
Dynamic and Interactive Documents - RDocbook Markup

Duncan Temple Lang

Table of Contents

Summary	1
Working with XML RDocbook	1
Graphics	6
SVG	6
Inline Graphics Content	8
Entities and XSL Stylesheet elements	8
Processing the Dynamic Document	11
XSL Files	11
Finding the XSL files	12
LaTeX output	13
Interactivity	15
i: elements	15
Generic i:object element	16
Multiple r:code References	17
The Update Button	17
Managing the Evaluation of Code nodes	18
Interactive Control Markup	19
Implementation	19
Type Annotations and Dependencies	22
Alternatives to wxWidgets	23

Summary

This document provides some details about how to write and process dynamic XML documents containing R code. The idea is to use an extended Docbook vocabulary for writing documents that can be rendered as HTML and PDF and processed programmatically in R and other languages to extract different elements. The document contains computer code which is evaluated and the results inserted into the document while it is being processed for rendering. This is the dynamic part. The second aspect of these documents is that they can be displayed as interactive HTML documents within an R-based browser. Support for these transformations and processing facilities is centralized in the *XDynDocs* package. This contains R functions to perform the processing and supporting XSL documents that provide the XSL rules for doing the transformations.

Working with XML RDocbook

We write a document using XML, and specifically an extended version of Docbook that has support for R-related elements. Docbook provides us with all the markup for writing technical documents such as sections, tables, figures, cross references, bibliographies. There is limited support for mathematical notation, but we

can use MathML for that (but processing MathML for rendering is more involved and incomplete). We add R code to the document where we want it to be evaluated and/or displayed using XML elements such as `r:code`, `r:plot`, `r:function`, `r:test`, `r:init`, `r:expr`, `r:options`, `r:func`, `r:var`, `r:pkg`. The purpose of each of these is described in table Table 1, “R-related XML elements for Docbook” (page 2).¹

Table 1. R-related XML elements for Docbook

Node	Description	Attributes
<code>r:code</code>	This identifies R code, typically to be executed. It supports and eval attribute which can have a value false or true to indicate whether it should be evaluated in the usual processing of a dynamic document. The code is formatted in the same manner as verbatim in LaTeX or PRE in HTML.	id, eval, r:output, i:display, i:update, r:capture.output - which evaluates the code and returns the output that was written to the console, i.e. using sink(). any R options, e.g. digits.
<code>r:commands</code>	takes a block of R code and parses it into expressions and then evaluates each expression in turn returning the code, preceded by a prompt, and the converted output as an XML node. The prompt can be specified on the node or picked up from the global XSL parameter or can be specified via the xslParam argument in R.	r:prompt - defaults to XSL top-level/global parameter r:prompt defined in dynamic.xsl
<code>r:function</code>	A code block that defines a function. This could define more than one function, but the intent is that we can easily evaluate the code to define a function by specifying just the id of this node.	id, eval
<code>r:plot</code>	The code creates a plot. This allows us to format the result properly in HTML, PDF, etc. by inserting an image from a file or creating a live graphics device. By using this element rather than a generic <code>r:code</code> , we can ensure that the plot is "printed" in the case of a lattice plot and we can manage the graphics devices automatically, e.g. create the file in which the plot will be saved and can cache the result to avoid regenerating a plot.	id, eval, i:update, i:display, r:width, r:height - in pixels (not inches), file - the name of the file to write the generated plot to (not used when creating an inline graphics device in interactive use), r:background - a color that is passed to the graphics device function as the <i>bg</i> argument. r:format - name of function

¹ Each of the nodes supports an `r:output` child which can contain a pre-computed display of the results which allows us to display these in cases where we do not want to evaluate the code. Similarly, one can specify an R object that is the result of the computation. This can be done via an attribute `r:result` whose value is a file name or URL identifying the location of the content. Alternatively, we can have an `r:result` child node whose contents are the serialized R objects, i.e. the save'd RDA, dump or dput format, suitably encoded. The `r:output` node supports an `r:format` attribute indicating the style of serialization.

Node	Description	Attributes
		that creates a graphics device, called with filename, r:width and r:height and r:background. r:inline - indicates to return the plot as an XML node, currently only meaningful for SVG. This allows us to avoid using auxiliary files. originalFile - gives the name of the file that contains the plot that is to be used if we don't run the code, i.e runCode = FALSE. This is the cached plot. It is different from file as when we do run the code, file will be overwritten whereas this one won't.
r:test	This is for code that validates the computations and ensures that things are working correctly. How this is processed depends on the options, but in some cases, we will have output from each expression within this element and in other cases we may suppress the output altogether but merely run this code and terminate the processing if it does not return TRUE .	id, eval
r:init	R code that is essentially initialization code that defines the inputs for the document or a section. This often identifies the important inputs that the reader might want to modify and have propagate to the subsequent code elements for evaluation.	id, eval
r:options	A node that specifies new values to be set via the <code>Roptions()</code> function. The values can be given via attributes, and in the future, will also be allowed as child nodes of this r:options element in the form <name> values </name> and values might be some rich content described in XML. For the most part, however, the values are strings or numbers and we can use XML attributes to specify the name = "value" pairs.	attributes are the arguments to <code>options()</code> .
r:frag	short-hand for fragment, this is intended to be used when the code in this element is referenced from another block of code and this code	

Node	Description	Attributes
	is not to be evaluated separately. This is the same as <code>r:code</code> with an attribute <code>eval</code> set to "false".	
<code>r:expr</code>	This is intended for use within regular text (i.e. not in a separate verbatim box) and is used to insert the value of an R expression. For example, we might have a sentence that says "there were <code><r:expr>nrow(failures)</r:expr></code> in the <code><r:expr>length(levels(failures\$month))</r:expr></code> months" and	<code>eval</code> , R-level options such as <code>digits</code> .
<code>r:value</code>	a node that contains one or more serialized R objects, often via <code>dput</code> but also as a binary RDA file that is then base64-encoded to be added to the XML document as text.	
<code>r:output</code>	This is intended to be a child of the <code>r:code</code> , <code>r:output</code> , <code>r:init</code> , <code>r:test</code> , etc. nodes and is used to include the output from the code in the associated node. This allows us to render the document without having to actually evaluate the code in the <code>r:*</code> nodes. So it is intended to capture the output from the original author's evaluation of the code. This is optional.	
<code>r:func</code>	reference to an R function. This is used to identify the reference to the function rather than having any impact on the evaluation of the R code. However, when we process a document within R, we can add information about the location of the help file, e.g. a link.	<code>pkg</code> , the name of the R package in which the function is located. This is optional.
<code>r:var</code>	referring to an R variable. Like <code>r:func</code> , this is used for rendering and not evaluation.	<code>name</code> to be used when we don't want the end <code></r:var></code> , e.g. <code><r:var name="source"/></code>
<code>r:arg</code>	a reference to an R function parameter.	<code>r:func</code> identifying the R function by name.
<code>r:pkg</code>	a reference to an R package.	
<code>omg:pkg</code>	a reference to an Omegahat package.	
<code>bioc:pkg</code>	a reference to a BioConductor package.	
<code>r:opt</code>	a reference to an R session-level option, i.e. accessible via <code>options()</code>	
<code>invisible</code>	a container node which causes the content/children to be evaluated but not displayed. This might also be done via an invisible attribute on the <code>r:code</code> , <code>r:plot</code> , etc. node	
<code>ignore</code>	a container node which causes the content/children to be entirely omitted from the processed document. This is a convenient way to "comment out" a portion of a document.	
<code>data</code> , <code>datalisting</code>	container for displaying verbatim data. This is equivalent to <code><pre>programlisting lang="data">....</code>	

Node	Description	Attributes
show, do	<p>Used to display one piece of code but run a slight different version. This is typically when we want to hide some of the details but still evaluate code with those details, e.g. creating a plot with axes' labels and a title:</p> <pre> <r:code> <show>plot(1:10)</show> <do>plot(1:10, xlab = "...", ylab = "...", main = "...") </r:code> </pre>	
r:remove, r:rm	<p>identify R objects that can be garbage collected. The content is raw text given as a comma-separated list of R names, or alternatively a sequence of <r:var> elements with the latter handling names with spaces, etc., e.g.</p> <pre> <r:rm> x, y, abc </r:rm> <r:rm><r:var>x</r:var><r:var>abc efg<r:var></r:rm> </pre> <p>In the future, we will do this programmatically via CodeDepends. These are not displayed in the document, but merely used to cleanup variables that are not needed in subsequent computations. These act as hints for the evaluator/processor.</p>	

As with all XML documents, the namespace prefix, e.g. r in r:code, is defined by the author of the document and one can use any word, e.g. rproj:code or R:code. The critical thing is that the prefix is associated with the same namespace URI that we use. These are

Table 2. Namespace URIs used in the Rdocbook Project

r	http://www.r-project.org
omg	http://www.omegahat.org
bioc	http://www.bioconductor.org
i	http://www.statdocs.org/interactive
s	http://cm.bell-labs.com/stat/S4
splus	http://www.insightful.com/S-Plus
c	http://www.open-std.org/JTC1/SC22/WG14
cxx	http://www.open-std.org/jtc1/sc22/wg21
py	http://www.python.org
pl	http://www.perl.org

The Emacs Lisp code in Rdocbook.el has additional name space mappings and a function to dynamically add one.

Graphics

Plots are generated and displayed via an `<r:plot >` node in the input document. This is very similar to a `<r:code>` and contains code that is evaluated. The big difference is that the code has a side effect of creating a plot and that is what is displayed. Since this does not just involve text being rendered, there are more controls available.

Graphics Formats

When an `<r:plot>` node is processed, first a graphics device is created. The particular type of device, or format of the output, is determined by first looking at the attributes of the node, and if no `r:format` attribute is present, using the graphics device in the `device` slot of the `DynamicOptions` object in the call to `dynDoc()`. This is a list with a single named element. The name gives the target format, e.g., JPG or PNG, and the corresponding element is a function which creates the appropriate graphics device. That function should take a file name as the first argument, the width and height of the image as the next arguments and should also have a `bg` parameter which will be used to pass the background color (as a string). We look for values for these arguments from the `<r:plot>` node and use defaults otherwise. The file name corresponds to the `file` attribute. If this is an absolute path name, it is used as-is, otherwise, it is assumed to be a file in the output directory specified in the call to `dynDoc()`. If no `file` attribute is present, a unique file name is generated.

The width and height attributes for the R graphics device are taken from the name-space qualified `r:width` and `r:height` attributes. These are passed directly to the graphics device creation function and so should make sense to it.

Note

We could do something with units here, a la SVG, but because we know nothing about the actual device, it is hard to make sense of them.

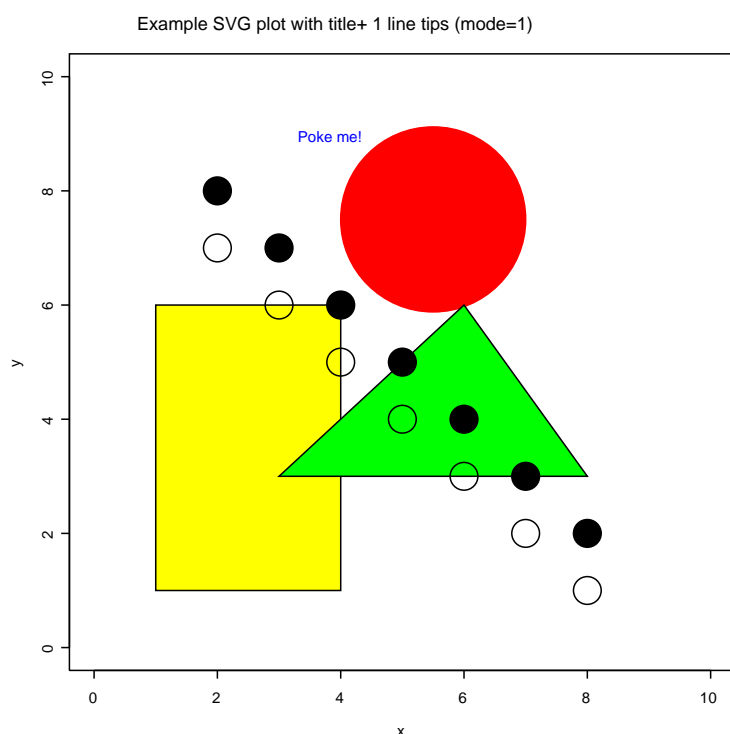
We use the `r:` namespace prefix as these values are R-specific. The node can also have a `width` and `height` attributes and these are passed on, as regular attributes of the result, e.g. as `width` and `height` attributes of an HTML `` node.

SVG

SVG (Scalable Vector Graphics) is an XML-based language for describing vector graphics. Vector graphics are nicer than bitmap/raster graphics as they scale nicely. In principal, we can draw at any resolution and then scale to a different level and get the same "quality" result. While JPEG and PNG files are easy to display inline within HTML documents, they do not scale well. When we resize a window, they either don't scale or become "pixelated". When we zoom, the same thing happens. PDF scales well as it is also uses vector graphics, but we cannot inline it within HTML documents. SVG, on the other hand, scales well and can be inlined in several browsers, and the number is increasing. It is not clear whether SVG will become a widely used format, but there is support in both Opera and Firefox and numerous tools for working with (creating, editing and viewing) SVG. SVG provides dynamic facilities for animation. SVG is also scriptable using JavaScript/ECMAScript and this allows it to provide interactive capabilities such as tooltips, hyperlinks, etc. One can even respond to use events on objects and, e.g., move them.

R has three SVG graphics device. With `libcairo` installed, we can use the `svg()` function in `grDevices`. Alternatively, we can use the `RSVGDevice` package by Jake Luciani. There is also `RSVGTipsDevice` which is derived from `RSVGDevice` and provides facilities for adding tooltips and hyperlinks. Such annotations of the SVG can be achieved with the [SVGAnnotation](#) package also. The following is an example of what it produces:

Figure 1. Output from RSVGTipsDevice



The core engine underlying these two packages needs some enhancements to make it produce high quality output in all circumstances. The most notable missing element is in doing computations involving text sizes as the font metric information is not available. Also, it would be nice to have more structure on the output. But R does not readily lend itself to being able to, for example, group drawings of lines and text to identify an axis or even collections of dots as the output of a call to `points()`.

It is possible to post-process SVG in R, or other environments, by reading it back as an XML document and deciphering its components and adding extra content.

There are several ways to generate SVG graphics for use in *XDynDocs* documents. As with many things in our system, the flexibility makes the description seem complex, but in fact it is reasonably straightforward. First, you chose to use SVG for an individual plot by specifying an `r:format` attribute on the `<r:plot>` node, or alternatively use SVG as the default. Alternatively, you can indicate that all `<r:plot>` nodes

without an `r:format` have a particular format by specifying a value for the *graphicsDevice* format. The second of these approaches is easy to setup. First, load your preferred SVG device package, e.g.

```
library(RSVGDevice)
```

Then, call *dynDoc()* with a value for *graphicsDevice* such as

```
dynDoc("file.xml", "HTML", graphicsDevice = list(svg = devSVG)
```

Here we associate the function with the name "svg" which is converted to upper-case and used as the value of the XSL variable **graphicsFormat**.

If we wanted to use the first approach for an individual plot, e.g. if one plot needs to be scalable for potential zooming or if it were to have some animation or interactivity within it, then we would have a node in our document something like

```
<r:code r:format="SVG" r:width="4" r:height="4" file="myPlot.svg">
hist(rnorm(100), prob = TRUE)
curve(dnorm(x), -4, 4, add = TRUE)
</r:code>
```

But for this to work, the processor must be able to find the function named "SVG". So we need to assign either **devSVG** or **devSVGTips** to that variable before the call to *dynDoc()*:

```
SVG = devSVG
```

Inline Graphics Content

When we create plots for inclusion in a document, we create them as separate files and refer to those files in the document, e.g. via a **<fo:external-graphics>** or HTML **** node. When we process the FO file to PDF, we end up with a single file. But when putting our HTML file on a Web site, we have to also transfer these auxiliary files. It is often convenient to simply inline the graphics directly into the target (HTML or FO) document. When using SVG, this is possible. We can create the node within R and return that. For XHTML, we can just insert the **<svg>** root node directly into the output document. Note that this needs to be XHTML and not just HTML. For FO, we would use an **<fo:instream-foreign-object>** to contain the **<svg>** node. This also makes the code slightly cleaner in the R-XSL interface as we would be returning nodes to display rather than a file name which then has to be transformed into an appropriate node. If the SVG device created an **<xml:internalNode>** directly, all the graphics could be done without I/O. There are many reasons we cannot move entirely to SVG, but it is useful to be able use it to do things more elegantly.

Entities and XSL Stylesheet elements

One of the useful aspects of LaTeX that the combination of XSL and DocBook does not directly support is the ability to define our own macros for frequently repeated content and to customize the appearance of the output. XML is used to separate content from appearance, so this makes sense. But what about repeated content. Entities allow us to define "macros" or constant/fixed content centrally which can be used in numerous places. We can define an entity within the internal DTD of a document and then refer to it within the document. For example, in order to refer to DocBook consistently rather than using Docbook and DocBook, I can define it as an entity and refer to that throughout the document. This can be done as follows:

```
<?xml version="1.0" ?> <!DOCTYPE article [
<!ENTITY Docbook "DocBook"> ]> <article
```



```
xmlns:r="http://www.r-project.org"> <para> &DTL </para> </article>
```

But what if I want to have XML content within the entity. For example, suppose I want to have a link to the DocBook web site. It is an error to define the entity as

```
<!ENTITY Docbook    "<ulink url="http://docbook.sourceforge.net">DocBook</ulink>">
```

which may seem natural. So what are we to do? We can use external entities. That is, we can put the content we want in a separate file and then define an external entity to reference that content verbatim. We could put `<ulink url="http://docbook.sourceforge.net">DocBook</ulink>` in a file name, say, `docbook-entity.xml`, and then define the entity at the top of our document to reference it as

```
<!ENTITY Docbook    SYSTEM "docbook-entity.xml">
```

Then we can use this as we would like.

We will end up with lots of small files, each corresponding to an entity. And we may well want to reuse this in other documents in other directories/folders so have to copy them there or else get the relative names right. To get around this, we can use identifiers when defining an entity such as

```
<!ENTITY Docbook    PUBLIC "foo" SYSTEM "docbook-entity.xml">
```

or we could use a URI for the file name and leave the catalog mechanism to resolve the local file name.

External entities act very much like XInclude but are less flexible as we cannot access subsets of the external document. So what's the point? Well, external entities will be "included" directly by the XML parser, assuming it is capable of it, whereas one might have to explicitly enable XInclude within the parser, e.g. with `xsltproc`. This is a minor issue. Generally, entities (internal or external) can be useful but we probably won't make use of them very often. They are mentioned here for completeness.

Entities work like macros, but what about adding LaTeX-like macro definitions for generating and formatting output. As we mention, XML separates content and appearance and that is good. But in our case, the document is being written not as regular data but to be displayed. We can provide a few templates to customize the layout by creating a separate XSL file and importing the "regular" one. And we will do this for each target format of interest, e.g. FO and HTML. Such files are easy to write:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">

  <xsl:import href="http://docbook.sourceforge.net/release/xsl/current/html/docbook"
    >

  <xsl:template match="docbook">
    <a href="http://docbook.sourceforge.net">DocBook</a>
  </xsl:template>

</xsl:stylesheet>
```

But these extra files mean that the document can no longer be distributed as a single file. Also, if we want to use a different base style sheet, we have to edit the file(s) rather than just specify the file name of the different style sheet in the call to the XSL transformer.

Instead, it seems reasonable in the case of our extended docbook format to allow authors to include one or more `<xsl:stylesheet>` nodes in the document itself. Because we potentially want to transform the document into both HTML and FO (to get PDF), we often have two style sheets. We identify which is which by adding a `format` attribute to the `<xsl:stylesheet>` node. This allows us to determine which style sheet to use.

The function `xsltApplyStyleSheet()` has a parameter `.merge` which can be used to control if it looks for an internal style sheet. This can be **FALSE** indicating that no attempt to use an internal style sheet is performed. Otherwise, `.merge` can be **TRUE** or a string. In either case, the function gets a `<xsl:stylesheet>` nodes from the input document. If there is only one and `.merge` is **TRUE**, this works well. If there is more than one, then it matches the value of `.merge` to the `target` attribute of these `<xsl:stylesheet>` nodes and so `.merge` should be a string such as "html" or "fo".

These embedded/internal style sheets can import the base style sheet and so be a self-contained stylesheet. In such cases, one can call `xsltStyleSheetApply()` without specifying the style sheet and the function will use the one in the XML document by itself. If however, one does specify a style sheet, that is imported into the style sheet taken from the XML file. If one wants to have the templates from the style sheet in the XML document added to the given XSL stylesheet, you need to use `mergeXSL()` and its additional parameters to do this.

One thing to remember is that our XML document now contains additional `<xsl:stylesheet>` nodes. These will appear in the output of the XSL transformation so we need to add a template to discard them, e.g.,

```
<xsl:template match="xsl:*" />
```

or alternatively and more simply, the author can wrap these nodes within an `<ignore>` or `<invisible>` node.

Let's look at a simple example. Consider the document

This is the first section and refers to the `<author>` tag which should get expanded to

The template for processing the `<xml:node>` node will be found in <http://www.omegahat.org/XDynDocs/XSL/html.xsl>. But we want to override the author node to be transformed to `Duncan Temple Lang`. So we add a template to this effect and put it in a stylesheet within the document yielding the complete document as

```
(XInclude) http://www.omegahat.org/Sxslt/examples/embeddedXSL.xml.
```

We can process this document with the command

```
doc = xsltApplyStyleSheet("embeddedXSL.xml", .merge = "html")
```

noting that we are telling `xsltApplyStyleSheet()` that we want to use the html style sheet within the XML document. And the relevant bit of the resulting document is

This is the first section and refers to the

```
<font xmlns="" class="xmlTag">&lt;author&gt;</font>
```

tag which should get expanded to

```
<a xmlns="" href="mailto:duncan@wald.ucdavis.edu">Duncan Temple Lang</a>
```

as we wanted.

We can generate the FO document also with the command

```
doc = xsltApplyStyleSheet("embeddedXSL.xml", "http://www.omegahat.org/XSL/fo/Rfo.xsl")
```

We specify the XSL style sheet to use in addition to the one in `embeddedXSL.xml`. This is because we omitted any `<xsl:import>` element within the FO style sheet in the XML document. The XSL file `Rfo.xsl` is implicitly imported into the local style sheet.

Processing the Dynamic Document

Within R, we can use the package *XDynDocs*. This provides a function `dynDoc` and the typical invocation requires one to specify the name of the file identifying the XML document to be processed. It defaults to generating PDF. However

XSL Files

While most of the details are transparent to R users (via the *target* and those who use the make rules, it can be useful to understand how the XSL files are structured.

There are seven target formats currently supported by the *XDynDocs* package, and these are divided into static, dynamic and both dynamic and interactive. The static formats are regular HTML, regular FO (and hence PDF) and LaTeX (and hence PDF, Postscript and DVI). The dynamic formats are the same: HTML, FO, LaTeX. The difference is that we evaluate the code nodes in the dynamic version and display the results/output. The dynamic and interactive version not only evaluates the code nodes, but also processes the interactive nodes (*i:**) and converts the entire document into a form that is more suitable for processing within R and displaying within the wxWidgets HTML "browser"/widget via R. This supports the display of interactive components within the HTML along with callbacks for the user events/interactions that re-evaluate the R code in the code nodes of the original document.

The XSL file `html.xsl` is used to create the dynamic (non-interactive) HTML format. It imports the `dynamic.xsl` file which provides the XSL templates for evaluating the `r:code`, `r:plot`, `r:expr`, ... nodes. The `html.xsl` provides the templates for rendering the DocBook (and R-related) nodes.

Similar to `html.xsl`, the `latex.xsl` provides the templates for converting/"formatting" the DocBook as LaTeX and uses `dynamic.xsl` to evaluate the R-related code nodes and generate the output in LaTeX format.

The `ihhtml.xsl` generates the interactive HTML content by importing `html.xsl` and providing additional templates for processing the *i:** nodes in the XML file.

Finding the XSL files

The `dynDoc()` function allows the caller to specify the target format using the `target` and provides short-hand names for the different possible targets, i.e. "HTML", "FO", "LaTeX", or "iHTML". From this, we can find the relevant XSL style sheet file within the `XDynDocs` package. However, the files it imports also have to be found and these can be slightly more problematic to find as the user/administrator might want to make use of alternative files. Except for the dblatex material (which is distributed under the GPL), all the files can be found within the `XDynDocs` distribution. We have included the most recent version of the DocBook XSL files that were available when the most recent version of the `XDynDocs` package was packaged and distributed. However, these Docbook XSL files change as do some of the Omegahat XSL files and it can be beneficial to point to a different set of files. We can do this via a catalog file which is how the XML tools in the XML and Sxslt packages attempt to map URIs and XML/DTD identifiers to different URIs and files.

The XSL files within `XDynDocs` refer to the `http://www.omegahat.org/XSL/` and `http://docbook.sourceforge.net/release/xsl/current/` and import files from there. So we need to map these URIs to local files so that we fetch them locally rather than via network requests. While we can map these within the package by providing a package-level `catalog.xml` file, we want the user to be able to add the relevant entries to other catalog files that she might be using.

If you are using the style sheets in `XDynDocs` outside of R, e.g. via `xsltproc`, you will probably want to use the `catalog.xml` file that is installed as part of `XDynDocs`. You can tell `xsltproc` and friends (i.e. `libxml2`) about this file by putting the fully-qualified file name as an element of the `XML_CATALOG_FILES` environment variable. Unlike most multi-element shell variables, this uses a space to separate elements rather than the more conventional `'.'`. So you can specify multiple catalog files in this variable and they are consulted in the order in which they are given. So if you already have a catalog file in, say, `~/catalog.xml` and you also want to use the one in `XDynDocs`, say `~/Rpackages/XDynDocs/XML/catalog.xml`, you would set the environment variable as

```
export XML_CATALOG_FILES="$HOME/catalog.xml
    $HOME/Rpackages/XDynDocs/XML/catalog.xml "
```

(or use `setenv` command if you use `csh` or `tcsh` as your shell.)

Alternatively, you can add the two entries from the `XDynDocs` using something like

```
xmlcatalog --noout --add rewriteURI
    'http://www.omegahat.org/XSL/' '...XDynDocs/XSL/' ~/catalog.xml
xmlcatalog --noout --add rewriteURI
    'http://docbook.sourceforge.net/release/xsl/current/' '...XDynDocs/XSL/do
    ~/catalog.xml
```

where you replace the `.../XDynDocs/XSL` with the actual location of the XSL directory within the installed `XDynDocs` package.

From within R, things are easier. The `XML` provides facilities for adding to and querying the global catalog table and this allows us to load additional catalog files. When the `XML` package first needs the catalogs, it loads the default ones from the `XML_CATALOG_FILES` environment variable or `/etc/xml/catalog` by default (at least on UNIX®). So any settings you have will be used and take precedence. But the `XDynDocs` package also loads its own `catalog.xml` file and so provides a backup or fall-through rewrite rule for local versions of the DocBook and Omegahat and `XDynDocs` XSL directories.

We can find all the imports within the XSL files using the following code. This helps us verify that we have all the files and that we have the relevant URIs mapped to local files in the catalog file.

LaTeX output

There are many reasons why producing the final output via LaTeX but using DocBook and XML as the format language is desirable. Using XML, we have a programmatically modifiable and queryable database as a document. We can do significant filtering both in R and in XSL. And with LaTeX, we can produce extremely high quality typeset output, including mathematical content. FO is very good and in several ways is easier to learn than LaTeX. But many of us are already familiar with LaTeX and learning a new technology is not very appealing.

There are several approaches to integrating DocBook and LaTeX. One is simply that we write everything in DocBook and then utilize XSL converters to map these elements to LaTeX. For example, we would write

```
<section><title>My Section</title>
```

and during the XSL transformation, this would be mapped to the LaTeX content

```
\section{My Section}
```

An example of this approach is given in `../tests/dblatex.xml`.

Another approach is that we use a very thin DocBook layer to make a LaTeX document into an XML document. Then we can declare name spaces and use `<r:code>` nodes and the like. The text is almost entirely in LaTeX. There is very little translation to LaTeX as the content is already in that format. But a major drawback is that we can access only a very limited number of the elements of the document programmatically and we cannot easily convert it to other formats such as (X)HTML. The more DocBook content we use, the more we can programmatically process the document.

An example of this style is given in `../tests/latex.xml`.

Regardless of which style we use, there will be some DocBook elements that we need to transform to LaTeX. There are two projects that provide XSL style sheets to aid in this. These are `db2latex` and `dblatex`. These are related and the latter is more recently developed. To some extent, the former is easier to customize and has been easier to get to work in our framework. However, both are relatively easy to adapt and we provide the basic XSL style sheets to use them within the Dynamic Documents framework.

We can use `dblatex` to generate LaTeX from our DocBook file. In our version, we can use a very minimal DocBook structure. Rather, we can have essentially raw LaTeX markup inside an `article` and `para` XML nodes, e.g.

```
<article xmlns:r="http://www.r-project.org">
<articleinfo>
<title>Using <latex/>{} for dynamic documents</title>
<author>
<firstname>Duncan</firstname><surname>Temple Lang</surname>
</author>
</articleinfo>
<para>
\def\myMacro{My Expanded Macro}
```

This is an example of a documentat that we want to convert to <latex/>.

It uses, for example, its own format for mathematical content such as

```
$$
E[X] = \int_0^{\infty} x f(x) dx
$$
```

```
<tex><![CDATA[
This is pure \TeX{}
which is passed through to
latex as is $x < 2$
]]></tex>
```

Here we introduce some R code

```
<r:init>
n = 100
</r:init>
```

And then we generate a sample of size \$n\$

```
<r:code>
summary(rnorm(n))
</r:code>
```

```
\begin{center}
\begin{tabular}{ll}{ll}
Info & value \\
\hline
$n$ & <r:expr>n</r:expr> \\
date & <current-date/>
\end{tabular}
\end{center}
```

It also uses a local macro such as \myMacro.

\section{A section} This is a regular <latex/>{} section. So we see that we are mixing <latex/> with \textit{DocBook}.

```
\section{Another section}
\includegraphics{logo.jpg}
```

```
<r:plot originalFile="logo.jpg">
plot(1:10)
</r:plot>
```

```
</para>
```

```
</article>
```

Our version overrides the escaping of LaTeX symbols so in our world \$ maps to \$ and not \\$. This allows us to use LaTeX markup directly. If one wants to use DocBook markup and preserve these escapes, then use the original docbook.xsl in the dblatex distribution and then import our latex.xsl file which provide the extensions to DocBook that are of interest, e.g. r:code, etc.

To process the resulting LaTeX file, you will need to have the docbook.sty and any related files installed. You can set the environment variable TEXINPUTS to

```
setenv TEXINPUTS ~/XML/dblatex-0.2.7/latex//::
```

You will also need to find definitions for the R-level environments. See Rdocbook.sty

Using dblatex:

```
dynDoc("latex.xml", "LaTeX", "../inst/XSL/dblatex.xsl", force = TRUE,  
      doc.collab.show = FALSE,  
      latex.output.revhistory = FALSE)
```

Using db2latex:

```
dynDoc("latex.xml", "LaTeX", "../inst/XSL/db2latex.xsl", force = TRUE,  
      latex.documentclass.article = "10pt")
```

Interactivity

One can add markup to the document that will be processed when the reader choses to view the document in our interactive viewer. So that these interactive elements are not displayed in standard rendering, we put this content inside `<interactive>`. Within this, we can have arbitrary markup which is typically used to position, annotate and layout the interactive controls. For example, we might arrange the controls in a table or place labels and images beside the controls.

When working with a document, we have authors and readers. There is one or more authors that create the original document. Then we may have later authors who add components to the document, e.g. interactive controls, branches for alternative approaches to a data analysis, etc. And then we have readers. Authors determine what they want the reader to see via their specification of the markup. However readers can chose how they want to see it and can specify options that control how the document and its nodes are processed. So there are two sets of options in effect.

The interactive tag has a ref to identify the associated r:code, etc. element(s). This is used when creating the environment for storing the interactive controls within this interactive node and getting the evaluation of callbacks, etc. right.

One can give give the newly created environment for the interactive node a name using the id attribute.

i: elements

To specify the controls, we can use regular HTML OBJECT tags. However, often the sort of interactive control we want in a document is one which allows the reader to change the value of a variable in an r:code, etc. element. So to do this, it is often more convenient to use specialized markup that leaves the details to our

rendering system. We have to identify the variable and the type of control and provide some parameters to give the control the appropriate characteristics. For example, suppose we have a simulation and a variable, **n**, controlling the sample size. This is an integer that should be positive and for practical purposes we put a limit on it of, say, 300. Then we might want to have a slider in the document with which the reader can set the value. We could mark this up with

```
<i:slider var="n" min="1" r:type="integer" max="300"/>
```

Alternatively, we might want a spin control and in this case, we might not want to specify a maximum value.

```
<i:spin var="n" min="1" r:type="integer" />
```

We can be even more "general" and merely specify the variable and its type and leave it to the rendering software to use an appropriate control. This allows the reader and her software to decide what control is generally appropriate for such a type. Sometimes, the author of a document will want to specify the control precisely, and other times this can be left to the reader's environment.

One can provide a name attribute in the `i:` element and if this is present, the newly created widget is assigned to this value within the environment associated with the `<interactive>` element in the XML document. This is useful when the controls need access to each other, i.e. a callback for one needs to update another. Although this can also be done by using the general `i:object` element and creating the two controls together, using the provided tools is often easier and the name attribute handles simple cases.

Generic `i:object` element

There also may be cases where we want to create a control for which we have not provided support via an XML element or for which we want to specify more parameters to customize it. In this case, we will want to provide R code that is used to create the control. We can do this with the general with the HTML element `OBJECT`. Since this control will typically alter one or more variables that are used in the `r:code` nodes, it is helpful to identify these and it is also useful to identify the code node(s) by name to ensure that the R code has access to those variables. Each `r:code`, etc. element has an associated environment and it is essential that the interactive controls are connected with the relevant environment. So rather than using the regular `OBJECT` tag, it is better to use `i:object` and a `ref`.

Note

We should use an `r:` namespace for the `ref` attribute.

```
<i:object type="R-generic" width="" height="" ref="r:code id">
```

```
</i:object>
```

The code is run just like an `OBJECT` handler for the `htmlViewer` in the *RwxWidgets* package. It will be evaluated within an environment that has the variables `tagHandler` (`RwxHtmlTagHandler`), `tag` (`wxHtmlTag`) and `parser` (`wxHtmlWinParser`) and `html` (`wxHtmlWindow`). The parent of the evaluation environment will be one for created for the interactive group. And its parent environment will be the one corresponding to the associated `r:code` node.

Multiple r:code References

If one needs to identify multiple r:code, etc. elements, you can specify the names as a comma-separated list in the ref attribute. If any of the names contains a comma, this will cause problems. So in these cases, choose a different separator character, e.g. |, that does not appear in any of the names and specify the values using this character as a separator and then specify the separator using the refSep attribute. For example,

```
ref="x,y,z|a,b|a third one" refSep="|" "
```

to yield "x,y,z", "a,b" and "a third one" uses the | character.

If there are multiple r:code nodes specified (see below), the environment in which the code will be evaluated will be associated with the last of these. Specifically, we create a new environment for evaluating code within the interactive node and the parent of that environment will be the environment associated with the "lowest" of the code nodes, i.e. the one latest in the document. The reason for this is that the code in the interactive elements will be able to "see" the variables in all of the nodes since the environments of the r:code, etc. nodes are chained together. Note that one will also see variables in any r:code environments in between the ones specified. For example, if we refer to code nodes A and C, we will also see the variables in B.

Referring to multiple r:code, etc. elements could potentially cause some issues in the evaluation model. For example, suppose we have four code nodes named A, B, C, D. In both A and C, we define a variable n and in B we use n, say to generate a sample of a certain size. Now, we define an interactive node that refers to both A and D. The environment for evaluating our interactive code and callbacks would be a child of D's environment. Now, if we ended up with a call to set A's version of n with an expression of the form

```
n <- value
```

this would change the definition in C, and not in A. As a result, we would end up with the wrong semantics. Instead, we want to have an expression of the form

```
assign(n, value, A.envir)
```

where A.envir is the environment associated with the r:code element with id A.

To avoid this ambiguity, when referring to variables in the i:* nodes, one should use both the name of the variable and also the id of the r:code, etc. node, e.g.

```
<i:slider var="n" ref="A" .../>
```

```
<r:code id="A">...</r:code>
```

One might also use a namespace-like notation of the form A::n,

```
<i:slider var="A::n" .../>
```

This raises the possibility of confusion when we are actually referring to a variable within an R namespace, so it is preferable to use the var & ref attribute pairing.

The Update Button

Generally, we might have several controls, each updating one or more variables that are used in subsequent r:code, etc. nodes. When we update one variable, we could recalculate all the subsequent code elements.

However, typically we will want to vary several inputs and then run the computations. And since the computations can be lengthy, we do not want updates to one variable to propagate instantaneously. Instead, for each `<interactive>` element, we add an "Update" button which the user can click to have their changes propagated through the calculations of the R code.

If there is only one control (i.e. one `i:` element) within a group and it is not a generic one with its own code, but rather tied to a variable, we can omit the Update button. Instead, the callback on the interactive control propagates the new value through the subsequent R code.

One can use the attribute `addUpdate` within the `<interactive>` element to override the creation of the Update button, specifying a value of `true` or `false`.

In addition to the Update button, we can also have a reset option on each `r:code`, etc. node. This could reset the controls to the specified value in the original document. It could also restore the values from the original evaluation of each code node.

In the presence of a reset button, we need to ensure that we can explicitly set the value of each widget to its original value. This is different from setting the value when we create it and, in particular, it means we need to assign each newly created widget to a variable. If an `i:*` element has a `name` attribute, we use this. This allows the author to know the name of the variable bound to the widget reference so she can use it in the computations. If no `name` attribute is specified, then we use the variable name by taking the value of the `var` attribute and concatenating it with the type of control. For example, the element

```
<i:slider var="n" .../>
<i:slider var="n" ref="A" .../>
```

would give `.n.slider` and `.A.n.slider`, respectively. If those variables already exist, then we generate a unique name. Since the author has not explicitly named it, there should be no reference to it in the code. We could provide a function for finding it however by storing the generated name and allowing the "user" to specify the `var` and `ref` arguments and the type.

Managing the Evaluation of Code nodes

Each code element will be displayed in the interactive document. (Ideally we would be able to expand or collapse it using dynamic HTML-like facilities, but within the `wxHtml` widget, this is probably not feasible.) But we do add a checkbox that indicates whether this node should be updated when inputs up-stream are changed. If this is checked, then the code is not reevaluated. It will be marked as such with a different color or some text that indicates it is out of date.

In addition to checkbox within the HTML document, we will also allow the user to interact with the code blocks via a tree widget which displays the code blocks symbolically. This is an outline view of the documentation or a table of contents that acts as a navigation tool. The reader can "lock" code elements here too to avoid recalculations. She can also jump to a particular code element and its output by double clicking on the element in the tree. Also, she can drag and drop one or more elements into another outline view and create a second document. Or she can create a branch/path in this document to explore alternative approaches.

In addition to the reader explicitly locking a code block, the author can indicate that she wants it to be locked and so not updated and the new results displayed. This is done with the `i:update` attribute and specifying a value `'false'`. This (currently) means that the code should not be reevaluated.

Interactive Control Markup

While it might be more direct to inline R code that uses wxWidgets to create the interactive controls, it is a good idea to use abstract markup of controls. There are several reasons. Firstly, using XML markup such as `i:slider`, etc. clearly separates the specification from the R language and so the same code can work with different implementations. For example, we might have MATLAB code that parallels the R code in the document and the interactive controls could be used for either version. Also, it is not obvious that wxWidgets is the right toolkit to be using in all circumstances. We might use Qt or Gtk and Mozilla. Having an abstract description of the controls allows us to provide alternative implementations. And indeed, one of the interfaces we think is potentially compelling is Microsoft Word and other word processors. Since Office has rich support and extensibility for XML and XSL, we could process the document in R by sending content to Word and have it render the output there along with the interactive controls. And the interactive controls would be very different there.

Implementation

We first transform an XML file into an HTML file using the `ihhtml.xsl` file. This transforms `r:code`, etc. nodes into regular HTML OBJECT elements. It also processes the `i:*` and interactive nodes. It adds the `ref` attribute from each ancestor interactive node to their child `i:*` nodes. We then take this document and pass it to the HTML widget and its parser. The parser hands control back to our handler functions in R when they encounter an "HTML" tag for which we have registered handlers. It is up to us to create and insert the relevant content for that node. We provide handlers for the `i:*` nodes and the OBJECT handlers. We could do this for `r:code` nodes directly, but have left it to XSL to transform these.

We need to evaluate the R code before the associated interactive code so that we have the correct values for the variables and so the "current values" in the controls are reflective of the actual values in the R code. Each `r:code`, etc. node has an `id` (which we either generate and add to the OBJECT node or is specified by the user) and this is how we associate the nodes. However, when using the HTML parser, we won't necessarily have seen the `r:code` node when we process the associated interactive node because the `r:code` node may appear later in the document, e.g. if the controls are to appear above the output. So in order to deal with this, we have to be able to fetch the `r:code` and evaluate it, and in the appropriate order (just in case an interactive node refers to an `r:code`, etc. node out of order.) Perhaps the simplest thing to do is to put all the code nodes in a special node at the top of the HTML document which treats them as a collection of named elements indexed by their `id`, i.e. a dictionary. Then we transform the `r:code` nodes for the HTML document to refer to the relevant `r:code`, etc. node in that dictionary. This allows us to read all the `r:code`, etc. nodes early in the parsing of the HTML document and as we encounter the interactive, `i:*` and `r:code`, etc. nodes to be able to ensure the code is evaluated. We do have to make certain to handle `r:code`, etc. nodes nested within ignore elements or which has an `eval="false"` attribute value.

Note that instead of creating the `r:code` dictionary when transforming the XML document to HTML, we could do this separately from the XSL transformation. Instead, we could use the XML parser to fetch these nodes from a second processing of the original XML document.

Given this reorganization, we can parse and render the HTML in a natural order. Suppose the `r:code`, etc. dictionary is contained within an HTML node named `r:codeDictionary`. When we read these elements, we can create a separate environment in which each will be evaluated. Subsequent `r:code` nodes need to see the variables created by earlier code blocks. So we arrange for the environment for an `r:code`, etc. element to have the environment of the previous `r:code`, etc. block as its child. For an interactive node, we find the environment for the associated `r:code`, etc. node. Then we create a new environment with that environment

as the parent. This allows the code in that interactive node, i.e. within the `i:*` elements, to directly access the variables in that `r:code`, etc. elements.

So when we see an interactive node, we create its new environment and make that the active one. The parser continues processing the child nodes which are typically HTML formatting nodes and then encounters an `i:*` node and so hands control to our R tag handler. We then create the widget/component, specify the callback action and return. For example, in our `isim.xml` document, we have an `i:slider` node to set the sample size variable `n`:

```
<i:slider var="n" min="1" max="300" r:type="integer" />
```

Our R tag handler function might be something like the following:

```
function(html, tag, parser)
{
  id = tag$GetParam("id")
  envir = getInteractiveEnvironment(id)

  win = parser$GetWindow()

  varName = tag$GetParam("var")
  tmp = tag$GetParam("r:envir") # code chunk id.
  if(tmp != "") {
    varName = c(varName, tmp)
  }
  # hostEnvironment = getEnvironment(tmp)
  }

  value = get(varName, envir)
  min = as.integer(tag$GetParam("min"))
  max = as.integer(tag$GetParam("max"))

  slider = wxSlider(win, wxID_ANY, value, min, max)

  assignTo = tag$GetParam("name")
  if(assignTo != "")
    assign(assignTo, slider, envir)

  hostEnvironment = parent.env(envir)

  slider$AddCallback(wxEVT_SCROLL_CHANGED,
    function(ev) {
      sl = ev$GetEventObject()
      assign(varName, sl$GetValue(), hostEnvironment)
      recalculateNodes(envir)
    })

  slider
}
```

When we specify this function, we need to ensure that it has access to the appropriate function *getInteractiveEnvironment()* . This will be provided by the top-level environment for our interactive document. So we must set the environment for this function to be able to see that function. This function will work on any *i:slider* node within the document, and the environment associated with the computations is obtained dynamically via the a call to *getInteractiveEnvironment()* . After this, all the calculations can be done programmatically within this general handler, i.e. it is not tied to a particular environment for a particular interactive node. The symbol *recalculateNodes()* will be found in this top-level document environment as that is, of course, independent of any particular node but can work generically when given an environment or id.

Note

We could define a function, say *updateVar()* , which does the assignment of the new value to the specified variable and then recalculates the nodes. For example,

```
updateVar =  
function(name, value, envir)  
{  
  assign(name, value, parent.env(envir))  
  recalculateNodes(id)  
}
```

Also we would define a function *getVarName()* and define it as

```
getVarName =  
function(tag)  
{  
  varName = tag$GetParam("var")  
  tmp = tag$GetParam("r:envir") # code chunk id.  
  if(tmp != "") {  
    varName = c(varName, tmp)  
    # hostEnvironment = getEnvironment(tmp)  
  }  
  
  varName  
}  
varName = getVarName(tag)
```

We would like to allow the user to customize the created widget by either providing their own code or callback or general code that is run after our code. This can be done by allowing content within the *i:** nodes, and not relying exclusively on attributes. We could have *i:callback* with an *append* attribute taking values true or false to indicate that the author wanted the code added to ours or run instead of ours. We could also use an enumeration of *before*, *after*, *instead*. And we could have an *i:create child* with code to create the widget. But this is of secondary importance as we will have support for a generic *i:object*.

The style of one of these general *i:* handler functions is to extract the information from attributes of the HTML node and use these to create the widget. Then we add one or more callbacks to handle the reader's interaction with this control to update variables and recalculate the subsequent parts of the document.

Note

If interactive nodes are allowed to be nested, then we use a stack for the active environments so that when we end one, we pop its environment from the top of the stack to leave environment from the parent interactive node active.

Almost all of the `i:*` handlers will need access to the environment and to the `wxHtmlWindow` object. So it would make sense for us to create a type of function (i.e. with a class) that has a signature that accepts the html window, the tag, the parser and the environment. The R handlers for the general OBJECT node in HTML are given the tag handler, the tag object, the parser and an optional environment. Since this has not been released yet, perhaps we should add the HTML window as an argument, before the environment. And we should use a class label to identify the function as supporting this "interface" rather than just counting the number of parameters it accepts.

In order to support `r:code`, etc. nodes being "locked" or not recalculated, we need only maintain a logical vector in the top-level environment indicating whether the `r:code`, etc. element is locked or not. This vector is indexed by the id values for the `r:code`, etc. nodes. When we go to recalculate the nodes, we check the value of the element for each node. Alternatively, we can add it as a slot in the `CodeBlock` class and look at that in `recalculateNodes()` . and this

The code associated with the nodes are maintained in the order in which they are to be evaluated and this makes it easy to process the chain of "down-stream" code nodes.

Type Annotations and Dependencies

We have in mind that a code node (e.g. `r:code`, `r:plot`, etc.) will be able to support markup describing its inputs and the outputs. These will specify the "variable names" and their types. The idea is to characterize the computational flow as we do in a program. Specifically we want to be able to allow authors and readers to create branches which represent alternative approaches to the analysis within a segment of the document. They would start at a particular node with a collection of outputs. They can then provide alternative computations on these values and connect back to a later node in the document with inputs of a similar type. For example, suppose a document describes a data analysis in which e-mail messages are classified as SPAM or HAM using CART. It starts by reading the e-mail messages and moves on to create derived variables which are then passed to the next node which fits a classification tree to these observations and the resulting classifier is used to predict entirely different messages. An author or reader might chose to investigate the use of a k-nearest neighbors approach in contrast to CART. This would be a branch from the derived variables node with the data frame of observations as inputs and a classifier as the output. It would then pass this to the prediction stage for the new data.

How feasible it is to swap one set of inputs for another depends on the code in the node to which they act as inputs. But given R's reasonably consistent modeling interface, it is not terribly unlikely. But it is greatly helped by providing descriptions of the inputs and outputs in terms of their types or classes. Precisely how we do this remains to be determined. But something of the form

```
<input>
  <var name="messages" r:type="data.frame" />
  ...
</input>
```

```
<output>
  <var name="model" r:type="rpart" />
</output>
```

Note that a branch may contain multiple steps or nodes that are sequential, i.e. so we would have parallel sequences of computations representing alternative approaches. And a node may have many branches emerging from it, not just 2. And a branch can have branches within it, i.e. inner- or sub-branches. For example, within the k-nearest neighbor approach, we might determine the metric and then the value for k in two separate steps, or use cross validation to determine both and then explore the fit. And a branch could have an end point that connects to an arbitrary node within the document, or even in another document. In other words, we can define a computational flow by allowing links to other nodes, local or remote. Again, in the context of the k-nearest neighbor fit, we might explore several metrics in parallel, choosing the value of k for each separately and exploring the confusion matrices for each.

Alternatives to wxWidgets

Having implemented this, we see that there is, as we expect, a natural separation between the XML processing and the rendering in the HTML widget and any HTML renderer would do. It is convenient to run this from within R but, as we did before ???, one can use a plugin to an existing browser that embeds an R engine within the browser. There are security issues and one has to connect the Java, JavaScript, C and R code.

Running the browser inside R means that we need to embed a rendering engine within R. There are several possibilities. khtml, gecko and gtkhtml are the three natural ones. But additionally, we can think of entirely platform-specific approaches such as using Word to render the XML via DCOM. In this setup, we would send Word an XML document or a sequence of XML chunks, or alternatively entirely process the document elements in R and add words and paragraphs, etc. to the Word document. And we would insert ActiveX controls to create the interactive components in the display and use the RDCOMEvents package to register R functions as event handlers for these controls. There are several

Using Gecko/Mozilla's embeddable HTML widget would mean that we get support for SVG, MathML, CSS, JavaScript. And the combination of SVG and an interface to JavaScript and the SVG objects via JavaScript could give us a style of updatable, interactive graphics in R.