following form:

```
ptweb [options] model [targetDirectory]
```

The "model" argument should be a MoML file. If no target directory is specified, then the name of the model becomes the name of the target directory (after any special characters have been replaced by characters that are allowed in file names). The options include:

- `-help`: Print a help message.
- `-run`: Run the model before the web page is exported, so that plot windows are included the export.

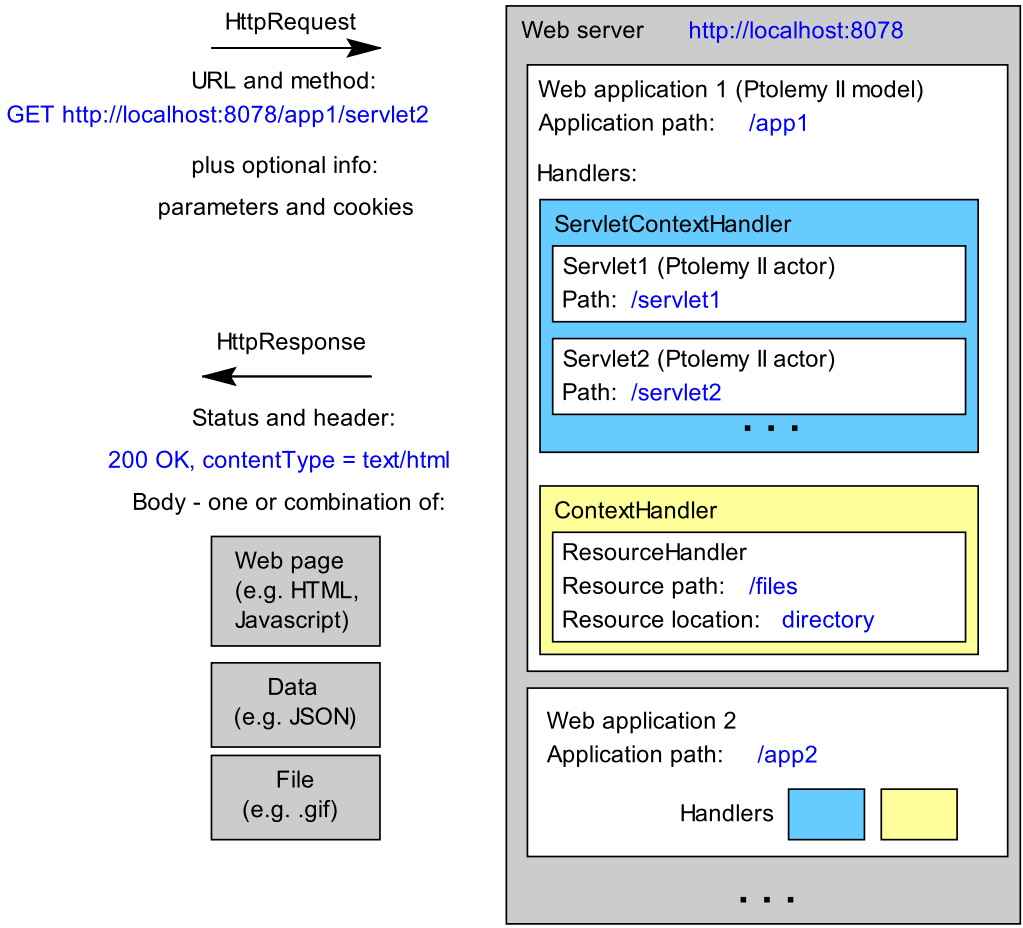


Figure 16.12: A web server hosts one or more web applications. Each application contains one or more request handlers. The web server receives HttpRequests and, according to the URL of the request, delegates the request to the appropriate request handler. The handler returns an HttpResponse.

application. When multiple applications are running on the same server, each application should have a unique application path prefix, so that the server can determine where to delegate requests.

Each application contains one or more request handlers. In our case, these handlers can be instances of the HttpActor actor (see box on page 584). Each handler registers a path prefix with the web application (this path prefix can again be the empty string). For HttpActor, the path prefix is given by the *path* parameter. When multiple handlers are running in the same application, each should have a unique path prefix, so that the application can determine where to delegate the request. For example, in Figure 16.12, the URL `http://localhost:8078/app1/servlet2` will be handled by application 1, which will delegate it to a Ptolemy II actor that has registered the prefix `servlet2`. If more that one prefix matches, then the server will delegate to handler with the most specific prefix. For example, if one handler has a blank prefix and the other has the prefix `/foo`, then all requests of the form `http://hostname:port/applicationPath/foo/...` will be delegated to the second handler, and all other requests to the first.

A second type of handler called a **resource handler** is also provided by the WebServer to handle requests for static resources such as files (Jetty class `ResourceHandler`). Again, this has a prefix which must appear in the URL. For example, in Figure 16.12, the URL `http://localhost:8078/app1/files/foo.png` references a file named `foo.png` that is stored on the server in a directory identified by a resource location attribute of the WebServer.

The response produced by a handler contains a status code, header, and the response body. The response body is the content for the user, for example, a web page, a file, or data formatted in JSON. The status code indicates whether the operation was successful, and if not, why not. There is a standard set of response codes for HTTP requests[†] The header contains information such as the content format (the **MIME type**[‡], the content length, and other useful information.

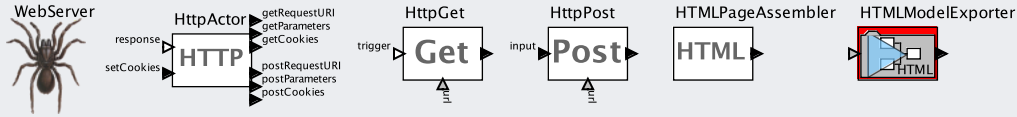
16.2.2 Constructing Web Services

The use of the WebServer and HttpActor are illustrated by the following example.

[†]See `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`.
[‡]See `http://www.iana.org/assignments/media-types/index.html`.

Sidebar: Components for Web Services and Web Pages

Some components that are particularly useful for constructing web services, accessing web pages, and building web pages are shown below:



- **WebServer**. An attribute that starts a Jetty web server (see `http://www.eclipse.org/jetty/`) when the model containing it is executed. This attribute routes incoming HTTP requests to objects in the model that implement an HttpService interface, such as HttpActor. This attribute has parameters for specifying the port on which to receive HTTP requests, an application path to be included in the URL accessing this server, directories in which to find resources that are requested, and a directory in which to store temporary files.

- **HttpActor**. An actor that handles **HTTP GET** and **HTTP POST** requests that match its path. This actor is designed to work with the DE director. The outputs contain the details of the request time stamped by the elapsed time (in seconds) since the server model started executing. This actor expects that for each output it produces, the model in which it resides will provide an input that is the response to the HTTP request.

- **HttpGet**. An actor that issues an HTTP GET request to a specified URL. This actor is similar to FileReader, but it only handles URLs, and not files.

- **HttpPost**. An actor that issues an HTTP POST request to a specified URL. The contents of the post are specified by an input record.

- **HTMLPageAssembler**. An actor that assembles an HTML page by inserting input text at appropriate places in a template file.

- **HTMLModelExporter**. An extension of the VisualModelReference actor that not only displays and executes a referenced model, but also exports that model to a web page using the techniques discussed in Section 16.1.

Example 16.3: The model in Figure 16.13 is a web service that asks the user to type in some text, then returns the "Ptolemnized" text, where all leading 'p's (but not including instances of 'th') are replaced with 'pt'. For example, 'text' becomes 'ptext', as shown in Figure 16.14. The text manipulation is accomplished by the PythonScript actor, which executes the Python code shown in Figure 16.15.

First, notice that the *stopWhenQueueIsEmpty* parameter of the DE director is set to false. Were this not the case, the model would halt immediately when run because there would be no pending events to process. Second, notice that the *enable-*

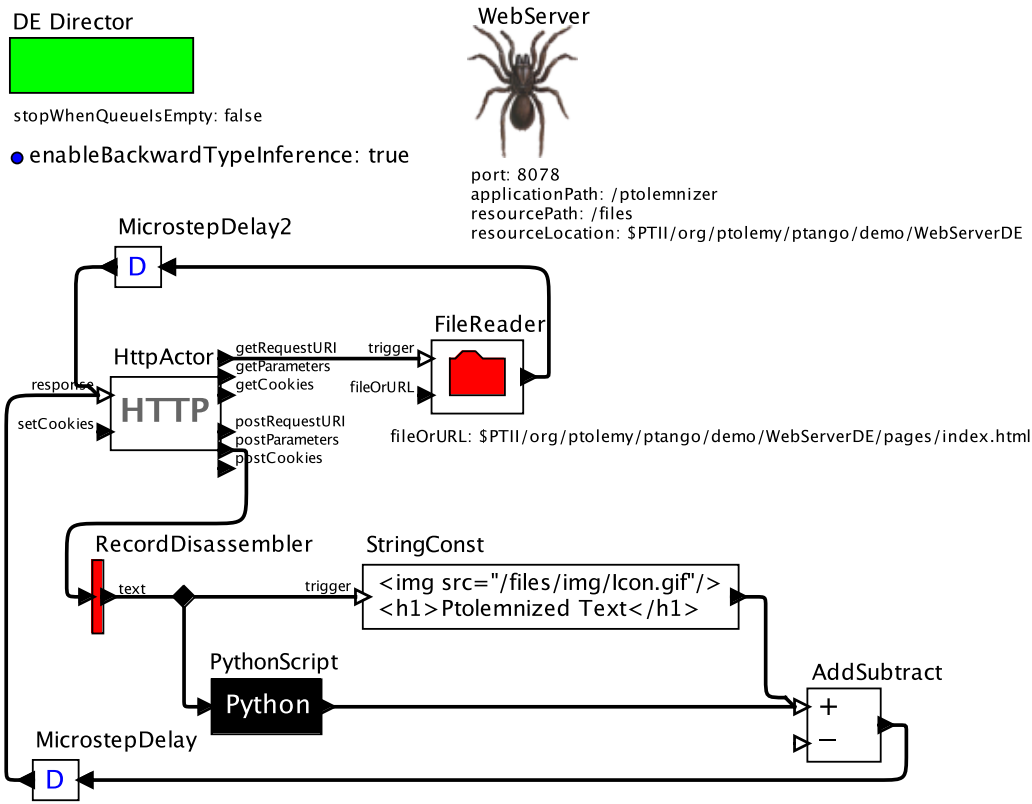


Figure 16.13: A simple web service implemented in Ptolemy II. [online]

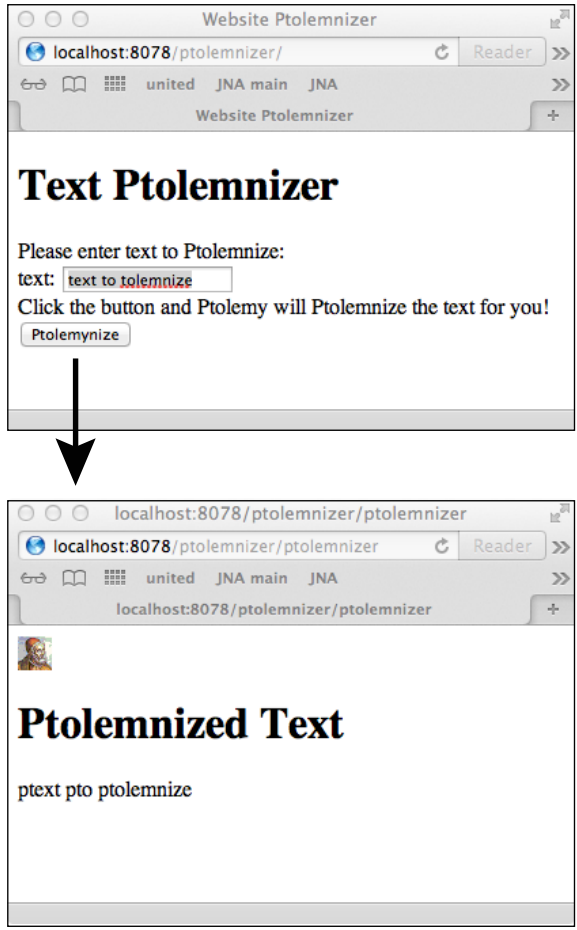


Figure 16.14: The web page returned by model in Figure 16.13 in response to an HTTP GET, and the page returned in response to a POST (triggered by the button).

```
from ptolemy.data import StringToken
class Main :
  "ptolemizer"
  def fire(self) :
    # read input, compute, send output
    t = self.in.get(0)
    s = t.stringValue()
    s = self.ptolemize(s)
    t = StringToken(s)
    self.out.broadcast(t)
    return

  def ptolemize(self, s) :
    l = list(s)
    length = len(l)
    if length == 0 :
      return ''
    if length == 1 :
      if l[0] == 't' :
        return 'pt'
      else :
        return l[0]
    if l[0] == 't' and l[1] != 'h' :
      l[0] = 'pt'
    i = 1
    while i < length - 1 :
      if l[i-1] == ' ' and l[i] == 't' and l[i+1] != 'h' :
        l[i] = 'pt'
      i = i + 1
    if l[-2] == ' ' and l[-1] == 't' :
      l[-1] = 'pt'
    return reduce(lambda x,y: x+y, l, '')
```

Figure 16.15: The Python code for the PythonScript actor in Figure 16.13.

BackwardTypeInference parameter of the model is set to true. This enables backward type inference, which in this case, results in the *postParameters* output of the HttpActor to have a record type with a single field called "test" of type string. The PythonScript actor specifies the type *string* on its input port, because the Python code expects a string.

When this model is executed, the WebServer launches a web service with an application path of `/ptolemnizer` on port 8078 of the local machine. The service is therefore available at `http://localhost:8078/ptolemnizer`. Accessing that URL in a web browser results in the top web page of Figure 16.14. How does this work?

When the web server receives an HTTP GET request with a matching application path, it delegates the request to the HttpActor. The actor requests of the director to be fired, and when the director fires it, it produces information about the GET request on its top three output ports. This model uses the URL of the GET request to trigger the FileReader actor, which simply reads a file on the local file system, the contents of which are shown in Figure 16.16. The contents of that file are sent back to the *response* input of the HttpActor, which then fires again. On that second firing, it collaborates with the WebServer to serve the response shown at the top of Figure 16.14. Note that MicrostepDelay actor is required in the feedback loop, as usual for DE models (see Section 7.3.2).

As you can see in Figure 16.14 and Figure 16.16, the web page that is served has a form, and pushing the "Ptolemnize" button results in an HTTP POST with the contents of the form. When this POST occurs, the WebServer again delegates to the HttpActor, which outputs the details of the POST on its lower three output ports. The *postParameters* port will produce a record token with a single field called "text." The RecordDisassembler extracts the value of this field, which is the text entered by the user into the form. The StringConst, PythonScript, and AddSubtract actor then construct an HTML response, which is sent back to the HttpActor. That response results in the page at the bottom of Figure 16.14.

The response to the POST includes an "img" element (see the StringConst actor in Figure 16.13). When the browser parses this response, this img element will trigger another HTTP GET. The WebServer has its *resourcePath* parameter set to `/files`, so the img src URL `/files/img/Icon.gif` will be handled by the resource handler rather than being delegated to an HttpActor (see Figure 16.12). That resource handler will search for a file named `img/Icon.gif` in the directory

given by the *resourceLocation* parameter. The small Ptolemy icon on the bottom page of Figure 16.14 is the result.

This example constructs a web service by composing a number of capabilities. It uses HTML to construct an interactive web page, and Python to process data submitted by a user. In effect, the Ptolemy model is serving as an orchestrator for a number of distinct software components.

16.2.3 Storing Data on the Client using Cookies

A **cookie** is small piece of data — specifically a (name, value) pair plus expiration and visibility information — that is stored by a web browser on the client side and returned to the web server along with subsequent HTTP requests. A web service can store state on the client using a cookie; for example, a web service can use a cookie to remember that the user has logged in. A **persistent cookie** is stored for a specified period of time (including indefinitely), whereas a **session cookie** is only stored until the browser window is closed.

HttpActor has basic support for getting and setting session cookies from a client browser. Specifically, HttpActor has a *requestedCookies* parameter whose value is an array of strings. This specifies the names of cookies that the web service sets or gets. It also has an input port *setCookies*, which accepts a record that assigns values to each of the named cookies. Finally, it has output ports *getCookies* and *postCookies* that provide a record with cookie values along with each HTTP GET or POST request.

Example 16.4: The model shown in Figure 16.17 uses cookies. The web service that this model implements uses cookies to remember the identity of a client over a sequence of HTTP accesses. The pages shown in Figure 16.18 illustrate how the service responds to an initial HTTP GET, an HTTP POST that stores the identity of a client "Claudius Ptolemaeus" as a cookie, a subsequent HTTP GET, and finally, an HTTP POST that deletes the cookie.

The model has two instances of HttpActor. The first one, labeled HttpActor1, has the default *path* parameter, which matches all requests. The second one, labeled

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <title> Website Ptolemnizer </title>
</head>
<body>

<div data-role="page" data-theme="c">
  <div data-role="header">
    <h1> Text Ptolemnizer </h1>
  </div>
  <div data-role="content">
    Please enter text to Ptolemnize:
    <form action="ptolemnizer" method="post" >

      <div data-role="fieldcontain" class="ui-hide-label">
        <label for="text">text:</label>
        <input type="text" name="text" id="text" value=""
                  width="80" placeholder="text to tolemnize"/>
        </br>
      </div>

      <div>
        Click the button and Ptolemy will
        Ptolemnize the text for you!
        <br/>
        <button type="submit" id="ptolemnize">
          Ptolemynize
        </button>
      </div>
    </form>
  </div>
</div>
</body>
</html>
```

Figure 16.16: The HTML code read by the FileReader actor in Figure 16.13.

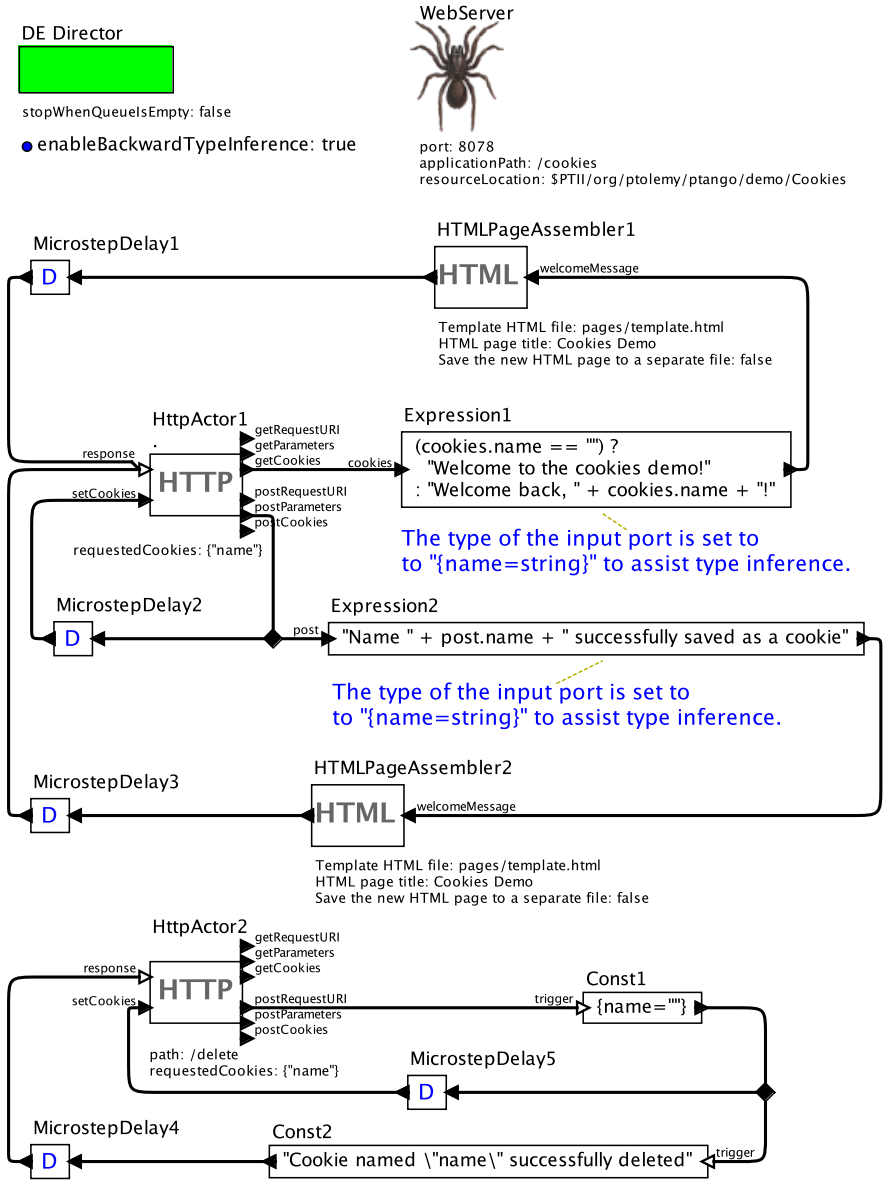


Figure 16.17: A model that gets, sets, and deletes a cookie on the client. [online]

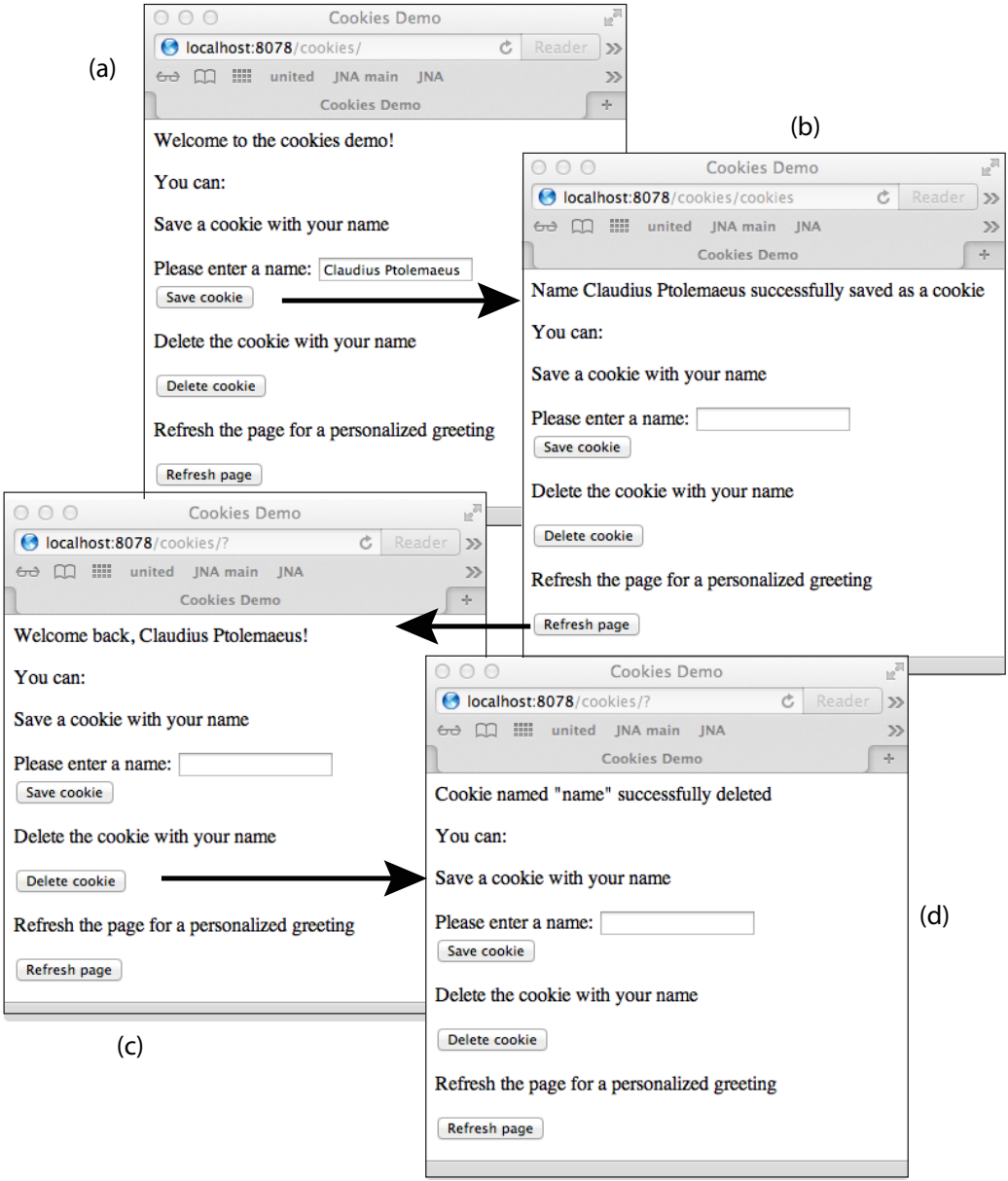


Figure 16.18: A sequence of web pages created by the model in Figure 16.17.

HttpActor2, has *path* set to `/delete`, so it will handle requests with URLs of the form `http://localhost:8078/cookies/delete`.

Both instances of HttpActor have parameter *requestedCookies* set to `{"name"}`, and array with one string. This instructs the HttpActor to check the incoming HTTP request for a cookie with the label `name`. The HttpActor produces a record on its *getCookies* or *postCookies* output port with the label `name` and the value provided by the cookie. If no cookie is found, the value is an empty string.

Note that an HttpActor actor always produces a record with the fields specified in *requestedCookies*, so downstream actors can always assume a record with the specified field. Hence, for example, the Expression actor named Expression1 in Figure 16.17 extracts the `name` field of the record using the syntax `cookies.name`. If value of the field is an empty string, then the model generates a generic welcome message, as shown in Figure 16.18(a). Otherwise, it customizes the page, as shown in Figure 16.18(c).

From the initial page, Figure 16.18(a), the user can specify a name and save a cookie with the name, which yields the response Figure 16.18(b). This is accomplished using an HTTP POST with parameter `name`. Notice in Figure 16.17 that the *postParameters* output port is fed back to the *setCookies* input port, so the response to this HTTP POST will be to set a cookie in the browser with whatever value is provided by the POST.

Clicking on the "Refresh page" button causes another HTTP GET, which now yields the customized page, Figure 16.18(c).

Clicking on the "Delete cookie" button sends a POST request to `http://localhost:8078/cookies/delete`. This request is mapped to HttpActor2. The response has two parts. First, Const1 sends a record with the label `name` and an empty string value to the *setCookies* port on HttpActor2. HttpActor2 interprets this as a request to delete the cookie. Note that, because of this implementation, the HttpActor actor will interpret any `RecordToken` label with an empty string value as a request to delete the cookie with that label. Hence, a missing cookie is equivalent to a cookie with an empty value. In addition, the model will generate a response confirming deletion of the cookie, Figure 16.18(d).

Assembling Web Pages

The model in Example 16.4 and Figure 16.17 serves some non-trivial web pages. To facilitate construction of these web pages, the model uses the HTMLPageAssembler actor. This actor inserts contents from its input ports into a specified template file, and outputs the resulting HTML page. The names of the input ports match HTML tag IDs in the template file.

Example 16.5: Figure 16.19 shows the HTML template referenced by the HTMLPageAssembler actors in Figure 16.17. Notice the **div** tag with ID "welcomeMessage." Notice further that the actors each have an input port named *welcomeMessage*, which has been added by the builder of the model. Whatever is received on this input port will be inserted into this div tag position in the response HTML page.

Note that the `Save cookie` and `Refresh page` buttons are HTML forms. These buttons perform the action specified when clicked. For example, the `Save cookie` button generates a POST request to the relative URL `cookies`, at `http://localhost:8078/cookies`, as specified by line 7. The `Refresh page` button generates a GET request to that same URL, as specified by line 24.

An alternative technique, also used in Figure 16.17, is to use JavaScript to update a page instead of returning a new page. This technique is known as **AJAX** (for asynchronous JavaScript and XML).

Example 16.6: The `Delete cookie` button calls the JavaScript function `deleteCookie()`, as shown on lines 17-18 of Figure 16.19. Figure 16.20 shows the `deleteCookie()` function definition. The function submits a POST request to the relative URL `cookies/delete`. If the request is successful, the response data are inserted into the HTML element with the ID `welcomeMessage` (overwriting any previous data). If the request is not successful, an error message is inserted into this element.

This example illustrates two reasons for using Ajax. First, returning a whole page is not necessary for the delete case. A simple message is sufficient. There are many cases

where a developer might want to insert a small update into a larger page. This promotes separation of concerns, where one developer could be responsible for the main page, and a second could be responsible for updates without having to know the structure of the rest of the main page. The second developer might also want to create a web service to provide data to many different pages.

```
    <body>
    <div>
        <div id="welcomeMessage">
        </div>

        <div> <p> You can: </p>  </div>
        <form accept-charset="UTF-8" action="cookies
                      method="post">
            <p> Save a cookie with your name </p>
            <p> Please enter a name:
                <input type="text" name="name" id="name"/>
                <br>
                <input type="submit" value="Save cookie"/>
            </p>
        </form>

        <div> <p> Delete the cookie with your name </p>
           <input type="button" value="Delete cookie"
            onclick="deleteCookie()"/>
        </div>

        <div> <p>
             Refresh the page for a personalized greeting
        </p> </div>

        <form name="input" action="/cookies" method="get">
            <input type="submit" value="Refresh page" />
        </form>

    </div>
    </body>
</html>
```

Figure 16.19: The HTML template referenced by the HTMLPageAssembler actors in Figure 16.17.

```
<!DOCTYPE HTML>
<html>
    <head>
        <script type="text/javascript"
          src="http://code.jquery.com/jquery-1.6.4.min.js">
        </script>
        <script type="text/javascript">
            function deleteCookie() {
                jQuery.ajax({
                      url: "/cookies/delete",
                      type: "post",
                      success: function(data) {
                          jQuery('#welcomeMessage')
                            .html(data);
                          },
                      error: function(data) {
                          jQuery('#welcomeMessage')
                            .html("Error deleting cookie.");
                      }
                    });
            }
        </script>
        <title>Cookies demo</title>

    </head>
```

Figure 16.20: The head section of the HTML template page used in Figure 16.17.

A more subtle reason for using Ajax is that the URL of the website remains unchanged, at `http://localhost:8078/cookies`, while still being able to use a URL structure for the delete web service, `cookies/delete`. If the URL were to change to `http://localhost:8078/cookies/delete`, this would cause problems when the user clicks on further buttons, because the button URLs are defined as relative URLs. E.g., the URL would then be `http://localhost:8078/cookies/delete/cookies`.

There are, of course, many other ways to create web pages to respond to HTTP requests. A particularly interesting possibility is to use the techniques covered in Section 16.1 above to generate web pages from Ptolemy II models. In fact, a web service model could include an instance of the HTMLModelExporter actor, which refers to another Ptolemy II model, executes it, generates a web page with the results, and returns the web page. This offers a particularly powerful way to combine models to provide sophisticated services.

16.3 Summary

Building web pages and web services by constructing models offers a potentially very powerful way to combine sophisticated components in a modular way. At a minimum, the ability to export a web page that documents a model is valuable, enabling teams of designers to more effectively communicate with one another. But more interestingly, the ability to incorporate web servers into models offers a particularly powerful way to combine distributed services.

Exercises

1. Figure 4.3, discussed in Example 4.3 of Chapter 4, implements a simple chat client that uses HTTP Get and HTTP Post to enable a client to chat with other clients on the Internet. In this exercise, we build a simple (and rather limited) web server that supports this client. This server will support exactly two clients, one that will use URL
   `http://localhost:8078/chat/Claudius`
   for its Get requests, and one that will use URL
   `http://localhost:8078/chat/Ptolemaeus`
   for its Get requests. Both will use the same URL,
   `http://localhost:8078/chat/post`
   to post chat data.

   (a) A key property of this server is that it must implement long polling, where it sits on an HTTP Get request until a chat client issues an HTTP Post, which provides some chat text, and then it responds to all clients that have pending Gets with the contents of the Post. To support this, create an actor-oriented class composite actor with two input ports, *get* and *post*, and one output port *response*. This class should queue a get request (at most one) and when a post arrives, if there is a pending get request in the queue, then it should respond with the contents of the post.

   (b) Use the class definition created in part (a) to build a web server that supports the two clients.

   (c) A limitation of the chat client in Figure 4.3 is that it does not stop gracefully. The stop button in the Vergil window eventually stops it, but not until the FileReader actor times out, which can take a long time. In a better design, the server would always respond to an HTTP Get request within some amount of time, given by parameter *maximumResponseTime*. It could respond with an empty string, and the client could then filter out empty strings so that it does not display them to the user. In this design, stopping the client will succeed within the *maximumResponseTime*. Modify your server and the client to implement this.

   (d) (Open-ended question) One of the limitations of the web server you have been asked to design is that only exactly two clients are supported. Another is that there is no authentication of clients. Discuss how to address these limitations,

and implement a more elaborate server that addresses at least one of these limitations.