Managing XML Documents Versions and Upgrades with XSLT

Vadim Zaliva, lord@crocodile.org

2001

Abstract

This paper describes mechanism for versioning and upgrding XML configuration files used in *FWBuilder* project.

1 Introduction

While most software programs should include configuration data, no common standard for presentation of such data exists. Windows programmers tend to use the registry, while UNIX programmers prefer plain text configuration files in either the */etc* directory or a *"dot-program name"* in the user home directory. Such files contain a variety of data formats. The recent surge of XML popularity may make it the perfect candidate for a unified language to present configuration data.

Besides syntax variety, another major challenge with configuration files is how to upgrade software without losing data. Common methods used by software developers include:

1. *Do nothing.* It is the user's responsibility to back up all data before an upgrade and to convert it to the new format once the upgrade is completed.

2. *Include a backup.* The upgrade procedure backs up old files before upgrading the software. The user must then merge the changed format back to the old files.

3. *Make new formats backward-compatible.* The developer makes an effort to keep new data formats backward-compatible so old files can be loaded into newer versions of the software. The new software is smart enough to fill in missing fields in old data.

4. Provide an upgrade procedure. The software recognizes and converts older files to the new format.

From the user's point of view, this last approach is, of course, the most desirable. Unfortunately, however, it can cause great pain for the developer.

Supporting all data formats throughout the life of a program entails considerable effort.

During work on Firewall Builder (see http://www.fwbuilder.org/), we developed an approach which simplifies the use of XML for versioning and maintenance of configuration files, described in detail below.

2 Preliminaries

Let's assume that your program works with several XML files, such as user preferences, GUI resources, and data. It is a good practice to use DTDs (document type definitions) to specify the format of each file type. While our approach does not require documents to have DTDs (which is allowed by the XML specification for well-formed documents), files with the same format should have the same document type.

Next, you need to decide on a versioning schema. While you may create a separate version sequence for each file in addition to a version number for the software itself, sometimes it is more convenient to assume that the data format version number is the same as that of the software. Our procedure allows either approach. Whatever versioning scheme you use, you must ensure that the version numbers can be compared to determine which is newer. A common software versioning scheme employing a series of digits separated by dots is supported in our library. If your scheme allows nondigits in version numbers, you will have to customize our code by writing a version-comparison function.

Let us look at an XML file fragment:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!DOCTYPE FWBuilderPreferences SYSTEM "fwbuilder_preferences.dtd">
3 <FWBuilderPreferences version="0.9.7">
```

Line 1 indicates that this is an XML file conforming to the XML version 1.0 standard, with UTF-8 text encoding.

Line 2 declares that this document is of type *FWBuilderPreferences*. Its DTD is *fwbuilder_preferences.dtd*

Line 3 opens the root element, whose name must match the document type declared in line 2. Here we use the attribute *version* to specify the data format version used in this document. In fact, this attribute is part of the user data. This is the only common part of user data that must be present in all data files. Without it, it would be impossible to distinguish one version from another. If you decide to use our procedure on existing project files that do not have this attribute, our code assumes that a missing version attribute corresponds to a certain initial version number, for example *0.0.0*.

To do the actual data conversion we use XSLT, a very convenient technology that allows a developer to write a style sheet to transform one XML document to another. The style sheet itself is an XML document.

3 Procedure

We will now assume the user has a document with a version 1.0 data format. A software upgrade is released using a version 1.1 data format. The new software includes an XSLT transformation that upgrades the format from 1.0 to 1.1. When version 1.2 comes out, it is desirable to be able to convert both 1.0 and 1.1 to the new format. This could be achieved either by writing a direct transformation from 1.0 to 1.2 or by using 1.1 as a transitional format ($1.0 \rightarrow 1.1 \rightarrow 1.2$). The best approach depends on your application. Our procedure allows either to be used.

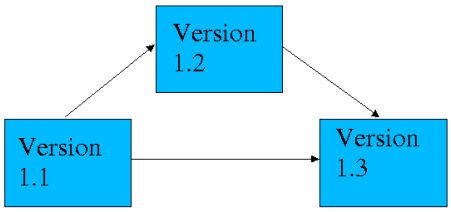
For each document type, the software vendor ships a series of XSLT style sheets enabling transformations among different version numbers. These are identified by the *original* version number, since the transformation procedure does not know in advance what will be the resulting version number produced by the transformation, except that it will be greater than or equal to the current one.

Our basic upgrade procedure is as follows:

1. Load the original data file.
2. Check the document format version number. If it is the same as the new one, we are done. If it is greater than the new one, display an error message indicating that the format is from a future version. Otherwise proceed to next step.
3. If the document format version number is less than the new one, locate and run the appropriate upgrade transformation for this document type, which is executed in memory.
4. If the resulting document format version number is still less than the new one, repeat step 3. If it is greater, it indicates a programming error during the transformation.
5. If the converted file is now in the desired format and this procedure is used in interactive mode, display a message indicating the data file has been converted and suggesting that the user save it.

It is important that each transformation update the *version* attribute to reflect the new data format version number.

Each transformation updates the data format to a newer version than the current one, but it does not have to be next one or the latest one since a transformation could skip a version or versions. Figure 3 shows an example in which format version 1.2 removes some attributes but 1.3 restores them to the way they were in 1.1. Thus, when upgrading from 1.1 to 1.3, the attribute values should be preserved, while when upgrading from 1.2 to 1.3 attribute values will have to be reset to the original defaults. This scenario grants writing two separate upgrade transformations for $1.1 \rightarrow 1.3$ and $1.2 \rightarrow 1.3$.



Typically, an incremental upgrade will be used. The software developer for each version needs to write just one transformation that upgrades from the previous one. This is enough to support all older format versions, provided you ship all transformations accumulated during previous releases.

The procedure can be interactive or noninteractive. In the noninteractive approach, you can use software to upgrade data files internally, shielding the user completely. When employing the interactive approach, in which when the user is led through the process of upgrading the data files (for example, instructing the user to choose the *Load* command from the *File* menu), you may wish to warn the user that files in the older format will be converted to the new format.

However, if a file is opened in read-only mode, there is no need to save the results of an upgrade transformation. Since all upgrades can be performed in memory without modifying the original file, this is an ideal approach for allowing a user to try a new software version with the option of returning to the previous version.

4 Implementation

The procedure described above was used in Firewall Builder. While it happens to have been written in C++, it could have been written in other languages as well, Java for instance. Firewall Builder uses *libxslt* and *libxml* libraries from the GNOME project, and the upgrade procedure is implemented as part of the projects API library, *libfwbuilder*.

The program includes several XML files. We used the upgrade procedure described earlier to update user preferences, data files, and resource files. All upgrade transformations are shipped in one directory, in which there are subdirectories for each version number. In each subdirectory there is one transformation file for each document type. For example, */usr/share/fwbuilder/migration/* contains these directories:

- *0.10.0/*
- *0.10.1/*

- *0.8.7/*
- *0.9.0/*
- *0.9.1/*
- *0.9.2/*
- *0.9.3/*
- *0.9.4/*
- *0.9.5/*

Directory 0.9.2, for example, might contain the file *FWBuilderPreferences.xslt*, an XSLT style sheet that converts XML documents of document type *FWBuilderPreferences* (a user preferences file) from version 0.9.2 to version 0.9.3 by adding an *Autosave* child element under the UI element:

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

<xsl:output method="xml" version="1.0"
    doctype-system="fwbuilder_preferences.dtd" indent="yes" encoding="utf-8"/>

<xsl:template match="FWBuilderPreferences">
  <FWBuilderPreferences version="0.9.3">
    <xsl:apply-templates/>
  </FWBuilderPreferences>
</xsl:template>

<xsl:template match="UI">
  <UI>
    <Autosave>false</Autosave>
    <xsl:apply-templates/>
  </UI>
</xsl:template>

<xsl:template match="*">
  <xsl:copy-of select="."/>
</xsl:template>

</xsl:stylesheet>
```

In our project we used a DTD to define XML file structure. Unfortunately, since DTD files themselves are not in XML format, they cannot be updated using this procedure (it will be possible in case of XML Schema). Because of this, we ship only the DTD for the most recent version. When we are loading a data file for the first time, we switch off DTD validation in the parser to allow loading files from older versions for which there is no DTD. Once it is loaded in memory, we perform the upgrade procedure. After the upgrade is complete and we have a file with the desired format version number, we revalidate it using the current DTD. This ensures that transformation results conform to the current DTD.

Most of the internal workings of the upgrade facility is hidden from the programmer. When there is a need to load the XML file, the programmer calls the following method:

```
xmlDocPtr XMLTools::loadFile(const std::string &file_name,
                             const std::string &type_name,
                             const std::string &dtd_file,
                             const UpgradePredicate *upgrade,
                             const std::string &template_dir,
                             const std::string &current_version
    ) throw(FWException);
```

This method takes care of all data loading, format conversion, and validation. If something goes wrong, an exception will be thrown . If it succeeds, the program returns a parsed document in current format. Let's take a brief look at the parameters:

file_name Name of the file to be loaded. It must exist and be readable.

type_name Expected document type. While loading, it will be compared with the document type within the document and an exception will be thrown if it does not match.

dtd_file DTD file name (relative to the *template_dir* parameter).

upgrade User can pass here a pointer to an instance of the class, which will be called to determine if the upgraded file should be saved in place of the old one after conversion. This predicate could be interactive or noninteractive. If it is interactive, it could (as it does in our case) generate a dialog box asking the user to make the decision about replacing the old file with the new.

template_dir Path to the directory where DTDs and upgrade transformations are stored (for example, */usr/share/fwbuilder/*). Transformations should be located in the *migration/* sub-directory. DTD files should be located in the top directory.

current_version Expected current format version. For example *"1.7"*.

You can download and view the complete implementation of the upgrade procedure described in this article in the source code of the Firewall Builder project.

References

[1] T. Bray, J. Paoli, C.M. Sperberg-McQueen, et al. Extensible Markup Language (XML). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210.*

[2] J. Clark et al. XSL Transformations (XSLT) Version 1.0. *W3C Recommendation*, 16(11), 1999.

[3] Firewall builder project. http://www.fwbuilder.org/.

[4] 'libxml' and 'libxslt' xml and xslt libraries for gnome project. http://www.xmlsoft.org/.