Enterprise-level Web Form Applications with XForms and XFDL

John **Boyer**

November 2005

Abstract

This paper describes a platform for the XML definition of secure, intelligent web-based applications. XForms provides a powerful model-view-controller (MVC) pattern that may best be described as a cause-and-effect XML processing model originated by XFDL. This paper describes a new version of XFDL that consumes, or *skins*, XForms. Hence, this paper presents the first integration of the standardized XML markup for expressing the core processing of a web-based form applications (XForms) with a host language (XFDL) that offers security, precision presentation, a document-centric capability, and other features that contribute to a more rich user experience.

Table of Contents

1. Introduction

In 1998, the W3C recommended XML as a grammar for defining interoperable software systems. Later the same year, an XML vocabulary named XFDL (the Extensible Forms Description Language) was published as a W3C Note, informing the W3C community of the need to standardize the next generation of secure, intelligent web-based forms applications. The key insights were that

- the web would become a platform for delivering more powerful applications than the simple shopping cart or pizza order forms of the time;
- the declarativeness of XML can be generalized beyond data to the core processing models of the data, simplifying the development and maintenance of the typically more sophisticated forms often found in insurance, health-care, financial and government applications.

In general, a **form** provides a rich user experience for the presentation of information and gather of responses to that information. But the information processing needs of these so-called sophisticated forms are as complex as for any application, and what makes a language desirable in a particular application domain is the set of languages features that advantage that domain.

1.1. XForms and the Model-View-Controller Pattern

In order to begin to meet the standardization challenge for this important class of web applications, the W3C [XForms] Recommendation presents an open platform for expressing the core model-view-controller (MVC) layers of forms applications. The **model** expresses an XML data processing framework that includes

- static input validation by [XML Schema Structures] and [XML Schema Datatypes]
- submission parameters for returning the completed data and for calling on web services to aid the user's input experience
- a declarative business rules engine for automatic calculations over the data to compute
  - intermediate or summative results (e.g. a purchase order total)
  - whether data satisfies data-dependent constraints (e.g. ensuring the upper page value of a print request is no less than the lower page value)
  - whether data is readonly (not modifiable by the model)
  - whether data is required for or relevant to submission

The **view** layer of XForms provides intent-based data access form controls as well as hierarchy and iteration constructs that allow the viewer layer to access nested data and data sequences in the model. The distinction between a model view and a presentation layer is discussed further in the next section on host languages.

The **controller** layer of XForms provides the ability to listen for [XML Events] and execute *imperative* action sequences that manipulate the model or a view (or both). It is interesting to note that, in XForms, the controller manipulations are not backed by a passive data model but rather by a active processing model that enforces a set of declaratively expressed business rules for the data. Hence, the business rules expressed in the XForms model extend the semantics of the controller layer actions. Moreover, an implicit aspect of controller layer actions is an update of the view layers that includes not only updated values but also dynamic response by iteration constructs to structural changes in the data.

I refer to this interplay between the model, view and controller layers as a **cause-and-effect** paradigm. The active participation of the model and views in the controller's data manipulations is made possible by a declarative style of expressing binding relationships with [XPath 1.0]. Because each imperative action is essentially tailored to the schema of the XML data by declarations in the model and view layers, XForms gains expressive power over pure scripting approaches by drastically reducing the lengths of imperative action sequences, thereby increasing maintainability and robustness and decreasing design pattern complexity.

1.2. Host Languages Provide Presentation Layers for XForms Views

The purpose of the XForms controls is to provide data access views of the model to containing applications. The most obvious view, of course, is a presentation layer for rendering the form to an end-user, and the XForms controls can indeed be used directly by a minimalist rendering engine. However, the intent of the XForms design is to allow sufficient abstraction so that XForms forms can be hosted by a XML-aware container applications that consume model views and augment them according to application-specific requirements.

At the horizon of the abstraction, model views could be created and used to drive back-end processing of data, e.g. workflows or data shredding for databases or content repository metadata. Generally, though, the short-term goal of the XForms view abstraction was to allow it to be *skinned* by (incorporated into) other host languages in order to meet the many diverse requirements that arise on the world wide web. One obvious host language is [XHTML], but VoiceXML could be used to satisfy advanced accessibility needs, and WML (or minimalist rendition) could be used on small devices.

1.3. XFDL as a Host Language for XForms in IBM Workplace Forms

The XFDL language is focused on providing rich user experience in forms that appear in in the business or enterprise context, which often has more stringent usability, security or auditability requirements than the general web context. The enterprise context tends to be an iteration over the business context; some requirements are needed by every business form while others are only needed in some forms. An example requirement from the business context is the need to present double precision numbers as formatted currency values, respecting locale issues. This includes rules for large number deconstruction, the symbol for the decimal place mark, the rules for the choosing the location and symbol of the currency mark, and rules for decimal place truncation or rounding. In XFDL, the presentation can be declared to a currency format; by default this converts an XForms data value like 10272.897814616084 to $10,272.90 in the presentation. Aside from the information processing requirements of forms in the general web context, the most common requirements in the business context are

- rich user experience for information presentment, including
  - multipage capacity
  - wizard front-ends to guide user through execution pathways of complex forms
  - precision layout and appearance control of high density forms that does not vary by computer
  - precisions screen presentation imposed on printed rendition
  - accessibility support
- rich user experience for gathering responses, including
  - data validation and formatting (dates, currencies, and patterns such as SSN, phone numbers, and postal codes)
  - organized compressed attachments

- enriched text
- accessibility support

- security scalability to meet all required levels of transaction auditability
  - digital signatures that integrate into browser certificate stores
  - signatures that leverage non-PKI authentication systems
  - signer can omit part of document (e.g. office-use-only section, other signature)
  - multiple signer scenarios
  - submissions in the web browser security/user authentication context
- ease-of use from single XML document architecture
  - offline processing of forms and incremental work via save/reload (suspend/resume)
  - ad hoc workflow capabilities such as emailing forms to role players
  - system developer focused on workflow of transaction data

Of particular interest are the benefits associated with the use of a **document-centric** architecture in XFDL-based systems. In a data-centric application, the data is bound to the MVC and presentation layers ephemerally. On the other hand, XFDL skins the cause-and-effect MVC layers of XForms with a precision presentation layer *in the same document*. **The full XFDL+XForms document, then, is the XML serialization of both the form application and its current state.** This has several advantages for end-users and application developers. Attachments can be added, input can be provided over an extended period of time, and digital signatures can secure all or required parts of the form, including not only the data but also its precision presentation. The end-user can be empowered to save the full document locally to disk and work offline or email the document to collaborators in an ad hoc workflow. The form remains a powerful collaboration tool even in formalized workflow. Once a form is completed, the full document can be archived in a records management system for transactions auditability, and the XML data can easily be harvested from the surrounding XML document to drive back-end data processing systems.

In essence, the form document becomes the fundamental unit that traverses the business process collecting all information related to a transaction. The application developer can focus on how the data related to a transaction must flow through a system without the distraction of having to architect the delivery of application fragments related to the transaction. This is analogous to the efficiency gain derived from programming in an object oriented language instead of a procedural language or assembly language.

This paper describes the first integration of XForms with a host language, XFDL, that is enabled with document-centricity and other capabilities that satisfy the full spectrum of requirements from the business context. The remainder of this paper presents an overview of the XFDL document structure, a description of how XFDL consumes the model views exposed by XForms, and what XForms features must be exploited to account for data persistence issues that arise within the current design of XForms (issues that affect any multi-user workflow as well as document-centric systems).

2. Introducing XForms

2.1. Components of an XForms Model

Despite the intent of XForms to express more complex web-based applications, it is still useful to begin with a simple example to illustrate the basics. We begin with a simple form to calculate the monthly payment and total payback of a compound interest loan. The markup below shows an XForms model containing a simple data instance for the details of a compound interest loan.

```
<xforms:model xmlns:xforms="http://www.w3.org/2002/xforms"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              schema="loan.xsd" functions="power">
      <xforms:instance xmlns="" id='loan'>
           <Loan>
               <StartDate>2005-09-20</StartDate>
               <Borrower>
                   <Name>John Q. Public</Name>
                   <Addr>123 Main St. Tinyville</Addr>
               </Borrower>
               <Principal currency="CDN">10000</Principal>
               <Duration>12</Duration>
               <InterestRate>5</InterestRate>
               <Payment>856.07</Payment>
               <TotalPayout>10272.84</TotalPayout>
           </Loan>
      </xforms:instance>

      <!-- More XForms instances for extra data -->
      <!-- XForms binds for model item properties and calculations -->
      <!-- XForms actions for constructor and destructor behaviors -->

      <xforms:submission id='loanSub' method='post'
action='http://example.org'/>
</xforms:model>
```

In many XForms applications, the data will have an associated schema to define data types and structural constraints on the XML. This could be done by placing the XML schema directly into the `xforms:model`, or more commonly (as above), by URI reference in the `schema` attribute.

For data type assignment, which is the most important to client-side processing, it is also possible to avoid using XML schema altogether by instead makingthe assignments with the XForms `type` model item property (**MIP**) in an `xforms:bind`. This is an important technique for forms that must operate on small, resource constrained devices. The additional markup below provides some assignment of types and a few other MIPs to some of the instance nodes.

```
<xforms:bind nodeset="Duration" type="xsd:nonNegativeInteger"/>

<xforms:bind nodeset="(Principal | InterestRate)"
             type="xsd:double" constraint=". > 0"/>

<xforms:bind nodeset="Borrower/*" required="true()"/>

<xforms:bind nodeset="(Payment | TotalPayout)"
             relevant="../Principal > 0 and
                       ../Duration > 0 and
```

```
                        ../InterestRate > 0"/>

<xforms:bind nodeset="StartDate" type="xsd:date"/>
```

Each `xforms:bind` above selects one or more nodes using the `nodeset` attribute, then another attribute such as `type` assigns a model item property. Note that the second bind uses both `type` and `constraint` MIPs to ensure that each of the two nodes is both a double number and positive. The constraint MIP is especially useful when the content validity of a node is dependent on the content of another node, such as the upper page number of a print request being no less than the data entered for the lower page number of that request.

The third bind above declares that all child elements of the `Borrower` must have non-empty content using the `required` MIP. This ensures that the name and address of the loan applicant are filled out before the transaction data can be successfully submitted. The final bind uses the `relevant` MIP to indicate that the payment and total payout nodes are not relevant unless the inputs that will be used to calculate them are given. Although relevance can affect submission, the view controls bound to these nodes are unavailable (typically invisible) until they become relevant.

The most important model item property of an instance node is its intrinsic value. In the case of the loan payment and total payout, these are derived by calculation over other instance nodes. The markup below shows how to declare value formulae using the `calculate` attribute.

```


<xforms:bind nodeset=TotalPayout" calculate="../Payment * ../Duration"/>

<xforms:instance id="rate" xmlns="">
      <rate></rate>
</xforms:instance>

<xforms:bind nodeset="instance('rate')"
             calculate="instance('loan')/InterestRate div 100.0 div 12.0"/>

<xforms:bind nodeset="Payment"
             calculate="if(instance('rate') > 0,
                            ../Principal * instance('rate')
                            div
                            (1.0 - power(1.0+instance('rate'), -../Duration)),

                            ../Principal * ../Duration)"/>
```

The first formula is the easiest since the total amount paid back is simply the product of the payment amount and the number of payments. However, it is also clear that this formula should be run last even though it appears to be first. The XForms compute system automatically works out the correct order for the calculations. The second `xforms:bind` calculates the result of a temporary variable that was created to simplify the main loan calculation. Any number of additional XForms instances can be added to the XForms model, and each can contain as much data as needed by the form author. In this case, we need to translate from the human version of interest rate percentage to the mathematical value needed by the loan formula.

The third `xforms:bind` has several interesting features. First, it shows the use of conditional logic. Using an `if()` function, the calculation determines whether it is appropriate to apply the compound interest formula. The compound interest formula itself requires exponentiation, which is not available in XPath. It is also not available in XForms 1.0 (though it is currently planned to appear in XForms 1.1). However, XForms does allow extension function to be added by implementations, and forms which use extension functions must declare the extra required functions in the `functions` attribute of the `xforms:model` element.

A controlling XForms action can be performed in response to the occurrence of an event. An XForms model has a number of events that can occur, but the most important one for data initialization is the *xforms-model-construct-done* event. The markup below shows how to hook these event and perform an initialization. In this case, if the start date of the transaction has not yet been set, then the current date is used.

```
<xforms:action ev:event="xforms-model-construct-done">
   <xforms:setvalue ref="StartDate" value="if(.='',substring(now(),1,10),.)"/>
</xforms:action>
```

The initializing action is the `xforms:setvalue`. Since it is the only action being performed for the event, the surrounding `xforms:action` could have been removed and the `ev:event` could have been placed directly on the `xforms:setvalue`. The `xforms:action` is more useful when multiple actions must be run in sequence in response to a single event.

2.2. The XForms Intent-based User Interface

An XForms view of the values and MIPs of an XForms model is constructed with the XForms user interface controls. The controls are bound to data instance nodes with a *single node binding*. For example, the markup below shows the `ref` attribute being used with `xforms:input>` controls to obtain such as the principal or interest rate for the loan application of the prior section, and with the `xforms:output` control to provide the results.

```
<xforms:input ref="Principal">
       <xforms:label>Enter principal:</xforms:label>
</xforms:input>

<xforms:input ref="Duration">
       <xforms:label>Enter duration:</xforms:label>
</xforms:input>

<xforms:input ref="InterestRate">
       <xforms:label>Enter interest rate:</xforms:label>
</xforms:input>

<xforms:output ref="Payment">
       <xforms:label>Monthly payment:</xforms:label>
</xforms:output>

<xforms:output ref="TotalPayout">
       <xforms:label>Total to be repaid:</xforms:label>
</xforms:output>

<xforms:submit submission='loanSub'>
       <xforms:label>Submit</xforms:label>
</xforms:submit>
```

Note that each view control has a child element called `xforms:label`. This element associates a meaningful label with the control, the presentation of which is not specified by XForms. It is could be read before the control's content in an auditory rendition or presented in a visual rendition at some location relative to the control (e.g. to the left, the right, or above) based on direction of language flow.

The `xforms:submit` presents a control that can be activated, such as a button. Upon activation, the IDREF in the `submission` attribute is used to select and begin execution of an `xforms:submission` in an XForms model. Any failure to meet the constraints or validate against the specified data types or the XML schema results in termination of the submission with an `xforms-submit-error`. The form author can attach more XForms actions to a submission so that additional tasks can be performed on either the success or failure of a submission. In the case of a failed submission, the form author may choose to show an error message. On success, a second submission could be performed.

XForms has several other user interface controls to collect input by other methods. For example, the `xforms:textarea` collects multiline input, the `xforms:select` provides a multiselection control, and the `xforms:select1` provides a single selection control. The exact type of single selection control is not specified by XForms. It could be a popup (drop down) list, a list box, or a set of radio buttons. Regardless, the user will be able to make one selection from among a set of choices, and that choice will deselect the previous choice.

The XForms view layer also contains controls for hierarchically grouping and iterating groups of other controls. The `xforms:group` can be used to logically contain a set of controls. This can allow the setting of a common base node for the single node bindings of controls in the group, and if the common base node is non-relevant, then the entire group of controls is made unavailable regardless of the relevance of the nodes bound to the contained controls. At the presentation layer, an XForms group can be styled to have a border and background color to visually relate the grouped controls.

The `xforms:switch` is similar to a group in many respects. It contains a number of `xforms:case` elements, only one of which is *selected* at a time. The group of controls in the selected case is presented and the controls in the non-selected cases are not operational and not presented.

Perhaps the most powerful user interface construct in XForms is the `xforms:repeat`, which iterates a template group of controls for each node of an XPath nodeset identifed by a *node-set binding*, which is typically given as an XPath expression in a `nodeset` attribute. Often, we think of the XForms repeat as generating a *table*, with each *row* being a copy of the template that is associated with one of the nodes from the nodeset. Indeed, the need to create of XForms model constructs that match the user interface iteration capabilities of the XForms repeat provides the most compelling use case for using the `nodeset` attribute in the `xforms:bind` The markup example below shows the essential binds and repeat for a simple purchase order. Take special note of the first bind, which shows that relative XPaths can be used so that a single bind can set up the calculations for every row of the purchase order.

```

<xforms:instance id="po" xmlns="">
   <purchaseOrder>
       <product name="Widget">
           <unitCost>3.00</unitCost>
           <quantity>6</quantity>
           <lineTotal>18.00</lineTotal>
       </product>
       <product name="Gadget">
           <unitCost>4.00</unitCost>
           <quantity>8</quantity>
           <lineTotal>32.00</lineTotal>
       </product>
       <subtotal>50.00</subtotal>
       <tax>7.00</tax>
```

```
        <total>57.00</total>
   </purchaseOrder>
</xforms:instance>

<xforms:bind nodeset="product/lineTotal"
             calculate="../unitCost * ../quantity"/>

<xforms:bind nodeset="subtotal"
             calculate="sum(product/lineTotal)"/>

<xforms:bind nodeset="tax" calculate="../subtotal * 0.14"/>
<xforms:bind nodeset="total" calculate="../subtotal + ../tax"/>

...

<xforms:repeat nodeset="product" id="POrepeat">
   <xforms:input ref="@namel">
       <xforms:label>Enter product name:</xforms:label>
   </xforms:input>

   <xforms:input ref="unitCost">
       <xforms:label>Enter cost per unit:</xforms:label>
   </xforms:input>

   <xforms:input ref="quantity">
       <xforms:label>Enter desired quantity:</xforms:label>
   </xforms:input>

   <xforms:output ref="lineTotal">
       <xforms:label>Total for this product:</xforms:label>
   </xforms:output>
</xforms:repeat>

 <xforms:output ref="subtotal">
     <xforms:label>Subtotal:</xforms:label>
 </xforms:output>
 <xforms:output ref="tax">
     <xforms:label>Tax:</xforms:label>
 </xforms:output>
 <xforms:output ref="total">
     <xforms:label>Total:</xforms:label>
 </xforms:output>
```

2.3. Controlling Models and Views with XForms Actions

An XForms action sequence can be performed to affect both model and view properties based on a user event. Perhaps the simplest example is to trigger an action sequence based on the user event.

```


<xforms:trigger>
```

```
   <xforms:label>Delete</xforms:label>
   <xforms:action ev:event="DOMActivate">
      <xforms:delete nodeset="product" at="index('POrepeat')"/>
      <xforms:setfocus control="POrepeat"/>
   </xforms:action>
</xforms:trigger>
```

In this example, the `xforms:trigger` could be visually presented as a button that, when pressed, activates an action sequence that deletes a node of data that corresponds to the currently focused row of the repeat table. Part of the semantics of the `xforms:delete` is that the calculated values in the model and the controls iterated for the repeat are automatically updated. Similar actions exist to perform other updates like inserting a data node, which would cause a new row of controls to appear.

3. The Structure of XFDL Documents

3.1. Pages, Items and Items within Items

The XFDL document element has the local name `XFDL` and typically declares the XFDL namespace as the default. To ensure backwards compatibility over time, the XFDL namespace includes the version number. The content model of the `XFDL` element consists of a `globalpage` element followed by one or more `page` elements. Both of those contain start with a `global` element. The `page` element can then contain any number of other child elements called **items**, each of which represents a single user interface control or grouping control. Each item contains zero or more child elements called **options**, which provide properties appropriate for the item.

```


<XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
      xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
      xmlns:ev="http://www.w3.org/2001/xml-events"
      xmlns:xforms="http://www.w3.org/2002/xforms"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <globalpage sid="global">
      <global sid="global">
          <!-- form global options -->
      </global>
   </global>

   <page sid="P1">
      <global sid="global">
          <!-- page global options -->
      </global>

      <!-- items below -->

      <field sid="NAME">
          <!-- field options -->
      </field>

      <popup sid="GENDER">
```

```
            <!-- popup options -->
        </popup>

        <!-- More items as needed -->

    </page>

    <!-- more pages as needed -->
</XFDL>
```

The pages, items and globals all must contain a `sid` attribute, which provides a **scope identifier** for the element. The scope identifier of an element must be unique among the children of its parent element. They provide an efficient referencing method, e.g. `P1.NAME` refers to the first `field` item above.

The XFDL schema recognizes a finite set of items, but it also allows custom items, which have a non-XFDL namespace but contain an `xfdl:sid` attribute. The namespace prefix declared above by `xmlns:custom` can be used to qualify custom items, but any other namespace could also be used. A list of the recognized items and their meaning appears below:

field	takes keyboard input, including single and multiple line plain text, passwords, and rich text.
label	displays a single or multiple line text message or an image.
button	allows the user to trigger a behavior, such as submitting the form or initiating the process of attaching a file.
action	a non-visual item that can trigger behaviors automatically, either once or repeatedly according to any multiple of a number of seconds.
popup	provides a single selection control of minimal appearance. Only the current choice is shown. The list of choices pops up when the control is activated, and hidden once a choice is made
list	provides single or multiple selection control. The list of choices is displayed compactly, and the currently selected choices are highlighted.
radiogroup	provides a single selection control with a full appearance using a set of radio buttons, which can be configured in a column, a row, or any arbitrary positionings.
checkgroup	provides single or multiple selection control with full apperance using a set of check boxes, which can be configured in a column, a row, or any arbitrary positionings.
combobox	provides an open single selection control of minimal appearance. The user can enter a line of text or choose from a popup list.
slider	provides a range control for integer and floating point values. The user slides a marker along a track to indicate a number in the range of the track.
pane	provides a common background, border and visibility control for a collection of contained items. It is also possible to switch the set of contained items being displayed.
table	provides a common background, border and visibility control for a collection of contained items that have been iterated from a template according to how much data needs to be displayed by the table.

Using the containment items, `table` and `pane`, an arbitrary level of nesting items is permitted because tables and panes can contain tables and panes. Moreover, the precision layout markup in XFDL is applied within the containers, so the layout needs of whole forms extends into the form fragments containined by panes and table rows. As a result, tables are not just iterations of simple linear combinations of form controls. Each row of a table can have arbitrary layout, including subtables that have arbitrary layout.

3.2. Options and Suboptions

The properties of each item are described by *option* subelements. A number of the options have further element depth, and these descendant elements are called **suboptions**. All options and suboptions in XFDL can be dynamically modified by attaching a `compute` attribute. Due to the need for computability and for suboption depth, item properties are expressed as elements, not attributes. Computable properties and their relationship to CSS-like control of rendition are discussed below and in Section 3.3, “Computes, Extension functions, CSS”.

The remainder of this section discusses the most important options in XFDL. It is not meant to replace the XFDL reference manual, though, as there are over a hundred options in total.

value	This option provides direct access to the data value associated with the item. Most input items update the value incrementally (as soon as the user provides input to the user interface control). Text input fields can also behave incrementally, but by default the the user interface control's contents are only committed to the value option on tab out or loss of focus. This option is not applicable to pane, table and action.
image	Associates an image with a button or label item, which if given overrides the rendering of text content from the `value` option.
rtf, texttype	Used to configure a field item for rich text support (texttype) and contain the rich text (rtf). The associated plain text is stored in the `value` option.
itemlocation	Used for precision positioning and sizing of the bounding box of an item. Items can be positioned or sized using absolute pixel coordinates, and they can also be positioned using any sequence of over 30 locators, such as aligning or expanding the top edge of an item with the bottom edge of another item. Locators appear as successive child elements of this option.
visible	Contains `on` or `off` to indicate whether the item is rendered. Default is `on`. If a container element (table, pane) is not visible, then its contained items are also hidden.
active	Contains `on` or `off` to indicate whether the item is enabled or disabled. Default is `on`.
readonly	Contains `on` or `off` to indicate whether the item is read-only or modifiable. Default is `off`. When an item is read-only, the end-user is prevented from modifying the user interface control, but the form may still programatically modify the `value` option, and the result will be rendered (if the item is visible).
label	Provides a text message immediately above the main user interface control for the item.
acclabel, suppresslabel	Used to provide higher fidelity accessibility support through ability to tailor the accessibility messaging experience without damaging the visual countenance of the form.

help	Associates a help message with the item.
format	This option provides numerous suboptions related to validating and presentation formatting of the `value` option. This option enables the ability to handle dates, currencies, SSNs, phone number, postal codes and other patterned input. A `message` suboption allows the form author to specify a message to be shown when the `value` option content is invalid. A `mandatory` suboption contains `on` or `off` to indicate whether or not user input is required.
type	Allows buttons and actions to call upon extended behaviors such as full document submission, digital signatures, and folder-organized compressed attachments.
url	Used in full document submissions. May be computed like any other XFDL option, allowing very lightweight workflow routing, for example.
fontcolor, bgcolor, labelfontcolor, labelbgcolor	Allows a color property to be specified as either an RGB triplet or by selecting one of over 700 color names. Background colors may also be transparent, which is actually the default for labels and container controls. The `compute` attribute can be used to associate the item with a globally defined style or to style the item's appearance based on its `value` option content.
fontinfo, labelfontinfo	Allows specification of font face, point size, and effects such as bold, italic and underline. The `compute` attribute can be used, for example, to associate the item with a globally defined style.
focused, mouseover, activated	virtual options that represent key events that occur on an item by switching between `on` and `off` content. These options are not serialized with the rest of the form as they are designed to be detected by XFDL computes.

It is not necessary to specify all of the options in an item since the options take default values. As examples, items are visible and active by default. Sometimes the default depends on the item, e.g. the default background color is white for fields and transparent for labels.

3.3. Computes, Extension functions, CSS

An XFDL `compute` attribute can be placed on any option or suboption of an item to declare an update rule that is automatically updated any time the nodes it references are changed. As a quick example, we see below the markup that varies some of a field's properties based on its value. The label text and font color are dynamically set based on the input value entered by the user, and the background color is changed when the field receives the focus or contains the mouse cursor.

```
<XFDL ... xml:lang="en">
    <page sid="P1">
        <global sid="global">...</global>
        <!-- More Items -->
        <field sid="IncomeTaxTotal">
            <readonly>on</readonly>
            <label compute="value >= '0' ? 'Your refund' : 'Please pay'"
                          >Your refund</label>
            <fontcolor compute="value >= '0' ? 'black' : 'red'"
                              >black</fontcolor>
            <value compute="some formula">$1,000.00</value>
            <format>
               <datatype>currency</datatype>
            </format>
            <bgcolor compute="focused=='on' || mouseover=='on'
                              ? '#C0FFFF' : 'white' ">white</bgcolor>
        </field>
        <!-- More Items -->
    </page>
</XFDL>
```

Observe that the results of a compute are placed into the content of the element. Also, note that the computes for `label` and `fontcolor` refer to `value` rather than the fully qualified reference, `P1.IncomeTaxTotal.value`. This is because XFDL scopes the references relative to their document location. If the page and item scope identifiers are omitted, then the page and item scope identifiers for the containing page and item are used.

The compute expressions in the example above contain the decision operator and some examples of comparators. Computes can also contain logical operations (or, and, not) and nested decisions. A compute, such as the one for value above, could also contain a formula involving arithmetic operations. All of these built-in operations understand data type, so adding a number to some value produces an appropriate result based on whether the value is a date, a currency, a number or a string. This is why, for example, the value option content above is comparable to `'0'` despite being decorated with presentation formatting.

There are over 70 built-in functions available to XFDL compute for performing all manner of string manipulations, financial calculations, trigonometric calculations, and so forth. Moreover, XFDL allows **extension functions** to be added via WSDLs and java classes. Such functions can perform any calculation, even computing the next move in a game of checkers, for example. Perhaps the most used function in pre-XForms XFDL forms is the `toggle()` function, which can detect a change of content of any option that it references. Combined with the decision operator, this can be to run an imperative sequence of functions. When the `activated` option of a button changes to `on`, then a web service function could be invoked. The `set()` function can be used to move the focus to a new field when the current field's value changes to an acceptable result. The functions `set()`, `duplicate()` and `destroy` can be used in combination when a button is pressed to change the form or its data.

Originated in 1996, the compute system is really a more general-purpose solution to address a super-set of the requirements that CSS targets. Therefore, only the simplest computes are needed to associate a style to an item property. The main difference is that the compute architecture was weighted toward facilitating item customizability, so the decision to consume the style is made by the item. However, it is easy for a design environment to offer item multiselection so that one or more styles can be assigned to any number of items (the CSS architecture is optimized for form design in a text editor). In fact, XFDL is the first host language for XForms that is capable of providing CSS-like functionality because XFDL exposes XForms instance data form XForms view controls to the `value` option of the associated XFDL item. Here is a simple example of creating a shared style:

```


<XFDL ... xmlns:styles="http://example.com/styles">
   <globalpage sid="global">
      <global sid="global">
         <styles:field>
            <styles:fontcolor>blue</styles:fontcolor>
            ...
         </styles:field>
      </global>
   </globalpage>
```

```
    <page sid="P1">
       <global sid="global"> ... </global>
       ...
       <field sid="Name">
           ...
           <fontcolor compute="global.global.styles:field[styles:fontcolor]"/>

       </field>
       <field sid="Address">
           ...
           <fontcolor compute="global.global.styles:field[styles:fontcolor]"/>

       </field>
       ...
    </page>
 </XFDL>
```

A few final points are raised by this example. First, note that the field style is a form global option, and the font color style is a suboption. So, XFDL references can refer to options and suboptions in a foreign namespace, and suboptions are referenced using array syntax. Second, note that a compute could be applied to the font color style, which allows dynamic changes to all of the options that consume the style.

3.4. Applying Computes with Foreign-Namespaced Elements

The XFDL compute system is meant to be applicable generally within the document, not just to XFDL options and suboptions. It would be useful for a future version of XML to consume this idea, only with some sort of XPath expression syntax. Until then, the feature is available in through the XFDL namespace. This means that the compute attribute has to be in the XFDL namespace.

There are numerous use cases for this feature. One is the computation of intermediate results. Another is computation of results in custom items. A third is to implement the notion of cascading styles. Continuing from the example above, one could have:

```


       <global sid="global">
          <styles:allInputs>
             <styles:fontcolor>blue</styles:fontcolor>
             <styles:bgcolor>255,255,128</styles:bgcolor>
             ...
          </styles:allInputs>
          <styles:field>
             <styles:fontcolor compute="styles:allInputs[styles:fontcolor]"/>
             <styles:bgcolor compute="styles:allInputs[styles:bgcolor]"/>
             ...
          </styles:field>
       </global>
```

4. The XFDL Skin for XForms

4.1. The General Notion of a 'Skin'

The guiding principle of the integration between XFDL and XForms is that XFDL is quite literally a *skin* that provides a sophisticated presentation layer for XForms view controls. This means that the user interface controls suggested by XFDL and all of its options and computes are used to *style* the XForms controls. At the markup level, the skin concept is reinforced by placing the XForms control at the option level inside an XFDL item so that the latter is a skin for the former even in the document serialization. The XFDL item's properties for font, border, coloring, positioning, size and so forth are thus applied to the user interface control associated with the XForms view control skinned by the XFDL item, and the XForms view control behaves like any other XFDL option that provides special properties to an XFDL item, such as a binding to instance data.

Of course, it is also the case that putting a compute on a `value` option is not appropriate because such computations are the domain of the `calculate` model item property of the XForms model. However, for most items, the data content obtainable from the single node binding of the XForms control is exposed to XFDL through the `value` option, so computes on other item properties can be based on the data coming from XForms by simply referencing the `value` option as was done before XFDL integrated with XForms.

XFDL as a skin also means that XFDL is malleable, taking on the shape of what is under it, to keep with the analogy. In other words, the XFDL items interact with XForms view controls to consume the instance data and model item properties (MIPs) exposed by the XForms view controls. Generally, this consumption occurs by setting the default value of the XFDL option or suboption based on the information from the XForms view control. However, in some cases, the more appropriate behavior is to combine the information from XForms and XFDL in some way. The details of the effects of XForms constructs on XFDL are discussed in detail in Section 4.5, “The Effects of XForms Constructs on the XFDL Presentation Layer”.

4.2. Location for XForms Models

The form global option `xformsmodels` is used to contain all `xforms:model` elements in an XFDL document. For example,

```
<XFDL ... >
   <globalpage sid="global">
      <global sid="global">
         <xformsmodels>
            <xforms:model ...>
                <!-- [schema], instances, binds, submissions, actions -->
            </xforms:model>
            ...
         </xformsmodels>
         ...
      </global>
   </globalpage>
   <page sid="P1">
      <global sid="global"> ... </global>
      ...
   </page>
</XFDL>
```

4.3. XFDL Skin Items for Atomic XForms Controls (Widget Library)

4.3.1. The *label* Item Skins *xforms:output*

Although the technology is designed to scale up to very large forms, it still helps to start with a simple example to help isolate the building blocks of all forms. To this end, we consider for the next few sections a form that might be used during an application for a mortgage. One might have to fill in various parameters like the principal, duration and interest rate, and the form would then report the monthly payment. We start with how an XForms view would expose that result to the presentation language via an `xforms:output` control. The XFDL item most frequently used to display non-editable text is the `label`, so that is the skin for `xforms:output`.

```
<label sid="MonthlyPayment">
    <xforms:output ref="/loan/mpymt"/>

    <format>
       <datatype>currency</datatype>
    </format>
</label>
```

XForms also allows a `value` attribute to appear instead of the `ref` to allow a simple formula to be calculated without having to store it in instance data. The `xforms:output` can also have an `xforms:label` to associate some information with the resulting formula. An example of this might be summing the results of a purchase order:

```
<label sid="POSubtotal">
    <xforms:output value="sum(/po/lines/line/total)">
       <xforms:label>Subtotal:</xforms:label>
    </xforms:output>

    <justify>right</justify>

    <format>
       <datatype>currency</datatype>
    </format>
</label>
```

The XFDL `label` item has a couple of special features that aid in presentation. While both the `xforms:label` and the `value` contribute to the display, only the latter is used to set the XFDL `value` option. This allows the XFDL `format` to process the numeric value. Second, it allows the `justify` option to operate only on the currency value label. This is useful for left aligning the text labels for subtotal, tax and total of a purchase order while right aligning the associated values.

4.3.2. The *field* Item Skins *xforms:input, xforms:secret,* and *xforms:textarea*

The appropriate XFDL item for typed input is the `field`, which therefore can be the skin for single-line (`xforms:input`), single-line write-only (`xforms:secret`) and multiline `xforms:textarea` XForms controls. Since the skinning pattern is identical, only the single-line input is shown below:

```
<field sid="LoanPrincipal">
    <xforms:input ref="/loan/principal">
       <xforms:label>Loan Principal</xforms:label>
    </xforms:input>
    <format>
       <datatype>currency</datatype>
       <presentation>
           <currencylocale compute="Currency.value->value"/>
       </presentation>
    </format>
</field>
```

In this example, the `xforms:input` binds to a node of instance data identifed by the XPath express `/loan/principal`. According to XForms, if the node doesn't exist, the field behaves as if it were bound to a non-relevant node (see Section 4.5.7, “The *relevant* MIP Affects the *visible* and *active* Options”). The `xforms:label` provides the default value for the XFDL options `label` and `acclabel`. The `value` option is not shown in the above serialization because it is transient in XFDL items containing XForms controls (see Section 4.5.9, “On the Transience of Options in XFDL Items that Skin XForms Controls”). However, the `value` is available during run-time for use in XFDL computes, and the `format` option casts it to a currency value. The locale of the currency is harvested by the XFDL compute system from the user selection made in the example of Section 4.3.3, “The *popup, list, radiogroup* and *checkgroup* Items Skin *xforms:select1*”.

4.3.3. The *popup*, *list*, *radiogroup* and *checkgroup* Items Skin *xforms:select1*

Perhaps the most common user interface control for obtaining a user choice is the popup list (a.k.a. the drop down list). This corresponds to the minimal appearance for the `xforms:select1`, which corresponds to the XFDL `popup` item. However, the form author may desire the compact appearance of an XFDL `list` or the full appearance of either a group of radio buttons (`radiogroup`) or a even a single-selection group of check boxes (`checkgroup`) The skin methodology for these items is the same except for the use of the appearance attribute. The minimal appearance is considered to be the default in XFDL, so the form author need not put the attribute when the skin is a `popup`, but the appearance attribute is required for the compact and full settings, and the setting of the appearance attribute must match the XFDL item skinning the `xforms:select1`.

```
<list sid="Currency">
 <xforms:select1 ref="principal/@currency" appearance="compact">
  <xforms:label>Choose currency</xforms:label>
  <xforms:item>
         <xforms:label>US Dollars</xforms:label>
         <xforms:value>en_US</xforms:value>
  </xforms:item>
  <xforms:item>
```

```
          <xforms:label>Canadian Dollars</xforms:label>
          <xforms:value>en_CA</xforms:value>
   </xforms:item>
  </xforms:select1>
  <itemlocation>
             <after>Principal</after>
             <alignt2t>Principal</alignt2t>
             <expandb2b>Principal</expandb2b>
  </itemlocation>
</list>
```

This example shows the XFDL `list` item as the skin for the single-selection control for choosing the currency locale. Although the user sees `US Dollars` and `Canadian Dollars`, the underlying value is the raw locale string, which is what the compute on the `currencylocale` references.

This example also shows a few of the suboptions available within the `itemlocation` option. The default size of a `list` is wide enough to surround the widest choice and tall enough to show all choices, plus the label (if any). However, this is not compact enough in this case. Since we would like to match the countenance of the Principal field, the height must be enough for the label and only one choice. First, the list item is moved so that its left margin is after the right of the Principal field. The top margin is then aligned with the principal field; alignment does not affect the size of the item. Finally, the last suboption changes the height so that the bottom margin of the list is aligned with the bottom of the Principal field (a negative expansion in this case). The width is not changed by this sequence of locators.

XForms supports an alternate syntax for specifying the choices for an `xforms:select1`. Instead of having one or more `xforms:item` elements, the `xforms:select1` may contain an `xforms:itemset` that implicitly generates one `xforms:item` per node identified in a node-set binding. XFDL supports this feature, but since the same technique is used in an `xforms:select`, the example is deferred to Section 4.3.4, “The *list* and *checkgroup* Items Skin *xforms:select*”.

The XFDL processor implicitly creates an XFDL item to correspond with the presentation of each `xforms:item`, whether explicitly declared or generated by `xforms:itemset`. This allows more powerful control of the presentation by adding XFDL options to each item using an `xforms:extension`. Again, the same technique is used with an `xforms:select` so the example is also shown in Section 4.3.4, “The *list* and *checkgroup* Items Skin *xforms:select*”.

4.3.4. The *list* and *checkgroup* Items Skin *xforms:select*

XFDL offers multiselect controls of both compact and full appearance by skinning `xforms:select` with the XFDL `list` and `checkgroup` items, respectively.

```


<!-- In the xforms:model -->

     <xforms:instance>
        <mainData>
            <chosenDVDs>1 3 5</chosenDVDs>
        </mainData>
     </xforms:instance>

     <xforms:instance xmlns="" id="DVDChoices">
        <data>
          <DVD title="The Matrix" code="1"/>
```

```
           <DVD title="The Wedding Singer" code="2"/>
           <DVD title="Tombstone" code="3"/>
           <DVD title="First Wives' Club" code="4"/>
           <DVD title="The Terminator" code="5"/>
           <DVD title="When Harry Met Sally" code="6"/>
         </data>
     </xforms:instance>

     <xforms:bind nodeset="chosenDVDs" constraint="string-length(.) &lt;= 5"/>
...

<!-- In an XFDL page -->

     <checkgroup sid="constrained_multiselect">
       <xforms:select appearance="full" ref="chosenDVDs">
          <xforms:label>Bonus selection-- Choose 3 DVDs for only a
buck!</xforms:label>
          <xforms:itemset nodeset="instance('DVDChoices')/DVD">
                <xforms:label ref="@title"/>
                <xforms:value ref="@code"/>

                <xforms:extension>
                   <labelfontinfo>
                      <fontname>Times</fontname>
                      <size>12</size>
                   </labelfontinfo>
                   <itemlocation>
                      <after compute="itemprevious"/>
                   </itemlocation>
                </xforms:extension>

          </xforms:itemset>
       </xforms:select>

       <labelfontinfo>
            <fontname>Times</fontname>
            <size>12</size>
            <effect>bold</effect>
       </labelfontinfo>

       <border>on</border>
       <bgcolor>cornsilk</bgcolor>

     </checkgroup>
```

The first interesting feature of this example is that it shows how to constrain the user to some subset of the available choices. The XForms constraint is used to prevent the user from submitting a form in which more than three choices have been made. The expression accounts for the need to use a character entity for the less-than symbol in an XML attribute, and it acccounts for the space separator between the code values. Because neither XPath 1.0 nor XForms 1.0 offer a function to count the space separators, the form author should make all values have the same length.

Due to the `nodeset` expression, a check box item is generated to present each `DVD` element in instance data that is matched by the node-set binding of the `xforms:itemset`. For each generated check box, the `xforms:label` is evaluated relative to the generating node from the nodeset binding and determines the textual label associated with the check box. Similarly, the `xforms:value` determines the data that goes into node referenced by the `xforms:select` when the corresponding check box is selected by the user.

The two XFDL options in the `xforms:extension` are copied to each generated check box item. The compute in the itemlocation automatically runs to determine the preceding checkbox. Every check box with a preceding check box is placed after its predecessor, resulting in a horizontal row of check boxes. The default behavior gives a vertical column of checkboxes because the default for `itemlocation` is to put each item below its predecessor (within the context of the container item).

The `checkgroup`, then, is considered to be a container item for the generated check boxes. The XFDL options outside of the `xforms:select1` affect the containing `checkgroup`. For example, the default border is off and the default background color is transparent, but this example makes alternate settings.

4.3.5. The *check* Item skins *xforms:input*

Sometimes input is the answer to a simple yes/no question. Rather than making the user type yes/no or true/false, XFDL allows a `check` item to skin an `xforms:input`.

```
    <check sid="single_check">
        <xforms:input ref="wantMembership">
            <xforms:label>Do you want to join the DVD club?</xforms:label>
        </xforms:input>
        ...
    </check>
```

4.3.6. The *combobox* Item skins *xforms:select1* and *xforms:input*

When the user's textual input is to be aided by a popup (drop down) of some kind, the XFDL `combobox` item is used to provide the assisted input capability. When the item skins an `xforms:select1`, then a popup list is available to the user. The XForms syntax is the same except that the `xforms:select1` must contain the attribute setting `selection='open'`.

```
   <combobox sid="PickYourPet">
      <xforms:select1 ref="/pet/type" selection="open">
            <xforms:label>Pick your pet:</xforms:label>
            <xforms:itemset nodeset="instance('usualPets')/choice">
               <xforms:label ref="@petname"/>
               <xforms:value ref="@petcode"/>
            </xforms:itemset>
      </xforms:select1>
   </combobox>
```

When the `combobox` skins an `xforms:input`, and the `format` option indicates that the datatype is a date, then a calendar chooser is available to assist user input.

```
<combobox sid="EndDate">
    <xforms:input ref="rental/endDate">
        <xforms:label>Agreement End Date</xforms:label>
    </xforms:input>
    <format>
        <datatype>date</datatype>
    </format>
</combobox>
```

4.3.7. The *slider* Item Skins *xforms:range*

Selection of a value within an integer or floating point range can be accomplished with the `slider` item, which skins the `xforms:range`. As usual, the XFDL options affect the presentation of the range control. The font information affects the tick mark labels, for example. This example also shows how to change from the default of no border and a transparent background.

```
   <slider sid="OverallRating">
       <xforms:range ref="rating" start="1" end="5" step="1">
            <xforms:label>Overall Rating:</xforms:label>
       </xforms:range>
       <labelfontinfo>
            <fontname>Times</fontname>
            <size>14</size>
            <effect>bold</effect>
       </labelfontinfo>
       <fontinfo>
            <fontname>Times</fontname>
            <size>10</size>
       </fontinfo>
       <border>on</border>
       <bgcolor>cornsilk</bgcolor>
   </slider>
```

4.3.8. The *button* Item Skins *xforms:upload*

The `xforms:upload` control allows for a single-file attachment. When skinned by an XFDL `button` item, the user can activate a file selection dialog by pressing the button, and the selected file's content will be placed in the referenced instance data node (base 64 encoded).

```
<button sid="GetThePicture">
    <xforms:upload ref="/data" mediatype="image/*">
        <xforms:label>Press me to attach a picture</xforms:label>
    </xforms:upload>
```

```
</button>
```

4.3.9. The *button* and *action* Items Skin *xforms:submit* and *xforms:trigger*

The `xforms:trigger` is used to activate a sequence of XForms actions, such as inserting or deleting data, changing the input focus or sending a submission. The `xforms:submit` acts as a shorthand for activating an triggering an `xforms:send` action. The XFDL `button` (and the automatic `action`) skins both of these XForms controls.

```


<!-- The hard way -->
<button sid="Submit">
    <xforms:trigger>
        <xforms:label>Submit Loan</xforms:label>
        <xforms:send ev:event="DOMActivate" submission="S"/>
    </xforms:trigger>
    ...
</button>

<!-- The easy way -->
<button sid="Submit2">
    <xforms:submit submission="S">
        <xforms:label>Submit Loan</xforms:label>
    </xforms:submit>
    ...
</button>
```

4.3.10. Custom Items

Single and multiple line data can be extracted from the instance without rendition to the end-user by using an item that is not in the XFDL namespace to skin the appropriate XForms control.

```


<my:simpleData xmlns:my="http://example.com" xfdl:sid="SimpleData">
     <xforms:input ref="data">
         <xforms:label/>
     </xforms:input>
</my:simpleData>
```

The instance data node referenced by the single node binding is available to the `xfdl:value`. So, other XFDL computes can refer to it using `SimpleData.value` and the XFDL `set()` function can be used to push data through the single node binding to the bound instance node. See Section 5.5, “Rich Text Support” for an example of using a custom item to capture the plain text associated with a rich text field.

4.4. XFDL Skin Items for Grouping, Switching and Repeating Controls

4.4.1. The *pane* Item Skins *xforms:group* and *xforms:switch*

A collection of XFDL items can be logically and visually contained using the XFDL `pane` item, which skins an `xforms:group` if the collection of items to be contained is invariant or the `xforms:switch` if the collection of items must be changed based on user events. A motivating example appears in Section 4.4.2, "The *table* Item Skins *xforms:repeat*".

As with the prior items that were used to contain either items, such as `checkgroup`, the options for background color and border exist and default to transparent and off, respectively. The default for the `visible` option is based on the relevance of the node referenced by the single node binding expressed in the underlying `xforms:group` or `xforms:switch`, or to `on` if the single node binding is not expressed. All of the items contained by a `pane` are invisible if the `pane` is invisible due either to the default or to an explicit setting.

4.4.2. The *table* Item Skins *xforms:repeat*

The `table` element is capable of iterating a collection of XFDL items once for each node of the nodeset binding of the `xforms:repeat` skinned by the table item. Any XFDL items can be placed into the `xforms:repeat`, including `pane` items and other `table` items. Here is an example:

```
<!-- Instance data, including repeat 'row' prototypical data -->
<xforms:instance id="po" xmlns="">
    <po>
       <order>
           <row>
               <product/>
               <unitCost>0</unitCost>
               <qty>0</qty>
               <lineTotal>0</lineTotal>
           </row>
       </order>
       <subtotal>0</subtotal>
       <tax>0</tax>
       <total>0</total>
    </po>
</xforms:instance>

<!-- column headers -->
<label sid="productColumn">
   <value>Product Name</value>
   <itemlocation>
      <y>75</y>
      <width>150</width>
   </itemlocation>
</label>

<label sid="unitCostColumn">
   <value>Unit Cost</value>
   <size>
      <width>15</width>
      <height>1</height>
   </size>
   <justify>center</justify>
   <itemlocation>
      <after>productColumn</after>
   </itemlocation>
</label>

<label sid="qtyColumn">
   <value>Quantity</value>
   <justify>center</justify>
   <itemlocation>
      <after>unitCostColumn</after>
      <width>100</width>
   </itemlocation>
</label>

<label sid="lineTotalColumn">
   <value>Line Total</value>
   <justify>right</justify>
   <itemlocation>
      <after>qtyColumn</after>
      <width>100</width>
   </itemlocation>
</label>

<!-- The table item, ignoring repeat template for now -->

<table sid="POTable">
   <bgcolor>cornsilk</bgcolor>
   <border>on</border>

   <xforms:repeat nodeset="order/row[position()!=last()]"
                  id="PORepeat">
       <!-- XFDL items to repeat go here -->
   </xforms:repeat>
</table>
```

The usual XFDL options appropriate for a container item are available to the `table` item with the usual defaults. The default for visibility is extended from single node binding to node-set binding in a natural way. The default is `off` if the node-set binding does not match any relevant nodes (and, of course, an empty node-set contains no relevant nodes).

The node-set binding excludes the last `row` node because techniques used later in this paper will preserve the last row to be used as prototypical data for inserting new rows. The `id` will also be used later to help determine the row that has the focus so that insertion and deletion operations can be performed relative to the focused row. Now we are able to see an example of content for the `xforms:repeat`

```


<pane sid="focusPane">
```

```
      <bgcolor compute="focused=='on' ? '#C0C0FF' : 'transparent'"/>
      <xforms:group ref=".">
         <combobox sid="Product">
            <xforms:select1 ref="product" selection="open">
               <xforms:label>Choose product</xforms:label>
               <xforms:itemset nodeset="instance('products')/product">
                  <xforms:label ref="@name"/>
                  <xforms:value ref="@code"/>
               </xforms:itemset>
            </xforms:select1>
            <suppresslabel>on</suppresslabel>
            <itemlocation>
               <expandr2r>productColumn</expandr2r>
            </itemlocation>
         </combobox>
         <field sid="unitCost">
            <xforms:input ref="unitCost">
               <xforms:label>Unit Cost</xforms:label>
            </xforms:input>
            <justify>center</justify>
            <suppresslabel>on</suppresslabel>
            <itemlocation>
                 <after>Product</after>
                 <expandr2r>unitCostColumn</expandr2r>
            </itemlocation>
            <format>
               <datatype>currency</datatype>
            </format>
         </field>
         <field sid="Qty">
            <xforms:input ref="qty">
               <xforms:label>Quantity</xforms:label>
            </xforms:input>
            <justify>center</justify>
            <suppresslabel>on</suppresslabel>
            <itemlocation>
                 <after>unitCost</after>
                 <expandr2r>qtyColumn</expandr2r>
            </itemlocation>
         </field>
         <label sid="LineTotal">
            <xforms:output ref="lineTotal"/>
            <justify>right</justify>
            <itemlocation>
               <after>Qty</after>
               <expandr2r>lineTotalColumn</expandr2r>
            </itemlocation>
            <format>
               <datatype>currency</datatype>
            </format>
            <acclabel>The total cost for all units of this product</acclabel>

         </label>
```

```
     </xforms:group>
</pane>
```

Any number of items can be placed in an `xforms:repeat`, which handles the collection of items as if they were surrounded by an anonymous `xforms:group`. However, in this example, the group is explicitly created with an XFDL `pane` item in order to get more control over the appearance of each table row. Specifically, the `bgcolor` option is overloaded with a compute that sets the background to a highlighted color if the `pane` has the focus. The focus is hierarchically assigned in XFDL, so the `pane` has the focus if an item within it has the focus. The net effect of this compute, then, is that the entire repeat table row that contains the focus is highlighted.

A second interesting feature of this example comes from the behavior of the `itemlocation` option. Observe that the locators make references within the containing group and also references to the column headers. Generally, this option behaves in a scoped manner with respect to hierarchically organized XFDL items, so XFDL items within inner nested repeats can refer to column headers in the outer nesting repeat (and so on out to the page level).

Finally, this example also shows additional accessibility support available in XFDL forms via the `suppresslabel` and `acclabel` options. The latter option can provide a screen-reader-only message. The `suppresslabel` option is more important to iterated XForms constructs as it allows the XForms view controls to retain their `xforms:label` for accessibility purposes without infesting the visual display with information that is obtained by other means within the form's visual design, such as column headers.

4.5. The Effects of XForms Constructs on the XFDL Presentation Layer

4.5.1. Single node binding and the *value* option

The single node bindings of most XForms view controls is transferred to the XFDL `value`, where it is formatted (if there is any formatting to do) and made available to the compute system and rendition engine. One exception occurs for XForms controls that have optional single node bindings, where the binding is only used to obtain a relevance setting. The second exception is for the `xforms:upload`, where the single node binding does not affect the display text but is also used for more than relevance. The third and final exception is that the single node binding attaches to the `rtf` option in rich text fields (see Section 5.5, “Rich Text Support”).

The single node binding is also used by controls capable of input to push data into the model whenever the associated XFDL `value` changes. Such changes can occur as the result of user input or as the result of XFDL computes setting data into the value. The default behavior is to convert the value content from its presentational format into raw data. Typically this is done according to the `datatype` given in the `format` option. All the specifications of the `format` option are reversed to create raw data that matches an XML schema data type that is associated with the XFDL `datatype`. The most usual applications for this feature are conversion of currency values into `xsd:double` and date values into `xsd:date`. The `format` option also contains a suboption that can turn off the unformatting so that, for example, long dates like `April 27, 1968` can be placed directly in instance data. The suboption is called `keepformatindata`.

4.5.2. The *xforms:label* Affects the XFDL *label* and *acclabel* Options

The `xforms:label` creates a weak binding only with the XFDL `label` and `acclabel` options. It provides only their default content. This allows either option, if they exist and are non-empty, to override the default. This is in keeping with the notion of XFDL as a skin over XForms. This means, however, that content from XForms cannot be accessed by the compute system using references to the XFDL options. Of course, the information from XForms labels can be accessed by querying the XForms labels or their referenced data nodes directly.

4.5.3. The *xforms:hint* and *xforms:help* Affect the XFDL *help* Option

XFDL has a system of associating help messages with atomic user interface items. When the user desires help for a control or enters invalid input, the help message appears in a tooltip. If the XFDL `help` option is not specified, then the underlying `xforms:hint` and `xforms:help` are concatenated and used instead.

4.5.4. The *xforms:alert* Affects the *format* Option's *message* Suboption

User input can be invalid due to failing the validity tests imposed by XForms on the associated instance data node or by failing the validity test of the XFDL `format` option (if one is given). When the user has entered invalid input, the `message` suboption of the `format` option is displayed in a tooltip. If the XFDL `message` is not specified, then the underlying `xforms:alert` is used instead.

4.5.5. The *readonly* MIP Affects the *readonly* Option

The `readonly` MIP of the instance node bound to a control by the single node binding affects whether the user can modify the data. The MIP sets the default for the `readonly` option, though contradicting the MIP results in data changes not be accepted by the model anyway.

4.5.6. The *required*, *type* and *constraint* MIPs Affect the *format* Option

The validity of the value of an item may be assessed by the `format` option, if one is given. These results are then combined with the information from the XForms model to determine the overall item validity. An item is invalid unless all channels report validity: any XML schema definitions, the `required`, `type` and `constraint` MIPs and the XFDL `format`. The `format` option does not have to be present for the XForms validity channels to be applied. If the result is invalid but the value is empty, then the item shows itself to be required. Otherwise, if the result is invalid and the value is non-empty, then the more severe invalidity countenance is displayed. Both states result in the tooltip being displayed with message content from Section 4.5.4, “The *xforms:alert* Affects the *format* Option's *message* Suboption”.

4.5.7. The *relevant* MIP Affects the *visible* and *active* Options

The `relevant` MIP affects atomic controls by setting the default for the `visible` option to `off`. The `active` option is also turned `off` (i.e. the item is disabled.

If a container item (`pane`, `table`, `radiogroup`, `checkgroup`) is not visible, then its contained items are not visible, regardless of their own XFDL settings and relevance.

There are two special relevance rules for the `table` item. First, it has a node-set binding, so it's `visible` option defaults to `off` only if none of the nodes in the node-set binding are relevant (or if the node-set binding produces an empty node-set). Second, for any non-relevant node, the corresponding row of items generated by the `table` is not shown, and the `table` geometry is collapsed to omit the space for the hidden items.

Other than non-relevance of a table row having an effect on the table size, the XFDL `visible` option, whether set explicitly or implicitly by relevance, does not change the item bounding box and therefore does not affect the layout of other items.

4.5.8. The Effects of *xforms:group*, *xforms:switch* and *xforms:repeat* on *itemlocation*

The `itemlocation` option provides over 30 keywords for use as a sequence of suboptions that manipulate the bounding box of an item. Some of the keywords make absolute settings, and others make settings by aligning or expanding a bounding box margin relative to another item or items, or by positioning an item relative to another item.

When an item is within a `table` or `pane`, the references from options like `itemlocation` are automatically scoped according to the logical hierarchy expressed in the document. Further information on and a markup example of this effect appear in Section 4.4.2, "The *table* Item Skins *xforms:repeat*".

4.5.9. On the Transience of Options in XFDL Items that Skin XForms Controls

When XFDL computes run, they change the contents of options. However, if the item containing a changed option also contains an XForms control, then the changes are not serialized because they can always be recomputed whenever the form is run once instance data is pushed out through the XForms view controls to the XFDL item presentation layer. A second rationale is that if it were necessary to save such information at the XFDL layer, then the item could not be used in an `xforms:repeat` anyway. A principal benefit of the transience is that the markup for the presentation layer is not changed, which means it can be digitally signed as a way of offering trusted forms application.

5. Advanced Issues

5.1. A Repeat Methodology for Persisted Data

The `xforms:insert` action is used to add a new subtree of instance data by copying an existing subtree. The attribute set was designed to be synchronized with the `xforms:repeat`, so the action has a node-set binding, the last element of which is duplicated. The `xforms:repeat` is then expected to automatically update the iteration of its template to allow a view of the additional data. Here is the trigger for adding a new row to the purchase order example of Section 4.4.2, "The *table* Item Skins *xforms:repeat*":

```
<button sid="Insert">
    <xforms:trigger>
        <xforms:label>Insert Row</xforms:label>
        <xforms:action ev:event="DOMActivate">
            <xforms:insert nodeset="order/row"
                           at="index('PORepeat')" position="after"/>
            <xforms:setfocus control="PORepeat"/>
        </xforms:action>
    </xforms:trigger>
</button>
```

When the user activates this button, the node-set binding obtains all the `row` elements, and the insertion operation duplicates the last `row` after the instance node corresponding the row of the repeat table that last had the focus, which is availabe from the `index()` function. The effect of the insertion is that the new node appears and that the index of the `xforms:repeat` is adjusted to indicate the position of the newly added node. Once this is done, the trigger's action sequence sets focus back from the activated button to the table. Since focus is hierarchic in XFDL, the table then passes the focus to the first focusable item in the currently indexed row (the new row).

The challenge with this scheme is that the newly created instance subtree is not empty, but rather contains the same data as the last row of the table. An easy solution would be to add `xforms:setvalue` actions after the `xforms:insert` to clear the new row. Of course this requires the form author to write a different action sequence for each table. then, if the schema for the data changes, then the action sequence has to be changed. The solution also does not scale to nested repeats because there is no way to clear an inner table (XForms 1.0 contains no looping actions).

It is better to have a general method that can clear the row of data without having to know the data structure. This would also facilitate the creation of design environment support for tables, and it would solve the nested repeat problem. The following are the components of a solution:

- omit the last node from the node-set binding of the repeat
- use an `xforms:bind` to make the last node non-relevant
- use the `xforms-model-construct-done` event to conditionally insert a second row of data if only row exists
- modify the delete button pattern to conditionally insert a second row of data if only one row exists.

The first of the above components was already shown in Section 4.4.2, "The *table* Item Skins *xforms:repeat*". The latter three are shown below. Two of the components call for a conditional insert action. Although XForms 1.0 does not offer conditional actions per se, XPath has predicates that are capable of keeping or omitting nodes based on conditions, and if the node-set binding produces an empty result, then actions like insert and delete perform no operation.

```
<!-- In the XForms model...  -->
<xforms:bind nodeset="order/row[last()]" relevant="false()"/>

<!-- For every row, last() returns the total number of row elements in the
     containing nodeset, so the condition last()=1 always returns false
     unless there is only one row, at which point a second row is inserted.
     The position is before, so that repeat indices will be set to 1. -->
<xforms:insert ev:event="xforms-model-construct-done"
               nodeset="order/row[last()=1]" at="1" position="before"/>

<!-- In the page... -->
<button sid="Delete">
   <xforms:trigger>
      <xforms:label>Delete Row</xforms:label>
      <xforms:action ev:event="DOMActivate">
         <xforms:delete nodeset="order/row" at="index('orderTable')"/>
        <xforms:insert nodeset="order/row[last()=1]" at="1" position="before"/>

         <xforms:setfocus control="orderTable"/>
      </xforms:action>
   </xforms:trigger>
</button>
```

In the XML fragments above, the conditional insert in the `xforms-model-construct-done` event causes a second `row` element to be added if there is only one. This starts the form with one relevant row of data, and that data is in the initial state. Since the user cannot modify the non-relevant row, it will remains in the initial state. When the user clicks the delete button, if the last relevant row is deleted, then the one non-relevant row is immediately inserted back into the data by the conditional insert in the delete button's action sequence. The end-user experience is that hitting delete on the last row simply clears the values in the row.

5.2. Accessibility

XFDL includes presentation layer extensions that facilitate better accessibility support than is typically achievable with other XForms implementations at this time. A key issue arises with the accessibility of tables driven by `xforms:repeat`. The problem is that many form authors are tending to provide empty `xforms:label` content for XForms view controls that are within an `xforms:repeat` because they don't want the labels to show up in every row of a repeat table.

In order to allow forms authors to specify XForms label content in repeated controls without sacrificing on the visual fidelity of the form, XFDL includes the `suppresslabel` option so that items can turn off the visual rendition of the label while still allowing it to be available to accessibility software. Furthermore, XFDL also supports the option `ac-clabel` as a means of creating very high fidelity accessibility support through tailored messaging that is more appropriate for screen reading than some visual labels tend to be. Markup examples of these options are available in Section 4.4.2, “The *table* Item Skins *xforms:repeat*”.

5.3. Preserving the Selected Case of a Switch

The `xforms:switch` contains a number of `xforms:case` elements, each of which is very similar to an `xforms:group` in that it is the container for a collection of zero or more items to be rendered. The purpose of this construct is to enable the user to switch between groups of items. The initial case of the switch is chosen as the first case with a `selected` attribute containing the value `true` (or the first case, if none contain `true`).

As the user interacts with the form, the `xforms:toggle` action is somehow triggered to change the selected case of the switch. However, this does not change the values of the `selected` attributes in the cases to reflect the current state of the switch. So, if the full document containing the form is saved and reloaded, the state of the switches is lost. Similarly, a user who submits data to a server one day and downloads the data to continue the next day will also have lost the switch states. Finally, part of the implementation experience of enabling switch in repeat (see Section 5.4, “Support for Switch in Repeat and IDREF into Repeats”) was the determination that the `selected` attributes of a switch would be unable to store the selected states of all switches generated by a repeat.

To solve this problem, XFDL supports the attribute `xfdl:state` on XForms switch elements. This optional attribute contains an XPath expression evaluated with a context node equal to the result of the single node binding. If the single node binding is not expressed, then the context node for the `xfdl:state` is obtained as if the single node binding were `ref="."`. If this attribute is given, resolves to a node, and that node contains the ID of a case in the switch, then the switch is initialized to that case (otherwise the normal initialization is used). Also, whenever the switch is toggled, the ID of the newly selected case is written to the node referenced by the `xfdl:state` (if the attribute is given and its XPath expression resolves to a node). If the XPath expression identifies more than one node, then the attribute is considered to resolve to the first node in the set (i.e. the attribute adheres to the first node rule). A markup example appears in Section 5.4, “Support for Switch in Repeat and IDREF into Repeats”.

This feature is particularly important because it allows the XForms control that is supposed to switch among sets of controls to be used in the implementation of what is often called *model-based switching*, which could previously only be done by using a set of XForms groups bound to some nodes whose relevance was changed as a means of switching groups. Using a switch is preferrable because the markup directly associates the cases involved in achieving a result.

5.4. Support for Switch in Repeat and IDREF into Repeats

As currently written, the XForms Recommendation does not recommend a methodology for how `xforms:switch` should behave when used within an `xforms:repeat`. Indeed, the meaning of IDREFs into repeated content is also not defined for the `index()` function, and the `xforms:setfocus` and `setindex` actions. The recommendation states that a future version will add the feature once implementation experience and user feedback are available. This has now occurred and the future version could possibly be XForms 1.0 Third Edition, or XForms 1.1 at the latest.

One issue that arose through implementation experience with XFDL was the switch state problem described in Section 5.3, "Preserving the Selected Case of a Switch". The main problem identified in the recommendation was how to make the `xforms:toggle` action operate on a switch that is within a repeat when the toggle indicates a switch case using an IDREF in its `case` attribute.

When an IDREF refers to an element that appears inside one or more XForms repeats, the IDREF can be contextualized. For the most part, the index of each containing repeat can be used to translate from an identified element of the source document and the iterations of that content by the repeats. Broadly speaking, then, the target of IDREFs is the repeat row that contains the focus. The discussion in the XForms working group is leaning toward a slightly more sophisticated contextualization for technical reasons, but the difference will not affect the more common use cases.

The guiding principle behind this refinement is that techniques used within a form should continue to work when the form is reused as a component in a larger form. For example, a delete trigger should be able to delete the current row of a repeat table regardless of whether or not that repeat table is within another repeat table. Similarly, the ability to set the focus or toggle a switch should not be lost when the controls involved are placed in a repeat. Here is some example markup for using switch in repeat:

```
<table sid="T">
    <xforms:repeat nodeset="some/nodes[position()!=last()]">
       ...
       <pane sid="P">
          <xforms:switch xfdl:state="@state">
             <xforms:case id="ETF">
                <!-- Controls for electronic fund transfer info -->
             </xforms:case>
             <xforms:case id="CC">
                <!-- Controls for credit card info -->
             </xforms:case>
          </xforms:switch>
          ...
       </pane>
       <radiogroup sid="R">
          <xforms:select1 ref="@state" appearance="full">
             <xforms:label>Credit Card or ETF?</xforms:label>
             <xforms:item>
                 <xforms:label>Credit card</xforms:label>
                 <xforms:value>CC</xforms:value>
             </xforms:item>
             <xforms:item>
                 <xforms:label>Electronic fund transfer</xforms:label>
                 <xforms:value>ETF</xforms:value>
             </xforms:item>
          </xforms:select1>
       </radiogroup>
    </xforms:repeat>
    ...
</table>
```

5.5. Rich Text Support

The goal with XFDL is to allow for more sophisticated types of user input than just plain text data entry and simple selection capabilities. One requirement that arises fairly regularly is the need of users to decorate their textual input. To meet this requirement, the XFDL `field` item supports the ability to accept multiline rich text as user input when the `texttype` option is changed from the default of `text/plain` to the value `text/rtf`. That declaration causes the `field` item to use the `rtf` option as the primary option for communicating content between the user interface control, the XFDL presentation layer, and the XForms view control. The `value` option becomes a secondary option for capturing the plain text shadow of the rich text.

```
<field sid="CommentField">
    <xforms:textarea ref="comment/richtext">
         <xforms:label>Comment:</xforms:label>
    </xforms:textarea>
    <scrollhoriz>wordwrap</scrollhoriz>
    <texttype>text/rtf</texttype>
</field>
```

Although the single node binding attaches the instance node to the `rtf` option, it is sometimes helpful to the application developer to get the plain text as well. This can be used for indexing purposes or be provided to drive other applications that have no rich text rendition capability. By storing the translation being created on the client-side XFDL processor, there is no need to build a server-side translation to meet those requirements. Delegation of this responsibility to the form application can be done with the following markup:

```
<custom:field xfdl:sid="customField" xmlns:custom="http://example.org">
    <xforms:textarea ref="comment/plaintext">
         <xforms:label>Plaintext of comment</xforms:label>
    </xforms:textarea>
    <custom:set_value xfdl:compute="toggle(CommentField.value)=='1'
                                   ? set('value', CommentField.value) : ''"/>
</custom:field>
```

5.6. Label and Button Images from XForms Instance Data

XForms includes the ability to obtain file content and upload it into instance data. This content could, for example, be a picture from a digital camera of a house to be sold. A rich user experience would show the uploaded picture to the user so that he or she can be sure that the correct file was selected. This seems to be the role of the `xforms:output`, but in XForms 1.0 this element is only capable of displaying text. There is a new attribute being added to `xforms:output` in XForms 1.1 called `mediatype` that will allow the output of images. Although it won't be interoperable until XForms 1.1, XFDL offers this feature now as an extension that is unavoidably necessity by the need to solve use cases like the one above. Example markup for this feature appears below:

```
<button sid="GetThePicture">
```

```
    <xforms:upload ref="/data" mediatype="image/*">
        <xforms:label>Press me to attach a picture</xforms:label>
    </xforms:upload>
</button>

<label sid="ShowThePicture">
    <xforms:output ref="/data" mediatype="image/*"/>
</label>

<button sid="SubmitThePicture">
   <xforms:submit submission="S">
      <xforms:label>
         <xforms:output ref="/data" mediatype="image/*"/>
      </xforms:label>
   </xforms:submit>
</button>
```

5.7. Attachments, Organized and Compressed

Although XForms 1.0 offers an attachment feature, it understandably has a few limitations in the first release of the language. The most important use case was to allow attachment of an image while being aligned with an XML schema type. Since the xsd:base64Binary type does not recognize compression and since images are typically compressed anyway, there is no compression feature for XForms attachments. However, if the user must attach a large file or many files that are not compressed, such as word documents, spreadsheets and other supporting documents of a financial filing, for example, then the size of instance data may become unwieldy. Moreover, if there are many files, a rich user experience feature would be to allow the user to attach multiple files with a single control and organize those files into a structure defined within the form.

XFDL allows these features as extensions to the `button` item that can be accessed by setting the button's `type` option to one of `enclose`, `extract` or `remove`. If only compression is required, then the button can be associated with a single attachment using the `data` option. A folder scheme can be specified with the `datagroup` option. Though the requirement has not yet arisen in our practice, the sufficiently motivated form author could use the XFDL compute system to allow the end-user to construct the set of folders for the attachments.

5.8. Calling Web Services

XForms 1.0 contains the ability to invoke web services that are based on [SOAP 1.2]. Although the [WSDL] descriptions of these services will not be directly consumable until a future version of XForms, a design environment can easily convert to the underlying SOAP envelopes and schema compliant XML hosted in XForms instances. The following parameterization of the XForms submission is needed:

- Use the `ref` attribute to indicate the SOAP request envelope.
- Set the `mediatype` attribute to `application/soap+xml; action=NameOfService`.
- Use `xforms:setvalue` actions in an `xforms-submit` handler to copy instance data to the SOAP request envelope.
- Use the `instance` attribute to indicate the instance in which to place the SOAP response envelope.
- Use `xforms:setvalue` actions in an `xforms-submit-done` handler to copy from the SOAP response envelope to instance data.

The key limitation to this approach is that only simple data can be sent and received. XForms 1.1 will allow arbitrary XML subtrees to be copied to and from SOAP envelopes via updates to the `xforms:insert` action. XForms 1.1 will also support [SOAP 1.1] by setting the `SOAPAction` HTTP header based on the `mediatype` attribute content, changing the HTTP content type from `application/soap+xml` to `text/xml`, and copying over the `charset`, if given, from the `mediatype` attribute.

5.9. Digital Signature Security and Transaction Non-Repudiation

A form is responsible for gathering the data of a transaction from a user (or users) so that the tranaction can ultimately take place between the user and the recipient of the form. If the recipient wants to be able to later prove the correctness of performing the transaction with or on behalf of the user, then the form application would benefit from the use of digital signatures.

A user can create a digital signature over any message (octet sequence). Once created the validation of that signature by the recipient ensures that the message has not changed and identifies the user who created the signature. Of course, the message signed should consist not only of transaction data but also a record of the presentation layer used to provide the data to the user. The reason is that the user is not reading the data but rather is interpreting the data through the presentation layer. Signing only the data would be somewhat analogous to signing only the ink or laser printer toner without affixing it to the paper.

Since XFDL keeps the presentation layer and data in the same file, it is easy to create digital signatures and add them to the bundle. The XFDL signature system supports signing the full document, but it also contains signature filters that allow part of the document to be omitted. The omission can be as simple as omitting the office-use-only section of a form so that it can be filled out after the user's signature is affixed. However, numerous other use cases exist for documents that have a non-trivial work flow associated with their completion. An easy example is the simplest form of multiple signer scenario, the signer/co-signer case. One person must sign the document first, and the document must then undergo the additional transition to allow a second signature. Since both signature are affixed to the document, the first signature must *in some way* omit the markup of the second signature so that the addition of the second signature does not corrupt the message validation performed for the first signature.

To enable signatures in an XFDL form, the form author must add one or more XFDL `button` items with the `type` option set to `signature`. Further options in the button allow for signature configuration, including specification of the cryptographic engine to use as well as the XML filtration to be performed on the document. The XFDL signature filter system provides an option called `signinstance` that allows the form author to omit parts of the data in XForms instances that must undergo transition after the signature is affixed. However, some functionality offered by XFDL has no analog in XForms and is therefore not stored in instance data. There are other XFDL options that allow these other markup extensions to be omitted as needed.

The most commonly asked question about signatures is "Why can't I just say what I want to sign rather than what should be omitted from the signature?" The answer is that XFDL allows the form author to do this, but this type of signature is usually insecure, so it is recommended that this feature be used only in special cases as an optimization to a previously created omission signature. The problem is this: if a signature filters a document based on finding what to sign, then it is easy to add arbitrary content to the document that doesn't match the filters. Therefore, the filters will validate the few bits of the document that its filters matched, but the newly added unsigned content may significantly alter the interpretation of that content. Therefore, it is better for the form author to be specific about what is allowed to change after the signature is affixed, and then sign everything that does not fit that specification. This way, anything added to the document that does not fit the form authors specification of what is allowed to change will cause the signature to become invalid.

5.10. Full Document Submission, Computable URL, and Server-Side Responsibilities

Another of the extended behaviors available from the XFDL `button` item is that it allows the submission of the entire form document rather than just the XML data being manipulated by the model. The `type` option supports the content keyword `done`, which submits the form to a given URL then closes the form. Submission without closing can be done with the `submit` keyword.

```
<button sid="Submit">
   <value>Submit Whole Form</value>
   <type>done</type>
   <url>http://example.org/.../processForm.php</url>
   ...
</button>
```

It is interesting to note that the submission URL is represented simply as another option of the item. As an option, it is subject to the machinations of the XFDL compute system. This means it is easy to set up rules within the form that dynamically decide where the form will go next. This can be used, for example, in very lightweight workflow scenarios.

Once a form document arrives on the server-side, there are several possible tasks that can be performed. The first would be to validate digital signatures, if any, to ensure that further work suggested by the form should indeed be performed. Next, the entire form could be archived for future transaction auditability and non-repudiation purposes. If the transaction data has now been completely gathered by the form, then the underlying instance data can be extracted from the form to drive back-end systems. This extraction is very easy because the entire document is XML. However, it may be necessary to use a module that is partially aware of XForms enough to initialize an XForms model so that non-relevant nodes can be pruned from the data extracted. On the other hand, if the form is not finished gathering transaction data from all parties involved in the transaction, then the form can be moved to the next state of the work flow so that it can be received and further transitioned by the next role player in the work flow sequence.

Bibliography

[SOAP 1.1] *Simple Object Access Protocol (SOAP) 1.1*, Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D., W3C Note, May 2000. Available at http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[SOAP 1.2] *SOAP Version 1.2*, Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., and Nielsen, H. F., W3C Recommendation, June 2003. Available at http://www.w3.org/TR/soap12.

[WSDL] *Web Services Description Language (WSDL) 1.1*, Christensen, E., Francisco, C., Meredith, G., and Weerawarana, S., W3C Note, March 2001. Available at http://www.w3.org/TR/wsdl.

[XForms] *XForms 1.0*, Dubinko, M., Klotz, L., Merrick, R. and Raman, T.V. (eds.), W3C Recommendation, October 2003. Available at http://www.w3.org/TR/xforms.

[XHTML] *XHTML 1.0 The Extensible HyperText Markup Language*, Pemberton, S., et al., W3C Recommendation, August 2002. Available at http://www.w3.org/TR/xhtml1/.

[XML Events] *XML Events: An Events Syntax for XML*, Pemberton, S., Raman, T.V., and McCarron, S. P. (eds.), W3C Recommendation, October 2003. Available at http://www.w3.org/TR/xml-events.

[XML Schema Structures] *XML Schema Part 1: Structures Second Edition*, Thompson, H. S., Beech, D., Maloney, M. and Mendelsohn, N. (eds.), W3C Recommendation, October 2004. Available at http://www.w3.org/TR/xmlschema-1/.

[XML Schema Datatypes] *XML Schema Part 2: Datatypes Second Edition*, Biron, P. V. and Malhotra, A. (eds.), W3C Recommendation, October 2004. Available at http://www.w3.org/TR/xmlschema-2/.

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

Biography

John **Boyer**
Senior Product Architect/Research Scientist
IBM Corporation [http://www.ibm.com]
Victoria
British Columbia
Canada
http://www.ibm.com/software

John M. Boyer is a Senior Product Architect and Research Scientist at the IBM Victoria Software Lab. Since 1993, John has worked to design and implement an XML-based platform for secure and robust web form applications. John is currently co-chair of the XForms working group, and he has co-authored and edited numerous W3C Recommendations, including XForms, XML Signatures, XML Canonicalization and Exclusive Canonicalization, and XPath Filtering. In 2001, John earned his Ph.D. in theoretical computer science (University of Victoria, Canada). He has published numerous refereed journal and conference papers and professional papers on algorithmics, computer security and XML-related technologies.