XML SCHEMA REPRESENTING AN EBIF TEMPLATE DEFINITION, METHOD FOR AUTO-GENERATING SCHEMATIC INSTANCES FROM ORIGINAL EBIF SOURCE CODE AND CONSTRAINING CUSTOMIZED INSTANTIATIONS OF THE RESULTING TEMPLATE

Mike McMahon and Lea Anne Dobbins
Comcast Media Center

Abstract

While there is no standardized source code language for EBIF, the commonly used authoring tools and compilers use XML as their source syntax. Peripheral XML standards such as XML Schema, XPath and XSLT can therefore be leveraged in validation, transformation and marshalling of EBIF source trees. Presented here is a methodology in which an arbitrary EBIF application, developed by any iTV vendor can be automatically "templatized" such that its original source tree is subsequently used to generate data driven, customized instantiations. Our ambition is to alleviate toolset incompatibility resulting from proprietary syntax, compilers and customization toolsets, thereby restoring the spirit of open standards to the end-to-end workflows associated with templating and customizing EBIF applications.

THE NEED FOR TEMPLATING IN EBIF

In order to re-purpose, brand or skin interactive television applications, a non-technical person (e.g. a brand manager) should be able to simply select an underlying template and supply the desired text, graphics and colors needed to generate a customized instantiation. In order to achieve this aim it is necessary to isolate the core logical and functional attributes of an application as a "template," whereby the customized data and stylistic attributes are supplied separately, in a non-technical and user friendly way, in order to define a given "instance." These capabilities need to be fluid enough to demonstrate unique branding and creative elements in the template "instance."

PROBLEM WITH CURRENT SOLUTIONS

Data, logic and presentation

There is a mechanism within EBIF to separate application logic, represented in binary form as a .PR file from application data, represented as a .DR file. This type of separation is certainly useful in order to iterate through data sets in applications such as those that fetch dynamic RSS feeds.

Nonetheless, logic, data and presentation remain largely coupled within EBIF. Consequently, true isolation of a core, logical template in order to expose only the stylistic and data qualities of an application to a non-technical brand manager is not feasible within the current construct. Presentational qualities and logical data binding directives are necessarily part of the core source code, as opposed to an accompanying properties file.

Even if such qualities were to be encapsulated in an external properties file there would still be need to enforce constraints such as string length, image format and dimensions, etc on any given set of instantiation properties. The most effective and safe way to generate a custom application is, therefore, to go back to the original application author with a set of requirements.

While this need not amount to much more than a copy, paste, compile effort on the part of the developer it is neither an efficient use of the developer nor the technology. In

addition, such an approach clearly raises a variety of quality assurance concerns as the underlying code base cannot be assumed to be a static entity. The custom instantiation would, therefore, need to go through a test cycle and although somewhat redundant, in many cases, this test cycle will need to be as comprehensive as that conducted on the original code base.

Proprietary Template Toolsets

There are several "template toolset" products available. Such products do indeed solve many of the problems identified above. They typically provide a simple, non-technical customization interface allowing a user to select from a pre-defined set of underlying templates and provide custom graphics, text and color schemes. These values are then used in a find and replace mechanism against the application's original source code such that a new, unique code base is generated and sent to a compiler, generating a custom instantiation. The more mature systems are additionally validating and constraining user input thereby enforcing output quality.

Such template toolset products unfortunately ship with a fixed set of baseline templates and corresponding customization GUIs. Customized instantiations are confined to the functional capabilities of those pre-canned templates. In order to add an additional functional template to the toolset one must work with that vendor to define and develop it.

It is generally not possible to ingest an application developed by a third party into these types of proprietary toolsets. Moreover, because "templatization directives" are not exposed, any new applications that are added to the system must be authored in a proprietary SDK, typically provided by that same vendor.

Therein lay the primary problem we seek to address: the lack of interoperability between EBIF authoring and templating or customization tools.

THE TEMPLATE DEFINITION SCHEMA

We aim to define an open standard XML schema which is intended to serve as a structured, strongly typed interchange between EBIF authoring tools and the template toolsets used for re-skinning and customizing specific instantiations of those applications.

Specifically presented here is an XML data model we call templateDefinition.xml and a functional reference implementation as it relates to ingesting a new application into a template toolset. The data model and methodology described herein allows an arbitrary EBIF code base, authored outside of and independently from the template toolset, to be ingested into the template toolset in a manner in which the "templatizable aspects and constraints" are identified and understood.

The application can then be added to the set of available templates, allowing the template toolset to reliably render, capture and validate the instantiation parameters in its customization GUI.

Our belief is that such a schematic representation of "where and how an application is customizable" should be adopted as an open standard, thereby allowing applications developed in any EBIF authoring tool to be ingested into external, third party customization tools.

Template Definition XML

Figure 1 below illustrates the crux of the data model. It provides document pointers to each file within the code base and, for each file, XPaths to the precise nodes and/or

attributes that could be reasonably and successfully customized. For each of these "templatizable items" the necessary constraints on the instantiation parameters are additionally defined.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <codeBase>
  + <doc location="TemplateTest/application/app-desc.xml">
  - <doc location="TemplateTest/application/page-resource/page/index.xml">
    - <templateItems>
      - <item id="31">
          <xpath>/mac/var[1]</xpath>
          <nodeType>var</nodeType>
        - <passThroughValues>
          - <attributes>
              <name>templateItem-theText</name>
              <type>string</type>
            </attributes>
          </passThroughValues>
          <transfromation default="Go Blue" type="element" />
        - <constraints>
            <constraint type="required">true</constraint>
            <constraint type="length">7</constraint>
          </constraints>
        - <originalNode>
            <var name="templateItem-theText" type="string">Go Blue</var>
          </originalNode>
          <name>templateItem-theText</name>
          <description>templateItem-theText</description>
        </item>
      </templateItems>
    </doc>
  + <doc location="TemplateTest/application/page-resource/style/palette.xml">
  </codeBase>
```

Figure 1: templateDefinition.xml

Physical Location of Source Code

The template toolkit needs build time access to the original source code in order to compile a given instance. We use URIs as pointers to the location of the original source code. With respect to third party IP, there are a couple options in terms of the physical location of the source code. It could be hosted by the original application developer and dynamically accessible to a template toolkit when compiling a custom instance. Alternatively, application authors could upload source code to the template toolkit. Either scenario necessitates contractual protection of the source code and associated IP.

Generating the Template Definition XML

The original developer of a given application is, clearly, the authority with respect to establishing which elements within the application could or should be safely customized. We therefore seek a mechanism whereby the original developer can compile a default instantiation while establishing specific text strings, variables, integers, colors or graphics as "customizable."

Likewise, the original developer is best able to define necessary constraints during the customization process. For example, the default value of a given text message within the application might be ten characters long. A message of twelve characters would be perfectly acceptable but a message of fifteen or more characters would cause line wrapping, detrimental to the visual appearance of the overall screen. It is therefore necessary to solicit not only the "templatizable aspects" of the application from the original developer, but also the corresponding constraints.

Our template definition XML, of course, describes both. The question, however, becomes how is that definition itself generated? Our view is that if such a data model were widely adopted it would likely be the case that EBIF authoring tools would automatically generate the template definition XML as a supplementary output of the authoring and compilation process. Perhaps application authors would highlight blocks of code and right click to bring up a dialogue box in order to capture the constraint definitions.

In lieu of template definition files generated from an authoring tool we had need to supply our own by way of an external, supplementary file. It is far from desirable to introduce the risk of human typos when creating the template definition XML as a freehand effort. It was also not possible to inject innocuous markup into the source code as the compilers would reject the syntax.

In order to automatically generate a compliant template definition XML file and remain both agnostic and innocuous to existing compilers we introduced a naming convention in the authoring syntax such that the application author prefixes potentially

customizable areas of the source code with *templateItem-*. This allows the original application author to surgically pinpoint specific areas of the source code that can, should or must be customized. Additionally, because this is a naming convention as opposed to an extension of the source syntax itself, it does not affect the existing compilers and can be used within any XML based EBIF source syntax.

Given such a naming convention within the underlying source tree we are able to programmatically traverse the whole of the application's source and extract the precise location of all "templatizable aspects" of the application as defined by the original application author. Figures 2 and 3 below illustrate an XSLT script that will traverse an EBIF source tree written in the TVWorks MAX syntax and generate a normalized template definition XML file. The logic in the XSLT will automatically derive the XPath and constraints. It takes a first pass through the source tree, indentifying each node flagged as "templatizable" by the author and holding them in memory.

```
<xsl:param name="srcDir" as="xs:string" required="yes" />
- <xsl:variable name="theCollection">
    <xsl:value-of select="$srcDir" />
    ?select=*.xml;recurse=yes
  </xsl:variable>
  <xsl:output method="xml" indent="yes" encoding="ISO-8859-1" />
- <xsl:template match="/">
  - <xsl:variable name="start">
    - <codeBase>
      - <xsl:for-each select="collection(iri-to-uri($theCollection))">
        - <xsl:variable name="theDoc">
            <xsl:value-of select="document-uri(.)" />
          </xsl:variable>
        - <xsl:element name="doc">
          - <xsl:attribute name="location">
              <xsl:value-of select="$theDoc" />
            </xsl:attribute>
          - <xsl:element name="templateItems">
            - <xsl:for-each select="document($theDoc)//node()">
              - <xsl:variable name="theAtt">
                  <xsl:value-of select="substring(@name,1,13)" />
                </xsl:variable>
              - <xsl:if test="($theAtt='templateItem-')">
                - <xsl:element name="item">
                  - <xsl:element name="xpath">
                      <xsl:value-of select="saxon:path()" />
                    </xsl:element>
                  - <xsl:element name="originalNode">
                      <xsl:copy-of select="." />
                    </xsl:element>
                  - <xsl:element name="nodeType">
                      <xsl:copy />
                    </xsl:element>
                  </xsl:element>
                </xsl:if>
              </xsl:for-each>
            </xsl:element>
          </xsl:element>
        </xsl:for-each>
      </codeBase>
```

Figure 2: XSLT first pass traverse

We then take a second pass through the memory tree in order to analyze individual node context, group and define each of the items:

```
- <xsl:if test="nodeType/var">
    <xsl:element name="nodeType">var</xsl:element>
  - <xsl:element name="passThroughValues">
    - <xsl:element name="attributes">
      - <xsl:element name="name">
          <xsl:value-of select="originalNode/var/@name" />
        </xsl:element>
      - <xsl:element name="type">
          <xsl:value-of select="originalNode/var/@type" />
        </xsl:element>
      </xsl:element>
    </xsl:element>
  - <xsl:element name="transformation">
      <xsl:attribute name="type">element</xsl:attribute>
    - <xsl:attribute name="default">
        <xsl:value-of select="originalNode/var" />
      </xsl:attribute>
    </xsl:element>
  - <xsl:element name="constraints">
    - <xsl:element name="constraint">
        <xsl:attribute name="type">required</xsl:attribute>
        true
      </xsl:element>
    - <xsl:element name="constraint">
        <xsl:attribute name="type">length</xsl:attribute>
        <xsl:value-of select="string-length(originalNode/var)" />
      </xsl:element>
    </xsl:element>
  </xsl:if>
- <xsl:if test="nodeType/color">
    <xsl:element name="nodeType">color</xsl:element>
  - <xsl:element name="passThroughValues">
    - <xsl:element name="attributes">
      - <xsl:element name="name">
          <xsl:value-of select="originalNode/color/@name" />
        </xsl:element>
      </xsl:element>
    </xsl:element>
  - <xsl:element name="transformation">
      <xsl:attribute name="type">attribute</xsl:attribute>
      <xsl:attribute name="targetAttribute">value</xsl:attribute>
    - <xsl:attribute name="default">
        <xsl:value-of select="originalNode/color/@value" />
      </xsl:attribute>
    </xsl:element>
  - <xsl:element name="constraints">
    - <xsl:element name="constraint">
        <xsl:attribute name="type">required</xsl:attribute>
        true
      </xsl:element>
    - <xsl:element name="constraint">
        <xsl:attribute name="type">fixedlength</xsl:attribute>
        6
      </xsl:element>
```

Figure 3: XSLT analyze and generate

This process results in a single, normalized template definition XML file. It is this file, in conjunction with the original source tree, which a third party template toolset can now ingest, interpret and reliably expose a corresponding customization interface. Insofar as the authoring tool and templating toolkit are independent pieces of proprietary

software from two different vendors, the template definition XML file serves as a data interchange able to abstract away those proprietary underpinnings and achieve interoperability between these two crucial components of the overall workflow.

REFERENCE IMPLEMENTATION

In order to exercise the data model and prove out the interoperability we have implemented a basic template toolkit to ingest an application and its template definition XML file. This is done as a web system with two login roles. The first role is that of an application developer wishing to upload and "templatize" their application. The second is for a "customizer," the individual interested in selecting from the overall set of available templates and generating a customized instantiation.

In this implementation Tomcat is used as web server and servlet container, Saxon as an XSLT processor and Oracle as a database. The database maintains names and descriptions of available templates as well as pointers to the original source tree and corresponding template definition XML file.

Application Upload and Template Definition

Application developers are presented with a simple HTML form page to enter the name and brief description of their application. The source tree is uploaded as a single .zip file which is unzipped into the server's file system and parsed by the XSLT script. The script discovers any "templatizable items" within the source tree and generates a single, templateDefinition.xml file. The developer is asked to provide some additional, human readable information to assist a customizer in understanding the significance of each customized item. We ask the developer to define a name and briefly describe each of the items. Once done, the template definition XML file is updated with the additional information and the developer has completed the upload. Figure 4 below illustrates the two upload screens as presented to the application developer.

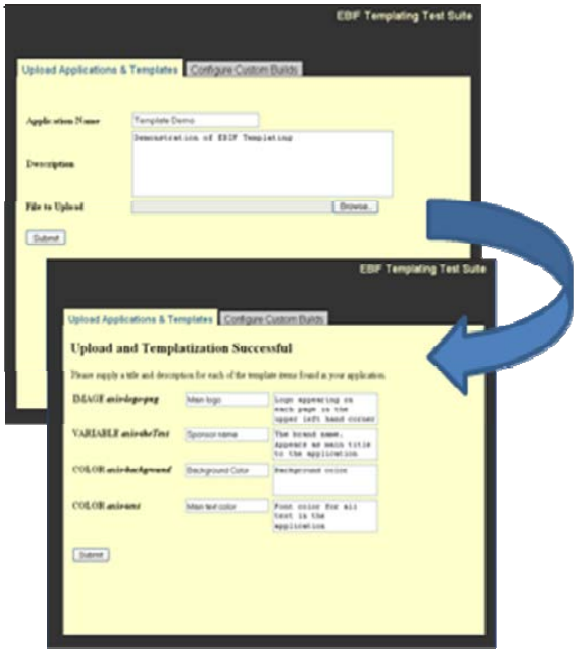

Figure 4: Uploading an application

Figure 5 below represents the logic and data flow associated with ingesting a new application and generating its template definition XML. Ultimately, we persist a name and description of the EBIF application as well as URI pointers to both the .zip file of the original source tree and generated template definition XML file.

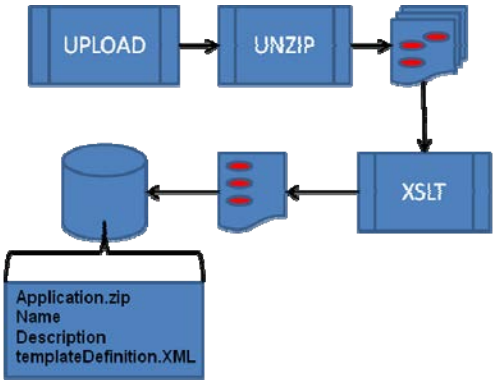

Figure 5: Ingest logic and data flow

The application is then deemed a template and ready for customization. Logging in as a customizer, the user is presented with a list of all templates in the database. Selecting any template will present the user with a screen for supplying the necessary customization values. Figure 6 below illustrates the two screens as presented to the customizer.

Figure 6: Customizing an application

It should be noted on the second screen that the template definition XML file itself is used to generate the customization interface. The customizer interacts with familiar HTML forms, where each input field is tailored to the attribute in question and constrained accordingly. Graphics have file upload fields, strings are constrained text input fields and colors are defined through a standard JavaScript color picker widget. The template definition XML is also used to generate field by field validation JavaScript such that when the customizer submits the form each field is validated by the web browser and it is impossible to post any values breaking the constraints defined by the original application author.

XSLT is then used as a find and replace mechanism against the original source tree, replacing the result of all XPath expressions found in the template definition file with new values in the instance definition file. The resulting source tree can then be compiled into the customized EBIF binary. Figure 7 represents the logic and data flow associated with the customization process.

Figure 7: Customization logic and data flow

Managing MSO and User Agent Variations

This methodology can be summarized as a normalized find and replace system with the actual customization achieved at build time. This technique achieves customization with respect to re-skinning and brand repurposing motivations. With respect to true end-to-end interoperability, however, we would be remiss if we did not address variations among such things as MSO navigational paradigms, user agent execution and integration with third party guide, DVR and VOD systems.

Each of these begs for their own level of unique customization, quite different and more complex than the surface level look and feel modifications desired by the non-technical brand manager. In addressing interoperability across navigational paradigms, consistency in UI dialogue screens, backend interoperability, etc, we nonetheless believe the methodology described here is a promising approach.

Where the non-technical brand manager seeks to replace specific application resources; an MSO, user agent or backend

system would need to replace whole components or methods calls within the application in order to achieve UI consistency across the plant or interoperability with backend systems, the user agent or other software on the set top box. For example, specific method calls to set DVR recordings or perform VOD telescopes may vary depending on the particulars of the guide, DVR or VOD system. Similarly, in the interest of uniformity, MSOs may seek to standardize navigational paradigms or such things as button labels, placement and on-click behaviors.

This is achievable at build time, whereby sets of pre-established blocks of source code, representing the unique, desired method calls and UI components are compiled into the application. Again, this is fundamentally a build time, find and replace mechanism not unlike the system we have described above. We believe, furthermore, that a well defined, standardized naming convention at the source code level would accommodate these needs. Application developers would indicate such replaceable blocks of code with naming conventions such as:

- templateItem-ConfirmationScreen
- templateItem-VODTelescope
- templateItem-DVRSetting

CONCLUSIONS

In order for EBIF to achieve critical mass it is essential to reduce the time to market and minimize the QC associated with individual applications while maximizing the level of creativity and flexibility in appearance of customized instantiations. This is best achieved by wide adoption of pre-approved templates and strong validation in customization tools. This aim must additionally and necessarily encourage innovation amongst a wide range of iTV vendors and independent application developers. This is the ecosystem from which new, compelling features and revenue models will be born and their applications will need to be made readily available as core templates within customization tools.

The naming convention, XML schema and XSLT transformations described here represent the underpinnings of potential standardizations, whereby application developers could easily designate "templatizable" aspects of their applications in a manner in which compilers and customization tools could reliably ingest, parse and process them. By implementing this in an open standard approach as presented here, authoring tools are decoupled from customization tools such that discrete components of the overall value chain become truly interoperable.

FUTURE WORK

Direct support in authoring tools

The XSLT used in this reference implementation to generate the template definition XML assumes that the TVWorks XDK is used as the authoring tool. While the general technique is theoretically agnostic to the particular XML authoring syntax, the node inspection logic within the XSLT is specific to the TVWorks MAX syntax.

This is only required to automatically generate a template definition XML file. The template definition XML itself is its own, standalone data model such that the XPath expressions and constraint definitions can be applied to any underlying XML based source syntax.

Ultimately, our view is certainly that EBIF authoring tools would intrinsically generate such template definition files and there would not be a requirement for the template toolset to generate one.

Validating color palettes and graphics

Color palettes are defined in the EBIF source code and all graphics included in the application are confined to those defined colors. The methodology described here allows a non-technical person to manipulate the underlying color palette and provide custom graphics. There exists potential conflict and limitations where an author might have a "core graphic" which must be preserved in all instantiations and the customizer finds the remaining colors cannot accommodate their desired graphic. The symptom is more pronounced on low-end environments, limited to sixteen colors.

Conventions surrounding graphics and color palettes should be explored. In a sixteen color environment things are clearly highly constrained, but it should be possible for an application author to provide and define (in the template definition XML) an acceptable set of potential palettes and any corresponding "core graphics." In the richer 256 color palettes the symptom is greatly alleviated and the solution is potentially as simple as earmarking a conventional set of a certain number of colors (64 or 128) as customizable.

Validation and Instantiation via Web Services

In our reference implementation we have used XSLT to inspect a set of EBIF source documents for particular naming conventions and node patterns. We did this in order to auto generate normalized template definition files in lieu of them being created by authoring tools. Whether or not authoring tools implement the schema, the inspection technique itself appears useful with regards to potentially automating some basic tests. For example, source code could be dynamically inspected using similar XSLT scripts for valid organization IDs, appropriate calls to terminate() methods, approved VOD handlers, correct HTTP POST parameters, allowable remote control keys, etc. These are examples of any number of scripted tests which could be used to perform automated validation based on an inspection of source syntax.

REFERENCES

[1] T. Bray et al, Extensible Markup Language (XML) 1.0, W3C Recommendation, November 2008 http://www.w3.org/TR/2008/REC-xml-20081126/

[2] J. Clark XSL Transformations (XSLT) 1.0, W3C Recommendation, November 1999 http://www.w3.org/TR/xslt/

[3] J. Clark and S. DeRose, XML Path Language (XPath) 1.0, W3C Recommendation, November 1999 http://www.w3.org/TR/xpath/

[4] H. Thompson et al, XML Schema Part 1: Structures, W3C Recommendation, October 2004 http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[5] P. Biron and A. Malhotra, XML Schema Part 2: Datatypes, W3C Recommendation, October 2004 http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/