From the programmer's point of view: XML for IVR and how DSD Schemas may help

..

Nils Klarlund http://www.research.att.com/~klarlund

..

Index

Abstract

IVR applications are notoriously difficult to construct. To simplify the task, many layers of abstractions are introduced. At the highest level, an application programmer is then mainly concerned about choosing pre-canned dialogues, which are filled in with a variety of parameters, such as prompts and timeout durations. In this article, we will study how XML and the DSD Schema notation may alleviate the burden of understanding a specialized notation with many interdependent parameters and defaults.

Introduction

Notwithstanding the XML hype, there is at least one fascinating and fundamental quantity that sets XML-based notations apart from ad hoc syntax: they encourage tree transformations---a technique that application programmers usually do not take advantage of. It is a hassle even to define a set of parse trees and a procedure by which they are constructed! XML completely circumvents this problem just by directly and simply being a notation for trees. Of course, mappings of trees into trees are a main ingredient of computer science; for example, such mappings are essential to building compilers, where the compilation process is partioned into several phases, most of which simply massage one intermediate tree format into another one.

The exciting development is that tree technology has already crept into everyday Web programming: CSS (Cascading Style Sheets) is a declarative notation for tree transformations that turn HTML into a visual formatting (page layout) model. For example, it's easy to specify a rule that makes H1 headers be typeset in a font that is 120% of the size of the base font. CSS also allows HTML elements to be categorized according to *classes*, which are programmer-defined labels such as in

```
<ul class="myclass">
  <li>
```

```
    This is my point.
  </li>
</ul>
```

Moreover, such classifications can be used to specify the formatting properties as in a CSS rule that forces all text in places like where "This is my point" is to be shown in red. More precisely, the rule specifies that text must be shown in red if it is contained in a `li` element that in turn is contained in an `ul` element with class attribute "myclass".

Unfortunately, CSS cannot accomplish tree transformations into any other domains than the non-XML formatting model. The W3C instead envisages that XSLT (eXtensible Stylesheet Language Transformations) together with the built-in default insertion mechanism of DTDs and XML schemas will cover transformational needs in general. However, XSLT likely is too complicated and too programmatic (in the sense of being a Turing-compete programming language) to be adopted by the majority of programmers---and even if it were to be well-known, there are many situations where its explicit programming model competes unfavorable with CSS. For example, it is becoming a common paradigm to use XSLT and CSS in conjunction for visual formatting applications, since they complement each other nicely.

In this article, we will study the benefits of XML for the problem of building IVR applications, especially with respect to default insertions. In particular, we will argue that DSDs (Document Structure Descriptions), see http://www.brics.dk/DSD/, offer special advantages for this problem when compared to other schema notations, like DTDs or XML Schema.

The IVR scenario

Our scenario calls for an application programmer to develop a little whimsical, interactive voice application that probes the mood of a customer. The programmer will use an XML notation called XPML (for Extensible Phone Markup Language), which is similar to HTML. The XPML notation is based on PML, the Phone Markup Language, as developed at AT&T Labs-Research. The main idea of PML is that simple HTML pages describe a finite-state machine, where intra-page hyperlinks become goto statements and text becomes synthesized speech; input fields corresponds to subdialogues for obtaining numbers and select elements become dialogues a la "for sales, choose 1; for customer service, choose 2,...". Naturally, each subdialogue construct provides numerals parameters for specifying prompts, timeout durations, and help messages in various error situations. As a further complication, that are several interdependencies among these parameters. For example, it was discovered during the development of PML that it is not sufficient to associate one dialogue style to each HTML element; rather, a set of interaction styles must be provided to through an `interaction` attribute to accommodate situations where there are many choices in a menu, the input of certain kinds of numbers, etc. Naturally, the kinds of prompts, along with many other parameters, are dependent on the value of this attribute.

The XPML notation as outlined here is preliminary and incomplete. It is similar to VoiceXML (a new dialogue markup language developed by AT&T, IBM, Lucent and Motorola); rather, we are interested in explaining some general issues surrounding the use of XML for interactive voice applications.

DSDs -- a schema notation for XML

DSDs are grammars for describing classes of XML documents, see http://www.brics.dk/DSD/. Thus, a DSD renders a judgment on every XML document: either it is *valid* or it is invalid. In this way, a DSD defines a class of XML documents, just like the more familar DTD concept of the XML 1.0 specification. A DSD processor takes as input an application document and a DSD; if it determines that the application document is that it according to the DSD, then it outputs the application document augmented with inserted defaults.

An effort is underway at the W3C to define XML Schema, a successor for DTDs, which are not very expressive. Judging from the size of the current specification, one may say that XML Schema is an order of magnitude more complicated than DTDs or DSDs. There are, however, many desirable properties that XML Schema cannot describe such as disallowed attribute combinations, or element content that depend on attributes. More importantly for the focus of this article, DSDs define generic, CSS-like default mechanisms

that allow an application programmer to specify defaults in a declarative way. The DSD processor ensures that only meaningful defaults are inserted, according to the DSD. This technique is a logical and practical extension of the default mechanisms that are already part of DTDs and XML Schema.

In this document, we will explain how XML and DSDs help both the application programmer and the platform programmer:

- DSDs aid the XPML programmer to choose the right syntactic constructs. DSDs are by themselves not easy too read because of the XML syntax, so we indicate how to present them in a more conventional BNF-like way that closely resembles the concrete syntax of the XPML notation.
- XPML programmers can easily check their documents for most errors using the DSD processor alone.
- XPML programmers can use the CSS-like default mechanism that comes with DSDs. Thus, XPML programs can be "styled" in a declarative and modular fashion.
- DSD descriptions significantly simplify the programming of an interpreter for XPML.

DSDs for syntax explanations

Let's suppose an application programmer is working on a program for demonstrating voice interaction. He wants to use XPML, but he doesn't know what XPML is except for some examples he has seen. Naturally, he'll use these examples for guidance, but the DSD may provide a readable, concise syntactic summary. We don't envisage that the programmer will read the DSD as an XML file. Instead, a hyperlinked HTML document may be produced by an XSLT stylesheet transformation. For example, the DSD definition of the element `XPML`, the top element of an XPML document, is shown below (left) through an XSLT stylesheet transformation in a way that strongly suggests the concrete syntax of an application document; the original DSD definition (right) will not be shown to the application programmer:

```
<XPML>ID=XPML:
   ( <head>
        Constraint head-constraint
     </head>[Defaultable],
     <body>
        Constraint body-constraint
     </body>)
</XPML>
```

```
<ElementDef ID="XPML">
  <Sequence>
    <BriefDoc>
      The head element may be omitted.
    </BriefDoc>
    <Element Name="head" Defaultable="yes">
      <Constraint IDRef="head-constraint"/>
    </Element>
    <BriefDoc>
      The body element is mandatory.
    </BriefDoc>
    <Element Name="body">
      <Constraint IDRef="body-constraint"/>
    </Element>
  </Sequence>
</ElementDef>
```

(In the HTML version shown here, the `BriefDoc` documentation strings are used as HTML `title` attributes, which produce a pop-up explanation when the mouse pointer is over the corresponding definition (try it if you are viewing this document in a 5th generation browser).) This particular snippet of a DSD specifies that the `XPML` element consists of a `head` element followed by a `body` element. The `head` is defaultable (which means that it may be omitted if a default for it has been specified) and its attributes and content are specified by the constraint named `head-constraint`. Similarly, the `body` element is specified by the constraint `body-constraint`. We will not go into any details about DSDs here---we hope that the informal explanations are sufficient. The XSLT stylesheet can be found at the DSD Web site; it is rather complicated, approximately 25 pages.

DSDs for debugging

Now, we will explore how schemas may help debug XML documents. Let's assume that the application

programmer's first attempt at the mood-probing XPML program is:

```
<?dsd URI="xpml-att.dsd"?>
<XPML>
  <head>
    <application name="HELLOWORLD"/>
    <maintainer address="karam@research.att.com" loglevel="2"/>
    <title>The Greeting Application</title>
  </head>
  <body>
    Welcome to greetings are us.
    <span nointerrupt="y"><audio url="/audioclips/greeting.vox"/></span>
    <a name="repeat"/>
    <menu name="feelings">
      <option dtmf="0">To end</option>
      <do><a href="#endit"/><comment>go to end point</comment></do>
      <option> If you are feeling like a cowboy. </option>
      <do> Howdy world! </do>
      <option> If you are feeling like a Canadian. </option>
      <do> Gid'day world, how's it going eh? </do>
    </menu>
    <a href="#repeat"/>
    <a name="endit"/>
  </body>
</XPML>
```

The programmer has inserted a `<?dsd URI="xpml-att.dsd"?>` processing instruction to denote that the document must conform to the DSD named `xpml-att.dsd`. He can now use the DSD processor to check the syntax of the document. It'll tell him:

```
Error in 'greetings-first-attempt.pml'

line 10: attribute 'nointerrupt' has illegal value 'y'

while checking attribute in constraint

"message-attributes", 'xpml-core.dsd' line 377
```

The DSD describes this constraint as follows:

```
ConstraintDef ID=message-attributes:
  nointerrupt="YesOrNo"[Optional]
```

Thus, the constraint declares the attribute `nointerrupt` whose string type is `YesOrNo`. In turn, the DSD details that

```
StringTypeDef ID=YesOrNo:
  ("yes" | "Yes" | "no" | "No")
```

So, the programmer must write `"yes"` , not `"y"`. This example illustrates our belief that DSDs provide concise and useful error messages that do not necessitate a complete understanding of the DSD itself.

DSDs for myriads of defaults

Once the above error is corrected, the DSD processor will accept the document and it'll insert all the default attributes and default elements specified by the XPML DSD. The resulting document is

```
<?dsd URI="xpml-att.dsd"?>
<XPML>
  <head>
    <application name="HELLOWORLD"/>
    <maintainer address="karam@research.att.com" loglevel="2"/>
    <title>
      The Greeting Application
    </title>
  </head>
  <body>
    Welcome to greetings are us.
    <span nointerrupt="yes"><audio url="/audioclips/greeting.vox"/></span>
    <a name="repeat"/>
    <menu asrmode="none" endchars="#" finaltimeout="5000ms" interaction="basic"
       interdigittimeout="4000ms" maxmisselected="3" maxtimeout="2" maxtterrs="3"
       name="feelings" timeout="0ms">
      <option dtmf="0">
        To end
      </option>
      <do><a href="#endit"/><comment>go to end point</comment></do>
      <option> If you are feeling like a cowboy. </option>
      <do> Howdy world! </do>
      <option> If you are feeling like a canadian. </option>
      <do>
        Gid'day world, how's it going eh?
      </do>
      <help>No help is available.</help>
      <initial><enumerate><option/> Press <emph><dtmf/></emph>.</enumerate></initial>
      <timeout>You have exceeded the time limit.</timeout>
      <toomanyerrors>Sorry, too many errors.</toomanyerrors>
      <counttimeout>Sorry, too many timeouts.</counttimeout>
      <pause> Pausing. Press pound sign to continue. </pause>
    </menu>
    <a href="#repeat"/>
    <a name="endit"/>
  </body>
</XPML>
```

It is similar to the original document except that a variety of timing and counting parameters, like `interdigittimeout="4000ms"` have been inserted. Also, various default messages used in error and help situations, like

```
<help>
  No help is available.
</help>
```

have been inserted. Voice programming is dependent on a great number of parameters whose tuning is often essential to obtaining the right performance. But they are not usually something that the programmer wants to explain in exquisite detail for every part of the PML document.

DSD defaults for customization

The XPML programmer may easily override defaults of his choosing. For example, he may change the `interdigittimeout` value whenever it's inserted by putting a DSD default declaration into the application document like:

```
<DSD:Default>
  <DefaultAttribute Name="interdigittimeout" Value="1000ms"/>
</DSD:Default>
```

Such a declaration can occur anywhere in the application document; its scope consists of all its siblings and their descendants. So in this case, the programmer may insert the DSD default declaration as a child of the `body` element; the effect is that the `interdigittimeout` attribute for the `menu` event will be inserted with value `1000ms`.

DSD stylesheets

The DSD defaults may be gathered in XML external parsed entities, which are just like XML documents except that multiple root elements are allowed. Below, we define a DSD stylesheet that overrides the default `help` element for the `menu` construct in two ways: for a `menu` without a `class` attribute, the message "We're sorry, can't help you more right now, but please call us at 1-800-greetings" is specified; for a `menu` with a `class` attribute of value `moody`, the default content of `help` becomes "How are you feeling, dud? press 1 to get relief",

```
<DSD:Default>
  <Context>
    <Element Name="menu"/>
  </Context>
  <DefaultContent>
    <help>
      We're sorry, can't help you more right now, but please call us at 1-800-greetings
    </help>
  </DefaultContent>
</DSD:Default>
<DSD:Default>
  <Context>
    <Element Name="menu">
      <Attribute Name="class" Value="moody"/>
    </Element>
  </Context>
  <DefaultContent>
    <help>
      How are you feeling, dud? press 1 to get relief
    </help>
  </DefaultContent>
</DSD:Default>
```

DSDs for evolving languages

In practice, it's rather impractical to describe a DSD as a single document. It's better to describe the main structure in a base document, while extensions---which can be platform specific additions or convenient abstractions--can be documented in follow-up documents.

The core of XPML XPML in essence is described in xpml-core.dsd. (For a more detailed explanation of this DSD, see below.) This language introduces an HTML-like structure and touchtone-based input modalities, including a number of specialized parameters. Also, defaults are provided for all parameters that a beginning XPML programmer should not need to worry about.

Adding speech markup to XPML Separately, in speech recognition extensions, it is described how the high-level construct `menu` may be further refined by speech recognition parameters. Specifically, the constraint that describes `menu` attributes and content is extended with a declaration of an attribute `asrmode` that has the value 'none', 'only', or 'plus':

```
asrmode="('none' | 'only' | 'plus')"[Optional];
```

Also, if the `asrmode` is different from `none`, then the constraint `menu-asr-constraint` applies:

```
If {Attributeasrmode="only" OrAttributeasrmode="plus"}
  Then{Constraintmenu-asr-constraint}
```

The `menu-asr-constraint` declares various other attributes such as `asrengine`, `asrurl`, etc.:

```
ConstraintDef ID=menu-asr-constraint:
  asrengine="Asr-engine-types"[Optional];
  asrurl="Url"[Optional];
  asrspotting="YesOrNo"[Optional];
  bargein="YesOrNo"[Optional];
  asrgrammarlib="YesOrNo"[Optional]
```

Adding interaction types to XPML In interaction extensions, a variety of additional ways of interpreting the `menu` element is provided. Each way is known as an *interaction*. The `interaction` attribute is declared below along with a conditional constraint that declares additional elements when the `attribute` has value `basic` or `optional`:

```
If {Attributeinteraction="basic" OrAttributeinteraction="optional"}
  Then{
    <counttimeout>
      Constraintmessage-attributes;
      Contentmenu-message-content
    </counttimeout>[Defaultable];
    <pause>
      Constraintmessage-attributes;
      Contentmenu-message-content
    </pause>[Defaultable]}
```

These declarations are part of a redefinition of a constraint `menu-constraint` that governs the attributes and content of `menu`s.

Combining it all DSDs may include other DSDs, so XPML is put together from the three parts just described in xpml.dsd.

Extending XPML with AT&T specific additions It's now possible to round off the description of XPML with properties, like metrics, that are specific to the AT&T platform. That's done by further extending the XPML DSD, see xpml-att.dsd.

DSDs for simplifying semantic processing

With a DSD processor, an XML document can be *normalized* by default insertion: since defaults cannot be removed by defaults in an application document, only overridden, the defaults given with the DSD itself provide a set of assumptions about the shape of the document that results from running the DSD processor on a valid document. For example, the XPML interpreter can assume `menu` elements that are fully filled in with timing attributes and content such as `help` and `error` messages.

How to read the XPML DSD

A DSD is itself an XML document, and as such barely readable. So the DSD is presented through a transformation that yields an HTML document that can be displayed by any browser. The displayed notation

that we've chosen here is one that emphasizes the concrete syntax of the XML documents specified by the DSD. The essential structure of XPML is described in the DSD xpml-core.dsd (html) (compare to xpml-core.xml (raw XML)). The first box framed in pink is an element definition that describes the `XPML` element itself. This element must be the root of any XML document conforming to the xpml-core DSD as specified in the second line of the header. The `XPML` element definition is named by a nonterminal ID, which happens also to be `XPML`. (When there are different uses of an element with the same name, the nonterminals allow us to distinguish their descriptions.) According to the element definition, the `XPML` element consists of two things in sequence:

- A `head` element, whose attributes and content are specified as the constraint *head-constraint*; and
- a `body` element, which has attributes and content specified as the constraint *body-constraint*

The *head-constraint* specifies

- a `title` element (the question mark after it means that it is optional),
- `meta` elements (zero or more as indicated by the star), each having a required `name` and `content` attribute, and
- *pragma* elements.

The order of these various elements is not important.

A note about syntax for content: parentheses containing a comma-separated list of content descriptions mean the same thing as in DTDs, namely all content that is a sequence of what's inside the parentheses; similarly, a |-separated list denotes a union.

The content known as *pragma* is defined in the third box. This content definition and element that is one of three different kinds.

In a DSD description, content and attribute declarations may be lumped together in constraints. This abstraction technique is illustrated in the constraint definition of menu-constraint. This is the constraint that is redefined twice to accommodate speech parameters and additional markup related to interaction types.

The remainder of this DSD description of the core of XPML should be rather self-explanatory. We use string types to characterize the allowed attribute values (unless a "?" indicates that there are no restrictions). When the introduction of additional attributes or content depend on the context, we use if statements that are conditioned on context expressions similar to those of CSS.

The components of the DSD for XPML

- The core: xpml-core.dsd (html), xpml-core.dsd (xml as .txt), xpml-core.dsd (xml);
- The speech extensions: xpml-speech.dsd (html), xpml-speech.dsd (xml as .txt), xpml-speech.dsd (xml);
- The menu interaction extensions: xpml-menu-interaction.dsd (html), xpml-menu-interaction.dsd (xml as .txt), xpml-menu-interaction.dsd (xml);
- The AT&T extensions: xpml-att.dsd (html), xpml-att.dsd (xml as .txt), xpml-att.dsd (xml);

Another HTML-like example

Here are links to an XPML example that contains form and input elements; note that lots of errors in such a form will not be caught by most other schema notations: `form` elements inside `form` elements, `input` elements not inside `form` elements, `input` elements of type `submit` with other attributes than `name` and `value`, `input` of type `submit` with a `value` attribute but without a `name` attribute, and so on.

- The form example: blue-widgets.pml (xml as .txt), blue-widgets.pml (xml);

Conclusion

We have shown through our IVR examples that schemas play an important role for the practical handling of domain specific languages expressed in XML. In particular, we have demonstrated that the CSS-like default mechanism provided by DSDs may be an easily graspable alternative to XSLT transformations.