A
Dissertation
On

DESIGN OF A NEW LANGUAGE BASED ON XFORMS + XFDL AND CONVERSION TO XFDL

Submitted in Partial fulfillment of the requirements
for the award of Degree of

MASTER OF ENGINEERING
(Computer Technology and Application**)**

Submitted By

MINAKSHI ANAND
College Roll No: 03/CTA/05
University Roll No. 2003

Under the Guidance of:
Prof. D Roy Choudhury
Department Of Computer Engineering
Delhi College of Engineering, Delhi

DEPARTMENT OF COMPUTER ENGINEERING
DELHI COLLEGE OF ENGINEERING
DELHI UNIVERSITY
2005-2007

CERTIFICATE

This is to certify that the work contained in this dissertation entitled **"Design of A New Language Based On XForms + XFDL and Conversion to XFDL "** by **Minakshi Anand** in the requirement for the partial fulfillment for the award of the degree of **Master of Engineering** in Computer Technology & Application, Delhi College of Engineering is an account of her work carried out under my guidance in the academic year 2006-2007.

This work embodies in this dissertation has not been submitted for the award of any other degree to the best of my knowledge.

Prof. D Roy Choudhury
Head of Department
Department of Computer Engineering
Delhi College of Engineering
Delhi

Dr. S C Gupta
Sr. Technical Director
National Informatics Center
Delhi

ACKNOWLEDGEMENT

It is a great pleasure to have the opportunity to extent my heartiest felt gratitude to everybody who helped me throughout the course of this project.

It is distinct pleasure to express my deep sense of gratitude and indebtedness to my learned supervisors Dr. S C Gupta and Prof. D Roy Choudhury for their invaluable guidance, encouragement and patient reviews. Their continuous inspiration only has made me complete this dissertation. Both of them kept on boosting me time and again for putting an extra ounce of effort to realize this work.

I would like to specially thank Kalpesh Kumar Meena and Dhiraj Kumar Singh for their constant support in the lab. I would also like to take this opportunity to present my sincere regards to my teachers Prof. Goldie Gabrani, Dr. S. K. Saxena, Mrs. Rajni Jindal, Mr. Manoj Sethi and Mr. Rajeev Kumar for their support and encouragement.

I am grateful to my parents for their moral support all the time, they have been always around to cheer me up in the odd times of this work. I am also thankful to my classmates for their unconditional support and motivation during this work. Being at DCE with them has been a lifetime experience for me, all the time we spend together enjoying life to its fullest, the birthday parties, placement parties and photo sessions discussing new topic or technology would remain with me forever.

I want to thank the IBM Research Scientist Dr. John M. Boyer for his valuable suggestions which helped us a lot during this project. Last but not least, special thanks to the members of World Wide Web Consortium (W3C).

Minakshi Anand
M.E. (Computer Technology and Applications)
College Roll No. 03/CTA/05
University Roll No. 2003
Department of Computer Engineering
Delhi College of Engineering, Delhi-110042

ABSTRACT

XForms is a W3C Forms standard that separates purpose from the presentation. The W3C XForms recommendation specifies 3 parts – model, view and control. XForms gives a way to specify data, the calculations and validations on the data in a declarative way. XML is used to specify data. For calculations, XForms provides several functions of its own and borrow functions form XPath as well. This enables us to write huge real world forms without the need of scripting. This makes coding forms much simpler than it is today. Though XForms is simple, it still contains some code which the form designer cannot easily relate to. In this thesis, we have tried to simplify various XForms tags.

XForms provides a device-independent view part which can be combined with any presentation option. XFDL (eXtensible Forms Description Language), developed as a simpler presentation language for business forms, has graduated to adopt XForms to represent data, calculations and validations. That is, it provides a good presentation option for XForms. But, it makes code (even for simple forms) quite big and repetitive. Though, XFDL is much simple and designer – friendly as compared to HTML or any existing Forms Standard, it may be tedious to write the same tags again and again. IBM Forms Designer is a product that helps design XFDL Forms but as it runs on eclipse, it consumes a lot of memory. Plus, it's a proprietary product. A designer may want to write code in a simple text or xml editor. To make this simpler, we have eliminated the repetitive part of the forms.

This thesis introduces a Forms Language that combines the aspects of XForms and XFDL and makes coding for the designer much simpler. The tags introduced are much more user-friendly. The implementation part includes a converter that converts the simplified code to XForms + XFDL code. Coding of the converter is done in java using the DOM API available in the Xerces (An XML Parser).

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Program Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
DTD	Document Type Definition
HTML	Hyper Text Markup Language
JAXP	Java API for XML Parsing
MIP	Model Item Properties
MVC	Model-View-Controller
P3P	Platform for Privacy Preferences
RFC	Request for Comments
SAX	Simple API for XML
UI	User Interface
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
XHTML	eXtensible Hyper Text Markup Language
XFDL	eXtensible Forms Description Language
XML	eXtensible Markup Language
XSL	eXtensible Style Sheets Language

1. INTRODUCTION

1.1 The Past, Present, and Future of Web Forms

As a general rule, the more interactive a web site is, the more heavily the site's designers rely on web forms, a general term for all different kinds of technologies used to gather information from users. Without forms, web sites are far less interesting. Form-less web sites were the norm in the early days of the Web and provided a one-way deluge of static information.

The addition of forms to Hypertext Markup Language (HTML), the primary language used in web pages, launched an entirely new way of surfing the Web. Using HTML forms, searching for information became possible on a worldwide scale. Sites such as Yahoo! quickly became the most popular "portals" of entry on the Web. Later, as developers pushed the limits of forms technology farther, web sites became even more interactive and customizable.

Shortly after the initial tempering of HTML, various individuals began considering the usefulness of forms alongside hypertext. HTML Version 2.0, as presented in a document called Request for Comments (RFC) 1866, was the first time that web forms were seriously considered for standardization. That RFC captured HTML as found in common use prior to June 1994. At this point, HTML already included forms, thanks to a 1993 proposal called HTML+.

Care and maintenance of the HTML family of specifications have since been handed over to the World Wide Web Consortium, or W3C. The last non-XML-based version of HTML was version 4.01, which didn't change forms processing much. New development of the standard is taking place on a closely related technology called XHTML, where the X indicates an XML foundation. XHTML 1.0 and 1.1 were largely concerned with details of the transition to XML and ways to combine vocabularies, not with major changes to the language.

XHTML 2.0, in contrast, is making some improvements that aren't compatible with earlier flavors of HTML. The largest such change is the adoption of XForms as a replacement for the older HTML forms technology.

The XForms standard is device- independent and therefore can be combined with any presentation language like HTML, XFDL (eXtensible Forms Description Language), SMIL (Simple Multimedia Integration Language), SVG (Scalar Vector Graphics) etc.

1.2 MOTIVATION

I always faced a lot of problems coding forms in HTML. Though, at a time when HTML was developed, it changed the entire web scenerio, coding of today's forms in HTML requires use of large amount of scripting and finally the code becomes unmanagable. So, much simpler standards based on XML are being developed. One such standard, I happened to use was XForms. It follows the basic rule that everything can be specified using markup, and there is no need of scripting. But, XForms has certain tags that difficult are to understand. So, we have tried to simplify these.

As XForms requires a presentation language for completeness, we used XFDL. In the process, we identified some loopholes in XFDL as well and thus, removed those in our new simplified language (an XML Language).

1.3 HTML

HTML is a non-proprietary format based upon SGML, and can be created and processed by a wide range of tools, from simple plain text editors - you type it in from scratch- to sophisticated WYSIWYG authoring tools. HTML uses tags such as `<h1>` and `</h1>` to structure text into headings, paragraphs, lists, hypertext links etc.

The introduction of the forms chapter in HTML 4.01 reads: "An HTML form is a section of a document containing normal content, markup, special elements called controls (checkboxes, radio buttons, menus, etc.), and labels on those controls. Users generally 'complete' a form by modifying its controls (entering text, selecting menu items, etc.), before submitting the form to an agent for processing (e.g., to a web server, to a mail server, etc.)."

Forms represent a structured exchange of data. In HTML forms, the structure of the collected data, called a *form data set*, is a set of name/value pairs. The names and values that are included in this set are solely determined by the controls present within the form, so that adding a new control element, as well as adding to the user interface, also adds a new name/value pair to the data set. Many authors take for granted this basic violation of the separation between the data layer and the user interface layer—a problem that XForms has gone to considerable lengths to alleviate.

1.4 XML

XML is a text-based markup language that is fast becoming the standard for data interchange on the Web. XML(e**X**tensible **M**arkup **L**anguage) provides a tag-based syntax for structuring data and applying markups to documents. But unlike HTML, XML tags *identify* the data, rather than specifying how to display it. Where an HTML tag says something like "display this data in bold font"(<b>...</b>), an XML tag acts like a field name in your program. It puts a label on a piece of data that identifies it (for example: <message>...</message>).

Documents that conform to XML may be made up of a variety of syntactic constructs such as **elements, namespace declarations, attributes, processing instructions, comments and text.**

1.4.1 Elements

```
<tagname></tagname>
<tagname/>
<tagname>children</tagname>
```

Elements typically make up the majority of the content of an XML document.Every XML document has exactly one top-level element, known as the *document element*. Elements have a name and may also have children. These children may themselves be elements or may be processing instructions, comments, CDATA sections, or characters. Elements may also be annotated with attributes.

1.4.2 Namespaces

```
<prefix:localname xmlns:prefix='namespace URI'/>
<prefix:localname xmlns:prefix='namespace URI'>
</prefix:localname>
<prefix:localname xmlns:prefix='namespace URI'>children
</prefix:localname>
```

Because XML allows designers to chose their own tagnames, it is possible that two or more designers may choose the same tagnames for some or all of their elements. XML namespaces provide a way to distinguish deterministically between XML elements that have the same local name but are, in fact, from different vocabularies. This is done by **associating an element with a namespace**. A namespace acts as a scope for all elements associated with it. Namespaces themselves also have names. A namespace name is a uniform resource identifier (URI). Such a URI serves as a unique string and need not be able to be dereferenced. The namespace name and the local name of the element together form a globally unique name known as a ***qualified name*.**

Namespace declarations appear inside an element start tag and are used to map a namespace name to another, typically shorter, string known as a ***namespace prefix***. The syntax for a namespace declaration is **xmlns:prefix='URI'**. It is also possible to map a namespace name to no prefix using a default namespace declaration. The syntax for a **default namespace declaration** is **xmlns='URI'**. In both cases, the URI may appear in single quotes (') or double quotes ("). Only one default namespace declaration may appear on an element. Any number of nondefault namespace declarations may appear on an element, provided they all have different prefix parts.

Examples:

Qualified and unqualified elements

```
<pre:Person xmlns:pre='urn:example-org:People' >
  <name>Martin</name>
  <age>33</age>
</pre:Person>
```

Qualified and unqualified elements using a default namespace declaration

```
<Person xmlns='urn:example-org:People' >
  <name xmlns=''>Martin</name>
  <age xmlns=''>33</age>
</Person>
```

Qualified elements

```
<pre:Person xmlns:pre='urn:example-org:People' >
  <pre:name>Martin</pre:name>
  <pre:age>33</pre:age>
</pre:Person>
```

Qualified elements using a default namespace declaration

```
<Person xmlns='urn:example-org:People' >
  <name>Martin</name>
  <age>33</age>
</Person>
```

1.4.3 Attributes

```
name='value'
name="value"
```

Elements can be annotated with attributes. **Attributes can be used to encode actual data or to provide metadata about an element**—that is, provide extra information about the content of the element on which they appear. Attributes appear as name/value pairs separated by an equal sign (=). Attribute names have the same construction rules as element names. Attribute values are textual in nature and must appear either in single quotes or double quotes. An element may have any number of attributes, but they must all have different names. **Unprefixed attributes** are not in any namespace even if a default namespace declaration is in scope.

1.4.4 Comments

```
<!-- comment text -->
```

XML supports comments that are used to provide information to humans about the actual XML content. They are not used to encode actual data. Comments can appear anywhere in the document. The character sequence -- may not appear inside a comment. Other markup characters such as less than, greater than, and ampersand (&),may appear inside comments but are not treated as markup.

1.4.5 The XML declaration

```
<?xml version='1.0' encoding='character encoding'
standalone='yes|no'?>
```

XML documents can contain an XML declaration that if present, must be the first construct in the document. An XML declaration is made up of as many as **three name/value pairs,** syntactically identical to attributes. The three attributes are a **mandatory version attribute and optional encoding and standalone attributes**. The order of these attributes within an XML declaration is fixed.

All XML declarations have a version attribute with a value that must be 1.0. XML documents are inherently Unicode, even when stored in a non-Unicode character encoding. The XML recommendation defines several possible values for the encoding attribute. If an XML document can be read with no reference to external sources, it is said to be a *stand-alone document*. Such documents can be annotated with a standalone attribute with a value of yes in the XML declaration.

1.4.6 Well-formed and Valid XML

All XML must be **well formed.** A well-formed XML document is one in which, in addition to all the constructs being syntactically correct, **there is exactly one toplevel element, all open tags have a corresponding close tag** or use the empty element shorthand syntax, and **all tags are correctly nested** (that is, close tags do not overlap). In addition, all the attributes of an element must have different names. If attributes are namespace qualified then the combination of namespace name and local name must be different. Similarly, all the namespace declarations of an element must be for different prefixes. All namespace prefixes used must have a corresponding namespace declaration that is in scope.

A **valid XML document** is one that conforms to a DTD (Document Type Definition) or an XML Schema. That is, the type and contents of all the elements are in conformance with the corresponding DTD or schema.

1.4.7 Why Is XML Important?

There are a number of reasons for XML's surging acceptance. This section lists a few of the most prominent.

1. Plain Text

Since XML is not a binary format, you can create and edit files with anything from a standard text editor to a visual development environment. That makes it easy to debug your programs, and makes it useful for storing small amounts of data. At the other end of the spectrum, an XML front end to a database makes it possible to efficiently store large amounts of XML data as well. So XML provides **scalability** for anything from small configuration files to a company-wide data repository.

2. Data Identification

XML tells you what kind of data you have, not how to display it. Because the markup tags identify the information and break up the data into parts, an email program can process it, a search program can look for messages sent to particular people, and an address book can extract the address information from the rest of the message. In short, because the different parts of the information have been identified, they can be used in different ways by different applications.

3. Easily Processed

As mentioned earlier, regular and consistent notation makes it easier to build a program to process XML data. For example, in HTML a <dt> tag can be delimited by </dt>, another <dt>, <dd>, or </dl>. That makes for some difficult programming. But in XML, the <dt> tag must always have a </dt> terminator, or else it will be defined as a <dt/> tag. (Otherwise, the XML parser won't be able to read the data.) And since XML is a **vendor-neutral standard,** you can choose among **several XML parsers,** any one of which takes the work out of processing XML data.

4. Hierarchical

Finally, XML documents benefit from their hierarchical structure. Hierarchical document structures are, in general**, faster to access** because you can drill down to the part you need, like stepping through a table of contents.

1.5 XHTML

The Extensible HyperText Markup Language (XHTML) is a family of current and future document types and modules that reproduce, subset, and extend HTML, reformulated in XML. XHTML Family document types are all XML-based, and ultimately are designed to work in conjunction with XML-based user agents. XHTML is the successor of HTML.

1.6 Limitations of HTML Forms, Advantages of XForms

1. According to developers, the most commonly cited problem with HTML forms is their **dependency on scripting languages**. Real-world HTML forms are reliant on script to accomplish many common tasks such as marking controls as required, performing validations and calculations, displaying error messages, and managing dynamic layout. This dependency results in complex documents, which are expensive and time-consuming to maintain, since a full-time programmer is practically necessary when dealing with that much script.
   XForms helps reduce the need for script in several ways: by defining a framework for simple, XPath-based calculations and validations, by providing better user feedback on the status of the form, through dynamic features such as repeating tables and optional sections, and through a system of *XForms Actions*—elements that cause commonly needed actions such as setting focus or changing a data value.

2. A second limitation of HTML forms is the **difficulty of initializing form data**, as commonly happens when web sites "remember" past users and provide them the courtesy of not having to repeatedly enter information. As shown earlier, each form control has its own unique way of defining initial data, with small bits of initialization data spread all across the document. This means that in order to process a blank form into a filled form, either a new document needs to be constructed piece by piece, or an existing document needs to be patched in numerous places—one of the reasons why template-replacement facilities are commonly found in application servers. Constructing such forms is CPU-intensive and leads to bottlenecks on high-volume servers.

In XForms, form data is provided from an initial XML file, which can be external to the form definition. Since XForms is also flexible enough to deal directly with most XML data formats, piping initial data into a form is typically a simple matter of pointing a `src` attribute to an existing XML data source.

3. A third limitation of HTML forms is that the **encoding formats**, urlencoded or multipart, represent only "flat" data, or name/value pairs. Many types of forms, including purchase orders, would benefit from a richer data representation.

   XML is a better foundation for most business documents than a flattened set of names and values. Since it has XML support as a fundamental requirement, XForms excels at helping users create those kinds of documents.

4. More subtle, but still serious, is a fourth fundamental design flaw in HTML forms: **a hidden assumption of a one-step process—from a client to a server**—with processing finishing there. In the real world, forms often travel in more complicated paths. For example, a vacation request form might go to a supervisor for approval, then to the human resources department, and finally to accounting for final processing. Managing HTML forms in such a workflow scenario involves reinterpreting the data format at every stage. Perhaps this is one reason why HTML forms aren't commonly seen in use for workflow.
   XForms enables a different pattern: it allows form data, as an XML file, to be routed to various workstations, as needed. At each stop, the data is loaded into a form, which provides a viewport into editing all or parts of the document, and submitted again. This process can be repeated as many times as necessary, with any number of participants.

1.7 OBJECTIVE

The objective of this thesis is to

- Design a new simplified language which gives the functionality of XForms + XFDL.

- The new language provides shorter and simpler code as compared to XForms + XFDL.

- Design a converter that converts the simplified code into XForms + XFDL code which can be displayed using the IBM Workplace Forms Viewer.

2. XFORMS

2.1 Introduction

Web applications and electronic commerce solutions have sparked the demand for better Web forms with richer interactions. XForms is the response to this demand, and provides a new platform-independent markup language for online interaction between a person and another, usually remote, agent. XForms are the successor to HTML forms, and benefit from the lessons learned from HTML forms.

"XForms" is W3C's name for a specification of Web forms that can be used with a wide variety of platforms including desktop computers, hand helds, information appliances, and even paper. XForms started life as a subgroup of the HTML Working Group, but has now been spun off as an independent Activity.

Traditional HTML Web forms don't separate **the *purpose* from the *presentation*** of a form. XForms, in contrast, are comprised of separate sections that describe what the form does, and how the form looks. This allows for flexible presentation options, including classic XHTML forms, to be attached to an XML form definition.

XForms is an XML application that represents the next generation of forms for the Web. By splitting traditional XHTML forms into **three parts—XForms model, instance data, and user interface**—it separates presentation from content, allows reuse, gives strong typing—reducing the number of round-trips to the server, as well as offering device independence and a reduced need for scripting.

The **XForms User Interface** provides a standard set of visual controls that are targeted toward replacing today's XHTML form controls. These form controls are directly usable inside XHTML and other XML documents, like SVG. Other groups, such as the Voice

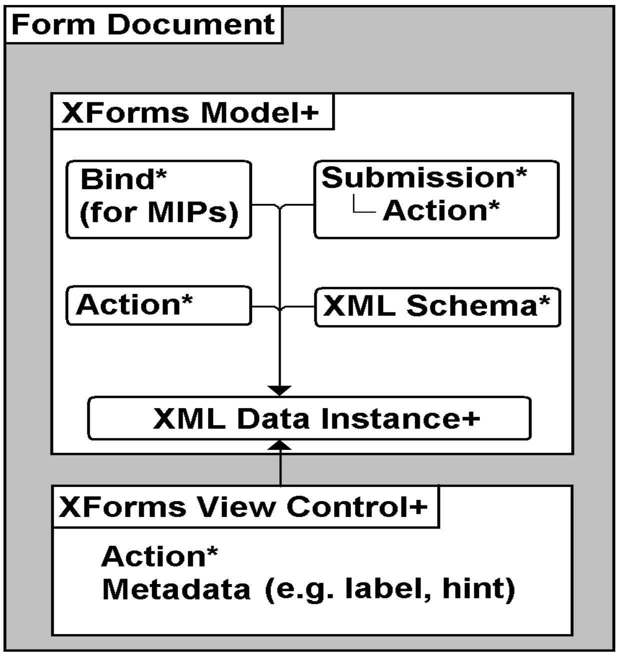


Fig. 2.1 Internal Architecture of XForms

Browser Working Group, may also independently develop user interface components for XForms.

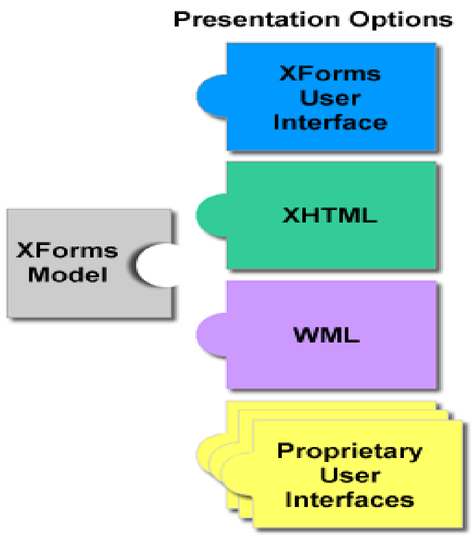


Fig. 2.2 XForms Presentation Options

An important concept in XForms is that forms collect data, which is expressed as XML **instance data**. Among other duties, the XForms Model describes the structure of the instance data. This is important, since like XML, forms represent a structured interchange of data. Workflow, auto-fill, and pre-fill form applications are supported through the use of instance data.

Finally, there needs to be a channel for instance data to flow to and from the XForms Processor. For this, the **XForms Submit Protocol** defines how XForms send and receive data, including the ability to suspend and resume the completion of a form.

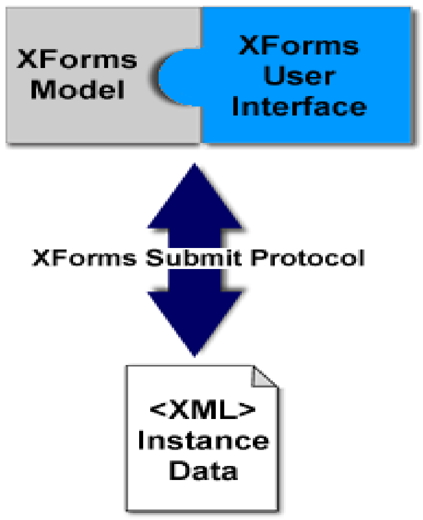


Fig . 2.3 XForms Submission

2.2 Goals of XForms

- Support for handheld, television, and desktop browsers, plus printers and scanners
- Richer user interface to meet the needs of business, consumer and device control applications
- Decoupled data, logic and presentation
- Improved internationalization
- Support for structured form data
- Advanced forms logic
- Multiple forms per page, and pages per form
- Suspend and Resume support
- Seamless integration with other XML tag sets

2.3 The XForms Model

XForms Model is the name given to the form description. That name was chosen mainly because it wasn't "data model," but also to evoke thoughts of the **Model-View-Controller (MVC) design pattern** in programming. In MVC, a model contains all the essential data, and one or more views provide a viewpoint to examine or interact with the data. The XForms Model is analogous to a MVC model, and form controls serve the function of views. (There's nothing that directly maps to a controller in XForms, though portions of the processing model and XForms Events play a similar role.)

A ***model item*** is the name for an XPath node with the addition of certain XForms properties, formally called *model item properties*. The connection between model item properties and form controls is called *binding*, which is accomplished through a set of XML elements that comprise the XForms Model.

2.3.1 Structural Elements

The XForms Model is made up of a number of different elements, outlined here.

1. **The model Element -** This element is the local root of the definition of the XForms Model. It is typically found in a non-rendered area of the containing document. **Eg -**the `head` section in XHTML can contain an XForms Model.

2. **The instance Element -** This element serves as a container for initial instance data. The contents of this element are simply data that will be both read and written during form interaction, nothing more.
   Instead of inline content, `instance` may use Linking Attributes (that is to say, `src`) to point to external instance data.

3. **The bind Element -** This element establishes conditions that are continuously applied to the instance data. With instance data defined neatly by the `instance` element, the question remains of how to annotate instance data nodes with properties necessary for forms. Each model item property is represented by an attribute on this element- `type`**,** `readonly`**,** `Required`, `relevant`, `calculate`, `constraint`, `p3ptype`.
   The properties are applied through an additional attribute, `nodeset`, which selects a node-set.

4. **The submission Element –** Specifies how,where and what data is submitted.

2.3.2 Common Attributes

The Common attribute collection contains the Binding Attributes—Single Node and Node-set. A number of situations in XForms call for a reference into instance data.

Binding attributes provide this feature. The following section describes these attributes :

- **ref** - Whenever the intent of the binding attributes is to select a single node, the `ref` attribute will be present. It contains an XPath path expression. In cases where the selected node-set happens to have more than one node, the *first node rule* applies, which removes all nodes other than the first, according to the order the nodes appear in the document.

- **nodeset** - Whenever the intent of the binding attributes is to select a node-set of any size, the `nodeset` attribute will be present. It contains an XPath path expression.

- **model** - In larger or more complex documents, it will be common to have multiple XForms Models. When this is the case, an additional attribute is needed to indicate to which XForms Model the binding attaches. The value of this attribute is of type IDREF, and so a `model` element in the same document must have an attribute of type ID with a matching value.

- **bind** - In some cases, such as when a graphic design professional who isn't concerned with XPath is laying out a form, it isn't desirable to have XPath strewn about on every set of binding attributes. The `bind` attribute, which takes precedence over any of `ref`, `nodeset`, or `model`, refers back to an already-defined node-set on a `bind` element. The value of this attribute is of type IDREF, and so a `bind` element in the same document must have an attribute of type ID with a matching value.

It's worth noting that the term binding, as used in XForms, can refer to two separate things. ***UI Binding*** occurs on an attribute of a form control element, and binds the form control to a particular model item. In dynamic forms, the association to a model item can jump around, causing the form control to be a window to different parts of the data at different times. The other use of binding, ***Model Binding***, occurs on the element `bind`, selecting an entire node-set to which a set of model item properties gets applied. It is a serious problem to have a dynamic model binding expression, since that complicates life behind-the-scenes for an XForms Processor, which can cause difficult-to-detect errors.

2.3.3 Model Item Properties

An individual property that can be applied to a node is called a model item property. Some of the properties are XPath expressions (called computed expressions in the specification), which the XForms Processor tracks and reevaluates as necessary:

- **Readonly** – This property signals whether a node is read-only, in which case form controls attached to the node won't allow the user to change data.
- **Required –** This property signals whether a value is required in this node for the form to be considered valid. A node satisfies the required condition when it is convertible into a string with one or more characters.
- **Relevant –** This property signals whether a node is currently relevant to the form. Form controls bound to non-relevant nodes are either disabled or completely invisible. Non-relevant nodes are not even submitted with the rest of the data.
- **Calculate -** This property defines a calculation used to determine the value of the node.
- **Constraint -** This property imposes an additional XPath-based constraint on the validity of the attached node.

The remaining properties are **static**, and don't get reevaluated:

- **type** – This property associates an XML Schema datatype with an instance data node. The unusual thing about this property is that it's technically unnecessary. The right XML Schema incantations can accomplish the same result. In many cases, however, using this model item property is more convenient than using XML Schema features.

- **p3ptype -** This property associates a P3P datatype identifier with a node.The **Platform for Privacy Preferences (P3P)** is a W3C specification that describes a machine-readable profile of what personally identifiable information is collected by a web site—especially forms. The main use of this property is to give P3P-compliant browsers enough information, at a granular enough level, to offer users flexible choices in how much personal information they give out, and to whom. Another use of this property is as a key for autocomplete features.

Some of the model item properties have an effect on child nodes as well. The **rules** for this behavior can be summarized like this:

- Setting a node to `readonly` sets all child nodes to readonly, unless specifically overridden.
- Setting a node to non-`relevant` sets all child nodes to non-relevant, unless specifically overridden.
- For all other model item properties, setting that property on a node has no effect on child nodes.

2.3.4 Making the Connection—Binding

A bind has two ends, one side in the XForms Model, and the other side at a form control. On the bind element within the XForms Model, the nodeset attribute holds the **Model Binding Expression**. On the other end, in the user interface, is the **UI Binding Expression.** This end may be bound two ways, using either IDREFs or XPath.

1. **With IDREFs** - The recommended way to perform binding is to put an `id` attribute on each `bind` element, and refer back to this with a `bind` attribute on each form control:

```
   <!-- in the XForms Model -->
      <xforms:bind nodeset="email" id="mybind"
      required="true()"/>
        ...
  <!-- later in the document -->
      <xforms:input bind="mybind"...>
```

This approach is distinguished by the use of the **`bind` attribute on form controls**. The main advantage of this approach is that it maintains separation between the model and the view. If the structure of the instance data were to change, only the attributes on the `bind` elements would need to be updated.

2. **With XPath** - Another way to bind is with XPath expressions on the form controls:

```
<!-- in the XForms Model -->
   <xforms:bind nodeset="email" id="mybind" required="true( )"/>
     ...
<!-- later in the document -->
   <xforms:input ref="email"...>
```

This approach is distinguished by the use of **`ref` attributes** on form controls. Many view this approach as simpler, since it cuts out one level of indirection. It is also more fragile, however, since the XPath expressions to locate nodes appear in two places. If the structure of the instance data were to change, both the attributes on the `bind` element and the `ref` attributes on the form controls would need to change.

2.3.5 Multiple Models

It's common to have multiple forms in the same document, and thus have multiple XForms Models. The document markup for this is straightforward:

```
<!-- in the XForms Model 1 -->
<xform:model id="m1">
    <xforms:bind nodeset="email" type="my:email"/>
  ...
</xforms:model>
<xforms:model id="m2">
    <xforms:bind nodeset="search" type="my:query"/>
  ...
</xforms:model>

<!-- later in the document -->
<xforms:input ref="email" model="m1"...>
<xforms:input ref="search" model="m2"...>
```

When using IDREF binding, this causes no additional problems, since the ID the form control points to is necessarily unique in the document. When using XPath binding, however, additional information is needed. In any document with two or more XForms Models, every XPath-style binding needs an additional attribute, `model`, to indicate which model is being bound to. By design, each XForms Model is a self-contained unit, and options for cross-model communication are limited.

2.3.6 Multiple Instances

A common scenario is that a form needs some extra data, perhaps for a calculation. In HTML forms, hidden fields could be used for this. But in XForms, the initial form data is XML, which is already widely deployed. Often, it's not possible to modify existing DTDs and XML Schemas to add new forms-specific elements and attributes to legacy XML. In these cases, it is possible to set aside additional XForms Instances as temporary storage.

On the markup side, this, too, is straightforward—using multiple `instance` elements:

```
<!-- in the XForms Model -->
<xforms:model>
   <xforms:instance id="formdata">
      <my:root>
        ...
      </my:root>
   </xforms:instance>
   <xforms:instance id="userid" src="scripts/getuserid"/>
      ...
   <xforms:bind nodeset="my:root/..."/>
   <xforms:bind nodeset="instance('userid')/..."/>
  ...
</xforms:model>
```

A similar problem to having multiple models occurs when you try to write a XPath expression that reaches across instances. By default, XPath expressions will always point into the first instance. The function `instance( )`, which takes an IDREF of an instance element, resets the XPath context to a different instance (but always within the same XForms Model).

2.4 XForms User Interface

2.4.1 Form controls

They are windows onto the form data kept in the XForms Model. In principle, this was true also for HTML forms, although the design of XForms makes a much sharper separation. The following sections describe the form controls included in XForms.

- **input** - This form control is quite similar to its HTML forms counterpart, as it permits the entry of any character data. There are some significant improvements, however, such as the ability to use an XML Schema datatype to optimize the user experience of entering the data.eg - date control can be entered through a calendar interface.

  ```
  <input ref="date">
  <!-- bound to node with XML Schema type xs:date   -->
  <label>Ship By:</label>
  </input>
  ```

- **secret -** This form control is nearly identical to its HTML forms counterpart. It offers only a cursory level of security, since the collected data isn't encrypted in any way, merely obscured for presentation. For end-to-end security, additional measures such as SSL are necessary.

- **output -** This is the only form control that doesn't accept user input. `Output` renders data from an XForms Model as inline text, normally indistinguishable from other text on the page.

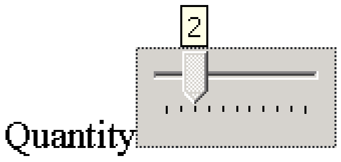
```
 <output ref="/my:employee/my:name">
  <label>Name:</label>
</output>
```

- **upload** - HTML forms had a limited file upload control, but the XForms version surpasses it in many ways.

```
<upload bind="attachment1">
   <label>Select a file</label>
   <filename bind="fname1"/>
   <mediatype bind="mt1"/>
</upload>
```

- **range** - This form control wasn't present in HTML forms. It provides an intuitive way to enter a bounded value. The upper and lower bounds are set by the attributes start and end, respectively, and the suggested interval by the attribute step.

```
<range start="0" end="10" step="1" ref="quan" model="po">
   <label>Quantity</label>
</range>
```



- **trigger** - This form control is similar to the HTML element `button` and in fact was called that in earlier XForms drafts. The final name emphasizes that this form control really is a trigger for XForms Actions—a push button is just one possible rendering. Other possibilities include images, hyperlinks, mouse gestures, and voice activation. The rendering of this form control is often similar to the submit element.

```
<trigger>
   <label>Login</label>
   ...
</trigger>
```

- **submit** - This form control is a specialization of trigger, with the effect of submitting the form. The submit parameters are taken from the element that matches the IDREF specified on the attribute submission.

```
<submit submission="formdata">
   <label>Buy</label>
</submit>
```

- **select1 -** This form control represents selection from a list with the intent of enforcing the selection of exactly one item.Any control that expresses the goal of picking things from a list, including conventional radio buttons and checkboxes fall into this category.

  Some graphic designers are unnerved about the generality of this form control, and would prefer to explicitly indicate that they want, say, checkboxes instead of a drop-down list. This is what the appearance attribute is all about.

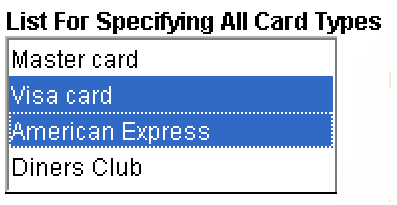
  appearance="full" - To always render all of the choices, a list of checkboxes or radiobuttons is used.

  appearance="compact" - To render a more compact list, a listbox that can have scroll bars to limit itself to a particular size, is used.

  appearance="minimal" - To render a minimal list, as little as a single item is shown, with additional choices appearing, like a drop-down menu, upon request.

- **select** - This form control represents selection from a list with the intent of allowing nothing, one thing, or multiple things to be selected. It shares many common features with select1.

```
<select ref="cctype">
   <label>List For Specifying All Card Types</label>
   <item>
         <label>Master Card</label>
         <value>MC</value>
   </item>
   <item>
         <label>Visa Card</label>
         <value>VI</value>
   </item>
    <item>
         <label>American Express</label>
         <value>AE</value>
   </item>
   <item>
         <label>Diners Club</label>
         <value>DC</value>
   </item>
</select>
```



- **help, hint, and alert** - In XForms, every form control can have a **`help` element**, which contains a message that's provided upon an explicit request (such as pushing the Help or F1 key). The help message is delivered in a way that is equivalent to a modeless message. Likewise, form controls can have a **`hint` element**, which contains a message that's shown at the discretion of the XForms Processor, for instance, if the user hovers the mouse over a form control for more than a given amount of time.

  A third kind of element, **`alert`**, contains a message to be shown to the user when an error condition (like a form control failing validation) happens.
  Like `label`, these elements can also get their contents from an external source (e.g., through the `src` attribute), or from the instance data (e.g., through `ref` and the other single-node binding attributes).

2.4.2 Grouping

Groups make possible a number of conveniences, but also have a functional aspect. In nearly every respect, a group is another kind of form control, and thus model item properties such as `relevant` and `required` can apply to a group, and override those properties on any contained form controls. This is most useful when an entire section of a form needs to change based on some condition in the instance data.Groups can also help authors with a couple of shortcuts:

- The group element can be a convenient place to declare the **XForms Namespace** as the default namespace, to reduce the clutter of repeated prefixes or additional declarations.

- **Binding attributes** can be declared, to set a new context node for any binding expressions that occur within the group

2.4.3 Dynamic Presentation

For years now, developers have used complicated scripting and other desperate measures to create "dynamic forms." For instance, one commercial product used HTML divs to represent individual "pages" of a multi-page form, with script to swap the current page. Using XForms, the same effect can be accomplished declaratively.

switch and case

The switch element is a container for case elements, usually two or more.At any given time, the contents of exactly one of the cases will be rendered in the final document, and the rest of the cases will be suppressed.One use of this is to provide tabbed interfaces.

```
<switch>
  <case id="default">Not Initialized</case>
  <case id="ready">Initialized</case>
</switch>
… <toggle ev:event="xforms-ready" case="ready" />
```

More generally, switch is useful for simulating pages, showing and hiding portions of the form, and enhancing the usability of forms by suppressing parts that don't matter at a given moment.

Each case element has a selected attribute, defaulted if necessary, that is visible to the host document, including DOM and CSS interfaces. Additionally, xforms-select and xforms-deselect events are dispatched to the individual cases, allowing event handlers to respond in a centralized manner to changes.

The actual switch is accomplished by an XForms Action named toggle, which takes a parameter of an IDREF that refers to the particular case that will become active.

Repeating Line Items

One of the most sorely missed features in HTML forms comes by many names: tables, grid controls, or line items. The basic concept is that many forms in common use don't fit in well with a flat list of form controls. The primary means of accomplishing this in XForms is the **repeat element**. The nodeset attribute of repeat selects a number of nodes, and the contents of the element, both from XForms and from the host language, are effectively repeated once for each resulting node. One way to think of this is to "unroll" the repeat, so that the following:

```
<repeat nodeset="item">
  <input ref="@quantity" .../>
</repeat>
```

has a similar effect, when the nodeset returns three item nodes, to:

```
<input ref="item[1]/@quantity" .../>
<input ref="item[2]/@quantity" .../>
<input ref="item[3]/@quantity" .../>
```

2.5 XML Events

An ***event*** is the representation of some asynchronous occurrence (such as a mouse click on the presentation of the element, or an arithmetical error in the value of an attribute of the element, or any of unthinkably many other possibilities) that gets associated with an element (targeted at it) in an XML document.

In the DOM model of events, the general behavior is that when an event occurs it is *dispatched* by passing it down the document tree in a phase called ***capture*** to the element where the event occurred (called its *target*), where it then may be passed back up the tree again in the phase called ***bubbling***. In general an event can be responded to at any element in the path (an *observer*) in either phase by causing an action, and/or by stopping the event, and/or by cancelling the default action for the event.

An ***action*** is some way of responding to an event; a ***handler*** is some specification for such an action, for instance using scripting or some other method. A ***listener*** is a binding of such a handler to an event targeting some element in a document.

2.5.1 The Old Way

In the design of HTML forms, script is used whenever some specific action is needed. For example, a form might have a button that copies values from a "ship to" section onto a "bill to" section. In HTML forms plus script, the following code would accomplish this:

```
<script type="text/javascript"> <!--
function copyAddresses(  ) {
  var frm = document.forms[0];
  frm.shipAddr.value = frm.billAddr.value;
  frm.shipCity.value = frm.billCity.value;
  frm.shipProv.value = frm.billProv.value;
  frm.shipPostCode.value = frm.billPostCode.value;
} --> </script>
```

It would then be activated by a button, with an event-specific attribute, specified like this:

```
<input type="button" id="cp" value="Copy values"
onclick="copyAddresses( )"/>
```

In terms of DOM Level 2 Events, this represents a registration of an observer on the input element, watching for the DOM click event at the target, and handling the event by calling a short script. As a result, the script in the onclick attribute will get called when the user clicks on the button.

Disadvantages of this approach:

- A special **hardwired attribute**, in this case `onclick,` is needed. This is inflexible and clutters the language.
- **Script is difficult to maintain**, especially when bits of script are scattered throughout the document.
- This **won't work in browsers that don't support scripting**.

2.5.2 Declarative Actions, Displacing Script

XML Events help specify the same thing as above, declaratively, that is, without the use of scripting. Handlers can come from two sources. XForms defines a number of handlers, called **XForms Actions**, discussed below. Additionally, the host language can define handlers, as is the case with script. With XForms Actions, the earlier example can be done without any script at all, like this:

```
<trigger>
  <label>Copy values</label>
  <action ev:event="DOMActivate">
    <setvalue ref="Shipping/Addr" value="../Billing/Addr"/>
    <setvalue ref="Shipping/City" value="../Billing/City"/>
    <setvalue ref="Shipping/Prov" value="../Billing/Prov"/>
    <setvalue ref="Shiping/PostCode" value="../Billing/PostCode"/>
  </action>
</trigger>
```

2.5.3 XForms Actions

The following sections describe all of the XForms Actions defined in XForms. Any of the following can be invoked in such a way that the processing described for the element happens in response to a given event.

- **xforms:delete** Deletes a row of elements from a table. The elements are first deleted from the XForms model, then the table's repeat deletes the visible items that were linked to those data elements.

- **xforms:insert** Allows you to add a row of elements to a table. This function copies a row of elements in the data model, then inserts the copy in the desired location in the data model. Once the copy is inserted in the data model, the table's repeat creates corresponding items that are displayed to the user.

- **xforms:message** Sets a message that is displayed to the user in a small dialog box. 3 levels of messages are provided – modal, modeless and ephemeral.

- **xforms:rebuild** Causes the form viewing application to rebuild any internal data structures that are used to track computational dependencies within a particular model.
- **xforms:recalculate** Causes the forms viewing application to recalculate any instance data that is affected by computations and is not up-to-date. This affects all data instances in the designated model.

- **xforms:refresh** Causes the forms viewing application to update all user interface elements linked to a particular model, so that they match the underlying data in the XForms model.

- **xforms:revalidate** Causes the forms viewing application to validate all instance data in a particular model. This ensures that all validation checks have been performed. In general, the XForms processor automatically runs the above 4 actions when required.

- **xforms:reset** Returns a particular XForms model to the state it was in when the form was opened. This allows the user the reset the contents of the form to their ″starting point″, which can increase usability of the form.

- **xforms:send** Triggers an XForms submission. The submission must already be defined in the XForms model.

- **xforms:setfocus** Sets the focus to a particular presentation element in the form.

- **xforms:setindex** Sets the index for the *xforms:repeat* element in a table. This determines which row in the table receives the focus. Rows use one-based indexing. This means that the first row has an index of 1, the second and index of 2, and so on.

- **xforms:setvalue** Sets the value for a specified element in the data model.

- **xforms:toggle** Selects one of the cases in an *xforms:switch* and makes it active. When one case is selected, all other cases in the switch are deselected.

2.5.4 XForms Events

Events are one of **four major types**:

1. **lifecycle** - An event that deals with setting up or tearing down the XForms engine.
2. **notification -** An event indicating something took place.
3. **interaction -** An event that triggers some kind of processing. Cancelling the event (when that is possible) stops the default processing.
4. **error handling -** An event indicating an error or some other unusual situation occurred.

2.5.5 Stages of XForms Processing

The life-cycle of an XForms processor can be divided into several categories, which is a useful viewpoint for a form author. By scrutinizing what you want to accomplish, you can narrow down the potentially huge list of events to a more manageable list.

- **Initialization**

Obviously, the first step is to initialize all the machinery underlying XForms. The initialization process was extensively discussed, the main decision point being how much stuff gets initialized before versus after the event happens. To make everybody happy, no fewer than three separate initialization events were defined, some of which fire multiple times.

- **Interaction**

The major portion of XForms processing involves interacting with the user, to the end of producing an XML document. During interaction, **two different kinds of events are important**: events that provide a notification that something happened, and events that *cause* something to happen. Both kinds of events are important to form authors.

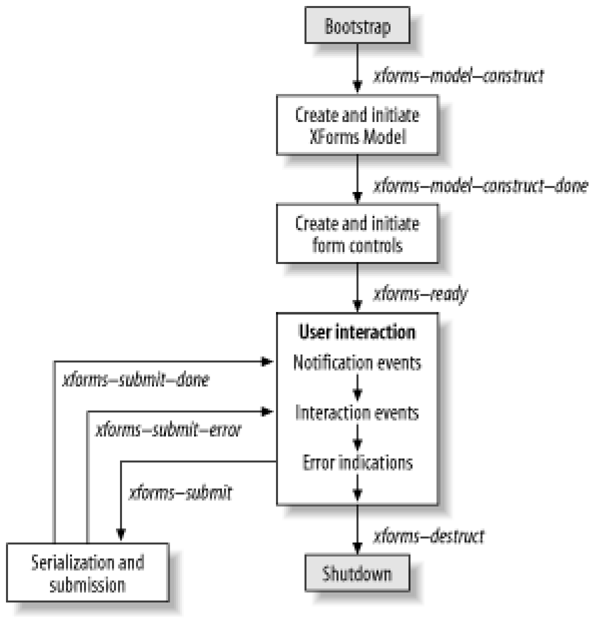


Fig. 2.4 Stages of XForms processing

- **Submit**

Sending the form data on its way is the goal of most forms, at least in theory.In practice, the submission attempt might not get very far, such as when the form contains invalid data. Other problems too, such as a crashed server, can prevent the submission from completing. For this reason, the submission process includes extra events that help the form author determine whether the submission was successful.

- **Deinitialization**

In HTML forms, submitting the form and loading a new page always happened at the same time. In XForms, however, the author has more granular control of the process. Thus, even though there is a single deinitialization event, it's still worth discussion of when it happens.

- **Error Condition**

Error processing is a tricky subject. In HTML forms, nearly any error is tolerated without generating error messages (but often at the expense of erratic or unpredictable behavior). In XForms, any number of problems can cause a fatal error, which will prevent a form from displaying properly.

2.6 Submit

The four main questions in submitting form data are ***when, what, where,* and *how***. The following sections discuss each of these questions.

2.6.1 When to Submit

Submit happens when the user presses the big button labeled "Submit," right? Formally, a submission is initiated **when an event called `xforms-submit`** arrives at the `submission` element. The reason for a separate event is so that submission can be requested in other situations, such as pressing Enter or meeting other conditions. The XForms Action `send` can explicitly send out the submission event, and the form control named `submit`, which otherwise behaves exactly like `trigger`, also dispatches an `xforms-submit` event.

2.6.2 What to Submit

XForms defines ways to reduce the amount of XML that gets selected for submission. One of the most powerful techniques for managing complexity is to **divide and conquer** using multiple XForms Models. Using **multiple `models`** can help but the strong separation between XForms Models can get in the way. Another option is to use **multiple `instances`**, which can use either inline or remote XML, within the same `model`. Once a particular instance document is selected, XForms still provides additional ways to prune the XML tree. One way is to specify a subtree of the instance data that gets submitted, using either the **`ref` attribute (XPath expression) or `bind` attribute (`IDREF` to a `bind` element).** Finally, any node that has a **`relevant` property** that evaluates to false will not be present in the submitted data.

2.6.3 Where and How to Submit

The questions of *where* and *how* are closely related, because the target **of submission is a URI**. The first part of a URI, called the ***scheme***, indicates the general approach for the submit transaction, as in "http," "file," or "mailto." The remainder of the URI gives more specific information on where the destination for the data is to be. Additionally, there need to be rules for how the in-memory instance data gets written down as a pattern of bytes on the wire.

2.6.4 The Submission Element

The submission element defines the parameters for serializing and submitting instance data. An XForms Model can contain any number of submission elements.Most of the action takes place in the **attributes** of this element.

- **ref and bind** - These attributes are functionally equivalent to binding attributes and are used to select a particular node that is selected, along with all its children.

- **action (required)** - This attribute holds a URI to which the submission will be directed.

- **method** - This required attribute provides additional information that, combined with the URI scheme in the action attribute, determines how the submission process will proceed. Possible values are "get", "put", "post", "form-data-post", "urlencoded-post".

- **replace** - This attribute describes what should happen with the response document (if any) that is returned from the submission action. Possible values are "all", "instance", "none".

- **separator** - This attribute, which applies only to urlencoded serialization, specifies the separator character to be used between name/value pairs. Possible values are "&" or ";". The default is ";".

- **mediatype** - This attribute selects the Internet media type to be associated with the serialized instance data, and has a default of application/xml.

3. XFDL

Extensible Forms Description Language (XFDL), developed by John Boyer and Tim Bray, is an application of XML that allows organizations to move their paper-based forms systems to the Internet while maintaining the necessary attributes of paper-based transaction records. More specifically, the features that paper forms offer (data structures, user interfaces, and transaction records), must be integrated into electronic commerce systems. XFDL was designed for implementation in business-to-business electronic commerce and intra-organizational information transactions.

XFDL replicates the legal requirements of Paper forms. These requirements are based on 3 key concepts : **security, non-repudiation, and auditability.** These concepts are based on a foundation of authorization, signer authentication, document authentication, and context preservation. Figure 3.1 illustrates the relationships between these concepts.

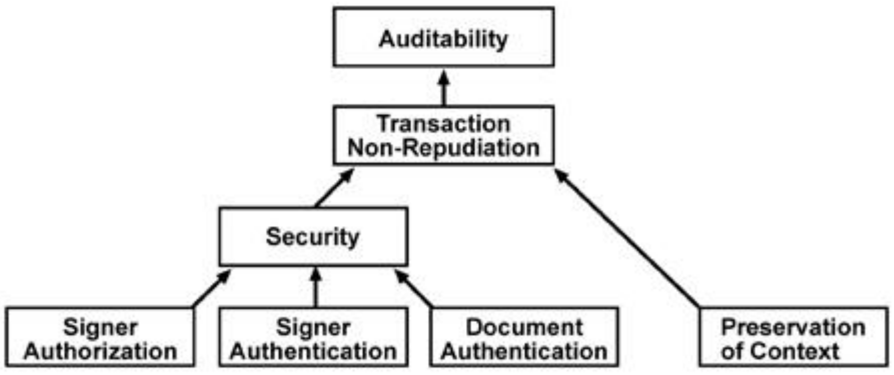


Fig. 3.1 Hierarchy of Transaction Record Validity

3.1 XFDL Features

The features of the language include support for a high-precision interface, fine-grained computations and integrated input validation, multiple overlapping digital signatures, and legally-binding transaction records.

1. XFDL is a Document-Centric Markup Language

XFDL takes a "document-centric" approach to digital forms by storing each necessary element of a viable transaction record in one securable file. It is the key feature in creating non-repudiable transaction records - the "message" must accurately record the full transaction context in order to achieve the essential purpose of digital signatures. Just like a paper form, the content, context, and structure of an XFDL form are all stored together. In the event of a dispute, an XFDL document can be recalled and used to prove the exact nature of a transaction. An organization can therefore create a "forms repository" of XFDL documents that provide legally-admissible transaction records for dispute reconciliation, business audits, chain-of-custody processes, and so on.

2. XFDL is a Computational Language

Unlike most XML languages, XFDL is also a programming language. XFDL is smart enough to make decisions, handle arithmetic, and respond to user input.
This computational power, or logic, is embedded into the XFDL document and becomes part of the transaction record that is digitally signed. Embedded logic also gives XFDL documents offline functionality, as an external connection to a script or some other source of programmatic intelligence is not required. This also makes XFDL documents useful for nomadic, or disconnected, data collection using portable computers and hand-held appliances.

3. XFDL is Assertion-Based

XFDL's computational logic is expressed using assertions. Unlike most traditional programming languages that function on a procedural mode, there is no thread of execution to be managed in XFDL. Creating computations in XFDL is much like using a spreadsheet. For example, to make Field1 the sum of Field2 and Field3, Field1 is "told" that this is the case. As the user interacts with the form, changing the values of Field2 and Field3, the XFDL language makes sure that the assertion is always true.

XFDL was engineered as an assertion-based language for **two reasons.** The first is readability. The type of logic that is normally used in complex forms is very easy to describe using assertions. The second is that it is very easy to freeze the exact state of a document when the user decides to save it, sign it, or send it to a server. Capturing the exact state of a program written in a procedural language would require the acquisition of a great deal of information, such as the heap, the call stack, the data and code segments, and so on.

4. XFDL is Human-Readable Plain Text

A core design tenet of both XML and XFDL is human readability. File formats for transaction records obviously need to be machine-readable, but can be more "future-proof" if they are also human-readable. XFDL is designed to create transaction records whose life span may be much longer than any particular Internet technology or vendor.

5. XFDL is a Publicly Accessible Open Standard Based on XML

XFDL derives several benefits from having a human readable XML syntax. Any document format, such as XFDL, which claims to be human readable must be a publicly accessible open standard. This is necessary to ensure that the semantics of the XML elements are known. As a natural consequence of this, organizations can share structured information gathered by XFDL documents with other organizations without a costly translation process. This is also valuable for organizations that want to share data among internal departments whose systems may be based on fundamentally different ontology.

6. XFDL is Extensible, Allowing Application-Specific/Server-Side Logic

Although the most important function on a non-repudiable transaction - the digital signature - occurs on the client machine, the full lifetime of a transaction can be quite complex and involve processing by modules other than a forms viewer. In a "fat-client" application design, these modules will run on the client machine, whereas the "thin-client" application architecture will have these modules on the server side. In either case, the ability to add custom functions to XFDL computes and, in general, to use XFDL's computation engine within custom items and options designed for modules other than a viewer is a key component in managing a transaction record's lifetime from a single source document.

7. Precision Layout for Dense Business and Government Forms

Unlike HTML and Java-based forms, XFDL documents allow precise recreation of the layout and appearance of paper forms on the Web. Although the acceptance of web technologies such as CSS and XSL promise to offer greater interface control, these technologies currently only address the need for precise presentation, not the need for viable transaction records. For instance, a style sheet cannot currently be inextricably bound to user-entered data.

8. Digital Signature Security and Flexibility

XFDL defines a technology-neutral digital signature interface that allows it to work with most common digital signature systems, such as RSA, DSS, biometric tokens, and so on. For cryptographic signatures, the signer's public key certificate is included with the encrypted hash. In addition to verifying the encrypted hash, the certificate authority signature on the signer's public key certificate is also verified using a certificate authority certificate on the verifying box. This simplifies the delivery of the signer's public key while preventing substitution attacks.

The digital signature hash value is not just computed on the entered data, as is the case with HTML forms. The XFDL elements that contain the content, context, and structure of the original form are "locked down" by the digital signature, in much the same way that a paper form is considered to be immutable once signed.

XFDL also supports **multiple overlapping signatures** that apply to different sections of a form. Many complex paper forms require multiple levels of input and approval. In this case, a multi-section form is used to gather the appropriate information and signatures. In practice, an XFDL form can be routed through an electronic workflow system, with each user adding the required information and digital signature until the document is complete.

3.2 The Structure of XFDL Forms

3.2.1 Top-Level Structure

An XFDL form is an XML 1.0 document whose root element tag is **XFDL**. This element must be in the XFDL namespace,

```
<XFDL xmlns="http://www.ibm.com/xmlns/prod/X 7.1"> ...</XFDL>
```

The XFDL element may contain many namespace attributes. By convention, the XFDL namespace is declared to be the default and it is also assigned to the prefix ’xfdl’.The XFDL element must contain a <globalpage> element as the first child element, followed by one or more <page> elements.

```
<!ELEMENT XFDL (globalpage, page+)>
```

The **<globalpage> element** must contain a single <global> element, which can contain zero or more option elements. These are referred to as form global options; they typically contain information applicable to the whole form or default settings for options appearing in the element content of pages. The <globalpage> and <global> elements must contain an attribute called *sid* which must be set to the value global.

```
<!ELEMENT XFDL (globalpage, page+)>
<!ELEMENT global (%options;*)>
<!ATTLIST globalpage sid CDATA #REQUIRED #FIXED "global">
```

A **<page> element** contains a <global> element followed by zero or more ’item’ elements. The options in the page’s global element typically contain information applicable to the whole page or default settings for options appearing within element content of items. The page global options take precedence over form global options. A page is also required to have a ’sid’ attribute, which provides an identifier that is unique among all <page> elements (sid is short for scope identifier).

```
<!ELEMENT page (global, %items;*)>
```

3.2.2 XFDL Items

An item is a single object in a page of a form. Some items represent GUI widgets, such as buttons, check boxes, popup lists, and text fields. Other items are used to carry information such as attached word processing documents or digital signatures.Each item must have a **sid attribute**, which provides a scope identifier that uniquely identifies the item from among all child items of its parent element.

An item can contain zero or more option elements. The options define the characteristics of the item. XForms user interface controls appear as options of XFDL items, and **the XFDL item is said to be the skin of the XForms form control** that it contains. XFDL allows elements in custom namespaces to appear at the item level (as long as they contain an xfdl:sid attribute. Following are the various XFDL Items :

- ***action* -** A non-visible item that can perform similar tasks to a button (print, cancel, submit, and so on) either after a certain period of time or with a regular frequency.
  Skin for: <xforms:submit>, <xforms:trigger>
- ***box* -** An item that provides a graphic effect used to visually group a set of the GUI widgets on the page. A box is drawn under all widgets on a page. This item is useful in some circumstances, but it is usually better to use a pane item (see below) to both visually and logically group related user interface elements.
- ***button* -** Performs one of a variety of tasks when pressed by the user, such as saving, printing, canceling, submitting, digitally signing the form, viewing documents enclosed in the form, and so on. A button can have a text or image face.
  Skin for: <xforms:submit>, <xforms:trigger>, <xforms:upload>
- ***check* -** Defines a single check box.
  Skin for: <xforms:input>
- ***checkgroup* -** Defines a group of checkboxes that operate together to provide a multiselection capability.
  Skin for: <xforms:select>, <xforms:select1>
- ***combobox* -** An edit field combined with a popup list; its value can be either selected or typed.
  Skin for: <xforms:select1> (select or type input), <xforms:input> (date selector)
- ***data* -** Used to carry binary information using base-64 encoding and compression, such as enclosed files or digital images, using base-64 encoding. This item appears when advanced XFDL enclosure mechanisms are used. When a basic <xforms:upload> is used, the data appears in an <xforms:instance> data node.
- ***field* -** Used to capture single- or multiple-line textual input from the user; it includes input validation and formatting features as well as enriched text capabilities.
  Skin for: <xforms:input> (single-line text), <xforms:secret> (single-line, write-only), <xforms:textarea> (for multiline plain text or enriched text)
- ***label* -** Shows either an image or a single or multiple line text value.

Skin for: <xforms:output>

- ***line-*** A simple graphic effect used as a separator.
- ***list -*** Shows a list box populated with choices from which the user may select one.
  Skin for: <xforms:select>, <xforms:select1>
- ***pane -*** Provides an hierarchic grouping capability for other items that are defined within the content of the pane. Also, may provide the ability to switch between multiple groupings.
  Skin for: <xforms:group>, <xforms:switch>
- ***popup -*** Shows either the text of the currently selected choice or a label if there is no selection; the popup provides a small button that causes the list of selectable choices to appear, from which the user may select one.
  Skin for: <xforms:select1>
- ***radiogroup -*** Defines a group of radio buttons. Initially none may be selected, but a maximum of one radio button can be selected within the group.
- ***signature -*** Receives the signature that ultimately results when a user presses a signature button.
  Skin for: <xforms:select1>
- ***slider -*** Creates a sliding control, similar to a volume control, that lets the user set a value within a specific range.
  Skin for: <xforms:range>
- ***spacer -*** An invisible GUI widget that facilitates spacing in the relational positioning scheme.
- ***table -*** Provides a template of XFDL items that are to be duplicated according to the amount of data available to be displayed. This item provides the ability to dynamically adjust the form rendition based on the amount of data and the amount of changes to that data.
  Skin for: <xforms:repeat>
- ***toolbar -*** Items associated with a toolbar item appear in a separate window pane above the pane for the form page; it is the typical location for page switching and other buttons as its contents are not printed if the form is rendered on paper.

3.2.3 XFDL Options

An option defines a named property of an item, page, or form. Options can appear as form globals, page globals, or as the contents of items.

- ***acclabel* -** Provides a special description of input items that is read by screen reading software.
- ***active* -** Specifies whether an item is active or inactive. In XFDL items containing XForms controls, the default for this option is set by the relevant model item property.
- ***bgcolor, fontcolor, labelbgcolor, and labelfontcolor* -** Specify the colors for an item or its label using either predefined names or RGB triplets in decimal or hexadecimal notation.
- ***border and labelborder* -** Control whether an item or its label has a border.
- ***colorinfo* -** Records the colors used to draw the form when the user signs the form.This is only necessary when the operating system colors are used instead of the colors defined in the form (which is a feature for users with vision impairments).
- ***coordinates* -** Receives the location of a mouse click on an image, if the image is in a button.
- ***cursortype* -** Displays different cursor icons when the user hovers over a button.
- ***data and datagroup* -** Used to create an association between data items and the buttons that provide file enclosure functionality.
- ***delay* -** Used in an action item to specify the timing for the event and whether it should be repeated.
- ***excludedmetadata* -** Used to store special information that is automatically excluded from signatures.
- ***filename and mimetype* -** Give additional information about an enclosed document.
- ***fontinfo and labelfontinfo* -** Defines the typeface, point size, and special effects (bold, italics, and underline) for the font used to display the item's value or label.
- ***format* -** Contains sub-elements that parameterize input validation for the item's value.
- ***formid* -** Defines a unique identifier for the form, such as a serial number.

- ***image* -** Identifies the data item containing the image for the button or label.
- ***imagemode* -** Specifies the display behavior of the image within the data item; the image may be clipped, resized, or scaled to fit the item.
- ***itemlocation, size and thickness* -** Help to define the location and size of the item.
- ***justify* -** Controls whether text in the item should be left, center, or right justified.
- ***label* -** Associates a simple text label with the item; labels can also be created independently with a label item.
- ***linespacing* -** Adjusts the spacing between lines of text in an item.
- ***mimedata* -** Used to store large binary data blocks encoded in bas-64 gzip compressed or base-64 format.
- ***next and previous* -** Link the item into the tab order of the page.
- ***padding* -** Defines how much extra whitespace is put around the pane item.
- ***pageid* -** Defines a unique identifier for a page, such as a serial number.
- ***printbgcolor, printlabelbgcolor, printfontcolor, and printlabelfontcolor* -** Provide the ability to set printing colors for each indicated option different from the display colors on the screen.
- ***printvisible* -** Determines whether an item should be visible when the form is printed. Has no effect on the visibility of the item on the screen.
- ***Printsettings* -** Parameterizes the paper rendition of a form.
- ***readonly* -** Sets the item to be readonly. In XFDL items containing XForms controls, the default for this option is set by the readonly model item property.
- ***rowpadding* -** Defines how much space is applied to the top and bottom of a table row.
- ***rtf* -** Contains the rich text value of rich text fields.
- ***requirements* -** Specifies the requirements for the Web Services to be used by the form.
- ***saveformat and transmitformat* -** Control how the form is written (XFDL, HTML) when it is saved or submitted.
- ***scrollhoriz and scrollvert* -** Control whether a text field item has horizontal and vertical scroll bars or whether it wordwraps, allows vertical sliding, and so on.
- ***texttype* -** Sets whether a field contains plain text or rich text.

- ***transmitdatagoups, transmitformat, transmitgroups, transmititiemrefs, transmititems, transmitnamespaces, transmitoptionrefs, transmitoptions, and transmitpagerefs*** **-** Work together to allow you to transmit form submissions.
- ***triggeritem*** **-** Set in the form globals to identify which action, button, or cell activated a form transmission or cancellation.
- ***type*** **-** Specifies whether the action, button, or cell item will perform a network operation, print, save, digitally sign, and so on.
- ***url*** **-** Provides the address for a page switch, or for a network link or submission.
- ***value*** **-** Holds the primary text associated with the item. In XFDL items that contain XForms controls, this option (and all options, such as those that are computed) are treated as transient, which means that any updates to the content are not serialized when the form is written because the updates are reflected in instance data.
- ***visible*** **-** Determines whether the item should be shown to the user or made invisible.
- ***webservices*** **-** Defines the nameof the Web Services used by the form.
- ***writeonly*** **-** Sets the item to be writeonly. This option is only for use with field items that do not skin XForms controls.

3.2.4 Implicit Options

There are some options that are defined within XFDL for the purpose of allowing them to be referenced without being defined by the form author. These options are dynamically added to the document object model (DOM) of the XFDL form while it is being processed, and they are removed when it is serialized. These options tend to be informational in nature or representative of events that can occur while the form is being processed.

- ***activated, focused, and mouseover*** **-** Indicates whether the form, page or item has been activated or focused or contains the mouse pointer.
- ***dirtyflag*** **-** In the form global item, this option indicates whether the end-user of the form viewing program has changed the form.
- ***focuseditem*** **-** At the page global level, records the scope identifier of the item that currently has the focus.

- ***itemprevious, itemnext, itemfirst, itemlast* -** Used to help create a doubly linked list of items in each page. The *itemprevious* and *itemnext* options occur in each item, and itemfirst and itemlast appear at the page global level.
- ***keypress* -** Records a keypress by the user that was not used as input to an XFDL item. The keypress is propagated upwards to the page and form global items.
- ***pageprevious, pagenext, pagefirst, pagelast* -** Used to help create a doubly linked list of pages in the form. The *pageprevious* and *pagenext* options occur in each page, and *pagefirst* and *pagelast* appear at the form global level.
- ***printing* -** In the form global item, this option indicates whether the form is currently printing.
- ***version* -** Appears in the form global item and defines the version of XFDL used to write the form. It is obtained from the XFDL namespace declaration.

3.3 Signatures in XFDL

Signatures are often used to sign only a portion of a document. Furthermore, a secondary signature is often used to sign the rest of the document while also endorsing the first part of the document. The classic example of this is the “For Office Use Only” section in any form. The implementation of digital signatures in XFDL must support scenarios like this, allowing both for filtering of what is signed and for overlapping signatures.

Furthermore, while digital signatures clearly identify the user, the application of digital signatures must also add a measure of security to the document itself. That is, once a document is signed, it should be impossible to change any of the information that was signed.

3.3.1 Applying Signature Filters

XFDL supports a **filtering system** for signatures. In effect, this allows any combination of form elements to be either included or excluded from a signature, which in turn allows forms to be divided into logical sections for the purposes of signing.

XFDL includes a series of signature filters. Each filter applies to a different cross-section of XFDL elements. For example, the *signitems* and *signitemrefs* filters control which items are signed or ignored, while the *signoptions* and *signoptionrefs* filters control which options are signed or ignored. Each level of filter also has an assigned order of precedence. For example, filters at the option level override filters at the item level.

By using these filters in combination, XFDL provides complete control over which elements are omitted from a signature (or alternately to indicate which elements should be included in a signature, though 'inclusive logic' filters should be used sparingly and with great care). The complete list of available filters is: ***signitems, signoptions, signpagerefs, signdatagroups, signgroups, signitemrefs, signnamespaces,*** **and *signoptionrefs*.**

3.3.2 Applying Multiple Signatures

Documents often require multiple signatures. Furthermore, it is common practice for some signatures to endorse other signatures. These secondary signatures can be said to overlap the original signatures, since they **sign both the document and the original signature.** For example, an insurance claim requires the claimant to sign the document. Later, the insurance adjuster may also have to sign the document, both to endorse the information provided by the claimant and to endorse information they have added to the claim.

XFDL allows any number of signatures in a single document. The signatures will sign separate portions of the form, or will overlap with other signatures, as specified by the filters used. For example, the first signature may use a set of filters that includes all elements in the top half of a page. The second signature may use a filter that includes the first signature and the top half of the page. Finally, a third signature might use a filter that includes the entire page and both the first and second signatures.

3.3.3 Securing Signed Elements

Paper documents rely on ink to secure the document. That is, once a document is signed, it is difficult to change the document because it is difficult to erase ink from paper. The very nature of paper and ink enforce the security of the document, since attempts to change the document generally leave detectable traces. In contrast, digital documents do not share this type of inherent security.

For this reason, the XFDL processor must provide the necessary security. Once an element in a document is signed, it is implicit that future readers should be unable to change that element.

3.3.4 Preventing Layout Changes

Once a document is signed, it is also implicit that the layout of that document should be secure. For example, if it were possible to move a paragraph, or even a line, the meaning of the document could be changed. To reflect this, any software processing XFDL must maintain the position of signed visual elements. This means that **both the position and the size of the visual elements must be secured.**

Thus, when a document is signed, the width, height, and position of all visible signed elements must be recorded. XFDL provides the **layoutinfo option** as a place to store this information within a given signature element. Furthermore, the layoutinfo option itself should be signed as part of the signature, ensuring that it cannot be changed.

The layout can later **be tested by re-calculating the position of all signed elements** and comparing this to the information stored in the layoutinfo option for that signature. If the information does not match, then the document has been modified and cannot be trusted.

The software processing the XFDL should perform this layout test at the following times:

- Immediately after a signature is created, it should test the entire document. This ensures that the process of generating the signature did not change the information.
- Whenever a page of the document is viewed, it should test the signed contents of that page.
- Whenever an item is computationally added, deleted, or moved, it should test the appropriate page.
- Whenever the details of a signature are viewed, it should test all portions of the document signed by that signature.

3.3.5 Preventing Exploitable Overlaps of Signed Elements

Unlike paper documents, digital documents also offer the potential for visual elements to overlap. For example, it is possible to create a block of text in a document, and to then obscure or hide that text with a second, overlapping block of text. In this scenario, even if the first block of text was secured with a signature, it would be possible to move the second block of text after the document was signed. This would change the meaning of the document by revealing information that was previously hidden.

Since the guiding principle of signatures is that ″**you sign what you see,**″ a scenario in which visual items are hidden or significantly overlapped cannot be allowed. If the signer cannot see elements of the form, then the signature cannot be considered valid. Exceptions for box items. Boxes are often used to visually create sections in a document, and will overlap other visual elements as a result. This overlap may be allowed.

3.4 The XFDL Compute System

XFDL computes are required to make changes to the presentation layer that are not related to data, such as color changes and so on. In general, using XForms computes to manipulate data and XFDL computes to manipulate the presentation layer will create a clean separation of duties that creates few conflicts.

An XFDL compute can be either a mathematical or conditional expression. A conditional expression has three parts separated by the ternary ?: operator. The first part is a Decision, which yields a boolean result. The consequences for a true and false boolean result recurse to the definition of Compute, permitting arbitrary nesting of decision logic.

```
Compute ::= Expr | Decision '?' Compute ':' Compute
```

The decision logic can apply logical-or (|| or 'or'), logical-and (&& or 'and'), and logical negation (!) to the results of logical comparisons.

A mathematical expression, denoted Expr, can include addition, subtraction, string concatenation (+.), multiplication, division, integer modulus, unary minus, and exponentiation.

4. DESIGN

A number of modifications are made in XForms and XFDL. The new language has the following features –

4.1 XML – Based Language

The WWW Forms technology started with HTML. HTML provided for the presentation but there was no common format for the representation of data. Then came XML which provided a common format for representation of data. Today, all web technologies are being developed around XML. A major criteria for acceptance of new languages is whether it would fit into the XML Workflows. XForms and XFDL are XML based. So, we had to ensure that any modifications we make conform with the XML syntax.

4.2 MVC Model

Our language follows a Model-View-Controller programming model as does XForms. But there is nothing in XForms that directly maps to the controller layer. Some consider the control part as the declarative calculations and validations present under the model tag of XForms. Others may consider the dynamic presentation part as the control because that helps us control the presentation. Still others consider XML Events and XFroms Actions as the control. We are in favour of the first concept and have separated the calculations and validations (which we consider the control) from the model part into the control part.

Under the model tag are the schema and instance data for the form. We provide a separate control tag which has **2 child elements - <calc> and <valid>** for calculations and validations respectively.

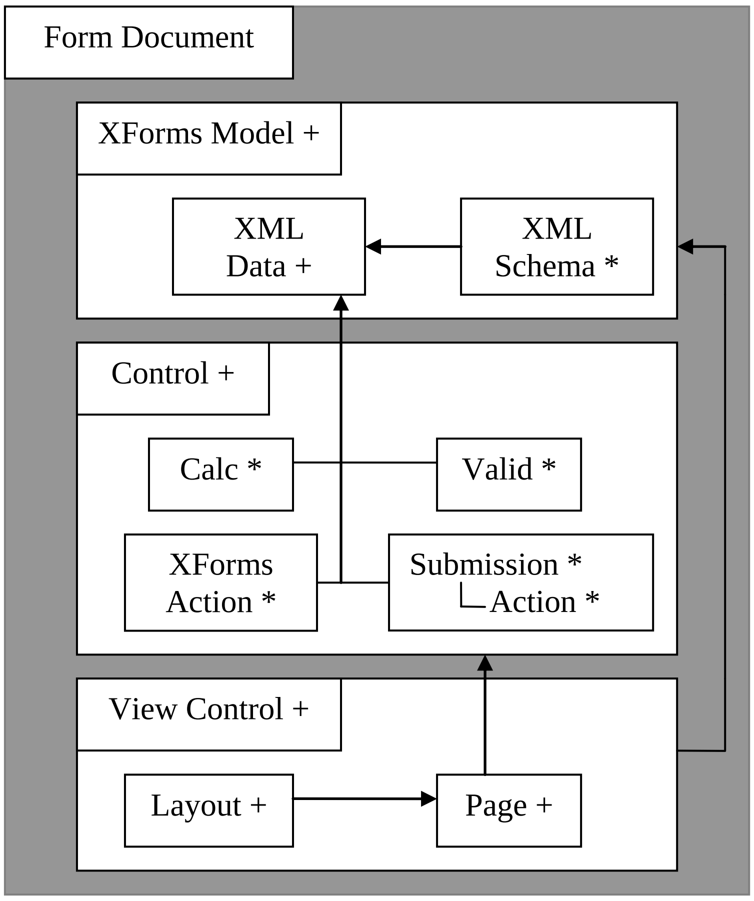


Fig. 4.1 Architecture of the new Language

4.3 Default Namespaces

The most commonly used namespaces in XForms + XFDL forms need not be specified in our language. As the converter knows which tag belongs to which namespace, it will be assigned the corresponding namespace.

The following namespaces are included into the file by default, others need to be specified. Or if we want to change the prefixes for the existing namespaces, namespace needs to be defined explicitly.

Namespaces

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

4.4 The <calc> tag

In XForms, the calculations are represented by **<bind>** statements that is to calculate the value of a variable, we bind it to the expression used to calculate its value. In our language, we have specified the calculations using the **<calc>** tag which has 2 child elements - **<var>** for variable whose value is being calculated, **<expr>** which contains the expression.

Example :

In the code given below, 4 <bind>s correspond to one <calc> because using our <calc> tag any no. of variables can be specified separated by commas. Four variables are specified in one <var> tag, <expr> contains the corresponding four expressions.

4.5 The <valid> tag

In XForms, the validations are also represented by **<bind>** statements that is we can bind various model item properties (MIPs) eg – relevant, required, type, p3ptype etc. to a variable. In our language, we have specify the validations using the **<valid>** tag which has 2 child elements - **<var>** for variable being validated, and the second tag depends on the property being specified eg- it is <relevant> for the relevant property.

Example :

XForms Code

```
<xforms:bind nodeset="/purchaseOrder/payment/as"
relevant="/purchaseOrder/totals/rowcount > 0"> </xforms:bind>
<xforms:bind nodeset="/purchaseOrder/payment/cc"
relevant="../as='credit'"> </xforms:bind>
<xforms:bind nodeset="/purchaseOrder/payment/exp"
relevant="../as='credit'" type="xsd:date"> </xforms:bind>
<xforms:bind nodeset="/purchaseOrder/payment/customernumber"
relevant="../as='cash'"> </xforms:bind>
```

New Code
`<valid path="purchaseOrder/payment"> <var> /as,/cc,/exp,/customernumber </var> <relevant> purchaseOrder/totals/rowcount &gt;0, /as='credit', /as='credit', /as='cash' </relevant> </valid>`

```
<valid path="purchaseOrder/payment">
      <var> /as,/cc,/exp,/customernumber </var>

<relevant> purchaseOrder/totals/rowcount &gt;0,
                /as='credit',
                /as='credit',
                /as='cash'
      </relevant>
</valid>
```

In the above code, again 4 <bind>s correspond to one <valid> because using our <valid> tag any no. of variables can be specified separated by commas. <relevant> tag specifies the comma-seperated property for the variables. But if the properties are different, they must be specified using separate <valid> tags as grouping can only be done for similar things.

4.6 Submission of Encrypted Data

XForms provides various encodings to submit the data but no encryption method is provided. Data submitted as XML can be read and understood by anybody. To protect the data, we have proposed a mechanism in which, if the submission element has a encryption attribute on it which specifies the encryption algorithm to be used, the data is submitted in encrypted form rather than xml (assuming that encoding specified is xml).

Example :

```
<submission id="submit1"  how="post" where="file:data.xml"
encryption="RSA"/>
```

In this example, the data is submitted as xml to the file data.xml. The text portion of the elements in this file will be encoded using the RSA encryption algorithm. Thus, text cannot be easily read by anybody. We will of course want to select an algorithm which is difficult to break, in order to provide maximum security.

4.7 Grouping of similar tags

As seen in the <valid> tag above, variables for which same property needs to be specified are grouped into one <valid>. The same concept in applied throughout the language. This reduces the length of of the code by a huge ratio. The effect can be best seen in real-world forms which have thousands of lines of code. Having a look at such a code reveals there is a large amount of repetition.

To remove this repetitive code, we eg- combine 4 <input> elements into one <input> element and separate the features of each by commas. This leads to a constraint in our language as compared to other XML based languages – the text cannot contain commas. But this problem can be solved by including the comma(,) into the list of prohibited characters for XML and like apostrophe is represented as **&apos:** , comma can be represented as **&com;** (All prohibited characters begin with a ampersand(&) and end with a semicolon(;)).

Example :

In the example below, <xforms:output> is enclosed within <xfdl:label> tag. As specified in the earlier chapters, all XForms tags must be enclosed within XFDL tags for the form to run on a XFDL processor. 4 outputs are specified. In our language, these can be grouped to form one <output> tag.

The resultant reduction in length can be noticed by just having a look at the form. Reducing the size of the code in important for a person who wants to type the form code in a simple textpad or wordpad because tools may not be always available. This will also result in minor reduction in the memory occupied by the form code though this is not a significant factor.

XForms Code

```
<label sid="cnt">
      <xforms:output ref="/purchaseOrder/totals/rowcount">
            <xforms:label>No. of items = </xforms:label>
      </xforms:output>
</label>

<label sid="sub">
     <xforms:output ref="/purchaseOrder/totals/subtotal">
            <xforms:label>Subtotal = </xforms:label>
      </xforms:output>
</label>

<label sid="tax">
      <xforms:output ref="/purchaseOrder/totals/tax">
            <xforms:label>Tax = </xforms:label>
      </xforms:output>
</label>

<label sid="total1">
      <xforms:output ref="/purchaseOrder/totals/total">
            <xforms:label>Total = </xforms:label>
      </xforms:output>
</label>
```

New Code

```
<output path="purchaseOrder/totals">
      <id> cnt,sub,tax,total1 </id>
      <what> /rowcount,/subtotal,tax,total </what>
      <label>No. of items = ,Subtotal = ,
            Tax = ,Total = </label>
</output>
```

4.8 Default Presentation Options

It is tedious to write code in XFDL using XForms because for the form to run on a XFDL processor, each XForms tag must be enclosed within a XFDL tag eg- <xforms:input> may be enclosed within <field>,<check> or <combobox>. But in most of the forms you see, you will find <xforms:input> enclosed within <field>. So, we have taken <field> as the default enclosure for <xforms:input>. If <xforms:input> needs to be enclosed within <combobox>, our language provides a **displayType attribute** on each XForms item. If displayType attribute is not provided, default enclosure is used.

This has been done for all the XForms form controls. The following table indicates the default enclosures for each.

Form Control	Possible Skins	Default Skin
output	label	label
input	field, check, combobox	field
secret, textarea	field	field
select	list, checkgroup	list
select1	list, popup, combobox, radiogroup, checkgroup	popup
range	slider	slider
upload	button	button
Submit, trigger	button, action	button
group, switch	pane	pane
repeat	table	table

Table 4.1 List of Default Skins

XForms Code	New Code
`<check sid="healthPlan_check"> <xforms:input ref="healthinfo/healthPlan"> </xforms:input> <label>Active Health Plan</label> </check>`	`<input displayType="check" id="healthPlan_check" what="healthinfo/healthPlan" label="Active Health Plan> </input>`

4.9 Seperation of Layout Information

Layout means the size and location of various items on the page being displayed. As layout information deals with the presentation, it is specified in XFDL and XForms is not concerned with it. In any real-world form coded using XFDL, you will find the

<itemlocation> tag placed within each and every item of the form.This tag is used to specify the x and y coordinates at which the item must be placed on the form. It may also specify the width and height of the item.

In the language that we propose, we separate this layout information from the actual item specification. That is ,the properties of the items is specified separately and layout separately.The itemlocation is also a property of the items, but as it is common to each item, it can be specified at a separate place.

A XFDL form has a number of pages and therefore, each page will have seperate layout. The information must be specified in the new **<layout>** tag which is the subelement of the <page> element. The example below shows how this can be done. Notice the reduction in size of the code. This is even more for larger forms.

XForms Code

```
<page sid="page1">
      <global sid="global"/>
      <label sid="persInfo_label">
            <value>Personal Information</value>
            <itemlocation>
                  <x>20</x>
                  <y>30</y>
            </itemlocation>
      </label>
      <label sid="name">
            <value>Name : </value>
            <itemlocation>
                  <x>20</x>
                  <y>60</y>
            </itemlocation>
      </label>
      <label sid="name_out">
            <xforms:output ref="info/name">
            </xforms:output>
            <itemlocation>
                  <x>60</x>
                  <y>60</y>
            </itemlocation>
      </label>
</page>
```

New Code

```
<page id="page1">
      <label>
            <id> persInfo_label, name </id>
            <value>Personal Information, Name :</value>
      </label>

      <output id="name_out" what="info/name"/>

      <layout>
            <items>
                  <id> persInfo_label, name, name_out
                  </id>
                  <coord> (20,30), (20,60), (60,60)
                  </coord>
            </items>
      </layout>

</page>
```

4.10 Modifications in the <table>

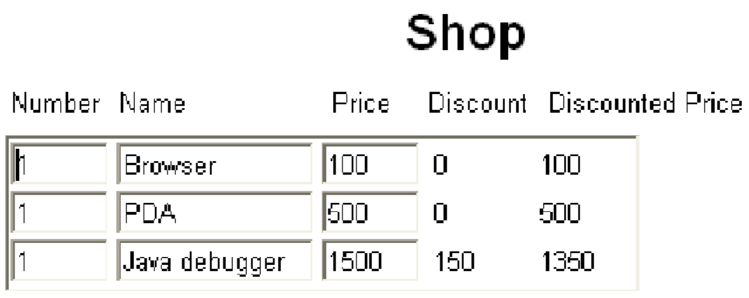
Most forms that display or input similar information for more than entity, use tables. Eg- A shop has several items, each having a name, price and may have some discount on it. Suppose, we want to represent this information for 10 items, then it is best done using tables.

XForms supports dynamic presentation through the **repeat** tag, which helps in creating a table whose rows can be dynamically increased or decreased. So, most of the tables seen in real-world forms have 2 buttons at the bottom, one for inserting rows into the table and other for deleting them.

But, XFDL just provides the table tag, and no explicit support for the header row of the table and buttons at the bottom. Even HTML tables have a header row but in XFDL, we have to create the header row using separate label tags as can be seen in the example below.

Example :

For creating the following table, the original code and the new code are shown –



Number	Name	Price	Discount	Discounted Price
1	Browser	100	0	100
1	PDA	500	0	500
1	Java debugger	1500	150	1350

There are **2 possible syntaxes** that we propose for <table> :

- **First code** below has labels for each input and output within the table. These **labels** can be used to provide the header row for the table but remember, these labels appear only once in the table and not as many times as the input or output appears.
- The **second code** shows an explicit **heading tag.**

Both the tables contain a trigger tag for the insert and delete buttons at the end of the table. Instead of creating separate buttons, we have included this within the table. The **<type> element** specifies the type of the button – whether it is for insertion or deletion. These types are introduced by us and are not build into XFDL.

XForms Code
<label sid="label1"> <value>Number</value> </label> **<label** sid="label2"> <value>Name</value> </label> **<label** sid="label3"> <value>Price</value> </label> **<label** sid="label4"> <value>Discount</value> </label> **<label** sid="label5"> <value>Discounted Price</value> </label>

```
<table sid="itemtable">
      <xforms:repeat id="repeat1"
      nodeset="/purchaseOrder/items/item">

            <field sid="field1">
                  <xforms:input ref="units">
                        <xforms:hint> The units of this item
                        </xforms:hint>
                  </xforms:input>
            </field>

            <field sid="field2">
                  <xforms:input ref="name">
                        <xforms:hint> The name of this item
                        </xforms:hint>
                  </xforms:input>
            </field>

            <field sid="field3">
                  <xforms:input ref="price">
                              <xforms:hint>The price of this item
                              </xforms:hint>
                  </xforms:input>
            </field>

            <label sid="output1">
                  <xforms:output ref="discount">
                  </xforms:output>
            </label>

            <label sid="output2">
                  <xforms:output ref="total">
                  </xforms:output>
            </label>

</xforms:repeat>
</table>
<button sid="insert">
      <xforms:trigger id="insert1">
            <xforms:label>Add item</xforms:label>
            <xforms:action ev:event="DOMActivate">
                  <xforms:insert at="index('repeat1')"
                  nodeset="/purchaseOrder/items/item"
                  position="after">
                  </xforms:insert>
            </xforms:action>
      </xforms:trigger>
</button>
<button sid="delete">
      <xforms:trigger id="delete1">
            <xforms:label>Delete item</xforms:label>
            <xforms:action ev:event="DOMActivate">
                        <xforms:delete at="index('repeat1')"
                        nodeset="/purchaseOrder/items/item">
                        </xforms:delete>
            </xforms:action>
      </xforms:trigger>
</button>
```

First Code

```
<table>
      <repeat what="purchaseOrder/items/item" id="repeat1">
            <input>
                  <id>unts,nm,pr</id>
                  <label>Number, Name, Price </label>
                  <what>/uints,/name,/price</what>
                  <hint>The units of this item,
                        The name of this item,
                        The price of this item
                  </hint>
            </input>
            <output>
                  <id>discnt,tot</id>
                  <label>Discount, Discounted Price</label>
                  <what>/discount,/total</what>
            </output>
      </repeat>
      <trigger>
            <id> insert1, delete1 </id>
            <label> Add item, Delete item </label>
            <type> insert, delete </type>
      </trigger>
</table>
```

Second Code

```
<table>
      <heading>
            <id>no1,name1,price1,dis1,disprice1</id>
            <value>Number,Name,Price,Discount,Discounted
            Price</value>
      </heading>
      <repeat what="purchaseOrder/items/item" id="repeat1">
            <input>
                  <id>unts,nm,pr</id>
                  <what>/uints,/name,/price</what>
                  <hint>The units of this item,
                        The name of this item,
                        The price of this item
                  </hint>
            </input>
            <output>
                  <id>discnt,tot</id>
                  <what>/discount,/total</what>
            </output>
      </repeat>
      <trigger>
            <id> insert1, delete1 </id>
            <label> Add item, Delete item </label>
            <type> insert, delete </type>
      </trigger>
</table>
```

4.11 The <select> clause

There are **2 clauses in XForms – select and select1**, to specify selection of one or more and one item respectively. We have combined these into one clause that is **<select> with a type attribute** whose possible values are **one or many**. The default value for the type attribute is many that means, the <select> without the type attribute is equivalent to XForms select.

Secondly,XFroms provides **2 versions of select** – one contains **the itemset** element that refers to the instance data and the in the other, **items** are explicitly specified. Following examples show the new code for each of these.

Example (1) :

XForms Code

```
<checkgroup sid="currency">
      <xforms:select ref="currency" appearance="full">

      <xforms:label >Select the currencies you accept:
      </xforms:label>

      <xforms:item>
            <xforms:label>US Dollars </xforms:label>
            <xforms:value> USD </xforms:value>
      </xforms:item>

      <xforms:item>
            <xforms:label>CDN Dollars </xforms:label>
            <xforms:value> CDN </xforms:value>
      </xforms:item>

      <xforms:item>
            <xforms:label>Euro</xforms:label>
            <xforms:value>Euro</xforms:value>
      </xforms:item>

      </xforms:select>
</checkgroup>
```

New Code

```
<select id=”currency” displayType =”checkgroup”
what=“currency”>

      <label> Select the currencies you accept:</label>
      <items>
              <label>US Dollars, CDN Dollars, Euro </label>
            <value> USD, CDN, Euro </value>
      </items>

</select>
```

Example (2) :

XForms Code

```
<checkgroup sid="currency">
         <xforms:select ref="currency" appearance="full">
            <xforms:label>Select the currencies you accept:
            </xforms:label>

            <xforms:itemset
            nodeset="instance(’currency’)/choice">
                  <xforms:label ref="@show"> </xforms:label>
                  <xforms:value ref="."> </xforms:value>
            </xforms:itemset>
      </xforms:select>
</checkgroup>
```

New Code

```
<select id=”currency” what="currency">
      <label>Select the currencies you accept: </label>

      <items what="instance(’currency’)/choice">
                  <label what="@show"> </label>
                  <value what="instance(‘currency’)/choice">
                  </value>
            </items>
      </select>
```

5. IMPLEMENTATION

5.1 XML Parsers

A parser is a piece of program that takes a physical representation of some data and converts it into an in-memory form for the program as a whole to use. An XML Parser is a parser that is designed to read XML and create a way for programs to use XML. The main types of parsers are : **SAX, DOM and pull.**

1. SAX

SAX stands for ***Simple API for XML***. Its main characteristic is that as it reads each unit of XML, it **creates an event that the calling program can use**. This allows the calling program to ignore the bits it doesn't care about, and just keep or use what it likes. The disadvantage is that the calling program must keep track of everything it might ever need.

2. DOM

DOM (***Document Object Model***) is an official recommendation of the W3C. It differs from SAX in that it builds the ***entire*** **XML document representation in memory** and then hands the calling program the whole chunk of memory. DOM can be very memory intensive as the entire tree has to be stored in the memory.

3. Pull Parser

SAX is a *push* parser, since it pushes events out to the calling application. *Pull* parsers, on the other hand, sit and wait for the application to come calling. They ask for the next available event, and the application basically loops until it runs out of XML.

Pull parsers are useful in streaming applications, which are areas where either the data is too large to fit in memory, or the data is being assembled just in time for the next stage to use it. It is designed to be used with large data sources, and unlike SAX which returns *every* event, **the pull parser can choose to skip events** (or in some implementations, whole sections of the document) that it is not interested in. The adapters are designed to work with both the SAX and the pull parser interfaces.

	SAX	**DOM**
Origin	Previously result of xml-dev community, started as Java only interface. Now maintained by the SourceForge organization.	W3C Organization recommendation. DOM is not an API.
Interface type	Primarily a java interface only. Now interfaces available on most programming languages	Language and platform – neutral recommendation.
Resource consumption	Limited impact	High impact on memory and processing resource as the DOM create in-memory representation of the XML document
How it operates	Event based interface. Events are triggered where the SAX parser encounters an XML tag.	Parses the XML document first, and then creates an in-memory representation of XML file as a nodes tree.
Document handling	Read-only parser	Can manipulate nodes and add , delete nodes.
Examples of Ideal situations for use	When memory/processing power is restricted. When XML documents size is very large and no random access is required.	When Random access of XML documents is required. For manipulation of in-memory structure of an XML document. When support for Namespaces is desirable.

Table 5.1 A Comparison of the SAX and DOM Parsers

Another way that parsers are classified is - **Validating versus non-validating parsers.** Validating parsers validate XML documents as they parse them, while non-validating parsers don't. In other words, if an XML document is well-formed, a non-validating parser doesn't care if that document follows the rules defined in a DTD or schema, or even if there are any rules for that document at all.

There are 2 reasons that we use a non-validating parser :

- **Speed and efficiency**. It takes a significant amount of effort for an XML parser to read a DTD or schema, then set up a rules engine that makes sure every element and attribute in an XML document follows the rules.

- **If you're sure that an XML document is valid** (maybe it's generated from a database query, for example), you may be able to get away with skipping validation. Depending on how complicated the document rules are, this can save a significant amount of time and memory.

How to use a parser

Generally, the following 3 steps are involved in programs using parsers :

1. Create a parser object
2. Point the parser object at your XML document
3. Process the results

5.2 The DOM Parser

When you parse an XML document with a DOM parser, you get a hierarchical data structure (a DOM tree) that represents everything the parser found in the XML document. You can then use functions of the DOM to manipulate the tree. You can search for things in the tree, move branches around, add new branches, or delete parts of the tree.

From a Java-language perspective, **a Node is an interface**. The Node is the base datatype of the DOM; everything in a DOM tree is a Node of one type or another.

DOM also defines a number of subinterfaces to the Node interface:

- **Element :** Represents an XML element in the source document.
- **Attr :** Represents an attribute of an XML element.
- **Text :** The content of an element. This means that an element with text contains text node children; the text of the element is *not* a property of the element itself.

- **Document :** Represents the entire XML document. Exactly one Document object exists for each XML document you parse.

Additional node types are: Comment, ProcessingInstruction, and CDATASection, which represents a CDATA section.

5.2.1 DOM APIs

JAXP, the Java API for XML Parsing specifies certain common tasks that the DOM and SAX standards leave out. Specifically, creating parser objects is not defined by the DOM or SAX standards.

The Document Object Model implementation is defined in the following packages:

- **org.w3c.dom -** Defines the DOM programming interfaces for XML (and, optionally, HTML) documents, as specified by the W3C.

- **javax.xml.parsers -** Defines the DocumentBuilderFactory class and the DocumentBuilder class, which returns an object that implements the W3C Document interface. This package also defines the ParserConfigurationException class for reporting errors.

We can use the DocumentBuilder newDocument() method to **create an empty Document** that implements the org.w3c.dom.Document interface. Alternatively, we can use one of the builder's parse methods to create a Document from existing XML data.

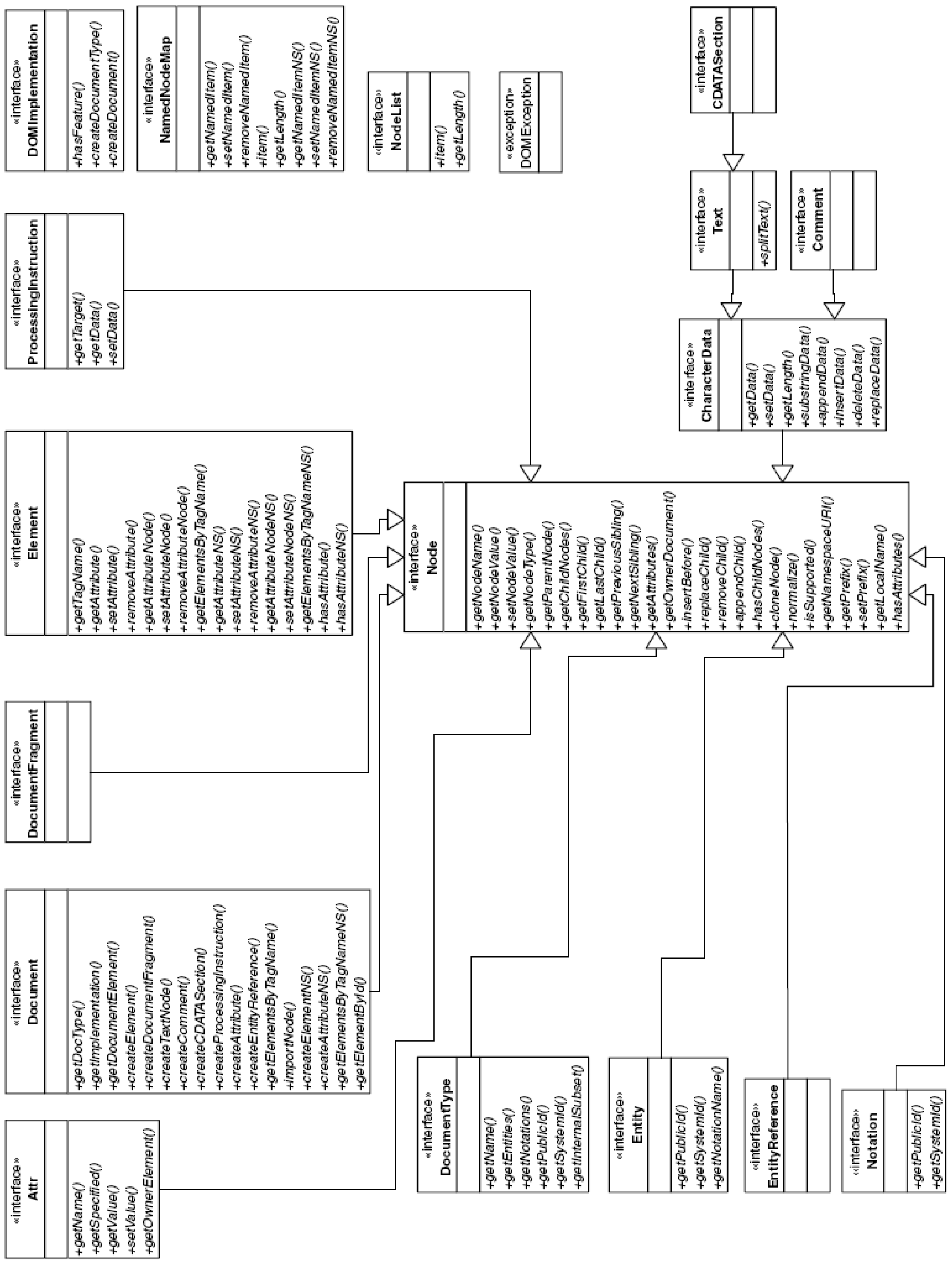


Fig. 5.1 The DOM Interface Hierarchy

5.2.2 Reading XML Data into a DOM

The following steps are involved in creating a DOM from an existing XML file :

1. Import the Required Classes

These lines import the JAXP APIs that we will be using:

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;

These lines import the exception details for exceptions that can be thrown when the XML document is parsed. DOMExceptions are only thrown when traversing or manipulating a DOM. Errors that occur during parsing are reported using a same mechanism as SAX:

import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

Finally, import the W3C definition for a DOM and DOM exceptions:

import org.w3c.dom.Document;
import org.w3c.dom.DOMException;

2. Declare the DOM

The **org.w3c.dom.Document** class is the W3C name for a Document Object Model (DOM). Whether we parse an XML document or create one, a Document instance will result.

static Document document*;*

3. Handle Errors

Next, we put in the error handling logic. The error-handling code for DOM and SAX applications are very similar:

try {
} **catch (SAXException sxe)** {
// Error generated during parsing
Exception x = sxe;

```
if (sxe.getException() != null)
x = sxe.getException();
x.printStackTrace();
} catch (ParserConfigurationException pce) {
// Parser with specified options can't be built
pce.printStackTrace();
} catch (IOException ioe) {
// I/O error
ioe.printStackTrace();
}
```

4. Instantiate the Factory

Next, we add the code highlighted below to obtain an instance of a factory that can give us a document builder:

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

5. Get a Parser and Parse the File

DocumentBuilder builder = factory.newDocumentBuilder();
document = builder.parse(new File(argv[0]));

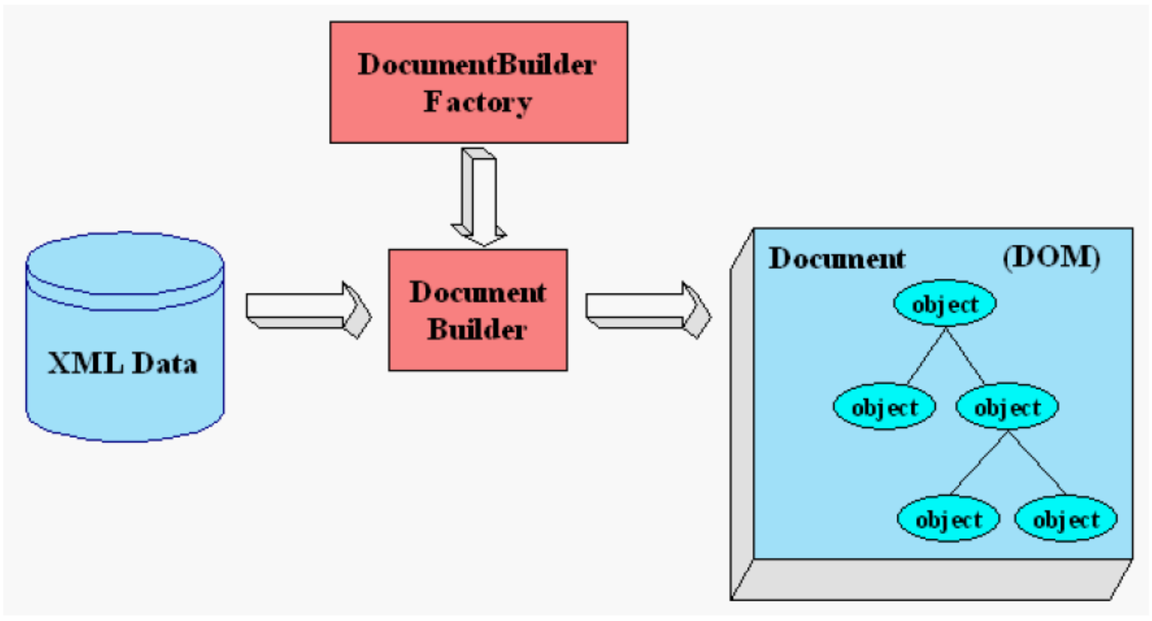


Fig. 5.2 Steps in creating a DOM Tree

6. Setting DOM parser features

After getting the document object, we may want to configure the factory. The most important methods are:

setValidating(boolean) - Sets the factory's validation property.
isValidating() - Returns true if the factory creates validating parsers, false otherwise.
setNamespaceAware(boolean) - Sets the factory's namespace-aware property.
isNamespaceAware() - Returns true if the factory creates namespace-aware parsers, false otherwise.
setIgnoringElementContentWhitespace(boolean) - Sets the factory's whitespace property. If this is true, the parsers created by the factory won't create nodes for the ignorable whitespace in the document.
isIgnoringElementContentWhitespace() - Returns true if the factory creates parsers that ignore whitespace, false otherwise.

5.2.3 Creating a new DOM

We are still going to create a document builder factory, but this time you're going to tell it create a new DOM instead of parsing an existing XML document.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
try {
DocumentBuilder builder = factory.newDocumentBuilder();
document = builder.newDocument();
```

In this code, you replaced the line that does the parsing with one that creates a DOM.

And since we are going to be working with **Element objects**, we add the statement to import that class at the top of the program:

```
import org.w3c.dom.Document;
import org.w3c.dom.DOMException;
import org.w3c.dom.Element;


//create an element XFDL and its children
Element root = (Element) document.createElement("XFDL");
document.appendChild(root);
```

```
root.appendChild( document.createTextNode("some") );
root.appendChild( document.createTextNode(" ") );
root.appendChild( document.createTextNode("text") );
```

For more details on DOM methods, refer [18].

5.3 The Conversion

The design chapter describes the modifications made to the XForms and XFDL. Our implementation part includes the conversion of this code into the XFDL code. The final form can be displayed in the IBM Workplace Forms Viewer which supports XForms + XFDL.

For the conversion, we have used the **Java Apache Xerces Parser** which comes with Java v1.4 and above. For lower versions of java, it can be integrated by downloading it from http://xerces.apache.org.

It is a fully conforming XML Schema processor.

- ✓ Supports Simple API for XML (SAX) 2.0.2
- ✓ Supports Document Object Model (DOM) Level 3
- ✓ Support XML 1.0 and Namespaces in XML 1.1 Recommendation

The conversion involves the following **steps :**

1. Read in the modified code and create a DOM Tree from it, by using a DOM Parser as decribed above.

2. This DOM Tree is subjected to the conversion code, which performs the appropriate conversion. The result is a new DOM tree.

3. The final DOM tree is then serialized to get an XML document (actually a XForms + XFDL document).

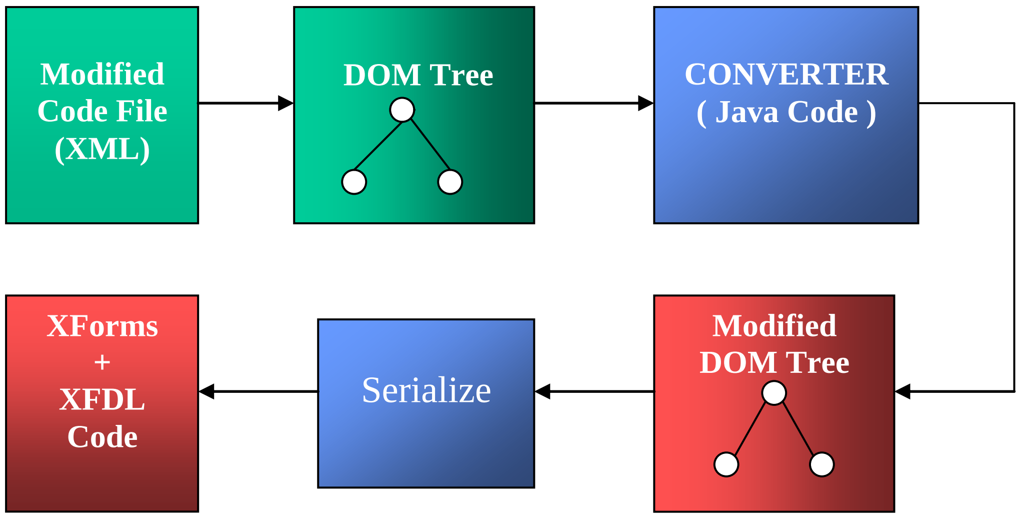


Fig. 5.3 Conversion Steps

5.4 Architecture of the converter

As can be seen from the figure below, the converter consists of 4 major parts (coded in java as 4 separate packages).

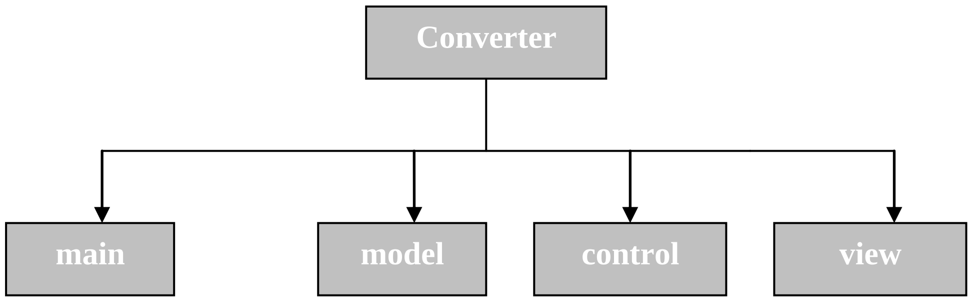


Fig. 5.4 Parts of the converter

5.4.1 The main package

This package contains the beginning code for the converter that is, code for parsing the XML file and create DOM tree. Also, it contains code to create the final XFDL document. It contains the following classes.

(i) convert – parses our modified code file and creates the DOM tree. It instantiates the createXFDL, createModelObject and createViewObject classes.

(ii) createXFDL – This class creates a new Document object, adds root element to it, and prints this to a XFDL file. It also sets the namespace attributes on the root element which specifies all the namespaces that can be used within the document.

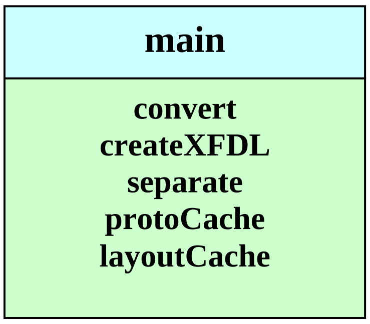


Fig. 5.5 The main package

(iii) separate - This class contains the code for converting the comma-seperated list of parameters into individual elements and appends the value of path to relative parameters. It is placed in the main class as it is a common class instantiated by various other classes. The code present here helps us to separate the similar tags that we did combine in our language.

(iv) prototype Cache – protoCache is used to store the prototype information. Whenever a new prototype is encountered in the globals section of the form, an instance of this class is created and it stores the prototype for later reference by any of the form controls.

(v) layout Cache – layoutCache reads the layout information within the <layout> tag and creates a cache that stores the ids and the corresponding location and size in formation. Each form control can call this class to obtain its location information.

5.4.2 The model package

This package contains the model part of the MVC paradigm. It contains the classes shown in the figure below :

(i) createModelObject – This class reads in the model elements and their properties from the modified code file, and creates the model objects. It also instantiates the createControlObject and createModelElement classes.

(ii) model – This class stores the model item properties namely model id, data file name or the actual data and schema.

(iii) createModelElement – This class takes the model object as input, and creates a model Element from it for storage in the new file. It instantiates the createBindSubmissionElement class as in the final XFDL file, bind and submission elements are part of the model itself.

(iv) createInstanceObject – A model can have various instances that represent the data instance. Each instance may be identified by an id. In our language, within the instance tag, we may give the file name or the actual data. This class reads the data within the <data> tag and stores it within the instance object.

(v) instance – This class stores the instance properties like nodelist of the children and the id.

(vi) createInstanceElement – This class creates the appropriate instance tag from the instance object. It checks if the child node of instance element is a text node. If it is, that means the user has given a file name which contains the instance, so it copies the file name into the src attribute. If the child is an element node, it imports all the children as such.

In the same way, there are 3 classes for handling the schema namely, createSchemaObject, schema, createSchemaElement.

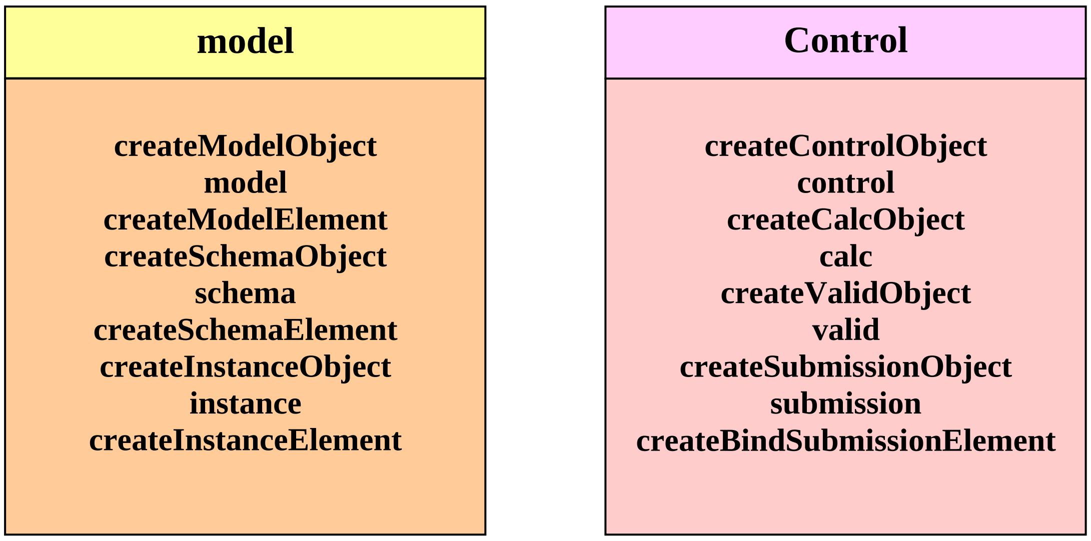


Fig. 5.6 model and control packages

5.4.3 The control package

It contains the code for **calculations, validations and submissions**. All these elements are converted to binds. It contains the following classes. The classes provide the similar functionalities as the corresponding classes in the model package. That is, createControlObject creates the control object that stores the control properties. As the name signifies, createBindSubmissionElement creates the bind elements from calc and valid elements and submission elements from the corresponding submission elements.

The Encryption handler

One modification made is the option for submission of encrypted data. In the encryption element, the user can mention encryption method to use. We use java inbuilt functions for the encryption. Whenever the submit button is clicked and encryption is set, the handler corresponding to the encryption is activated and data is submitted in the encrypted form. The handler is of course provided on the submit event.

5.4.4 The view package

This package contains a number of sub-packages for each specific presentation element. Following figure shows the view package, its classes and its sub-packages. View contains the **following classes –**

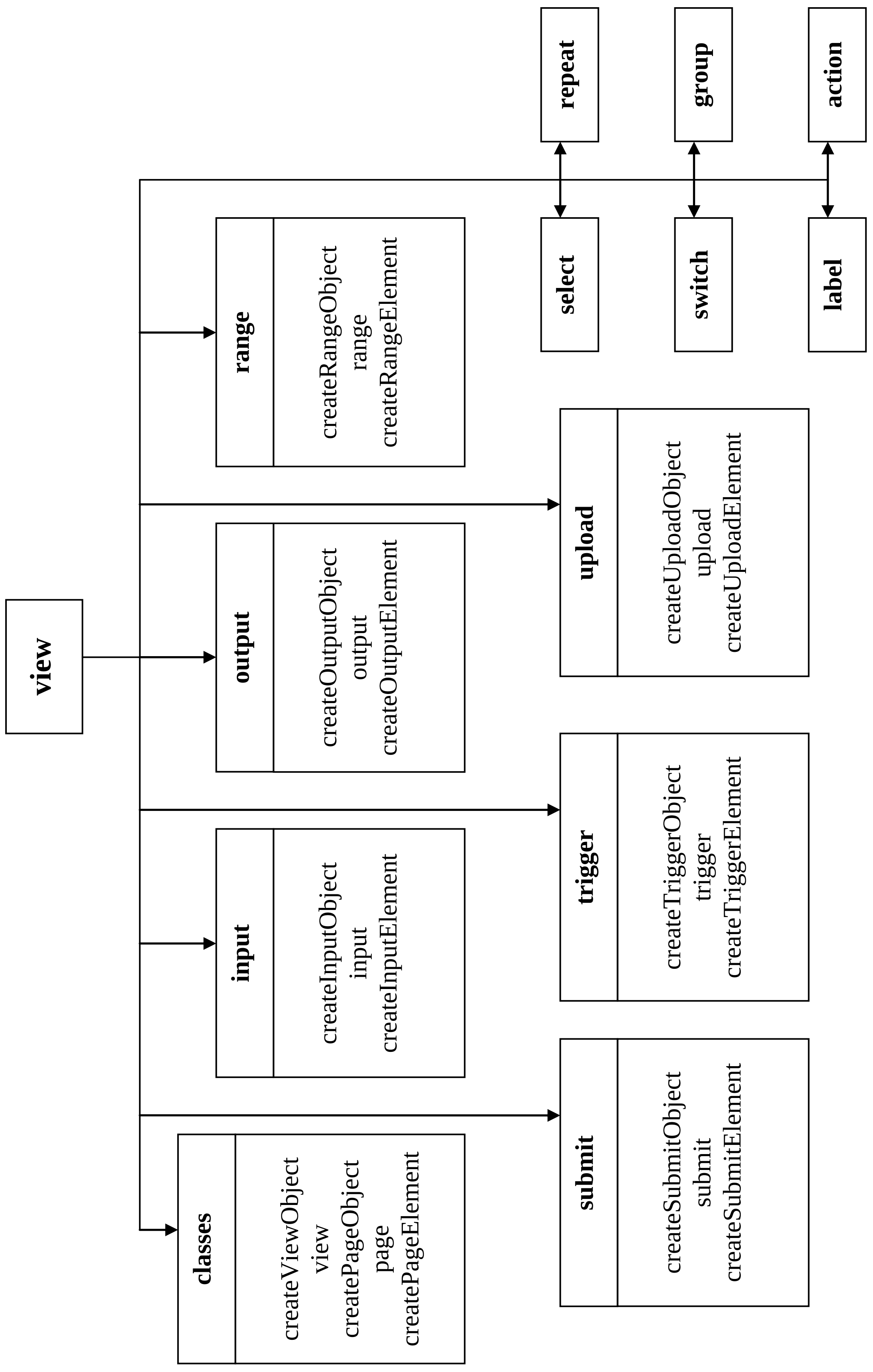


Fig. 5.7 The view Package

- createViewObject
- view
- createPageObject
- page
- createPageElement

There is no view element present in the XFDL file. The packages **input, output, range, submit, trigger and upload** have been specified in detail. The other packages namely **action, select, switch, repeat, group** further contain sub-packages which are depicted in separate package diagrams.

1. **The select package –** It contains the 3 files for creating the select element itself and a **package items** for creating the item or itemset element from the items element. Items further calls the label,value and copy classes.

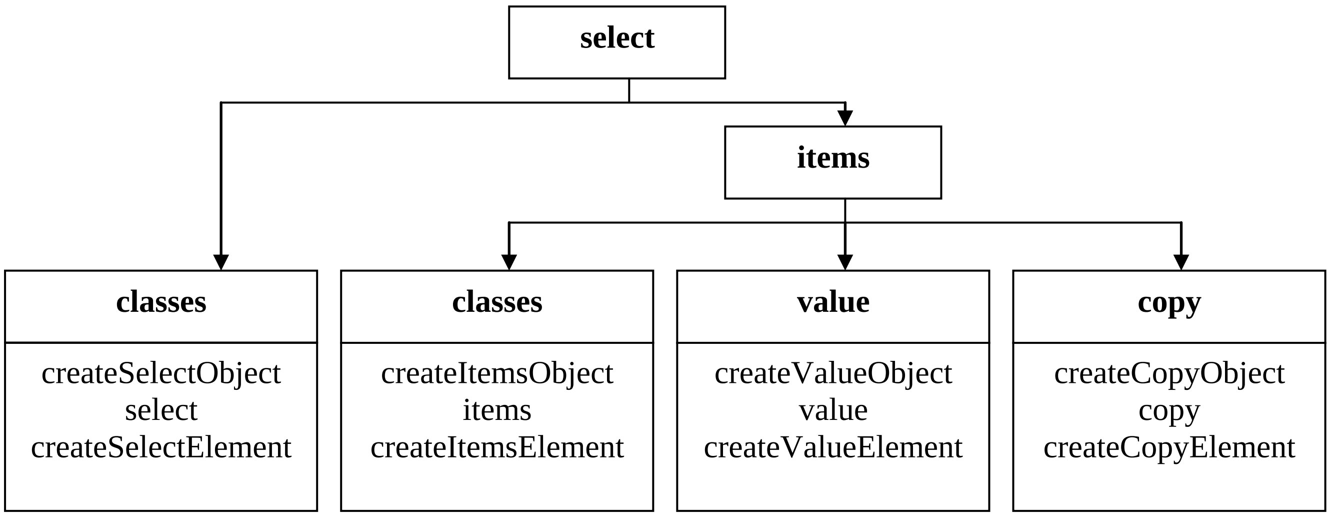


Fig. 5.8 The select package

2. **The pane package** – An XFDL pane may contain xforms:switch and xforms:group. The switch inturn contains case and its classes. Group module may call any of the form controls.

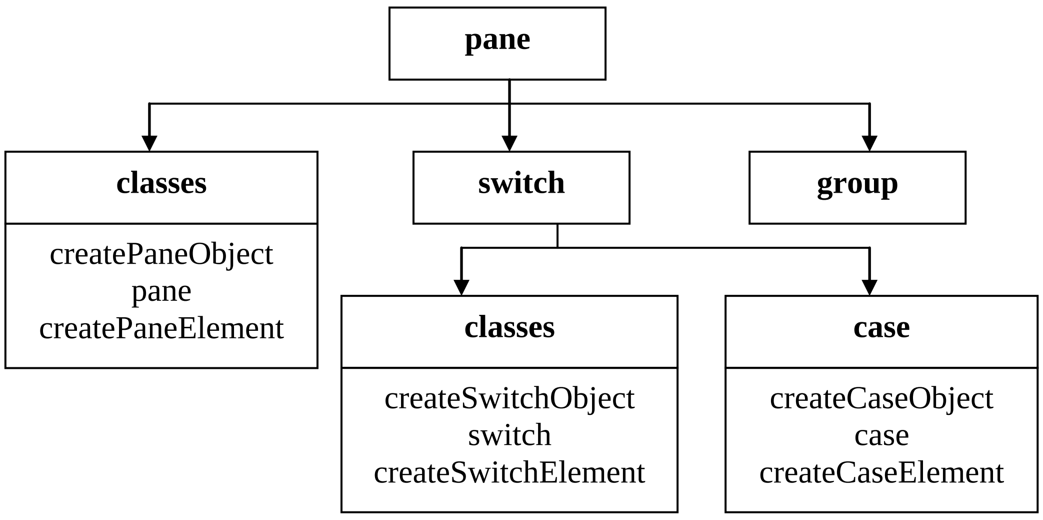


Fig. 5.9 The pane package

3. **The table package –** The table package contains classes for drawing the table, extracting the heading form the labels, creating the repeat construct and the default buttons – add and delete.

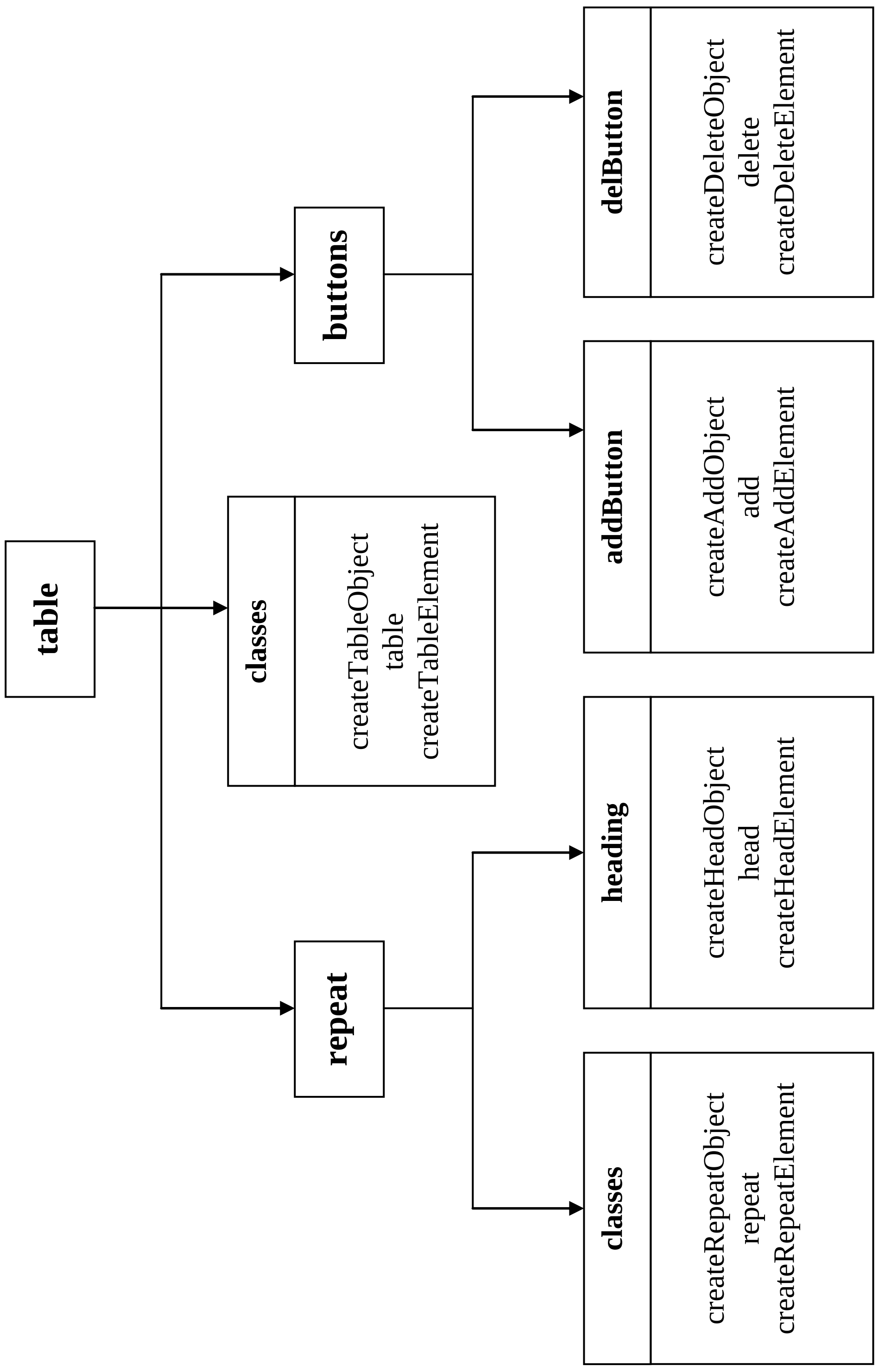


Fig. 5.10 The table Package

4. **The action package –** The action package contains the following classes – createActionObject, action, createActionElement and subpackages as shown in the figure.

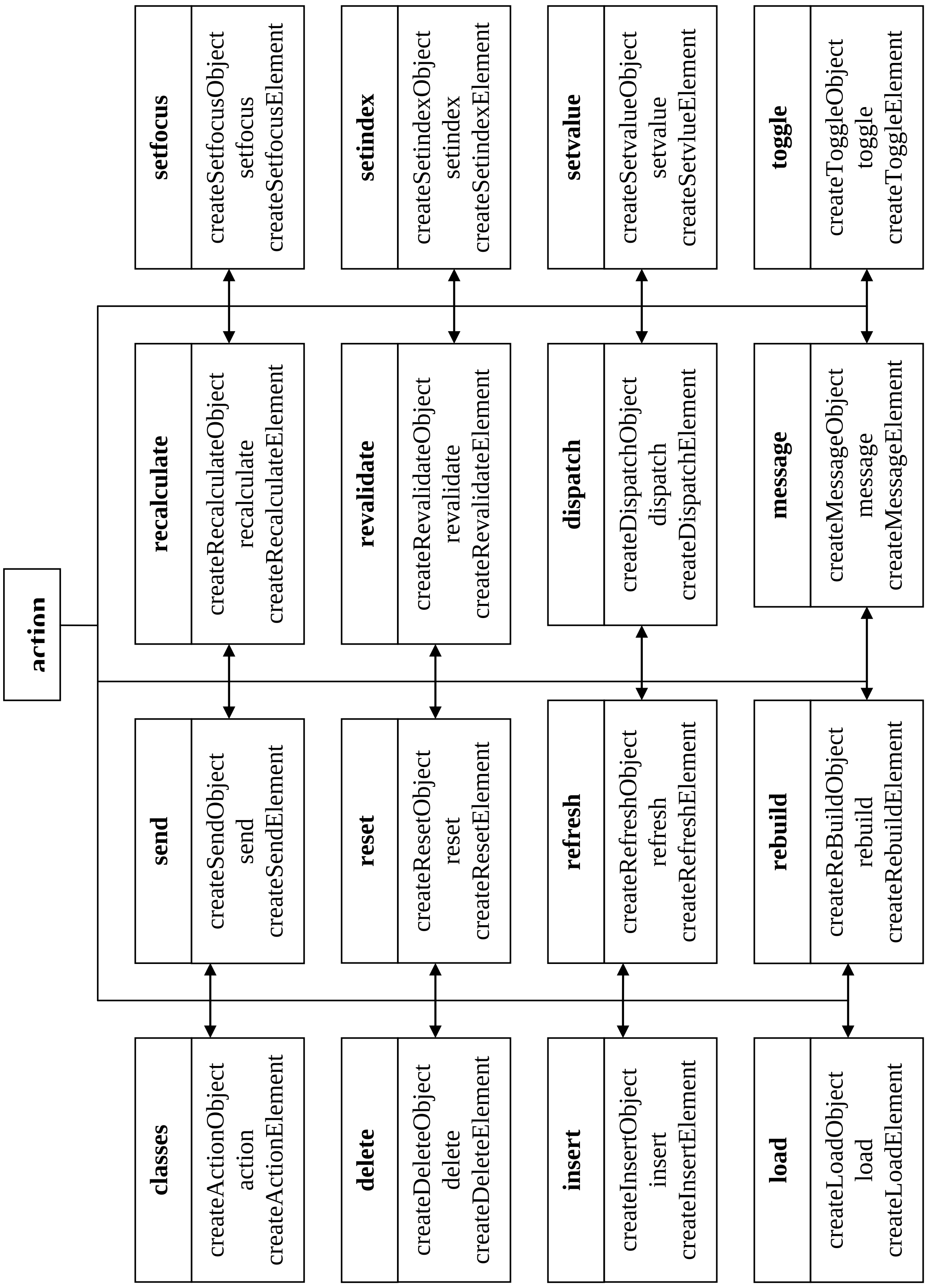
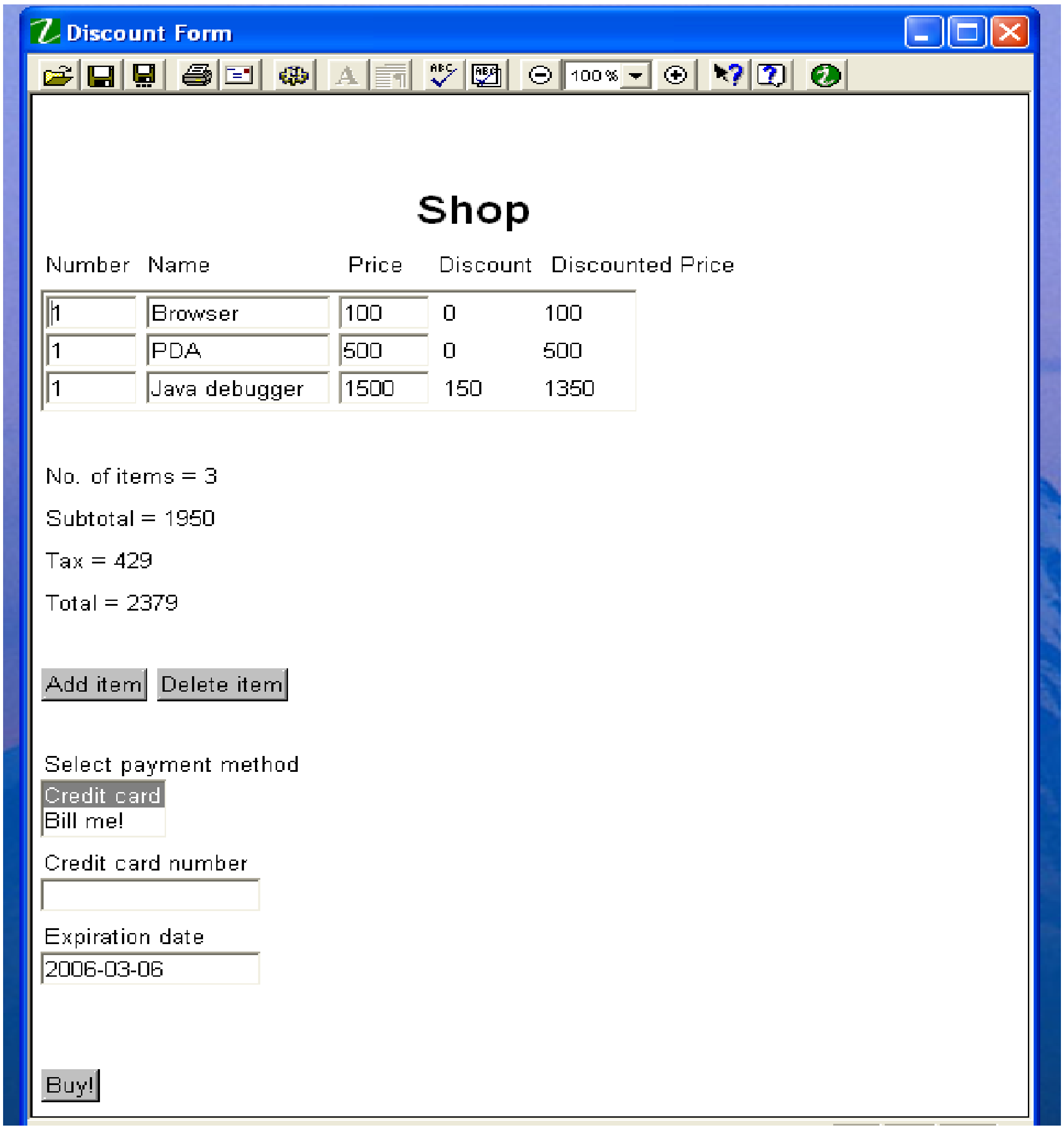


Fig. 5.11 The action Package

6. RESULTS (SAMPLE FORMS)

6.1 SAMPLE FORM 1 (DISCOUNT FORM)



Shop

Number	Name	Price	Discount	Discounted Price
1	Browser	100	0	100
1	PDA	500	0	500
1	Java debugger	1500	150	1350

No. of items = 3

Subtotal = 1950

Tax = 429

Total = 2379

Select payment method

[✓ Credit card]
Bill me!

Credit card number

Expiration date
2006-03-06

purchaseOrder.xml (The Data file)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<purchaseOrder xmlns="" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">

      <items>
            <item>
                  <name>Browser</name>
                  <units>1</units>
                  <price>100</price>
                  <total>0</total>
                  <discount>0</discount>
            </item>

            <item>
                  <name>PDA</name>
                  <units>1</units>
                  <price>500</price>
                  <total>0</total>
                  <discount>0</discount>
            </item>

            <item>
                  <name>Java debugger</name>
                  <units>1</units>
                  <price>1500</price>
                  <total>0</total>
                  <discount>0</discount>
            </item>

      </items>

      <totals>
            <subtotal>0</subtotal>
            <tax>0</tax>
            <total>0</total>
            <rowcount>0</rowcount>
      </totals>

      <info>
            <tax>0.22</tax>
      </info>

      <payment>
            <as>credit</as>
            <cc/>
            <exp>2006-03-06</exp>
            <customernumber>123</customernumber>
      </payment>

</purchaseOrder>
```

XForms + XFDL Form

```
<?xml version="1.0" encoding="UTF-8"?>
<XFDL xmlns:custom=”http://www.ibm.com/xmlns/prod/XFDL/Custom”
xmlns:ev="http://www.w3.org/2001/xml-events"
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:xforms="http://www.w3.org/2002/xforms"
xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<globalpage sid="global">
      <global sid="global">

         <xformsmodels>
            <xforms:model>

               <xforms:instance src="purchaseorder.xml">
               </xforms:instance>

               <xforms:bind nodeset="/purchaseOrder/items/item/units"
               type="xsd:integer"></xforms:bind>

               <xforms:bind nodeset="/purchaseOrder/totals">
                  <xforms:bind
                  calculate="sum(../../items/item/total)"
                  nodeset="subtotal"></xforms:bind>

                  <xforms:bind calculate="round(../subtotal *
                  ../../info/tax)" nodeset="tax"></xforms:bind>

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

                  <xforms:bind calculate
                  ="sum(/purchaseOrder/items/item/units)"
                  nodeset="rowcount"></xforms:bind>
               </xforms:bind>

               <xforms:bind calculate=" ((../units * ../price)>1000)
               *(../units * ../price * 0.1)" nodeset
               ="/purchaseOrder/items/item/discount">

               </xforms:bind>

               <xforms:bind calculate="../units * ../price -
               ../discount" nodeset="/purchaseOrder/items/item/total"
               relevant="../units > 0"> </xforms:bind>



               <xforms:bind nodeset="/purchaseOrder/payment/as"
               relevant="/purchaseOrder/totals/rowcount > 0">

               </xforms:bind>

               <xforms:bind nodeset="/purchaseOrder/payment/cc"
               relevant="../as='credit'"> </xforms:bind>
```

```
                <xforms:bind nodeset="/purchaseOrder/payment/exp"
                relevant="../as='credit'" type="xsd:date">
                </xforms:bind>
                <xforms:bind nodeset
                ="/purchaseOrder/payment/customernumber"
                relevant="../as='cash'"> </xforms:bind>

                <xforms:submission action="file:data.xml" id="submit1"
                method="put" ref="/purchaseOrder" replace="none">
                </xforms:submission>

             </xforms:model>

          </xformsmodels>

      </global>
</globalpage>

<page sid="page1">

      <global sid="global">
            <label>Discount Form</label>
      </global>

      <label sid="label1">
            <value>Shop</value>
            <itemlocation>
                  <x>200</x>
                  <y>50</y>
            </itemlocation>
            <fontinfo>
                  <fontname>Helvetica</fontname>
                  <size>15</size>
                  <effect>bold</effect>
             </fontinfo>
      </label>

      <label sid="label2">
            <value>Number</value>
            <itemlocation>
                  <below>label1</below>
            </itemlocation>
      </label>

      <label sid="label3">
            <value>Name</value>
            <itemlocation>
                  <after>label2</after>
                  <width>105</width>
            </itemlocation>
      </label>

      <label sid="label4">
            <value>Price</value>
            <itemlocation>
                  <after>label3</after>
                  <width>45</width>
            </itemlocation>
      </label>
```

```
<label sid="label5">
      <value>Discount</value>
      <itemlocation>
            <after>label4</after>
      </itemlocation>
</label>

<label sid="label6">
      <value>Discounted Price</value>
      <itemlocation>
            <after>label5</after>
      </itemlocation>
</label>

<table sid="itemtable">
      <border>on</border>
      <xforms:repeat id="repeat1" nodeset
      ="/purchaseOrder/items/item">

            <field sid="field1">
                  <xforms:input ref="units">
                        <xforms:hint>The units of this
                        item</xforms:hint>
                  </xforms:input>
                  <itemlocation>
                        <width>50</width>
                  </itemlocation>
            </field>

            <field sid="field2">
                  <xforms:input ref="name">
                        <xforms:hint>The name of this item
                        </xforms:hint>
                  </xforms:input>
                  <itemlocation>
                        <width>100</width>
                        <after>field1</after>
                  </itemlocation>
            </field>

            <field sid="field3">
                  <xforms:input ref="price">
                        <xforms:hint>The price of this item
                        </xforms:hint>
                  </xforms:input>
                  <itemlocation>
                        <width>50</width>
                        <after>field2</after>
                  </itemlocation>
            </field>

            <label sid="label7">
                  <xforms:output ref="discount">
                  </xforms:output>
                  <itemlocation>
                        <width>50</width>
                        <after>field3</after>
                  </itemlocation>
            </label>

            <label sid="label8">
```

```
                  <xforms:output ref="total">
                  </xforms:output>
                  <itemlocation>
                        <width>50</width>
                        <after>label7</after>
                  </itemlocation>
            </label>

      </xforms:repeat>
      <itemlocation>
            <below>label6</below>
      </itemlocation>
</table>

<label sid="label9">
      <xforms:output ref="/purchaseOrder/totals/rowcount">
            <xforms:label>No. of items = </xforms:label>
      </xforms:output>
      <itemlocation>
            <below>itemtable</below>
            <offsety>20</offsety>
      </itemlocation>
</label>

<label sid="label10">
      <xforms:output ref="/purchaseOrder/totals/subtotal">
            <xforms:label>Subtotal = </xforms:label>
            <xforms:hint>Subtotal</xforms:hint>
      </xforms:output>
      <itemlocation>
            <below>label9</below>
      </itemlocation>
</label>

<label sid="label11">
      <xforms:output ref="/purchaseOrder/totals/tax">
            <xforms:label>Tax = </xforms:label>
            <xforms:hint>Tax</xforms:hint>
      </xforms:output>
      <itemlocation>
            <below>label10</below>
      </itemlocation>
</label>

<label sid="label12">
      <xforms:output ref="/purchaseOrder/totals/total">
            <xforms:label>Total = </xforms:label>
            <xforms:hint>Total</xforms:hint>
      </xforms:output>
      <itemlocation>
            <below>label11</below>
      </itemlocation>
</label>

<button sid="insert">
      <xforms:trigger>
            <xforms:label>Add item</xforms:label>
            <xforms:action ev:event="DOMActivate">
                  <xforms:insert at="index('repeat1')" nodeset=
                  "/purchaseOrder/items/item" position
                  ="after"></xforms:insert>
                  <xforms:setvalue ref=
                  "/purchaseOrder/items/item[index('repeat1')]/
                  price" value="'100'"></xforms:setvalue>
                  <xforms:setvalue ref=
                  "/purchaseOrder/items/item[index('repeat1')]/
                  units" value="'1'"></xforms:setvalue>

            </xforms:action>
      </xforms:trigger>

      <itemlocation>
         <below>label12</below>
         <offsety>20</offsety>
      </itemlocation>
</button>

<button sid="delete">
      <xforms:trigger>
            <xforms:label>Delete item</xforms:label>
            <xforms:action ev:event="DOMActivate">
                  <xforms:delete at="index('repeat1')" nodeset=
                  "/purchaseOrder/items/item"></xforms:delete>
            </xforms:action>
      </xforms:trigger>
      <itemlocation>
            <after>insert</after>
      </itemlocation>
</button>

<list sid="payment">
      <xforms:select1 appearance="compact" ref=
      "/purchaseOrder/payment/as">
            <xforms:label>Select payment method</xforms:label>

            <xforms:item>
                  <xforms:label>Credit card</xforms:label>
                  <xforms:value>credit</xforms:value>
            </xforms:item>

            <xforms:item>
                  <xforms:label>Bill me!</xforms:label>
                  <xforms:value>cash</xforms:value>
            </xforms:item>
      </xforms:select1>
      <itemlocation>
         <below>delete</below>
         <offsety>20</offsety>
      </itemlocation>
</list>

<field sid="field4">
      <xforms:input ref="/purchaseOrder/payment/cc">
            <xforms:label>Credit card number</xforms:label>
      </xforms:input>
      <itemlocation>
            <below>payment</below>
            <width>120</width>
      </itemlocation>
</field>
```

```
      <field sid="field5">
            <xforms:input ref="/purchaseOrder/payment/exp">
                  <xforms:label>Expiration date</xforms:label>
            </xforms:input>
            <itemlocation>
                  <below>field4</below>
                  <width>120</width>
            </itemlocation>
      </field>

      <field sid="field6">
            <xforms:input
ref="/purchaseOrder/payment/customernumber">
                  <xforms:label>Customer number</xforms:label>
            </xforms:input>
            <itemlocation>
                  <below>field5</below>
                  <width>120</width>
            </itemlocation>
      </field>

      <button sid="subm">
            <xforms:submit submission="submit1">
               <xforms:label>Buy!</xforms:label>
            </xforms:submit>
            <itemlocation>
                  <below>field6</below>
            </itemlocation>
      </button>

</page>

</XFDL>
```

New Code

```
<xforms>

      <model>
            <!-- Standard Namespaces included by default -->
            <data>purchaseOrder.xml</data>
      </model>

      <control>
            <calc path="purchaseOrder/items">
                  <!-- Relative Path starts with / -->
                  <var>/subtotal,/tax,/total,/rowcount</var>
                  <expr>sum(purchaseOrder/items/item/total),
                        round(purchaseOrder/totals/subtotal *
                        purchaseOrder/info/tax),
                        purchaseOrder/totals/subtotal +
                        purchaseOrder/totals/tax,
                        sum(purchaseOrder/items/item/units)
                  </expr>
            </calc>

            <calc path="purchaseOrder/items/item">
                  <cond>(/units * /price)>1000</cond>
                  <var>/discount</var>
                  <expr>
                        /units * /price * 0.1
                  </expr>
            </calc>

            <calc path="purchaseOrder/items/item">
                  <var>/total</var>
                  <expr>
                        /units * /price - /discount
                  </expr>
            </calc>

            <valid path="purchaseOrder/payment">
                  <var>/as,/cc,/exp,/customernumber</var>
                  <relevant>purchaseOrder/totals/rowcount &gt;0,
                            as='credit', as='credit', as='cash'
                  </relevant>
            </valid>

            <valid path="purchaseOrder">
                  <var>/items/item/units,/payment/exp</var>
                  <type>xsd:integer,xsd:date</type>
            </valid>

            <submission id="submit1" how="put" where="file:data.xml
            what="/purchaseOrder"/>

      </control>

      <view>

         <page id="page1" title="Discount Form">
            <label id="label1">
                  <value>Shop</value>
                  <fontinfo>
                  <fontname>Helvetica</fontname>
                        <size>15</size>
                        <effect>bold</effect>
                  </fontinfo>
            </label>

            <table border="on">
                  <repeat what="purchaseOrder/items/item"
id="repeat1">

                        <input>
                              <id>field1,field2,field3</id>
                              <label>Number,Name,Price</label>
                              <var>/uints,/name,/price</var>

                              <hint>The units of this item,
                                    The name of this item,
                                    The price of this item
                              </hint>
                        </input>

                        <output>
                              <id>label7,label8</id>
                              <label>Discount,Discounted
Price</label>
```

```
                <var>/discount,/total</var>
            </output>
        </repeat>
</table>

<output path="purchaseOrder/totals">
        <id>label9,label10,label11,label12</id>
        <var>/rowcount,/subtotal,/tax,/total</var>
        <label>No. of items = ,Subtotal = ,Tax = ,Total =
        </label>
        <hint>,Subtotal,Tax,Total</hint>
</output>

<trigger id="insert">
        <label>Add item</label>
        <action event="DOMActivate" path=
        "purchaseOrder/items">
                <insert what="/item" at="index('repeat1')" />
                <setvalue what=
                "/item[index('repeat1')]/price"
                value="'100'"></setvalue>
                <setvalue what=
                "/item[index('repeat1')]/units"
                value="'1'"></setvalue>
        </action>
</trigger>

<trigger id="delete">
        <label>Delete item</label>
        <action event="DOMActivate">
                <delete what="purchaseOrder/items/item"
                at="index('repeat1')" />
        </action>
</trigger>

<select type="one" id="payment" what=
"purchaseOrder/payment/as" diplayType="list">
        <label>Select payment method</label>
        <items>
                <value>credit,cash</value>
                <label>Credit card,Bill Me</label>
        </items>
</select>

<input path="purchaseOrder/payment">
        <id>cc,exp,cust</id>
        <var>cc,exp,customernumber</var>
        <label>Credit card number,Expiration date,Customer
        number </label>
</input>

<submit id="subm" what="submit1" >
        <label>Buy!</label>
</submit>

<layout>
        <item id="label1" coord="(200,50)"/>

        <items>
                <id>field2,field3,label7,label8</id>
                <after>field1,field2,field3,label7</after>
```

```
                              <width>100,50,50,50</width>
                        </items>

                        <items>
                              <id>itemtable,label9,label10,label11,label12,
                              insert,payment,subm</id>
                              <below>label1,itemtable,label9,label10,
                              label11,label12,delete,field6</below>
                              <offsety>0,20,0,0,0,20,20,0</offsety>
                        </items>

                        <item id="delete" after="insert"/>

                        <items>
                              <id>field1,field4,field5,field6</id>
                              <below>label1,payment,field4,field5</below>
                              <width>50,120,120,120</width>
                        </items>

                  </layout>

            </view>
</xforms>
```

The above code is much simpler and shows about 50% reduction in length.

data.xml (The file to which data is submitted)

```
<purchaseOrder xmlns="" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
   <items>
      <item>
         <name>Browser</name>
         <units>1</units>
         <price>100</price>
         <total>100</total>
         <discount>0</discount>
      </item>
      <item>
         <name>PDA</name>
         <units>1</units>
         <price>500</price>
         <total>500</total>
         <discount>0</discount>
      </item>
      <item>
         <name>Java debugger</name>
         <units>1</units>
         <price>1500</price>
         <total>1350</total>
         <discount>150</discount>
      </item>
   </items>
   <totals>
      <subtotal>1950</subtotal>
      <tax>429</tax>
      <total>2379</total>
      <rowcount>3</rowcount>
   </totals>
   <info>
      <tax>0.22</tax>
   </info>
   <payment>
      <as>credit</as>
      <cc></cc>
      <exp>2006-03-06</exp>

   </payment>
</purchaseOrder>
```

Another form was simplified, taken from the sample forms provided with the IBM Viewer v2.6. It consists of 2 pages. This is a very complicated form and has been simplified using our design. But due to limitation of space, we are not able to provide the codes here. The original code was about 48 pages in length and the modified code is 28 pages, a **reduction of about 42%.** This is the only measure that can be provided here.

7. CONCLUSION AND FUTURE WORK

Forms are an important part of the Web, and they continue to be the primary means for enabling interactive Web applications. Web applications and electronic commerce solutions have sparked the demand for better Web forms with richer interactions. HTML Forms are no more sufficient to fulfill the requirements for today’s forms. XForms is the response to the demand for more sophisticated forms, and provides a new platform-independent markup language. But, XForms is difficult to use and this is one of the major reason for forms still being coded using HTML + scripting. As form developers already know some form of scripting, they prefer to design forms in this traditional way.

XForms needs to be made simpler and developer-friendly for it to become popular. We have tried to achieve this in this thesis. As XForms requires a presentation language, after much research on various presentation options, we found XFDL as the most suitable option for e-commerce and other sophisticated forms. But, we found a scope for simplification in this language as well.

The modifications made include – seperation of the data and control parts in XForms, provision for default presentation options for XForms + XFDL forms, seperation of layout information from the items of the form, improvisation in the table tag, combining similar tags into one by using the comma operator. The above modifications make the code simpler to understand and write, and smaller in size.

A converter is designed which converts our modified code into XForms + XFDL code. The resultant form can be displayed using the IBM Workplace Forms Viewer. The future work includes design of an engine which can support the modified code. That is, instead of converting to the XFDL code, an independent engine can support this language.

Also, XHTML can be simplified along the same lines. XHTML must also include the concept of defining some features and reuse it again and again in the form. This concept is supported by CSS (Cascading Stylesheets) but CSS is a non-XML standard. The requirement is to have a XML version for CSS which can be used with XHTML.

APPENDIX A

Mapping Tables

Following are the tables mapping the modified language **with XForms + XFDL languages :**

General Features

Purpose	**XForms + XFDL**	**Our Language**
XML Based	Yes	Yes
Root Element	<XFDL>	<xforms>
Default Namespaces	No, Namespaces must be explicitly defined <XFDL xmlns= "http://www.ibm.com/xmlns/prod/XFDL/7.0" xmlns:xforms= "http://www.w3.org/2002/xforms" xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom" xmlns:ev="http://www.w3.org/2001/xml-events" xmlns:xsd= "http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">	Yes, these namespaces are defined by default.
Other Namespaces	<XFDL xmlns:prefix=”URI” ……….>	<xforms xmlns:prefix=”URI”>
globalpage	<globalpage sid="global">	XXX
Global options	<global sid="global">	<global>
Identfier	sid (scope identifier)	id (identifier)
Page definition	<page>	<page>
Seperation of common information	XXX	<global> <common tag id=”id”> common info

reused using **use**		</common tag> </global> <common tag use=”id”/>
actions	Xforms Actions	Same
functions	XForms + XPath + XFDL Functions	Some additional functions namely concat, str, max, min, atan2, avg, count, and some financial functions
events	XML Events	Same
Custom options	Allowed	Allowed, namespace for custom items is by default “custom”

Model and Control

Purpose	**XForms + XFDL**	**Our Language**
XForms Model	<xformsmodels> (All models within this tag)	XXX
To specify a data model	<xforms: model id=”modelId” schema =”filename” ………> <xforms:instance src=”filename”/> Or <xforms:instance> ….Instance data… </xforms:instance> <xforms:schema> Schema </xforms:schema> . …Control part … </xforms:model>	<model id=”modelId” schema =”filename”> <data>filename Or ….Data…. </data> <schema> filename or xml schema </schema> </model>
To specify instance	<xforms:instance>	<data>
control	<xforms:model ….. events=” ” function=”QName”> <xforms:bind nodeset=”Path” calculate or property …. > </xforms:bind> <xforms:submission …. /> <xforms:action> …….. </xforms:action> </xforms:model>	<control events=” “ function=”QName”> <calc> calculations </calc> <valid> validations </valid> <submission ….. /> <action> …….. </action>

calculations	<xforms:bind nodeset=”Path” calculate=”expr”/>
	<xforms:bind 2 ….
	Separate binds <xforms:bind>
		<calc>
		    <var> var1,var2 … </var>
		    <expr> expr1,expr2...
		    </expr>
		</calc>

validations	<xforms:bind nodeset=”Path” relevant or another property …. =”condition”/>
	<xforms:bind 2 ….
	Separate binds <xforms:bind>
		<valid>
		    <var> var1,var2 … </var>
		    <any property ....> expr1,expr2...
		    </any property>
		</valid>

submission	<xforms:submission id=”id” action= “file or URI” ref=”what to submit “ method= “http method (get|put|post)” ……. />	<submission id=”id” where= ”file or URI” what=” ” how=”method” ………. />
Submission of encrypted data	No, data submitted as xml is visible to all	Yes, <submission id=”id” encryption= ”encryption algo” ………. />

View (Form Controls)

Purpose	XForms + XFDL	Our Language
Input (field)	<field sid=”id”> <xforms:input ref=”Path” ……. > <xforms:label>label </xforms:label> help|hint|alert|action </xforms:input> </field>	<input id=”id” what=”Path” ……. help|hint|alert|action|label > </input>

More than one input	<field sid=”id1”>
	<xforms:input ref=”Path” ……. >
	    <xforms:label>label
	    </xforms:label>
	    help|hint|alert|action
	
	</xforms:input>
	</field>
	<field sid=”id2” ….. same as above>
	</field>
		<input ……. >
		    <id> id1,id2 </id>
		    <what> Path1 </what>
		    <help|hint|alert|action|label>
		</input>

Default Enclosure	No, eg- field must always skin input	Yes

secret	<field sid=”id”> <xforms:secret ref=”Path” …… > help|hint|alert|action|label </xforms:secret> </field>	<input type=”secret” id=”id” what=”Path” ……. help|hint|alert|action|label > </input>
textarea	<field sid=”id”> <xforms:textarea ref=”Path” …… > …… </xforms:textarea> </field>	<input type=”textarea” id=”id” what=”Path” … > ……… </input>
output	<label sid=”id”> <xforms:output ref=”Path” …… > …… </xforms:output> </label>	<output id=”id” what=”Path” … > ……… </output>
select	<checkgroup sid=”id”> <xforms:select ref=”Path” …> <xforms:label>… </xforms:label> <xforms:item>... <xforms:label>… </xforms:label> <xforms:value>… <xforms:value> </xforms:item> <xforms:item>... <xforms:label>… </xforms:label> <xforms:value>… <xforms:value> </xforms:item> <xforms:select> </checkgroup>	<select displayType=”checkgroup” id=”id” what=”Path” > <label>… </label> <items> <label>label1,2,… </label> <value>value1,2… </value> </items> </select>
select1	<select1 …>	<select type=”one” …>
Set of items within the select clause	Itemset	items
Table heading	As separate labels before the table tag	Labels within the elements of the table
Insert/delete buttons	As separate buttons outside the table, have to be explicitly linked to the table through the ref attribute	Part of the table itself. Advantage : need not be linked to the table <trigger> <id>but1,but2</id> <type>add,delete</type>

		<label>……</label> </trigger>
Seperation of layout information	No, coordinate information present in each item	Yes, separate layout for each page
Location and size of items	<itemlocation> <x> x-coordinate </x> <y> y-coordinate </y> <width> ….. </width> <height> …. </height> ………….. </itemlocation> within each item	<layout> <items> <id> id1, id2 ….</id> <coord>(x1,y1), (x2,y2) , ... </coord> <width> … , … </width> <height> … , …</height> </items> ………… </layout> at the end of page
coordinates	<x> x -coord </x> <y> y-coord </y>	<coord> (x,y) </coord>

REFERENCES

XML

[1] Aaron Skonnard, Martin Gudgin ,"Essential XML Quick Reference, A Programmer's Reference to XML, XPath,XSLT, XML Schema", Addison – Wesley Publication

[2] Tim Bray, October, 2000, "Extensible Markup Language (XML) 1.0 (Second Edition)", available at http://www.w3.org/TR/REC-xml .

[3] Tim Bray 1998, 2000, "Namespaces in XML", available at http://www.w3.org/TR/REC-xml/-names.

[4] developerWorks, August 2002, "Introduction to XML".

[5] Erik T. Ray ,"Learning XML, 2nd Edition", published by O'Reilly.

[6] Elliotte Rusty Harold and W. Scott Means*, "XML in a Nutshell, 2nd Edition",* published by O'Reilly.

XForms

[7] Micah Dubinko, "O'Reilly XForms Essentials"

[8] "XForms1.0, W3C Recommendation,14Oct,2003 ",available at http://www.w3.org/TR/2003/REC-xforms-20031014/

[9] Steven Pemberton, "XForms for HTML Authors", W3C Submission, 28 October 2003, available at www.w3.org/MarkUp/Forms/2003/xforms-for-html-authors.html

[10] Steven Pemberton, "XForms for HTML Authors", W3C Submission,2006-08-08, available at http://www.w3.org/MarkUp/Forms/2006/xforms-for-html-authors-part2.html

[11] XForms - The Next Generation of Web Forms, W3C Submission, available at www.w3.org/MarkUp/Forms/

[12] Richard Cardone, Danny Soroker, Alpana Tiwari, "Using XForms to Simplify Web Programming."

[13] Steven Pemberton, "XForms Quick Reference", W3C Submission, available at www.w3.org/MarkUp/Forms/2006/xforms-qr.html

XFDL

[14] "XFDL Specification", IBM Workplace Forms, version 2.7.

[15] J. Boyer, T. Bray, & M. Gordon, "Extensible Forms Description Language (XFDL) 4.0", W3C Note, available at: http://www.w3.org/TR/NOTE-XFDL

[16] Barclay T. Blair and John Boyer , "XFDL: Creating Electronic Commerce Transaction Records Using XML"

[17] John Boyer, "Enterprise-level Web Form Applications with XForms and XFDL", IBM Corporation, November 2005

Parser

[18] Eric Armstrong, "Working with XML - The Java API for Xml Parsing (JAXP) Tutorial", [Version 1.1, Update 31 -- 21 Aug 2001]

[19] Xerces Parser, available at http://xerces.apache.org

[20] developerWorks, July 2003, "Understanding DOM".

[21] Brett McLaughlin , "Java and XML, 2nd Edition".

[22] Doug Tidwell , "XML programming in Java technology, Part 1,2,3".

[23] LeHors, Arnaud, "Document Object Model (DOM) Level 2 Core Specification", available at http://www.w3.org/TR/DOM-Level-2-Core/ ,1999