ELSEVIER

Formsheets and the XML forms language

Anders Kristensen *

HP Labs (Bristol), Filton Road, Bristol, BS34 8QZ, UK

Abstract

This paper presents XForm — a proposal for a general and powerful mechanism for handling forms in XML. XForm defines form — related constructs independent of any particular XML language and set of form controls. It defines the notion of *formsheets* as a mechanism for computing form values on the client, form values being arbitrary, typed XML documents. This enables a symmetrical exchange of data between clients and servers which is useful for example for database and workflow applications. Formsheets can be written in a variety of languages — we argue that the document transformation capabilities of XSL stylesheets make them an elegant choice.

Keywords: XML; Form; Formsheet; XForm; XSL

1. Introduction

HTML defines a number of elements which taken together allows authors to construct *forms* — elements which can be used to solicit input from a user [13]. HTML forms have proven themselves extremely useful and after hyperlinking must be said to be the most important way of performing user interaction on the Web. With XML becoming the standard format for data exchanged between applications on the Web it is interesting to reconsider what forms are and how they might work in the context of XML [6]. This paper proposes a notion of XML forms which is quite different from HTML forms which we shall call *XForms*.

1.1. HTML forms

The following brief description of HTML forms is largely taken from [13]. A form in HTML is an instance of the `form` element. It is a part of a document which contains normal markup as well as a set of special elements called form *controls* (text fields, checkboxes, menus, etc.). Users interact with the form through its controls, completing the form before submitting it to a remote entity, typically a Web server, for processing.

Controls have a *name*, an *initial value*, and a *current value*, each of which is a text string. The name is given by the `name` attribute while the initial value of most controls may be specified with the control element's `value` attribute. The current value is first set to that of the initial value but may thereafter be modified through user interaction and scripts. The *form value* or *form data set* is a set of name-value pairs corresponding to the names and current values of some of its controls. This is what

* E-mail: ak@hplb.hpl.hp.com

is sent to the server when a form is submitted for processing.

1.2. XML forms

Where HTML defines a specific vocabulary for forms, e.g. `form`, `input`, `button`, etc. elements, XML is a general syntax — a language for defining languages — and as such doesn't have a built-in notion of forms. As XML works at a different level than HTML it is not surprising to find that XML forms should operate differently from HTML forms, in particular it would be nice if the abstract notion of what a form is could be made to apply to different concrete applications of XML. The forthcoming XML based version of HTML [12] is an important such application but by no means not the only one.

The proposal in this paper is to define a *generic* mechanism for how to do forms in XML. There are several parts to the proposal:

Form recognition:
Has to do with how form elements are recognized. We define the syntax used to assert form existence and to describe form characteristics in documents. This mechanism is similar in style to that used by the XML Linking Language [10] in that it doesn't define an XML language per se but rather syntax which can be used in conjunction with a variety of languages (in SGML parlance the form definition is like an *architectural form*).

Form values:
Form values are themselves XML data sets. In the simplest case a form value is just an XML encoded set of name-value pairs. This differs from HTML form data sets mainly in allowing for form fields with structured values. More interesting, we introduce the notion of *formsheets* to denote the specification of how to construct the value of a form upon submission. The formsheet has access to the document and form state, or *current value*, and controls the construction of the corresponding form value. The proposal allows for multiple formsheet languages to be used. We show how the document transformation capabilities of the Extensible Stylesheet Language (XSL) makes it a prime candidate [7], but using traditional programming languages is also an option.

Form submission:
Works much like for HTML forms, but is defined in terms of XLink and hence is more general.

Typed input fields:
It would be very useful for forms to be more intelligent about the kind of data they are soliciting. We discuss what sort of typing system could be associated with form input fields, and present the XForm use of XML namespaces to allow user agents to provide a form with client-specific default values.

The XForms proposal doesn't define the actual elements making up a form — the form controls — neither does it specify the user interaction behaviour and semantics of such elements. This is left to XForm compatible XML languages. Examples are given in a (hypothetical) 'wellformed HTML' application of XML, using the HTML form elements but in an XForm compliant manner.

We believe that specifying the core properties of forms independently of specific data and layout elements is a big advantage as it means that the same basic mechanism can be used regardless of the exact nature of the XML language at hand. This is analagous to how linking, stylesheets, and scripting are defined independently of the languages that use them, and this approach generally leads to better and more modular standards.

The rest of this paper is organized as follows. The XForm subjects mentioned above: form recognition, data set construction, submission, and typed form controls, are discussed in sections Sections 2–5. Section 6 shows how workflow-like applications can be built using XForms and other XML technologies, and Section 7 discusses related work.

2. Form recognition

Analogous to the workings of XLink, the existence of a form is asserted by the presence of a *form element*. An XForm aware processor recognizes an element as asserting the existence of a form, by looking for the presence of a designated attribute named `xf:form`, where `xf` denotes the XForm namespace identified in this paper by the URL `http://www.w3.org/TR/XForm` (XML namespaces are defined in [5]). Any element can be used to assert form pres-

ence by including this attribute. However, a particular XForms compliant XML language may choose to single out certain elements as being form elements, e.g. by defining *fixed* or default attribute values in a document type definition, thus avoiding having to list them in document instances themselves.

The XForm attributes are defined as a set of parameter entities which can be included in custom DTDs:

```
<!ENTITY % xform-core.att
   "form      CDATA  #REQUIRED
   attributes CDAT   #IMPLIED"
>
```

The `xf:form` attribute has three defined values:

`global`: indicates that the element is a form element and that the form value is computed by executing a formsheet on the entire document at the time of submission.

`scoped`: also indicates presence of a form, but one for which the value is computed using the form element as the root, i.e. rather than executing the formsheet on the whole document it is executed only on the form element. This makes for simpler formsheets and corresponds to how HTML forms work.

`submit`: this value is used to signal that the element acts as an implicit reference to a containing scoped form element. Activating the element implies submitting the corresponding enclosing form.

So the difference between `scoped` and `global` forms is that the former gets its value only from descendant elements and its own attributes whereas the entire document can contribute to the value of the latter. `Submit` elements are allowed only within a lexically enclosing `scoped` form element and its effect is to trigger submission of that enclosing form. It is provided primarily as a mechanism for enabling a modular XML based version of HTML to be defined to be in conformance with XForms.

The typical use of forms as we know them from HTML is to gather input from a user, submit it to a server, and getting a reply back with a response document. In other words a form behaves exactly like a hyperlink except that it allows for the collection and submission of data from the user agent, and it allows for the use of HTTP request methods other than GET [9]. Hence we define a form to be an XML link with additional attributes specifying form-specific properties. In XLink terminology forms are usually inline links, i.e. the form element serves as one of its own resources (as does both links and forms in HTML), but we don't rule out the use of out-of-line forms. Similarly forms can be either *simple* or *extended* links.

The HTML `form` element corresponds to a simple, inline link. Allowing forms to be extended links permits functionality such as multi-ended links with additional information such as titles and roles. This enriches the notion of forms but has little impact on how they are defined or how they operate. Hence this is not discussed further.

2.1. Attribute remapping

Problems may occur with attribute names as XForm is applied to existing XML languages. An XForm compliant DTD may choose to remap XForm attribute names in the manner of XLink. The attribute `xf:attributes` consists of an even number of tokens. This list is interpreted as a set of pairs where the first element is the name of an XForms attribute and the second is the name it is mapped to. For example, to map `action` to `link` one would have:

```
<!ATTLIST form
  xf:form       CDATA  #FIXED "scoped"
  xf:attributes CDATA  #FIXED "action
                                  link">
```

3. Constructing form values

When submitting a form to a processing agent the user agent must first build the form value. XForm values are either sets of name-value pairs in some representation or they are arbitrary XML documents constructed by a formsheet. For name-value data sets we allow the representations defined for use in HTML and add an XML encoding. The second class of form values is much more general and flexible, and can be used to construct form values in any XML language.

Form value construction is effected by the following form element attributes:

```
<!ENTITY % xform-value.att
   "enctype   CDATA  #IMPLIED
   charset    CDATA  #IMPLIED
   formsheet  CDATA  #IMPLIED
   form-lang  CDATA  #IMPLIED
   result-ns  CDATA  #IMPLIED"
>
```

The `enctype` attribute specifies the MIME type of the form data set. For MIME-like transport protocols, this will appear in the `Content-Type` header of the form submission. If the `formsheet` attribute is not specified form value construction depends only on the `enctype` attribute, which MUST then be defined and take on one of the following defined values:

- `application/x-www-form-urlencoded`
- `multipart/form-data`
- `text/xml`
- `application/xml`

The first two are as defined for use in HTML [13,11] and the `xml` variants are described below.

3.1. Form values in a generic XML encoding

The MIME types `text/xml` and `application/xml` are defined as generic types for carrying XML encoded data [14]. The `application` subtype is defined to overcome constraints placed on character set encodings for `text` top-level MIME types. The two types are otherwise identical.

When a form doesn't specify a `formsheet` the corresponding data set is a set of name-value pairs. When `enctype` is one of the two `xml` subtypes this set is represented as a `map` element which consists of a sequence of named `item` elements, each of which represents a single form field:

XML DTD for encoding of name-value pairs

```
<!ELEMENT map (item*)>
<!ELEMENT item ANY>
<!ATTLIST item name CDATA #REQUIRED>
<!ATTLIST item href CDATA #IMPLIED>
<!ATTLIST item type CDATA #IMPLIED>
```

The name of the form field is given by the mandatory `name` attribute of item elements. The value of a form field is either available 'inline' as the contents of the corresponding item element or, if the `href` attribute is defined, 'out-of-line' as the contents of the resource identified by that URI. The resource may be transported as part of the same data unit, e.g. as a separate MIME bodypart, or may be remote.

Map items are considered unordered and there is no requirement that they have unique names. We shall refer to such data sets as being *XForm-map* encoded. This DTD has an XML namespace identified by the URL `http://www.ietf.org/XML/NS/map`.

An HTML form with input fields for *name*, *tel*, and *email* might then result in a XForm-map value of:

```
<?xml version="1.0"?>
<map xmlns="http://www.ietf.org/
                    XML/NS/map">
 <item name="name">Joe Bloggs</item>
 <item name="tel">+1-222-333-4444</item>
 <item name="email">joe@example.com
 </item>
</map>
```

The XForm `charset` attribute specifies the character set used in the XML encoded 'document' but field values can contain any characters whatsoever as long as the generating user agent obeys standard XML encoding rules.

When carried over a MIME-like transport, e.g. HTTP or Internet email, the `Content-Type` header field takes the value of the forms `enctype` attribute. When used with HTTP XForm-map encoded data sets are always carried in the request body, i.e. `method="GET"` which places data in the request-URI is not allowed.

3.1.1. Structured values

Field values are not limited to being simple text strings but can be any arbitrarily complex, but well-formed, XML structure. The following data set contains an XML encoded digital business card [8] together with other data items, all of which possibly originate from an online form:

```
<?xml version="1.0" encoding="utf-8"?>
<xm:map xmlns:xm="http://www.ietf.org/XML/NS/map"
        xmlns:vc="http://www.ietf.org/XML/NS/vCard">
```

```
  <item name="vcard">
    <vc:vCard version="3.0">
      <fn>Joe Bloggs</fn>
      <n><family>Bloggs</family><given>Joe</given></n>
      <tel>+1-22-333-4444</tel>
      <email>bloggs@example.com</email>
    </vc:vCard>
  </item>
  <item name="notify">yes</item>
  <item name="org-type">Software Development</item>
  <item name="heard-of">From a friend.</item>
</xm:map>
```

When a map associates a name with a value which is itself an XML element this element may be typed by giving it an XML namespace attribute. The vCard namespace identifier used in this example is fictional — none is currently specified. Namespaces are more suitable as a typing mechanism than DTDs or formal public identifiers as they were designed to apply to individual elements of a larger XML document, not necessarily to the document as a whole.

3.2. Constructing form values using formsheets

The encoding schemes discussed above are provided primarily for simplicity and backwards compatibility with HTML forms. The 'native' XForms method of constructing form values is the much more general idea of using formsheets. A formsheet specifies how to transform the source document into a form value. It is denoted by the `formsheet` attribute which is a URL. The `form-lang` attribute specifies the MIME type of the formsheet language and can be omitted when it can be inferred from the context, e.g. from an HTTP `Content-Type` header field.

The major advantage of formsheets is that they allow us to construct the form value as being *any* sort of XML document. This means it becomes possible to submit *typed* data rather than using 'stupid', generic encodings of form values. The typing mechanism used is that of MIME and XML namespaces. Form data is assigned a namespace identifier by the generating formsheet and the MIME type is given in the `enctype` attribute.

Fig. 1 illustrates how formsheets would typically be used in Web applications.

The client retrieves an XML document from a Web server and has it rendered using a stylesheet. The document contains a number of form input elements whose values changes in response to user input. When the form is submitted the formsheet executes on the 'current value' of the source document. This results in a 'data' XML document which is then submitted to the server.

In the situation where a stylesheet has been applied to the original document in order to construct a renderable XML flow-object tree (as is the case for XSL stylesheets), the formsheet operates on the formatting object tree rather than the original tree. This is so because form controls wouldn't necessarily be

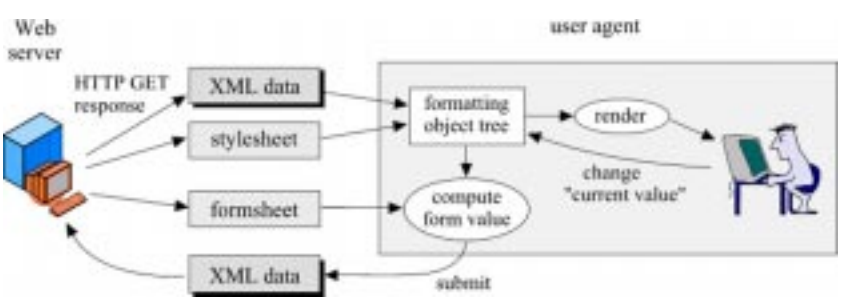

Fig. 1. Typical use of formsheets to produce XML form data.

present in the original document, but would often be added by the XSL tree transformation step. This is not an issue with stylesheet languages such as CSS which doesn't have transformational capabilites.

3.2.1. Example: using XSL as the formsheet language

The following example illustrates form value construction and submission behaviour for forms with formsheets expressed in XSL. XSL consists of two parts: a general-purpose transformation language which is used to map an XML source document into an XML target document, and a formatting vocabulary — itself an XML language — which is used to visually render a document. We use the transformation part to transform a document containing an XForm into the data document. The elements of the source document that contributes to the form data document are, by definition, the forms set of controls.

Suppose a server wishes to prompt a client for a user ID and a password. It can do this by using an HTML like form:

```
<form xf:form="scoped" action="/cgi-bin/logon" method="post"
      enctype="text/x-userid"
      formsheet="userid.xfl"
      form-lang="text/xsl">
  User ID: <input type="text" name="userid"/><br/>
  Password: <input type="text" name="passwd"/><br/>
  <button xf:form="submit">
</form>
```

Suppose we want to generate form data sets of the form:

```
<user>
  <id>aladdin</id>
  <password>open sesame</password>
</user>
```

The "userid.xfl" XSL formsheet could then be defined as follows (**boldface** text is passed through to the output uninterpreted by the XSL transformation engine):

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
   <xsl:template match="/">
    <user>
      <id>
        <xsl:value-of
          expr="/form/input[attribute(name)='userid']/attribute(value)"/>
      </id>
      <password>
        <xsl:value-of
          expr="/form/input[attribute(name)='passwd']/attribute(value)"/>
      </password>
    </user>
  </xsl:template>
</xsl:stylesheet>
```

This XSL formsheet consists of a single template rule whose pattern matches the form element of the source document. The result is the contents of the template element with the `xsl:value-of` elements substituted for the current values of the *userid* and *passwd* `input` fields. The form value is

submitted with an HTTP `Content-Type` header of `text/x-userid`. The XML form data may have a namespace identifier associated with it, although in this case it doesn't.

If for whatever reason we wanted the form value to be encoded as:

```
<user id="aladdin"
                password="open sesame"/>
```

doing so would be a simple matter of changing the `userid.xfl` formsheet. No changes are needed for the document itself.

When, as in this example, the form is declared to be `scoped`, the formsheet executes on the form element only, i.e. for the purpose of computing the form value this element is the document root. In contrast forms declared `global` executes on the whole document. This reflects the requirement that `scoped` forms contain all its constituent controls.

As is the case for stylesheets formsheets will typically be held as complete resources separate from the XML data they are applied to, but may also be embedded directly in documents. Embedded formsheets would have an `ID` attribute and would be referenced from form elements using '#' fragment identifiers in the `href` URL.

3.2.2. Using JavaScript as the formsheet language

An obvious alternative to using XSL transformations for computing form values is to use a scripting language together with the Document Object Model [1]. JavaScript, for example, is popular and is often present in Web user agents. Which language is preferred is largely a matter of taste. XSL is designed to do XML to XML conversions and hence is quite elegant for this type of application, whereas scripting languages are computationally complete, i.e. more powerful, and has the larger number of devotees at this time.

3.2.3. Benefits of formsheets

Using formsheets to control the encoding of form values has some significant benefits over the simple name-value data sets of HTML.

3.2.4. Symmetry

With HTML based Web applications servers send data to clients in HTML pages which contain both semantic and presentational markup. Clients passes form data to servers either as URL-encoded strings in the request URI or in the body of HTTP requests or as a MIME encoded multipart. With XML and XSL it is now possible for the server to send XML encoded data to the client without any presentational markup. XForms allows us to use the same data encoding to be used in the other direction — from client to server. What's more the data is *typed* using MIME media types and XML namespace information. A client can edit a database table record by record and have data shipped in identical formats in both directions.

Being able to pass data around using a single (semantic) encoding is not just a theoretical nicety, it means that we don't have to rely on special-purpose server-side processing in the form of CGI programs to make sense of the data. Standard components can make sense of it a priori. This makes it easier to string together applications (Web based or otherwise) as the data is self-describing. In a sense it helps dissolve the client-server distinction as content can flow in both directions using identical representations.

3.2.5. Generality

Formsheets operate on the source XML document and can construct arbitrary data sets. The generality of formsheets means that it is possible (and even straightforward) to construct a formsheet that performs the task of encoding the value of a form using the XForm-map encoding. In other words, the formsheet mechanism has the appealing property of having this other form data set encoding as a special case and the XForm-map representation can be formally defined as a specific XSL formsheet.

The power of formsheets doesn't come for free. Writing XSL transformation scripts is not trivial and it is not realistic to expect everyone to master it. The definition of form elements in a specific XML language might therefore choose to define the formsheet to have some default or *fixed* value.

4. Form submission

As in HTML a constructed form value is submitted to the processing agent using the protocol defined by the method attribute. The action attribute is the same as the href attribute in XLink and is the URI of

the server side resource to which the form value gets submitted.

```
<!ENTITY % xform-submission.att
   "action   CDATA  #IMPLIED
    method   CDATA  'post'"
>
```

5. Typed form controls

It is desirable to support some sort of typed data entry, i.e. to be able to specify certain types of constraints on data to be entered into forms. This would have at least two applications: it would allow user agents (UA) to check the validity of data entered before submitting it to a server, thus providing immediate feedback and generating less network traffic, and secondly it would potentially enable the UA to fill in known values automatically. A good example of this, as anyone who has spent much time on the Web will testify to, is name and address information prompted for by many Web sites as part of a registration procedure. Rather than having to repeatedly type in this information it would be convenient if browsers stored it and simply filled in the right form fields with this information as the default value, without ever submitting anything without explicit consent, of course, and only for form controls without a default value of their own. This is maybe particularly useful for information which is not easily entered via a keyboard, such as base64 encoded digital signatures or images; vCards, for example, optionally carries photos.

These two applications may require different kinds of typing mechanism. The former is much like types found in programming languages (strings, integers, booleans, etc.). Checking conformance to such datatypes could be done using scripting languages in conjunction with forms but relying on procedural verification for a problem that essentially calls for a declarative approach is not very elegant. Also it doesn't allow UAs to use specialized user interfaces for types such as dates.

The approach taken in XForm is to use the datatypes specified in the Document Content Description framework for denoting 'basic' datatypes [4]. The DCD also allows constraints like maximum and minimum values to be specified. The following is an example of an input field using the DCD dt attribute (short for *datatype*) for requesting an integer value between 0 and 100:

```
<input name="interest" DCD:dt="int"
    DCD:min="0" DCD:max="100" />
```

The other application — filling in default values for frequently used data — must in some way deal with application semantics. Again we use namespaces, but now rather than referring to datatypes defined by a single XML language we need to be able to refer to elements of *any* language. We define an XForm dt attribute for this purpose. The value of this attribute consists of a namespace identifier followed by the name of an element in the corresponding XML language:

```
<!ENTITY % xform-types.att
   "dt       CDATA       #IMPLIED"
>
```

So to continue the example of a server soliciting contact information from a UA, we note that digital business card information is addressed by the vCard specification which has defined an XML encoding [8]. A form requesting individual elements of a vCard structure might look like:

```
<h:html xmlns:h="http://www.w3.org/TR/REC-html40"
        xmlns:xf="http://www.w3.org/TR/XForm"
        xmlns:vc="http://www.ietf.org/Schemas/XML/vCard">
  <body>
    <form xf:form="scoped" action="/cgi-bin/vCard" method="post">
      <input name="firstname" xf:dt="vc:given"/>
      <input name="lastname"  xf:dt="vc:family"/>
      <input name="tel"       xf:dt="vc:tel" vc:tel.type="WORK"/>
      ...
```

```
    </form>
  </body>
</h:html>
```

while a form prompting for an entire vCard might look like this:

```
<form xf:form="scoped" action="/cgi-bin/payment" method="post">
      <input name="person" xf:dt="vc:vCard"/>
      <input name="credit-card" type="text"/>
      <input name="exp-mnth" DCD:dt="int" min="1" max="12">
      <input name="exp-year" DCD:dt="int" min="1998" max="2010">
      ...
</form>
```

The former example is fairly simple to deal with for a UA as the specified datatypes, i.e. the `given`, `family`, and `tel` vCard elements, are defined to contain flat textual data only, i.e. they cannot have child elements of their own. This means that a UA which 'knows about' vCards and is configured with vCard information for its user can initialize those form fields with the appropriate values, while UAs which doesn't know about vCards will just ignore the `dt` datatype specifications and thus degrade gracefully to their usual untyped behaviour. Note that the form author may include further attributes from the target datatype namespace. An example of this is the `vc:tel.type` attribute in the example. Within a `tel` element of a vCard the `tel.type` attribute further qualifies the content of that element. The UA may wish to take notice of such extra information but the semantics of these attributes are, of course, determined by the target datatype, not the XForm specification.

In the case where the form is requesting a complete vCard or any other element with its own internal structure, there is no simple way to degrade service gracefully in case the UA doesn't know the specified type, and 'knowing' the type may involve writing custom code to handle the user-interface for forms requesting those datatypes anyway, and so is more work. One idea would be to use the namespace identifier of unknown datatypes to attempt to retrieve a DTD or a DCD which could then be interpreted to construct a UI which would allow editing complex structures of that type. Another idea would be to define alternative input elements for UAs which does not support a particular datatype (in the style of the HTML NOFRAME element for use by UAs not supporting frames). Neither of these ideas seem very attractive and it may be better to rule out the use of structured datatypes, at least initially.

6. Workflow applications

The generality of formsheets means we have a lot of flexibility in how form values are constructed. In particular it is possible to reconstruct the original document completely or partially. This has some interesting and maybe surprising applications. One such application area is workflow-like messaging systems. Consider the following scenario.

Jack composes a memo and wants to send it to a number of people in turn. Each recipient gets to read it, add comments, and forward it to the next person in the chain. This can be accomplished using XForms in an XML document representing the memo. The memo has some text (plain or marked up) representing the actual message content. In addition it has a list of recipient names and email addresses and a list of comments made so far together with the names of the persons making them. Fig. 2 shows the XML encoded memo as it might look by the time it reaches Joe.

Joe's user agent will render the memo according to the stylesheet referenced in the document. It will display the `title`, `author`, and `to` elements along with the body and existing comments, and will add a form element at the bottom of the page where Joe can add his own comments. The `action` attribute of the form element contains a *mailto* URL which is the email address of the next recipient (Ann in this case). The formsheet associated with the form

```
<?xml version="1.0"?>
<?xml:stylesheet type="text/xsl"
                 href="http://example.com/workflow/memo.xsl"?>
<memo xmlns="http://example.com/workflow/memo.dtd">
    <head>
        <title>My Beautiful Memo</title>
        <author>Jack</author>
        <recipients>
            <to>ed@example.com</to>
            <to current="true">joe@example.com</to>
            <to>ann@example.com</to>
            <to>jack@example.com</to>
        </recipients>
    </head>

<body> This is the body of the text. This is what the recipients will see in
turn and have a chance to add their comments to. A comment in this example is
some unformatted text along with an identification of who made the
comment.</body>

    <comments>
        <comment by="ed@example.com">This is really good, Jack!</comment>
    </comments>
</memo>
```

Fig. 2. The memo document by the time it reaches Joe.

reconstructs the original structure shown in Fig. 2, only now marked with a new 'current' recipient and with an additional `comment` element. Form submission in this case means emailing the memo to the next recipient in the list. As Jack is himself the last recipient the memo will end up with him.

This example can be extended in a variety of ways. For example, the document could be an acquisition form retrieved from a Web server with a bunch of information to be filled out and a link for submitting it. When the submit link is traversed the document is send to a list of managers in turn for them to add comments and their (authenticated) sign-off. When the last signature has been added the request goes to the purchasing department and from there to an online order tracking system. In this example the formsheet may change underway, i.e. at some point the formsheet used to transition to the next step generates a document which will contain a form which uses a different formsheet.

All this can be achieved using just standard XML technologies: XML, XLink, XSL, and XForm together with standard transports such as HTTP and email. The construction of the formsheets that accomplishes it is not trivial but not hopelessly complicated either. Luckily it is not something everyone would have to do. Standard communication patterns like the ones above can be codified once and for all, but it's worth noting that as long as user agents implement the standard XML technologies listed above they will be able to take part in any new 'communication pattern' of this sort that anyone can come up with.

7. Related work

7.1. HTML

Earlier sections have already discussed some of the differences between HTML forms and XForms, the most important ones being the 'computability' and typing of XForm values. Here are some more.

Decoupling forms from form controls:

HTML defines a set of form controls. These are the UI elements (*widgets*) which interacts with the

user in order to accept values. Hence form controls always have a 'current value' which changes dynamically in accordance with user input. The addition of scripting to HTML potentially makes *all* elements dynamic, as a script may change content and attribute values of elements. HTML forms are still limited to submitting the value of form controls, though. XForms doesn't make the same distinction between form controls and other elements. The formsheet mechanism is general enough to allow submitted data to be assembled from all parts of a document. One could even imagine having XML documents with XForms without having any form controls in the HTML sense but still be able to capture (part of) the state of the document at some point in time and submit it.

Hidden fields:
This means, for example, that there is no need for special *hidden* fields (a form control which is not rendered and which contributes a fixed value to the form data set). This is so because *any* element can effectively act as a hidden field. The stylesheet in this case is written so as not to render the element while the formsheet will use its value. This is an improvement on HTML hidden fields as XForm hidden fields can be structured XML, i.e. they can have arbitrary attributes and child nodes of their own!

Controls need not be named:
Another consequence of using formsheets is that form controls need not have names, as form values are not necessarily name-value pairs.

Forms as links:
The fact that global forms are not associated with its form controls through lexical scoping means that the same set of form controls can be used by several forms. And any linking element can be extended to be a form element, so whereas HTML submit elements are buttons (or, less commonly, image maps), with XForms we can allow *any* element to trigger form submission. For example, wellformed HTML might allow `A` links to act as forms, a stylesheet language defining a *tabbed pane* element could submit an XForm data value when the user changed pane etc.

The following DTD approximates an XForm compliant definition of HTML forms. The parameter entities are defined in the HTML specification and some details (like scripting related attributes) have been left out for simplicity.

```
<!ELEMENT FORM (%block;)+>
<!ATTLIST FORM
  xf:form       CDATA        #FIXED  "scoped"
  xml:link      CDATA        #FIXED  "simple"
  action        %URI;        #REQUIRED
  method        (GET|POST)   GET
  enctype       CDATA        "application/x-www-form-urlencoded"
  formsheet     CDATA        #IMPLIED
  form-lang     CDATA        #IMPLIED
  charset       CDATA        #IMPLIED
  result-ns     CDATA        #IMPLIED
  additional XLink attributes
>
<!ELEMENT INPUT EMPTY>
<!ATTLIST INPUT
  xf:attributes CDATA         #FIXED  "form type"
  type          %InputType;   #CDATA
  name          CDATA         #IMPLIED
  value         CDATA         #IMPLIED
>
```

This definition largely allows backwards compatibility with HTML4.0 while allowing the use of formsheets and XForm-map encoded form values.

7.2. XFDL

A number of companies offer products in the area of digital, networked forms processing, but to our knowledge only one proposal exists for a standardized XML form language for use on the Web: the Extensible Forms Description Language [3].

The approach taken in XFDL is very different from that of XForms. XFDL is an XML application and defines a fixed set of form elements, structural markup, GUI display elements, and scripting capabilities all within the same language. The emphasis seems to be on defining a markup language (form controls and other markup) which allows for the construction of visually pleasing on-line forms and which is powerful enough to faithfully reproduce their paper-based equivalents. Additionally XFDL adds scripting functionality (for checking form values on clients) through it's own scripting language. It doesn't seem to address form value construction or typing.

In contrast the approach taken in designing XForms was to define forms processing separate from other pieces of the XML puzzle such as stylesheets, linking and scripting and make it as modular and generic as possible. XForms and XFDL operates at different levels of abstraction, and it would be possible to define an XForm compliant XML form language using the XFDL form controls and standard XML stylesheet and scripting support.

8. Conclusion

This paper has presented XForms-a proposal for what forms could be taken to mean in XML. The proposal is a radical rethinking of HTML forms and generalizes those, while ensuring the possibility of defining a 'wellformed' HTML DTD which is XForm compliant. The main contributions of this paper are the idea of using 'formsheets' to compute form values, the proposal for how to specify *typed* input fields, and the definition of a forms framework for XML which dissociates intrinsic properties of Web forms from any specific markup language and in particular from any specific set of form controls. This broadens the scope of XForms and makes the basic mechanisms applicable to input devices other than keyboards and mice.

Designing protocols to be modular and single-function has been a very successful design principle on the Internet. XForms is intended to address fundamental aspects of XML based forms processing and to fit in well with other XML technologies such as XML itself, XML namespaces, XSL, XLink, action sheets [2], and scripting languages.

References

[1] V. Apparao et al., Document Object Model (DOM) Level 1 Specification, W3C Recommendation, October 1998, http://www.w3.org/TR/REC-DOM-Level-1/

[2] V. Apparao, B. Eich, R. Guha and N. Ranjan, Action sheets: A modular way of defining behavior for XML and HTML, W3C Note, June 1998, http://www.w3.org/TR/NOTE-AS

[3] J. Boyer, T. Bray and M. Gordon, Extensible Forms Description Language (XFDL) 4.0, W3C Note, September 1998, http://www.w3.org/TR/NOTE-XFDL

[4] T. Bray, C. Frankston and A. Malhotra (Eds.), Document Content Description for XML, W3C Note, July 1998, http://www.w3.org/TR/NOTE-dcd

[5] T. Bray, D. Hollander and A. Layman (Eds.), Namespaces in XML, W3C Recommendation, January 1999, http://www.w3.org/TR/REC-xml-names/

[6] T. Bray, J. Paoli and C.M. Sperberg-McQueen (Eds.), Extensible Markup Language (XML) 1.0, W3C Recommendation, February 1998, http://www.w3.org/TR/REC-xml

[7] J. Clark and S. Deach (Eds.), Extensible Stylesheet Language (XSL), W3C Working Draft, August 1998, http://www.w3.org/TR/WD-xsl

[8] F. Dawson and P. Hoffman, The vCard v3.0 XML DTD, work in progress, November 1998, draft-dawson-vcard-xml-dtd-02.txt.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk and T. Berners-Lee, Hypertext Transfer Protocol — HTTP/1.1, RFC 2068, January 1997.

[10] E. Maler and S. DeRose (Eds.), XML Linking Language (XLink), W3C Working Draft, March 1998, http://www.w3.org/TR/WD-xml-link

[11] L. Masinter, Returning values from forms: multipart/form-data, RFC 2388, August 1998.

[12] S. Pemberton et al., XHTML 1.0: The Extensible HyperText Markup Language, W3C Working Draft, February 1999, http://www.w3.org/TR/WD-html-in-xml/

[13] D. Raggett, A. Le Hors and I. Jacobs (Eds.), HTML 4.0

Specification, W3C Recommendation, April 1998, http://www.w3.org/TR/REC-html40

[14] E. Whitehead and M. Murata, XML Media Types, RFC 2376, July 1998.

Anders Kristensen is a member of the research staff at Hewlett-Packard Laboratories in Bristol, U.K. His interests includes an array of WWW and Internet technologies, distributed systems, object-orientation, software development, design patterns, and framework design. Anders holds an M.Sc. degree in computer science and a B.Sc. in mathematics from Aarhus University, Denmark. Home page: http://www-uk.hpl.hp.com/people/ak/.