FanXE — The Fantastic XML Editor

Thomas C. Allard

Banking Analysis Section
Division of Monetary Affairs
Board of Governors of the Federal Reserve System
20th Street & Constitution Avenue NW, Washington DC, 20551
(tallard@frb.gov)

October 2005

Abstract

FanXE[1] is an XML editor written in [**incr Tcl**] using **tdom** and **Tkhtml**. A developer builds XSLT style sheets that transform nodes in the DOM into HTML forms editable in the GUI. When the form is submitted, the DOM node is rebuilt. **FanXE** can maintain XML documents under revision control or within a PostgreSQL database. The base class is easily extensible as demonstrated with three subclasses. **FanXE** can act as a command-line XSLT engine and includes XSLT extensions that allow style sheets to execute Tcl commands in a safe interpreter.

1 Introduction

On my first foray into XML, I needed a highly customizable GUI for editing XML documents. This included choosing which nodes are displayed to the end user and the ability to edit nodes and their children in a single form. The end-user must not be required to know anything about XML. Experienced users should be able to view the underlying XML and use XPath expressions to search the documents. Developers should be capable of writing **FanXE** applications with limited or no Tcl experience (extensive XML/XSLT knowledge, however, is required for developers).

It was also important that my editor be the face of different applications and therefore it had to be easily extensible. The "front page" can be either a static HTML page or dynamically generated HTML from a method inside the class. Links in the HTML can be either XPath expressions to nodes in the XML documents, links to methods in the class, or just regular web links.

[1]Pronounced "fancy."

FanXE uses the Document Object Model (DOM) to operate on all XML documents. All DOM operations are performed by the tdom package.

FanXE reads an XML file called *fanxe.xml*[2] which determines which XML documents will be read into the DOM, which nodes will be displayed in the GUI and how they will be labeled, which style sheets apply to each node, and all other configuration details for the GUI. **FanXE** also loads a user configuration file.[3] The user configuration file can include additional DOMs to load and can override any other aspect of the GUI configuration. Both the *fanxe.xml* file and the user configuration file can be edited within the **FanXE** GUI.

2 GUI Overview

The **FanXE** GUI editor is split into three panes: a tree view of all XML DOMs, an HTML view pane, and a tabbed pane. The layout of panes is user-configurable (see section 2.4).

2.1 Tree Pane

The Tree pane (Figure 1) presents multiple DOMs in a `tixTree` widget (see section 5.3.1). Not all branches of the DOM are shown in the tree, just those configured in the *fanxe.xml* configuration file (see section 2.4).

2.2 View Pane

Selecting any node in the Tree pane will run a style sheet associated with that node and display the HTML

[2]The path of the *fanxe.xml* file is set with the `-xmlDir` option. Subclasses have their own `-xmlDir` option set and load their own set of XML documents.

[3]Stored in the user's $HOME/.tk/fanxe directory with "***classname*.xml**" as the file name.

Figure 1: Tree pane

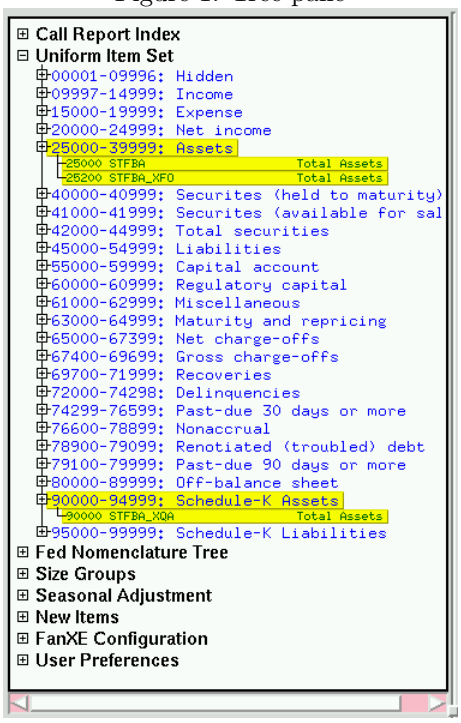


Figure 2: View pane

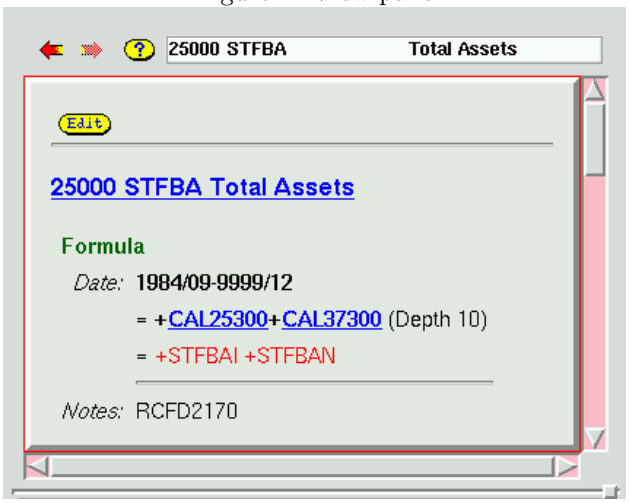
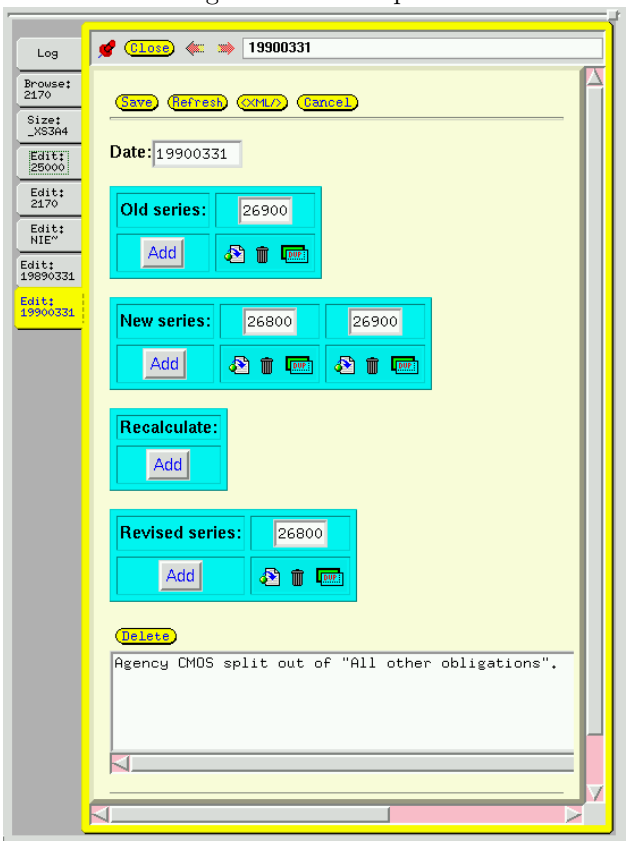


Figure 3: Tabbed pane



results in the View pane (Figure 2). The View pane is a **Tkhtml** `html` widget which is a fairly full-featured web browser (see section 5.2). **FanXE** supports a special **FanXE** hostname (see section 2.5) as well as a special `fanxe://` URI (see section 2.6). **FanXE** does not support HTML forms except for editing the nodes (see section 2.7).

Clicking on the Help icon ( ) will display help information. The URI for the help file is configured in the *fanxe.xml* file and it can be a `fanxe://` URI.

2.3 Tabbed Pane

The Tabbed pane (Figure 3) contains the log messages and tabbed **Tkhtml** windows. It is built with the **BLT** `tabset` widget (see section 5.3.2). Middle clicking on any node in the Tree pane or middle-clicking on any link will open the link in a new tab instead of using the View pane.

Style sheets associated with nodes may contain HTML form elements for editing the node (see section 2.7). More commonly, the style sheet will present a non-editable summary of the node and include an Edit button that will run a second style sheet that is always opened in a new tab. **FanXE** uses specially constructed HTML form elements that, when submitted, reconstruct the DOM node. If the node is already being edited, the previous tab will be moved to the foreground

and the edit style sheet will not be reapplied.

Edit style sheets will commonly have Save, Refresh, XML/ and Cancel buttons (including the *fanxe_templates.xsl* style sheet and calling the `fanxe:buttons` template will add these buttons). Note that Save only saves the changes in the DOM but does *not* write the XML document to disk. You will be warned before exiting if you have changes that have not been saved to a file. The Refresh button rebuilds the node and reapplies the style sheet without changing the original DOM. The XML/ button will display the node (with current edits) as pretty-print XML[4] in a new tab. The Cancel button will abort editing and close the tab.

Clicking on the "Pin Up" icon ( ) or right-clicking on the tab will detach that tab and open it in a new window allowing you to view multiple tabs simultaneously. Clicking the "Pin Down" icon ( ) will reattach the tab.

2.4 GUI Configuration

Figure 4 shows the Tree editor. In this example, only two elements of the *Uniform Item Set* DOM are displayed in the **FanXE** Tree pane: the <**Section**> element and its child <**Series**>. When the DOM is loaded, the <**Section**> elements will be shown open in the Tree (the "Open when loaded" box is checked).

The "Title" entry for the <**Section**> element in Figure 4 contains the following code:

```
set title "[node @Range]: [node @Title]"
```

The `node` command is aliased to the DOM node object for that element and this Tcl code is executed in a safe Tcl interpreter. The resulting value of the *$title* variable is used as the label for that branch in the Tree.

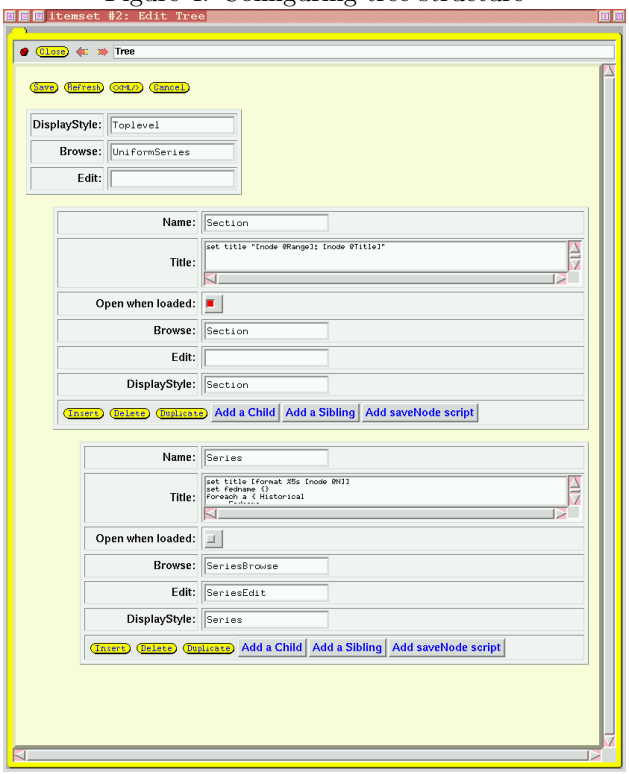
When a user clicks on a <**Section**> element in the Tree, the *Section.xsl* style sheet[5] is applied against that node and the HTML results are displayed in the View pane. When a user clicks on a <**Series**> element, the *SeriesBrowse.xsl* style sheet is applied and results are displayed in the View pane. The *SeriesBrowse.xsl* style sheet includes an Edit button. When a user clicks the Edit button, the *SeriesEdit.xsl* style sheet is applied to that node and the HTML results of that style sheet are displayed in a new tab (if that node is already being edited, the tab is simply moved to the foreground).

The "DisplayStyle" entry in Figure 4 references a `tixDisplayStyle` (see section 5.3.1) defined in the

[4]Pretty-print XML is created by the *xmlverbatim.xsl* style sheet written by Oliver Becker (available at: *http://www.informatik.hu-berlin.de/~obecker/XSLT/*).

[5]All style sheets are loaded and stored in memory when the GUI starts.

Figure 4: Configuring tree structure



fanxe.xml or user configuration file and can be edited within **FanXE** (Figure 5). This defines the font shape, relative size, and color of the element label in the Tree.

FanXE also allows you to use the GUI editor to completely customize the layout of the GUI. You can control the base font size, colors and any additional X resources. You can also customize the layout of the panes (Figure 6).

All customization options can be overridden by the user's personal configuration.

Figure 5: Defining DisplayStyles

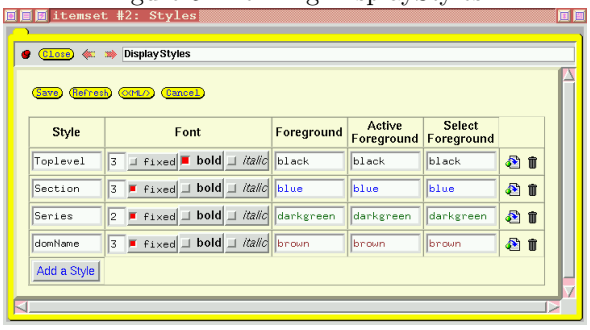
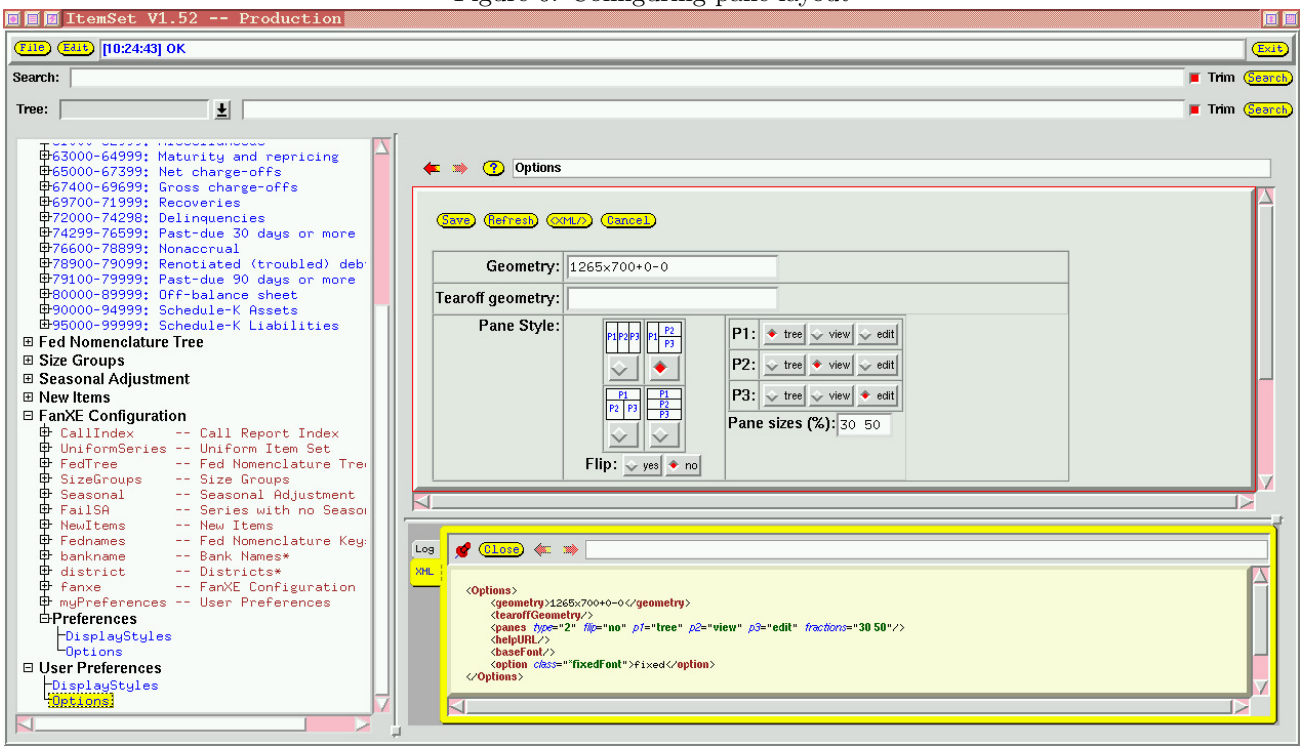


Figure 6: Configuring pane layout



2.5 Linking to Methods and Images

FanXE was designed so that it could be the front-end interface to any application. It needs to be able to call methods from the **Tkhtml** widgets. Using **FanXE** as the hostname in http links accomplishes this. These links are in the form:

`http://FanXE/method`[?*arg1*][...][&*argN*]

where `method` is the name of the public method being called and *arg1*...*argN* are arguments to that method.

You can also link to a variety of internal images (such as all the buttons like Edit and Save) using the **FanXE** hostname.[6]

2.6 Searching the DOMs

FanXE can search the DOMs with either XPath or regular expressions. For both search methods, selecting the "Trim" button will show only nodes in the tree that match and will collapse all other nodes. If "Trim" is not selected, matching nodes will simply be highlighted (and opened if closed) while leaving the rest of the tree in the same state.

[6]The built-in images come from a package I developed called **imageLib**, which is itself maintained via **FanXE**. The `ShowImages` method will display all available images and their names in a new tab.

FanXE's **Tkhtml** widgets also support a special URI scheme (`fanxe`) of the form:

`fanxe://DOMname/`*XPath*

where **DOMname** is the name of the DOM (defined in *fanxe.xml*) and *XPath* is any valid XPath expression.

XSLT style sheets can use these URIs to link one XML document to another.

2.7 Editing Nodes

The magic of **FanXE** is the ability to convert an XML node into an HTML form that can be edited and later turned back into a new XML node. The trick is creating HTML <**form**> elements with specially constructed *name* attributes.

The *name* attribute is made up of one or more `tag:ID` pairs separated by a forward slash ("/"). At the end of the `tag:ID` pairs, the *name* may optionally be followed by an at sign ("@") and an attribute name. The full path of the node must be included in the *name* attribute.

Each `tag:ID` pair includes the element's tag and a unique ID associated with that element (usually created by the XSLT `generate-id()` function or the `fanxe:path()` function). The ID can be any string as long as it does not include a colon (":"), slash ("/") or

at sign ("@"). Also, if the ID begins with `COPY`, it will be treated in a special way (see below).

For example, the following XML contains four elements:

```
<Root>
  <Branch>
    <Leaf Attribute=""/>
  </Branch>
  <Branch/>
</Root>
```

This can be represented by four HTML <**form**> elements with the following *name* attributes:

```
Root:r1
Root:r1/Branch:b1
Root:r1/Branch:b1/Leaf:l1@Attribute
Root:r1/Branch:b2
```

Note that you do *not* have to have an HTML <**form**> element for each XML element. Referencing `Root:r1/Branch:b1` implicitly creates the `Root:r1` element.

As in all XML documents, you can only have a single root element.

The value of an attribute or a text node is determined by the current contents of the HTML <**form**> elements. See Table 1 for HTML <**form**> elements supported by **FanXE**.

Nodes with an ID that begins with `COPY` are used to copy all child elements of the original DOM into the new DOM without creating an HTML <**form**> element for each node. The value of the HTML <**form**> element will be appended to the XPath of the original node and the entire sub-tree will be replicated in the new DOM.

The <**input** ***type="submit"***> element can be used to add, insert, delete or duplicate nodes within the node being edited.

See Table 2 for a portion of the XSLT style sheet that generated the HTML in Figure 5.

3 Command Line Interface

FanXE was written with the **[incr Tcl]** object oriented extension. A command line script, `fanxe`, can be used to access any public method. Usage of the script is:

```
fanxe [CLASS OPTIONS --] [method [ARGS]]
```

The script will instantiate a **FanXE** object with any options you specify. If no `method` is specified and the X environment is available, the script will default to the `GUI` method and start the graphical user interface. Only public methods can be called from the script. When

Table 1: Supported HTML <**form**> elements

<**input**
type="submit"
name=[REQUIRED]
value=[REQUIRED]>

<**input**
type="image"
name=[REQUIRED]
value=[REQUIRED]
src=[REQUIRED]>

<**input**
type="checkbox | radio"
name=[REQUIRED]
value=[REQUIRED]
showvalue=[OPTIONAL; **default**=`false`]
(NOTE: nonstandard HTML)
checked=[OPTIONAL; **default**=FALSE]
disabled=[OPTIONAL; **default**=FALSE] >

<**input**
type="text"
name=[REQUIRED]
value=[REQUIRED]
disabled=[OPTIONAL; **default**=FALSE]
size=[OPTIONAL; **default**=20] >

<**input**
type="hidden"
name=[REQUIRED]
value=[REQUIRED]>

<**textarea**
rows=[OPTIONAL; **default**=5]
cols=[OPTIONAL; **default**=80] >
VALUE
</**textarea**>

<**select**
size=[**OPTIONAL; default=# options**]
multiple=[OPTIONAL; **default**=FALSE]
disabled=[OPTIONAL; **default**=FALSE] >
<**option**
value=[REQUIRED]>
OPTION-LABEL
</**option**>
</**select**>

Table 2: XSLT fragment to edit `DisplayStyles`

```
<!-- [...] -->
<xsl:for-each select="Style">
  <tr><td>
  <!-- DisplayStyles/Style@Name -->
  <tt>
  <xsl:element name="input">
    <xsl:attribute name="type">
      <xsl:text>text</xsl:text>
    </xsl:attribute>
    <xsl:attribute name="size">
      <xsl:text>10</xsl:text>
    </xsl:attribute>
    <xsl:attribute name="name">
      <xsl:value-of select="fanxe:path()"/>
      <xsl:text>@Name</xsl:text>
    </xsl:attribute>
    <xsl:attribute name="value">
      <xsl:value-of select="@Name"/>
    </xsl:attribute>
  </xsl:element>
  </tt>
  <!-- [...] -->
</xsl:for-each>
<tr><td>
<!-- Insert DisplayStyles/Style -->
<font color="#0000FF">
<xsl:element name="input">
  <xsl:attribute name="type">
    submit
  </xsl:attribute>
  <xsl:attribute name="size">
    <xsl:text>0</xsl:text>
  </xsl:attribute>
  <xsl:attribute name="name">
    <xsl:value-of select="fanxe:path()"/>
    <xsl:text>/Style:ADD</xsl:text>
  </xsl:attribute>
  <xsl:attribute name="value">
    <xsl:text>Add a Style</xsl:text>
  </xsl:attribute>
</xsl:element>
</font>
</td></tr>
<!-- [...] -->
```

creating subclasses (see section 4), I always create a script with the same functionality.

In addition to the `GUI` method, **FanXE** has a number of methods that can be called with or without the X environment. I will briefly discuss a few of these. All public and protected methods and variables are documented in the man page.[7]

3.1 `transform` Method

The `transform` method is an interface to **tdom**'s XSLT engine, one of the fastest and least buggy XSLT engines available (and a primary reason for its inclusion in **FanXE**; see section 5.1 for more details).

You specify an XML file (or a DOM already loaded in memory) and a style sheet and the XSLT engine will transform the results. You can pass parameters to the style sheet, and you can have the results either serialized or saved as a new DOM. I added a *`-document`* option to add support for the `xsl:document` extension allowing you to serialize the output into multiple files.

In addition to all the standard XSLT 1.0 functions and elements, I've written a few XSLT functions in Tcl. These include `fanxe:tcl` which will execute the string argument in a safe Tcl interpreter and return the results and `fanxe:mtime` which will return the file modification time of the file name passed (the safe interpreter does not include the `file` command).

The `exslt`[8] extension elements work as-is with **tdom** (and therefore with **FanXE**), but the functions distributed with `exslt` do not work with **tdom**. I partially implemented the `exslt:seconds` and `exslt:date` functions with Tcl code within **FanXE** and these functions are available with the `transform` method.

Subclasses of **FanXE** can define their own Tcl-written XSLT extensions.

3.2 `compare` Method

FanXE can store XML documents under revision control (see section 4.1.3). The `compare` method will compare any two revisions of the XML document.

Each document will be split into segments specified by an XPath expression (the default is `//Series`). Each XPath segment needs a unique key attribute.

Serialized XML text will be returned with the changes. Nodes in the original revision will be marked either as <**Change** ***Type="Replace"***> if replaced or <**Change** ***Type="Delete"***> if deleted. Nodes in the final revision will be marked as

[7] The man page is automatically generated by a script from the source code. As long as the methods and variables are properly commented, generating up-to-date documentation is simple.

[8] *http://www.exslt.org/*

Table 3: Sample Output of `compare` method

The description of MDRM="B565" was changed and MDRM="C869" was added.

```
<Differences Old="revision 1.20" New="revision 1.24">
  <Change Type="Replace">
    <Series MDRM="B565">Federal Home Loan Bank advances remaining maturity 1-3 years
      <Item FFIEC="31" MNEM="RCFD" BeginDate="20010331" EndDate="99991231" Location="RC-M_5.a.(2)"/>
      <Item FFIEC="41" MNEM="RCON" BeginDate="20010331" EndDate="99991231" Location="RC-M_5.a.(2)"/>
    </Series>
  </Change>
  <Change Type="With">
    <Series MDRM="B565">FHLB advances remaining maturity 1-3 years
      <Item FFIEC="31" MNEM="RCFD" BeginDate="20010331" EndDate="99991231" Location="RC-M_5.a.(2)"/>
      <Item FFIEC="41" MNEM="RCON" BeginDate="20010331" EndDate="99991231" Location="RC-M_5.a.(2)"/>
    </Series>
  </Change>
  <Change Type="Add">
    <Series MDRM="C869">Cash and balances due from dep. insts; items not subj to risk-wghting
      <Item FFIEC="31" MNEM="RCFD" BeginDate="20050630" EndDate="99991231" Location="RC-R_34._B"/>
      <Item FFIEC="41" MNEM="RCON" BeginDate="20050630" EndDate="99991231" Location="RC-R_34._B"/>
    </Series>
  </Change>
</Differences>
```

<**Change** ***Type="With"***> if replaced or <**Change** ***Type="Add"***> if added. See Table 3 for a sample of the output.

3.3 `fork` Method

When loading XML documents or XSLT style sheets stored under revision control (see section 4.1.3), **FanXE** will always try to load the revision with the symbolic name defined by the class option `-rev` (the default value is "Production"). To begin working on changes while retaining the original symbolic name, you can use the `fork` method. This will identify all files stored in RCS used by **FanXE** and create a new symbolic name (e.g. "Development").

Note that **FanXE** is doing nothing more than labeling revisions with the new tag. It does not actually create a new branch in RCS and has no means for merging changes from different revisions. I typically execute "`fork Production Development`", make changes under "Development," generate a difference report between revisions, and then execute "`fork Development Production`".

4 Application Examples

The best way to see the other features of **FanXE** is to look closely at various subclasses that I have developed. Since first writing **FanXE** in October 2002, I have developed about a dozen applications using **FanXE** as the basis. I'll discuss three applications here and their unique requirements.

4.1 ItemSet: Report Form Meta Data

The **ItemSet** class was the first application written with **FanXE** and was the driving factor of its design. The purpose of this application is to define and maintain a *Uniform Item Set* of commercial bank financial data that can be adjusted for mergers. Using the *Uniform Item Set* XML document, we create SAS[9] and FAME[10] programs (via XSLT) to build micro- and macro- databases of commercial bank balance sheets and income statements.

FanXE was originally written with the goal of being the front-end GUI editor for this application. When developing it, I knew that I would need an editor for other XML applications and I was determined to make **FanXE** as modular as possible. I also knew that I would want to write applications that used the XML documents without the GUI (an XSLT processor at the minimum), so the **FanXE** GUI was always intended to be only one portion of the application.

[9] *http://www.sas.com/*

[10] *http://www.fame.sungard.com/products/fame9.html*

4.1.1 Background on Merger-Adjusted Call Reports

Every quarter, all commercial banks in the United States are required to file balance sheet and income statement information (the FFIEC Call Report[11]). The Federal Reserve Board publishes an annual article in the Federal Reserve Bulletin on the profitability of commercial banks in the United States. We use Call Report data for our analysis.

On the Call Report income statement, data are reported year-to-date. When one bank merges with another, the surviving bank does not always report the income of the acquired bank. Since we are interested in the profitability[12] of banks, underestimating income biases our estimates down. As more and larger banks merge,[13] the bias becomes more severe.

To capture the missing income data, we create a merger-adjusted version of the Call Report which adds the prior-period year-to-date income of all predecessors of mergers (as well as an estimate of the current-quarter income) to the successor's income statement.

However, banks file different report forms if they have foreign offices and smaller banks generally report less detail than larger banks. These forms also change over time (historically, different forms even requested entirely different details of loan categories). We therefore first have to create a common set of items that are reported on all forms and track when items are added, deleted or redefined. We then derive estimates of the details not reported by smaller banks.[14] This defines our *Uniform Item Set* which we can then adjust for mergers.

4.1.2 Defining the Meta Data

We wanted to store all the structural information about the original report forms as well as our *Uniform Item Set* in XML documents. These files would constitute the meta data for our application. They contain no financial data, just an index of Call Report series and the definition of the series in our *Uniform Item Set*.

XML is a perfect language for this type of data because of its natural tree structure.[15] In the *Uniform Item Set*, some fields are inputs (direct mappings from the raw Call Report) while others are formulas (sums of other series in the *Uniform Item Set*). Each item can have a different definition for different time periods. Some items are used to derive other detail items for different reporters.

4.1.3 Revision Control

For this application, I also needed to keep track of all revisions, and I needed the ability to generate difference reports whenever the Call Report changed. I therefore designed **FanXE** with the ability to store documents using the Revision Control System (RCS).

FanXE does not require that documents be stored in RCS and the RCS features can be turned off (`-useRCS 0`). However, when **FanXE** detects a file is stored in RCS (and `-useRCS` is on), it will automatically check documents in and out of RCS when loading and saving.

The `compare` method (see section 3.2) was written to build these reports. The **ItemSet** subclass adds a `diffHTML` method which calls `compare`, applies a style sheet to the results (which adds a summary and links in HTML) and posts the HTML on our internal website.

4.1.4 Cross-Referenced DOMs

When the Call Report changes, I use **FanXE** to manually edit the Call Report Index. I then run a method in the **ItemSet** class that will identify and open for editing any nodes in the *Uniform Item Set* DOM that are effected by the changes. Both XML documents are reasonably large[16] and **FanXE**'s performance is excellent.

In addition to the Call Report Index and the *Uniform Item Set*, I also maintain the size groups and lists of series that will be seasonally adjusted in **ItemSet**. All of the XML documents loaded by the **ItemSet** class are related to each other. I added the `fanxe` URI scheme (see section 2.6) to **FanXE** for this reason. For example, when browsing any node in the *Uniform Item Set*, you can click on the original series name and be taken to its entry in the Call Report Index, which will tell you for what dates that series is available and its location on the reporting forms. Alternatively, if you find an item of interest on the Call Report form, you can look

[11] *http://www.ffiec.gov/ffiec_report_forms.htm#CallReports*

[12] One measure of profitability is "Return on Assets," defined as the ratio of total income over a period to total average assets over that period.

[13] In 1985, there were nearly 15,000 commercial banks. Today, there are fewer than 8,000.

[14] We use the average aggregate proportions of the missing detail from larger banks and apply those ratios to the micro data of the smaller banks.

[15] When I first created this database, all information about the report forms were stored in SAS data sets. This was not very efficient and the SAS programs that parsed the data sets to build the database and create HTML references were complex. A lot of fields were repeated (the series name and attributes were repeated for each date change) and others were often blank or used for different purposes depending on whether the specific item was an input or a formula. Making the transition from SAS to XML was not difficult because I was able to export the SAS data sets to XML and use XSLT style sheets to transform them into the tree structure that I desired.

[16] The Call Report Index XML document is over 12,000 lines. The *Uniform Item Set* XML document is over 25,000 lines.

it up in the Call Report Index and click on the series to see where it is used in the *Uniform Item Set*.

4.1.5 CGI Interface to the *Uniform Item Set*

Once all changes are final, another method is called that builds static HTML index pages for the Call Report Index and the *Uniform Item Set*. In addition to the static pages, I have built CGI applications that can query these XML documents. One query, for example, will recursively follow all formulas in the *Uniform Item Set*, cross-reference it with the Call Report Index, and show a table with the locations of all input series from each report form.

4.2 BCBreaks: Storing XML in PostgreSQL

In addition to the quarterly Call Report data, we maintain weekly balance sheets of large commercial banks and a sample panel of small commercial banks and branches and agencies of foreign banks. From week to week, we need to make corrections, called "breaks," to the aggregate time series from these reports to account for mergers, revisions, and panel adjustments.

Each break record is stored in an XML document with data about the institution responsible for the break as well as attributes about the break such as the start and end dates and levels of the break. The **BCBreaks** class is an interface for storing the breaks.

An XSLT style sheet is applied to the breaks XML document to create a FAME program that applies the breaks in the database.

Like the **ItemSet** application, this data was originally stored in SAS data sets. This had a number of drawbacks, including the fact that only one person could edit the data set at a time and, if left open, the application would crash when attempting to apply the breaks.

I decided to store each break in a PostgreSQL database to allow multiple users to edit simultaneously. I needed to modify **FanXE** to handle XML documents stored in something other than flat files. I added the <**loadDOM**>, <**saveDOM**> and <**saveNode**> elements to the *fanxe.xml* configuration file. These elements can contain Tcl scripts to load or save the entire document or to save a single node.

The **BCBreaks** class has methods to handle loading and saving nodes from the PostgreSQL database. The <**saveNode**> script adds, deletes or modifies the break records as users edit the DOM. In addition to making the change, a log is written with the action and *inode* that was edited and a `NOTIFY tableUpdated` command is sent to the PostgreSQL database.

When the GUI is started, it registers `pg_listen` to wait for `tableUpdated` events. When an event is triggered, the PostgreSQL log is referenced and any modifications are mirrored in real time in the GUI. If you have a node open for editing and another user deletes it, it will disappear from your screen. As soon as a user adds a new break, it will appear in everyone's GUI.

4.3 SDMX: Time Series XML Data

SDMX[17] is a new standard (ISO/TS 17369:2005 SDMX) for exchanging statistical data among government agencies. The Federal Reserve Board is planning to adopt SDMX for our new Downloadable Data Project. Content providers within the Board will export data from FAME to SDMX-ML and provide the XML documents to the public website maintainers. Maintainers are currently developing the front-end interface that will allow the public to query and download the vast amounts of data the Board collects and maintains.

The **SDMX** application is used by content providers inexperienced with XML to build the complex SDMX-ML documents.

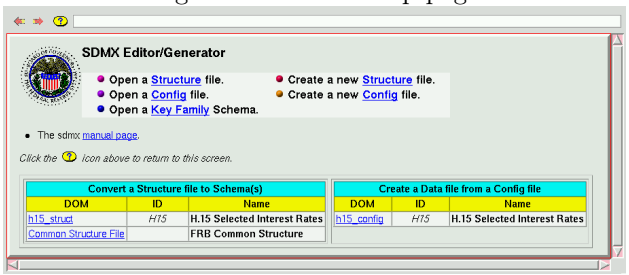
4.3.1 Application Front End

FanXE needed to be more than a simple XML document editor. While command line interfaces are useful, memorizing and issuing commands makes for a complicated work flow. Therefore, **FanXE** needed to be able to issue commands and show help specific to the application.

The help page ( ) for the **SDMX** class is dynamically generated from one of its public methods based on XML documents loaded into memory (Figure 7). This allows users to create and edit SDMX structure, schema and data files.

[17] *http://www.sdmx.org/*

Figure 7: **SDMX** help page



4.3.2 Cloning DOMs

Other **FanXE** applications discussed so far had hard-coded XML documents that were always loaded. In the case of **SDMX**, each user will need to load their own XML documents and may need to open multiple files of the same type.

I added the `domClone` method to **FanXE** that will load a new XML document created by copying the structure (Tree layout, style sheets used, etc.) of an existing XML document. **SDMX** users can use this to load new structure and configuration files.

5 Choosing Packages

My site is fairly homogeneous. All users connect to a their section's Linux server[18] via Exceed.

Because I do not have to worry about shipping a product and I have absolute control of what packages are available on the Linux servers, I have not been conservative with the use of other packages. **FanXE** has a liberal mix of widgets from **Tix**, **BLT**, **iwidgets** and more. **FanXE** was developed with **TclPro** 1.4.1 (which uses **Tcl/Tk** 8.3).

This section provides a brief overview of why I chose some packages over others.

5.1 tdom versus tcldom

In the first versions of **FanXE**, I actually included support for both **tdom** and **tcldom**. The **tdom** package at that time (version 0.7.4) had some XSLT bugs that I couldn't work around, but was otherwise much faster than **tcldom** (which used libxslt for XSLT functions). After reporting my problems to the **tdom** list, Rolf Ade was quickly able to correct the bugs. Within three months, all bugs I had found had been resolved. At this point, the speed showdowns between **tdom** and libxslt demonstrated that **tdom** was well ahead of the competition. Rolf and others have done an excellent job quashing bugs and keeping **tdom** ahead of the rest. There are no known outstanding bugs in **tdom**. It does not leak memory and has a perfectly reasonable memory footprint. However, it does not yet support XSLT 2.0 functions.

5.2 Tkhtml versus ihtml

Building an editor that was going to view HTML obviously required an HTML widget. I initially tried the **iwidgets** `ihtml` widget, but it simply couldn't handle nested tables. **Tkhtml**, however, has been able to handle every nested table I have ever thrown at it.

Tkhtml doesn't support CSS and, to date, there hasn't been much development. There has been some recent activity and promise of a new version, but to date the current version of **Tkhtml** does everything I need.

5.3 Tix versus BLT

I consider some of the **BLT** package to be irreplaceable (e.g. `blt::exec`). Every widget available in **Tix** has a **BLT** counterpart, yet I still chose to use some **Tix** widgets over some **BLT** widgets.

5.3.1 `tixTree` versus `blt::hierbox`

I needed a tree widget for the Tree pane (see section 2.1). This widget needed to be high performance, so only C-coded widgets were considered. Between **Tix** and **BLT**, **Tix** won my speed test by only a slight margin.

I ultimately chose **Tix** because I liked the `tixDisplayStyle` (see section 2.4). This allows me to configure and reconfigure the font for every node in the tree based on the element.

I also have more familiarity with the **Tix** API as I have been using it for mega-widgets for as long as I have been programming in **Tcl/Tk**. The lack of ongoing support for **Tix** has made me consider replacing the **Tix** widgets and I may yet do so. The API for **Tix** is noticeably different from **BLT** and others so that this will not be a simple process. So far, by the time I've upgraded to the latest version of **Tcl/Tk**, someone has always had a patch to keep **Tix** alive. I continue to hope that someone will keep **Tix** going.

5.3.2 `blt::tabset` versus `tixNoteBook`

For the Tabbed pane (see section 2.3) I needed a tabset widget. **Tix** has `tixNoteBook`, **iwidgets** has `iwidgets::tabset` and **BWidget** has `NoteBook`. Only the **BLT**'s `blt::tabset` offers detachable tabs. I considered the ability to view and edit multiple nodes at the same time as a critical feature and chose **BLT** for this reason alone.

6 Availability

FanXE was created as part of my work for the Federal Government. It is therefore not copyrightable and available in the Public Domain. **FanXE** is available at *http://wklink.freeshell.org/*.

[18] We recently migrated from Solaris® 2.8 to Red Hat® Enterprise Linux® Version 3. None of my **Tcl/Tk** code required any changes.

7 Thanks

I have stood on the shoulders of many giants to build **FanXE**. Thanks to John Ousterhout for creating **Tcl/Tk**, Jeff Hobbes and the rest of the **Tcl/Tk** core team maintaining everything, Michael McLennan for **[incr Tcl]**, Jochen Löwer and Rolf Ade for **tdom**, D. Richard Hipp for **Tkhtml**, George Howlett for **BLT**, Ioi Lam for **Tix** (and others who have kept it working), and the entire community for providing answers whenever I've had questions.