the DOMImplementation object. rewritten. Anyone able to write it up better? simplest from a specification point of view. Opinions? whatever, let me know. The idea is to make events bubble through this element and then up to the document, the bit about inserting into the document is just there to define it. I just want this easy to implement. I don't really care either way as far as the edge cases go here so long as they are defined, so let me know what you prefer, even if it is the status quo.

Proposed XHTML Module: XForms Basic

Working Draft 4 December 2003

This version:

http://www.hixie.ch/specs/html/forms/xforms-basic-1

Latest version:

http://www.hixie.ch/specs/html/forms/xforms-basic

Previous version:

http://lists.w3.org/Archives/Member/w3c-archive/2003Sep/att-0014/hfp.html

Editor:

Ian Hickson, Opera Software, ian@hixie.ch

© Copyright 2003 Opera Software.

Abstract

The HTML4 form semantics are extended with new types, new attributes for defining constraints, new DOM interfaces for validation, and new events for dependency tracking. A repeating model is introduced for declarative repeating of form sections. Mechanisms for XML submission and initialization of forms are added to the form submission model. HTML4, XHTML1.1 and the DOM are thus extended in a manner which has a clear migration path from existing HTML forms, leveraging the knowledge authors have built up with their experience with HTML so far.

Status of this document

This is a work in progress! This document is changing on a daily if not hourly basis in response to comments and as a general part of its

development process. Comments are very welcome, please send them to htmlforms@damowmow.com and cc www-archive@w3.org. Thank you.

It is very wrong to cite this as anything other than a work in progress. Do not implement this in a production product. It is not ready yet! At all!

This document currently has no official standing within the W3C at all. It is the result of loose collaboration between interested parties over dinner, in various mailing lists, on IRC, and in private e-mail. To become involved in the development of this document, please send comments to the address given above. **Your input will be taken into consideration.**

This is a working draft and may therefore be updated, replaced or rendered obsolete by other documents at any time. It is inappropriate to use Working Drafts as reference material or to cite them as other than "work in progress".

To find the latest version of this working draft, please follow the "Latest version" link above.

Table of contents

---

1. Introduction

The following features are considered requirements for this specification:

- Backwards compatibility (where possible).
- Ease of authoring for authors with limited knowledge about XML, data models, etc, but familiar with commonly used languages such as HTML and ECMAScript.
- Basic data typing, providing new controls for commonly used types.
- Validation on the client side (while recognizing that server side validation will still be required).
- Dynamically adding more fields (repeating structures) on the client side.
- XML submission.
- The ability to initialize forms from external data sources.

This specification is *not* a subset of XForms 1.0.

1.1. Relationship to HTML

This specification is an extension to [XHTML1]. It clarifies and extends the semantics put forth in [HTML4] for form controls and form submission. It is expected to be implemented in ordinary HTML user agents alongside existing forms technology, and indeed, some of the features described in this draft have been implemented by user agents as ad-hoc, non-standard extensions for many years due to strong market need.

1.2. Relationship to XForms

In general, [XForms] and this specification have different target audiences. XForms is aimed at the specialist form authoring world, for products that will not typically be sent over, or used as part of, the World Wide Web. On the other hand, this specification is expected to be implemented by general Web browsers, and used by Web authors targeting users that use those browsers. In addition, it is expected that this specification could be used to implement XForms 1.0 by transforming XForms documents into XHTML using this specification's features.

This specification attempts to add some of the functionality of XForms with a minimum impact on the existing, widely implemented forms model. Where appropriate, backwards compatibility, ease of authoring, and ease of implementation have been given priority over theoretical purity.

The following features of XForms have not been addressed:

- The separation of the instance data model, data typing, field interdependencies, and submission information from the content model and interface elements.
- The ability to create arbitrary XML fragments to be filled in before submission (although this may be possible by dynamically modifying the submission XML as the form is submitted).
- The option for field relationships, event handling, and other features to be implemented in a declarative fashion.
- The ability to edit local XML files directly. (While technically not defined by the XForms 1.0 specification, UAs have generally implemented such a feature since it is easy to extend the XForms model in that way.)

The majority of the features that XForms supports using declarative syntax are, in this specification, handled by using scripting. Some new interfaces are introduced to simplify some of the more tedious tasks.

1.3. Conformance requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Diagrams, examples, and notes are non-normative. All other content in this specification is intended to be normative.

This specification includes by reference the form-related parts of the HTML4, XHTML1.1, DOM2 HTML, DOM3 Core, and DOM3 Events specifications ([HTML4], [XHTML1], [DOM2HTML], [DOM3CORE], [DOM3EVENTS]). Compliant UAs must implement all the semantics of those specifications to claim compliance to this one.

1.4. Terminology

This specification refers to both HTML and XML attributes and DOM attributes, often in the same context. When it is not clear which is being referred to, they are referred to as **content attributes** for HTML and XML attributes, and **DOM attributes** for those from the DOM. CSS properties are just referred to as properties.

2. Extensions to form control elements

HTML `input` elements use the `type` attribute to specify the data type. In [HTML4], the types are:

`text`
: A free-form text input, nominally free of line breaks.

`password`
: A free-form text input for sensitive information, nominally free of line breaks.

`checkbox`
: A set of zero or more values from a predefined list (in the limiting case of the list only containing one value, this is equivalent to a boolean).

`radio`
: An enumerated value.

`submit`
: An enumerated value, with the extra semantic that it must be the last value selected and initiates form submission.

`file`
: An arbitrary file with a MIME type and optionally a file name.

`image`
: A coordinate, relative to a particular image's size.

Three other types, `reset`, `hidden`, and `button`, are also available, but are either not real data types, or not modifiable by the user, and so are not considered here.

In addition, HTML also provides a few alternate elements that convey typing semantics similar to the above types, but use different data models:

`select`
: An enumerated value, much like the `radio` type.

`select multiple`
: A set of zero or more values from a predefined list, much like the `checkbox` type.

`textarea`
: A free-form text input, nominally with no line break restrictions.

`button`
: A submit button with a richer content model than an `input` element button. (Can also be used for `reset` and `button` buttons.)

This specification includes these types, their semantics, and their processing rules, by reference. Compliant UAs must follow all the guidelines given in the HTML4 specification except those modified by this specification.

These types are useful, but limited. This section expands the list to cover more

specific data types, and introduces attributes that are designed to constrain data entry or other aspects of the UA's behaviour.

In addition to the attributes described below, some changes are made to the content model of HTML form elements to take into account scripting needs. Specifically, the `form`, `legend`, `select`, and `optgroup` elements may now be empty (in HTML4, those elements always required at least one element child, or, in the case of `legend`, at least one character of text).

Similarly, as controls no longer need to be contained within their `form` element to be associated with it, it is possible that authors would prefer to declare the page's forms in advance, in the `head` element of XHTML documents. This is therefore allowed, although only when the `form` element is empty.

> ***Note: The `form` element's `action` attribute is no longer required, and there is now an implied form for orphaned form controls. These changes are described in other parts of this specification.***

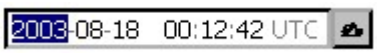
2.1. Extensions to the `input` element

Several new types are introduced for the `type` attribute. As with the older types, UAs are encouraged to show specialized widgets for these types, instead of requiring that the user enter the data into a text field.

The formats described below are those that UAs must use when submitting the data. They do not necessarily represent what the user is expected to type. It is the UA's responsibility to convert the user's input into the specified format.

`datetime`

A date and time encoded according to [ISO8601] with the time zone set to UTC, e.g.: `1995-12-31T23:59:59Z`. User agents are expected to show an appropriate widget.

> ***Note: This specification does not specify how the widget should appear. It could be something like this:***
>
> 
>

UAs may display the time in whatever time zone is appropriate for the user, but should be clear to the user that the time is globally defined, not time-zone dependent. The submitted date and time must be in the UTC timezone.

`date`

A date encoded according to [ISO8601], e.g.: `1995-12-31`. User agents are expected to show an appropriate widget, such as a calendar.

expdate

A date consisting of a year and a month encoded according to [ISO8601], e.g.: `1995-12`. This type is used most frequently for credit card expiry dates.

time

A time encoded according to [ISO8601] with no time zone, e.g.: `23:59:59`. User agents are expected to show an appropriate widget, such as a clock. UAs should make it clear to the user that the time does not carry any time zone information.

number

An arbitrary precision real number, written out with an optional minus sign ("-"), a decimal integer, a decimal point (".") and a decimal fractional part, together forming a number representing the base, followed optionally by the lowercase literal letter "e", another optional minus sign, and a decimal integer exponent representing the index of a power of ten with which to multiply the base to get the resulting number. For example, negative-root-two, to 32 significant figures, would be `-1.41421356237309504880168872420969e0`, while the radius of the earth given in furlongs would be `3.17053408e4`. This format is designed to be compatible with `scanf(3)`'s `%f` format and similar parsers while being easier to parse than required by some other floating point syntaxes. If the exponent part is omitted it is assumed to be zero. Note that `+0`, `0e+0`, `+0e0` are invalid numbers (the minus sign cannot be replaced by a plus sign for positive numbers, it must simply be dropped) and UAs must therefore not serialize numbers in those formats.

integer

An integer, written as a series of decimal digits optionally prefixed by a minus sign ("-").

email

An e-mail address, as defined by [RFC822] (the `mailbox` token, defined in RFC822 section 6.1).

uri

A URI, as defined by [RFC2396] (the `absoluteURI` token, defined in RFC2396 section 3).

To limit the range of values allowed by the above types, two new attributes are introduced, which apply to the date-related, time-related, and numeric types:

min

Gives the minimum value (inclusive) of the field, in the format specified for the relevant type. If absent, or if the value is not in the expected format, this should be treated as negative infinity (or the equivalent for the relevant type).

max

Gives the maximum value (inclusive) of the field, in the format specified for the relevant type. If absent, or if the value is not in the expected format, this should be treated as positive infinity (or the equivalent for the relevant type).

UAs must not submit forms that contain fields whose values do not match their types (ERROR_TYPE_MISMATCH) or whose values are outside the allowed range (ERROR_RANGE_UNDERFLOW, ERROR_RANGE_OVERFLOW).

Note that a field with a `max` less than its `min` can never be satisfied and thus would block a form from being submitted. This is not considered an error condition.

Note that fields may have no value, in which case they aren't "successful", do not take part in submission, and therefore do not need to have their values match their types (unless they are required fields).

Note: For more details on the meaning and relevance of the term "successful", see HTML4 section 17.3.2.

The following form uses some of the types described above:

```
<form action="..." method="post" onsubmit="verify(event)">
 <p>
  <label>
   Quantity:
   <input name="count" type="integer" min="0" max="99" value="1" />
  </label>
 </p>
 <p>
  <label for="time1"> Preferred delivery time: </label>
  <input id="time1" name="time1" type="time" min="08:00:00" max="17
  <input id="time2" name="time2" type="time" min="08:00:00" max="17
 </p>
 <script type="text/javascript">
  function verify(event) {
    // check that time1 is smaller than time2, otherwise, swap them
    if (event.target.time1 &gt;= event.target.time2) { // ISO8601 t
      var time2 = event.target.time2;
      event.target.time2 = event.target.time1;
      event.target.time1 = time2;
    }
  }
 </script>
</form>
```

Note: Servers should still perform type checking on submitted data, as malicious users or rogue user agents might submit data

intended to bypass this client-side type checking.

The `size` attribute of the `input` element is deprecated in favor of using CSS to specify the layout of the form.

2.2. Extensions to the `select` element

In addition to the new types above, the `select` element is extended to allow for a free-form input control which has author-specified auto-completion values.

`editable`
: This attribute specifies that the `select` element's value can be one other than those specified in the list of `option` elements.

For example, a Web page may wish to ask a user to enter a filename, offering some suggested names but allowing the user to enter other names as well:

```
<select name="filename" editable="editable">
  <option>My First Holiday</option>
  <option>untitled1.txt</option>
</select>
```

When this attribute is specified, the DOM `type` attribute ([DOM2HTML]) has the value "edit-one".

If both `editable` and `multiple` are specified, the UA must allow the user to enter multiple free-form values. This kind of interface is sometimes seen in mail user agents for their "To:" fields, for instance.

When both these attributes are specified, the DOM `type` attribute has the value "edit-multiple". The DOM `HTMLSelectElement` interface is extended to have a new attribute `values` of type `DOMStringList` ([DOM3Core]) that represents the currently selected or entered values.

In documents conforming to this specification, `select` elements need not have any `option` elements.

2.3. Extensions to the `textarea` element

The `rows` and `cols` attributes of the `textarea` element are no longer required attributes. When unspecified, CSS-compliant browsers should lay the element out as specified by CSS, and non-CSS UAs may use UA-specific defaults, such

as, for visual UAs, using the width of the display device and a height suitable for the device.

The `textarea` element may have a `wrap` attribute specified. This attribute controls the wrapping behaviour of submitted text.

`soft`
: This is the default value. The text is submitted without line breaks other than explicitly entered line breaks. (In other words, the submitted text is exactly as found in the DOM.)

`hard`
: The text is submitted with explicit line breaks, and in addition, line breaks added to wrap the text at the width given by the `cols` attribute. (These additional line breaks can't be seen in the DOM.)

Authors should always specify a `cols` attribute when the `wrap` attribute is set to `hard`. When `wrap="hard"` is specified without a `cols` attribute, user agents should use the display width when wrapping the text for submission. This will typically mean that different users submit text at different wrapping widths, defeating much of the purpose of client-side wrapping.

CSS UAs should *render* `textarea` elements as specified by the `'white-space'` property, although UAs may have rules in their UA stylesheet that key the default 'white-space' property values based on the `wrap` element for `textarea` elements.

2.4. Extensions to existing attributes

In addition to the new attributes given in this section, some existing attributes from [HTML4] are clarified below. These, and other attributes from HTML4, continue having the same semantics as described in HTML4 unless specified otherwise.

`accept`
: The file upload control (`<input type="file">`) uses the `accept` attribute to specify a comma-separated list of content types that a server processing the form will handle correctly. In this specification, this attribute is extended as follows:

  - MIME types may have a subtype of `*`, for example:

    ```
    <input type="file" name="avatar" accept="image/*"/>
    ```

    In this way, the `accept` attribute may be used to specify that the server is expecting an image, a sound clip, a video, etc, without specifying the exact list of types.

- UAs should use the list of acceptable types in constructing a filter for a file picker, if one is provided to the user.

- If the UA wishes to let the user create the file prior to upload, it should use the `accept` attribute's MIME type list to determine which application to use.

  > One recent use for sound file upload has been the concept of *audio blogging*. This is similar to straight-forward Web logging, or diary writing, but instead of submitting textual entries, one submits sound bites.
  >
  > The submission interface to such a system could be written as follows:
  >
  > ```
  > <form action="/weblog/submit" method="post">
  >  <label>
  >   Attach your audio-blog sound file:
  >   <input type="file" name="blog" accept="audio/*"/>
  >  </label>
  >  <input type="submit" value="Blog!"/>
  > </form>
  > ```
  >
  > A compliant UA could, upon encountering this form, provide a "Record" button instead of, or in addition to, the more usual "Browse" button. Selecting this button could then bring up a sound recording application.

  This is expected to be most useful on small devices that do not have file systems and for which the only way of handling file upload is to generate the content on the fly.

- UAs must not submit files that do not have a MIME type conforming to one of the MIME types listed as acceptable (ERROR_TYPE_MISMATCH). (UAs may, however, allow the user to override the MIME type to be one of the allowable types, e.g. if the file is originally incorrectly labeled.)

- If an `accept` attribute is set on a `form` element, it sets the default for any file upload controls in that form. (This is done by the file upload controls first checking their attribute, and if they don't have one, checking their form's.)

`action`

The `form` element's `action` attribute is no longer a required attribute. If omitted, the default value is the empty string, which is a relative URI pointing at the current document.

`disabled`

The `disabled` attribute applies to all control types, including `fieldset` (in HTML4 the `disabled` attribute did not apply to the `fieldset` element), except the `output` element.

`maxlength`

This attribute applies to `text`, `password` and `file` input types. In particular, it does not apply to the date-related, time-related, and numeric field types, or to the `email` or `uri` types. ***In HTML4, this attribute only applied to the `text` and `password` types.***

When this attribute is specified on `text` or `password` controls, the server must not submit a form with the control having more than the specified number of characters (ERROR_TOO_LONG). For details on counting string lengths, see [CHARMOD].

When specified on a file upload control, it specifies the maximum size in bytes of the content. UAs must not submit files bigger than this size (ERROR_TOO_LONG).

> ***Note: Servers must still expect to receive, and must be able to cope with, content larger than allowed by the maxlength attribute, in order to deal with malicious or non-conforming clients.***

`name`

Some names in this version of HTML forms have predefined meanings, allowing UAs to fill in the form fields automatically. These names, and their semantics, are described in [RFC3106].

`readonly`

This attribute applies only to `text`, `password`, `email`, `uri`, date-related, time-related, and numeric input types, as well as the `textarea` element. Specifically, it does not apply to radio buttons, check boxes, file upload fields, `select` elements (editable or not), or any of the button types; the interface concept of "readonly" values does not apply to button-like interfaces. (The DOM `readonly` attribute ([DOM2HTML]) obviously applies to the same set of types as the HTML attribute.)

Other attributes not listed here retain the same semantics as in [HTML4].

2.5. The `pattern` attribute

For the `text`, `email` and `uri` types of the `input` element, the `select` element when it has the new `editable` attribute set, and the `textarea` element, a new attribute, `pattern`, is introduced to specify patterns that the strings must match.

When specified, the `pattern` attribute contains a regular expression that the field's value must match before the form may be submitted (ERROR_PATTERN_MISMATCH).

```
<label> Credit Card Number:
 <input type="text" pattern="^[0-9]{10}$" name="cc" />
</label>
```

The regular expression language used for this attribute is the same as that defined in [ECMA262].

UAs must refuse to submit forms that contain fields whose values do not match their patterns.

In the case of the `email` and `uri`, the `pattern` attribute specifies a pattern that must be matched *in addition* to the value matching the generic pattern relevant for the field. If the pattern given by the attribute specifies a pattern that is incompatible with the grammar of the field type, as in the example below, then the field could never be satisfied. (A document containing such a situation is not technically invalid, but it is of dubious semantic use.)

```
<form>
 <p>
  This form could never be submitted, as the following required field
  can never be satisfied:
  <input type="uri" pattern="^[^:]+$" required="required" name="test"/>
 </p>
</form>
```

When the value doesn't match the field's type, a ERROR_TYPE_MISMATCH error occurs; when the value doesn't match the pattern, a ERROR_PATTERN_MISMATCH error occurs.

2.6. The `required` attribute

Form controls can have the `required` attribute specified, to indicate that the user must enter a value into the form control before submitting the form.

The `required` attribute applies to all form controls except check boxes, those with the type `hidden`, image inputs, buttons, `fieldset`s, and `output` elements. It *can* be used on controls with the `readonly` attribute set; this may be useful in scripted environments. For disabled controls, the attribute has no effect.

User agents must not submit forms that have form controls marked as required that do not have values (ERROR_REQUIRED). For radio buttons, exactly one radio button from each set must be checked.

> Here is a form fragment showing two required fields and one optional field.

> A user agent would not allow the user to submit the form until the "name" and "team" fields were filled in.

```
<ul>
  <li>Name: <input type="text" name="name" required="required" /></
  <li>Team:
    <select name="team" required="required">
      <option value="foxes">The Foxes</option>
      <option value="ferrets">The Ferrets</option>
      <option value="kittens">The Kittens</option>
    </select>
  <li>Comment: <input type="text" name="comment" /></li>
  </li>
</ul>
```

2.7. The `form` attribute

All form controls can have the `form` attribute specified. The `form` attribute gives the ID of the `form` element the form control should be associated with, and overrides the relationship between the form control and any ancestor `form` element.

Setting an element's `form` attribute either to a non-existent ID or to one that identifies an element that is not an HTML `form` element disassociates the form control from its form, leaving it unassociated with any form. On the other hand, an element whose `form` attribute is the empty string (as in `form=""`) is treated like an element with no `form` element ancestor (even if it has such an ancestor).

When set on a `fieldset` element, this also changes the association of any descendant form controls, unless they have `form` attributes of their own, or are contained inside forms that are themselves descendants of the `fieldset` element.

When forms are submitted, reset, or have their form controls enumerated through the DOM, only those controls associated with the form are taken into account. A control can be associated only with one form at a time.

A `form` attribute that specifies an ID that occurs multiple times in a document should select the same form as would be selected by the `getElementById()` method for that ID ([DOM3CORE]).

> In this example, each row contains one form, even though without this attribute it would not be possible to have more than one form per table if any of them span cells.

```
<table>
 <thead>
  <tr>
   <th>Name</th>
```

```
      <th>Value</th>
      <th>Action</th>
     </tr>
    </thead>
    <tbody>
     <tr>
      <td>
       <form id="edit1" action="/edit" method="post">
        <input type="hidden" name="id" value="1"/>
        <input type="text" name="name" value="First Row"/>
       </form>
      </td>
      <td>
       <input form="edit1" type="text" name="value"/>
      </td>
      <td>
       <input form="edit1" type="submit" name="Edit"/>
      </td>
     </tr>
     <tr>
      <td>
       <form id="edit2" action="/edit" method="post">
        <input type="hidden" name="id" value="2"/>
        <input type="text" name="name" value="Second Row"/>
       </form>
      </td>
      <td>
       <input form="edit2" type="text" name="value"/>
      </td>
      <td>
       <input form="edit2" type="submit" name="Edit"/>
      </td>
     </tr>
    </tbody>
   </table>
```

2.8. The `autocomplete` attribute

All form controls except the various push button controls and `hidden` and `output` controls, can have the `autocomplete` attribute set. The attribute takes two values, `true` and `false`. The default, when the attribute is not specified, is `true`.

A true value means the UA is allowed to store the value entered by the user so that if the user returns to the page, the UA can pre-fill the form. A false value means that the UA must not remember that field's value.

> Banks frequently do not want UAs to pre-fill login information:
>
> ```
> <p>Account: <input type="text" name="ac" autocomplete="false" /></l
> <p>PIN: <input type="text" name="pin" autocomplete="false" /></li>
> ```

Note: In practice, this attribute is required by many banking

institutions, who insist that UAs implement it before supporting them on their Web sites. For this reason, it is implemented by most major Web browsers already, and has been for many years.

2.9. The `inputmode` attribute

The `inputmode` attribute applies to the `input` element when it has a `type` attribute of `text`, `password`, `email`, or `uri`, to the `select` element when it has a the `editable` attribute set, and to the `textarea` element.

This attribute is defined to be exactly equivalent to the `inputmode` attribute defined in the XForms 1.0 specification (sections E1 through E3.2) [XForms].

2.10. The `help` attribute

Any form control can have a `help` attribute specified. This attribute contains a URI that the UA may use to provide help information regarding the active field.

This specification does not specify how help information should be used, but for example, the UA could show a small pop-up window if the user focuses such a control and pressed the `F1` key, or could show the help information in a side-bar while the relevant control is focused.

Note: This attribute is added mainly because XForms has it, to show that it would be trivial to add to HTML as well. However, there is some doubt that it is actually a useful feature. The XForms `hint` element is already supported in HTML, as the `title` attribute.

2.11. The `output` element

The `output` element acts very much like a `span` element, except that it is considered to be a form control for the purposes of the DOM. It has no attributes beyond the common attributes and the `form` attribute. Its value is given by its contents, which must be only text (like the `textarea` element). Its value can be set dynamically via the `value` DOM attribute, thus replacing the contents of the element.

The *initial value* of the `output` control is stored in a mutable `defaultValue` DOM attribute of type `DOMString`. This is similar to the way `textarea` elements work, except that the contents of an element for `output` controls reflects the *current value* not the initial, or default, value. See [HTML4] section 17.2 for the definiton of the term "initial value".

The `output` element is never successful for form submission. Resetting a form does reset its `output` elements.

> The following example shows two input fields. Changing either field updates an `output` element containing the product of both fields.
>
> ```
> <form>
>  
>   <input name="a" type="number" value="0"> *
>   <input name="b" type="number" value="0"> =
>   <output name="result" onforminput="value = a.value * b.value">0</
>  
> </form>
> ```
>
> This would work something like the following:
>
> 0 * 0 = 0
>
> ***Note: The `forminput` event is defined in the section on new events.***

2.12. The implied form for form controls with no form element

When a form control has no form element ancestor and no form attribute, or when the form attribute is set to the empty string, it is called an **orphan form control**. Orphan controls are bound to an anonymous `HTMLFormElement` DOM node that is associated with the document and has its `parentNode` set to the document node, but does not (initially) appear in the document. If it is inserted into the document by some evil QA engineer, or otherwise manipulated, it does not lose its role as guardian of the orphan form controls, although if then removed from the document again, its `parentNode` does not revert to the document node.

The implied form is only created when it is required. When created, it is added to the document's DOM's `forms` collection.

This form can be submitted like any other form; it partakes in all the normal event handling and so forth. Since by default forms have `action` attributes that point to the base URI (typically the document itself), and by default use the GET URI scheme, the obsolete `isindex` element can be replaced with the following more styleable alternative:

```
<p>Keywords: <input name="keywords"> <input type="submit"></p>
```

The only difference is that this would be submitted using `application/x-www-form-urlencoded` instead of simply causing the keywords to be appended as one query string.

2.13. Handling unexpected elements

There are several elements that are defined as expecting particular elements as children. Using the DOM, or in XML, it is possible for authors to violate these expectations and place elements in unexpected places.

Authors must not do this. User agent implementors may curse authors who violate these rules, and may persecute them to the full extent allowed by applicable international law.

Upon encountering such an invalid construct, UAs must proceed as follows:

For non-empty `input` elements

By default, the form control must replace the contents of the element in the rendering with the form control widget. Using CSS3 Generated Content [CSS3CONTENT] or XBL [XBL], however, it is possible for the author to override this behaviour.

For `output` elements containing elements

The `defaultValue` DOM attribute is initialized from the DOM3 Core `textContent` attribute ([DOM3CORE]). Setting the element's `value` attribute is defined to be identical to setting the DOM3 Core `textContent` attribute. While the element contains elements, they are rendered according to the CSS rules.

For `textarea` elements containing elements

The `defaultValue` DOM attribute is identical to the `textContent` DOM attribute both for reading and writing, and is used to set the initial `value`. The rendering is based on the `value` DOM attribute, not the contents of the element, unless CSS is used to override this somehow.

For `select` elements containing nodes other than `option` and `optgroup` elements, and for `optgroup` elements containing nodes other than `option` elements

Only the `option` and `optgroup` elements take part in the `select` semantics. Unless otherwise forced to appear by a stylesheet, other child nodes are never visible.

For `option` elements containing nodes other than text nodes

The value of the control, if not specified explicitly, is initialized using the `textContent` DOM attribute's value.

As far as rendering goes, it is left largely up to the UA. Two possibilities are sensible: rendering the content normally, just as it would have been outside the form control; and rendering the initial value only, with the rest of the content not displayed (unless forced to appear through some CSS).

Note: It should be noted that while nesting a form inside a

`select` control may look cool, it is extremely poor UI and must not be encouraged.

For `option` and `optgroup` elements that are not inside `select` elements
The elements should be treated much like `span` elements as far as rendering goes.

Other invalid cases should be handled analogously.

3. Repeating form controls

Occasionally forms contain repeating sections, for example an order form could have one row per item, with product, quantity, and subtotal fields. The **repeating form controls model** defines how such a form can be described without resorting to scripting.

Note: The entire model can be emulated purely using JavaScript and the DOM. With such a library, this model could be used and down-level clients could be supported before user agents implemented it ubiquitously. Creating such a library is left as an exercise to the reader.

3.1. Definitions

Note: In this section, a number of references are made to namespaces. For authors who are only using HTML or XHTML, the definitions below ensure that no namespaces need appear in the document (except the namespace on the root element). Thus, such a reader can simply gloss over the parts that mention namespaces.

In order to implement such a form declaratively, a new global attribute is introduced: the **`repeat`** attribute. When placed on elements in the `http://www.w3.org/1999/xhtml` namespace, it must be a namespace-free attribute, and when placed on other elements, it must be an attribute in the `http://www.w3.org/1999/xhtml` namespace.

The effect of this attribute depends on its value, which is a space-separated list of tokens with no leading spaces. The first token is mandatory and must be either `template` or `repeated`. If the attribute is present but either is empty, or starts with whitespace, or has a first token that is neither of these two values, it

has no effect.

The second token is optional, if present it must be an integer. If the second token is not an integer, it is ignored. The third token, and any subsequent tokens, if any are present, are always ignored.

3.1.1. Repetition templates

An element in the `http://www.w3.org/1999/xhtml` namespace with the `repeat` attribute in no namespace, or an element in any other namespace with the `repeat` attribute in the `http://www.w3.org/1999/xhtml` namespace, with the first token of the attribute's value equal to `template`, is a **repetition template**.

Repetition templates may occur anywhere. They are not specifically associated with any form.

Every template has an index associated with it. By default, the index has the value 0. However, if the aforementioned `repeat` attribute has a second token, and that token is an integer (one or more digits in the range 0-9, optionally with a leading minus sign), then the index is set to that value.

Unrecognized tokens must be ignored.

```
<div repeat="template 1 "/> <!-- A template, index set to 1. -->

<div repeat="template +1 3"/> <!-- A template, index set to 0
                                  (second and third tokens ignored). -->

<div repeat="template 2 3"/> <!-- A template, index set to 2
                                 (third token ignored). -->

<div repeat="invalid template "/> <!-- Not a template. -->

<div repeat=" template "/> <!-- Not a template (leading whitespace). -->
```

3.1.2. Repetition blocks

An element in the `http://www.w3.org/1999/xhtml` namespace with the `repeat` attribute in no namespace, or an element in any other namespace with the `repeat` attribute in the `http://www.w3.org/1999/xhtml` namespace, with the first token of the attribute's value equal to `repeated`, is a **repetition block**.

Repetition blocks should only occur as following siblings of repetition templates. If an element is declared as a repetition block but does not have a previous sibling that is a repetition template, then it can only take part in certain aspects of the repetition model (namely deletion and movement, and not addition). Such elements are termed **orphan repetition blocks**.

Every repetition block has an index associated with it. By default, the index is equal to the number of repetition blocks present in the DOM tree between the repetition block and its template (so the first block's index defaults to 0, the second to 1, and so forth). The index of orphan repetition blocks defaults to 0. However, if the repetition block's `repeat` attribute has a second token, and that token is an integer (one or more digits in the range 0-9, optionally with a leading minus sign), then the index is set to that value.

```
<div>
 <div repeat="template"/> <!-- The template for the next few elements.
 <div repeat="repeated"/> <!-- A simple repetition block, index 0. -->
 <div repeat="repeated -5"/> <!-- Another, index -5 -->
 <div repeat="repeated"/> <!-- A simple repetition block, index 2. -->
 <div repeat="nothing"/> <!-- Just a normal element. -->
 <div repeat="repeated"/> <!-- A simple repetition block, index 3. -->
</div>
<div repeat="repeated"/> <!-- Orphan repetition block, index 0. -->
```

3.2. New form controls

Several new button types are introduced to support the repetition model. These values are valid types for both the `input` element and the `button` element.

`add`
: Adds a new repetition block.

`remove`
: Removes the nearest ancestor repetition block.

`move-up`
: Moves the nearest ancestor repetition block up one.

`move-down`
: Moves the nearest ancestor repetition block down one.

These control types can never be successful.

In addition, to support the `add` type, a new attribute is introduced to the `input` and `button` elements: `template`.

`template`
: Specifies the repetition template to use.

These are described in more detail in the next section.

3.3. The repetition model

A repetition template should not be displayed. In CSS-aware user agents, this should be achieved by including the following rules, or their equivalent, in the UA's user agent stylesheet:

```
@namespace html url(http://www.w3.org/1999/xhtml);
:not(html|*)[html|repeat="template"],
:not(html|*)[html|repeat^="template "],
html|*[|repeat="template"],
html|*[|repeat^="template "] { display: none; }
```

Any form controls inside a repetition template are associated with their form's `templateElements` DOM attribute, and are *not* present in the form's `elements` DOM attribute, unless the relevant form is inside the template itself. Since only controls in the `elements` attribute can be successful, controls inside repetition templates that would be part of forms outside the template can never be successful and cannot be pre-filled directly when the form is pre-seeded. However, see the section on seeding a form with initial values for details on how repeating blocks can be pre-filled.

3.3.1. Addition

If an `add` button is activated, and it has a `template` attribute, and the element, in the same document, with the ID given by the `template` attribute in question, is a repetition template as defined above, then that element's template replication behaviour is invoked. (Specifically, in scripting-aware environments, the element's `addRepetitionBlock()` method is called with a null argument.)

If an `add` button is activated, and it has no `template` attribute, but the element has an ancestor that is a repetition block that is not an orphan repetition block, then the repetition template associated with that repetition block has its template replication behaviour invoked with the respective repetition block as its argument. (Specifically, in scripting-aware environments, the element's `addRepetitionBlock()` method is called with a reference to the DOM Element node that represents the repetition block.)

When a template's replication behaviour is invoked (specifically, when either its **`addRepetitionBlock()`** method is called or its **`addRepetitionBlockByIndex()`** method is called) the following is performed:

1. The template examines its following siblings, up to the next repetition template or the end of the block, whichever comes first. For each sibling that is a repetition block (as defined above), if the repetition block's index is greater than or equal to the template's index, then the template's index is increased to the repetition block's index plus one. The last repetition block examined will be used in a later step.

2. If this algorithm was invoked via the `addRepetitionBlockByIndex()` method, and the value of the method's index argument is greater than the template's index, then the template's index is set to the value of index argument.

3. A clone of the template is made. The resulting element is the new repetition block element.

4. If the new repetition block element is in the `http://www.w3.org/1999/xhtml` namespace, then the `repeat` attribute in no namespace on the cloned element has its value changed to `repeated`. Otherwise, the `repeat` attribute in the `http://www.w3.org/1999/xhtml` namespace has its value changed to `repeated`.

5. If this algorithm was invoked via the `addRepetitionBlockByIndex()` method, the new repetition block element's index is set to the method's index argument. Otherwise, the new repetition block element's index is set to the template's index.

6. If the new repetition block has an ID attribute (that is, an attribute specifying an ID, regardless of the attribute's namespace or name), then that attribute's value is used as the template name in the following steps. Otherwise, the template has no name. (If there is more than one ID attribute, the "first" one in terms of node order is used. [DOM3CORE])

7. If the template has a name, then, for every attribute on the new element, and for every attribute in every descendant of the new element, any occurrences of a string consisting of an underscore, the template's name, and another underscore, is replaced by the new repetition block's index. (For example if the template is called "order", and the new repetition block's index has the value 2, and one of the attributes of one of the descendents of the new repetition block is "comment._order_._comment_", then the attribute's value is changed to "comment.2._comment_".) This is performed without paying attention to the types of attributes, and is done to *all* descendants, even those inside nested forms, nested repetition templates, and so forth.

8. The attribute from which the template's name was derived, if any, is removed from the new repetition block element.

9. If the first argument to the method was null, or if the argument to the function does not designate a repetition block belonging to this repetition template, then the new element is inserted into the parent of the template, immediately after the last repetition block found in the first step above, or after the template itself if there were no such repetition blocks.

10. Otherwise, the new element is inserted into the parent of the template, immediately *before* the node passed as the method's argument.

11. The template's index is increased by one.

For an example, see the example section below.

3.3.2. Removal

If a `remove` button is activated, and the element has an ancestor that is a repetition block as defined above, then the nearest such ancestor's template deletion behaviour is invoked. (Specifically, in scripting-aware environments, the element's `removeRepetitionBlock()` method is invoked.)

When a repetition block's deletion behaviour is invoked (specifically, when its **`removeRepetitionBlock()`** method is called) the following is performed:

1. The node is removed from its parent, if it has one.

This occurs even if the repetition block is an orphan repetition block.

For an example, see the example section below.

3.3.3. Movement of repetition blocks

The two remaining button types, `move-up` and `move-down`, are used to move the current repetition block up or down the sibling repetition blocks.

If a `move-up` or `move-down` button is activated, and the element has an ancestor that is a repetition block as defined above, then the nearest such ancestor's template movement behaviour is invoked in the relevant direction. (Specifically, in scripting-aware environments, the element's `moveRepetitionBlock()` method is called; for `move-up` buttons the argument is -1 and for `move-down` buttons the argument is 1).

When a repetition block's movement behaviour is invoked (specifically, when its **`moveRepetitionBlock()`** method is called) the following is performed, where *distance* is an integer representing how far and in what direction to move the block (the argument to the method):

1. If *distance* is 0, or if the repetition block has no parent, nothing happens and the algorithm ends here.

2. Set *target*, a reference to a DOM Node, to the repetition block being moved.

3. If *distance* is negative: While *distance* is not zero and *target*'s `previousSibling` is defined and is not a repetition template, set *target* to this `previousSibling` and, if it is a repetition block, increase *distance* by one (make it less negative by one).

4. Otherwise, *distance* is positive: While *distance* is not zero and *target*'s

`nextSibling` is defined and is not a repetition template, set *target* to this `nextSibling` and, if it is a repetition block, decrease *distance* by one. After the loop, set *target* to *target*'s `nextSibling` (which may be null).

5. Call the repetition block's parent node's `insertBefore()` method with the `newChild` argument being the repetition block and the `refChild` argument being *target* (which may be null by this point).

This occurs even if the repetition block is an orphan repetition block.

Moving repetition blocks does not change the index of the repetition blocks.

In addition, user agents must automatically disable `move-up` buttons (irrespective of the value of the `disabled` DOM attribute) when their repetition block could not be moved any higher according to the algorithm above, and when the buttons are not in a repetition block. Similarly, user agents must automatically disable `move-down` buttons when their repetition block could not be moved any lower according to the algorithm above, and when the buttons are not in a repetition block. This automatic disabling does not affect the DOM `disabled` property.

3.4. Example

The following example shows how to use repetition templates to dynamically add more rows to a form in a table.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
 <head>
  <title>Form Repeat Demo</title>
  </head>
 <body>
  <form action="http://software.hixie.ch/utilities/cgi/test-tools/echo"
   <table>
    <thead>
     <tr>
      <th>Name</th>
      <th>Count</th>
      <th></th>
     </tr>
    </thead>
    <tbody>
     <tr repeat="template" id="row">
      <td><input type="text" name="name__row_" value=""></td>
      <td><input type="text" name="count__row_" value="1"></td>
      <td><input type="remove" value="Delete Row"></td>
     </tr>
     <tr repeat="repeated">
      <td><input type="text" name="name_0" value="John Smith"></td>
      <td><input type="text" name="count_0" value="1"></td>
      <td><input type="remove" value="Delete Row"></td>
```

```
      </tr>
     </tbody>
    </table>
    <p>
     <input type="add" value="Add Row" template="row">
    </p>
   </form>
  </body>
 </html>
```

Initially, one row would be visible, with two text input fields, one having the value "John Smith" and the other having the value "1".

If the "Add Row" button is pressed, a new row is added. The first such row would have the index 1 and so the controls would be named "name_1" and "count_1" respectively.

If the "Delete Row" button above is pressed, the row would be removed.

This example does not demonstrate nested repeat blocks, reordering repetition blocks, and inserting new repetition blocks in the middle of the existing sequence, all of which are possible using the facilities described above.

4. The forms event model

The following events are considered form events:

- {"http://www.w3.org/2001/xml-events", "change"}
- {"http://www.w3.org/2001/xml-events", "formchange"}
- {"http://www.w3.org/2001/xml-events", "input"}
- {"http://www.w3.org/2001/xml-events", "forminput"}
- {"http://www.w3.org/2001/xml-events", "formerror"}
- {"http://www.w3.org/2001/xml-events", "submit"}
- {"http://www.w3.org/2001/xml-events", "reset"}

Some of the above are mainly described in [DOM3EVENTS] and [HTML4]. This section introduces the new events and new semantics for the existing events.

4.1. Bubbling semantics

Since form controls no longer need to be descendants of their form elements,

the semantics of form events bubbling are changed slightly.

If a bubbling form-related event is targeted at, or bubbles into, a form control with a `form` attribute pointing to a valid form (which, if the attribute is the empty string, would be the implied form), it continues bubbling not at the node's parent, but at the specified form element.

For example, in the following document:

```
<application>
 <fieldset xmlns="http://www.w3.org/1999/xhtml" form="">
  <button> Test </button>
 </fieldset>
</application>
```

...clicking the button generates an event that is targeted on the button, bubbles up to the fieldset, is redirected straight to the anonymous implied form, and from there bubbles to the document (recall that the anonymous form node is parented to the document node by default). The root element never sees the event during the bubbling phase.

If a bubbling form-related event bubbles into the document node without passing through a `form` element, it is similarly redirected so that it continues bubbling at the implied form. (In the esoteric case where the implied form has been inserted into the document, this will cause the event to bubble into the document node twice.)

4.2. Scope resolution for ECMAScript in HTML event handler attributes

The scope chain for ECMAScript executed in HTML event handler attributes links from the activation object for the handler, to its `this` parameter (the event target), to the form, to the document, to the default view (the window).

The event handler is passed one argument, *event*, corresponding to the event object.

> ***Note: This definition is intentionally backwards compatible with DOM Level 0. See also ECMA-262 Edition 3, sections 10.1.6 and 10.2.3, for more details on activation objects. [ECMA262]***

4.3. Change events and input events

In [DOM3EVENTS] and [HTML4], the `change` event is fired on a form control element when the control loses the input focus and its value has been modified since gaining focus.

To address the need for more immediate feedback mechanisms, this specification introduces the **`input` event**. This event is fired on a control whenever the value of the control changes due to input from the user, and is otherwise identical to the `change` event. (For example, it bubbles, is not cancelable, and has no context information.)

4.4. Events to enable simpler dependency tracking

Sometimes form controls are inter-dependent. In these cases, it is more intuitive to specify the dependencies on the control whose value or attributes depend on another's, rather than specify which controls should be affected by a change on the element that changes. For this reason, two new events are introduced, **`formchange`** and **`forminput`**.

These events are in the same namespace as the other form events, do not bubble, cannot be canceled, have no context information, and have no default action.

The default action of a `change` event is to fire a `formchange` event at each element in the form's `elements` and `templateElements` lists, in document order, with the `elements` list processed first. Note that template controls *are* affected. This is important as it enables templates to be kept up to date so that when they are cloned, they already reflect the form's state.

The `input` element analogously invokes the `forminput` event on all the form's controls as its default action.

4.4.1. Declarative dependency tracking

Due to the way in which scope resolution is defined, a subset of ECMAScript that is completely declarative can be specified for dependencies. Doing so is left as an exercise to the reader.

> In the following example, the text field is only enabled if the checkbox is checked.
>
> ```
> 
>  <label>
>   <input type="checkbox" name="subscribe">
>   Subscribe to daily newsletter
>  </label>
> 
> 
>  <label>
>   E-mail:
>   <input type="email" name="email" disabled="disabled"
>             onformchange="disabled = !subscribe.checked">
>  </label>
> ```

```
    </p>
```

4.5. Form validation

With the introduction of the various type checking mechanisms, some way for scripting authors to hook into the type checking process is required. This is provided by the **`formerror`** event (in the `http://www.w3.org/2001/xml-events` namespace).

When a form is submitted, each control in that form, in document order, is checked for validity. For each control that fails to comply with its constraints (i.e. each control whose `validity` attribute is non-zero), a `formerror` event must be fired on the control.

The `onformerror` attribute (on `input`, `textarea` and `select` elements) can be used to write handlers for this event.

The `formerror` event bubbles, as described in an earlier section.

This event is cancelable. The default action depends on when the event was fired. If it was fired during form submission, then the default action is UA-specific, but is expected to consist of focusing the element, alerting the user that the entered value is unacceptable in the user's native language along with explanatory text saying *why* the value is currently invalid, and aborting the form submission. UAs would typically only do this for the first form control found to be invalid; while the event is dispatched to all invalid controls, it is simpler for the user to deal with one error at a time. When fired by script calling the `validate()` method (i.e. not during form submission), the event has no default action.

> The following example shows one way to use this event.
>
> ```
> <form action="..." method="post">
>  
>   <label>
>    Byte 1:
>    <input name="byte" type="integer" min="0" max="255" required="re
>           onformerror="failed(event)" />
>   </label>
>   <output name="error"/>
>  
>  <script type="text/javascript">
>   function failed(event) {
>     // a control can fail for more than one reason; only report one
>     form.error.value = 'The value is wrong for a reason I did not e
>     if (event.target.validity & event.target.form.ERROR_TYPE_MISMAT
>       form.error.value = 'That is not an integer.';
>     else if (event.target.validity & event.target.form.ERROR_RANGE_
>       form.error.value = 'That integer is less than 0.';
>     else if (event.target.validity & event.target.form.ERROR_RANGE_
> ```

```
          form.error.value = 'That integer is more than 255.';
        else if (event.target.validity & event.target.form.ERROR_RANGE_
          form.error.value = 'You did not enter a value.';
      }
    </script>
  </form>
```

5. Form submission

Processors conforming to this specification must use a slightly different algorithm than the [HTML4] form submission algorithm (HTML4 section 17.13.3), as described in this section.

When the user agent submits a form (e.g., in response to the user activating a submit button), it must perform the following steps.

1. **Step one: Identify all form controls**

   All the controls that apply to the form, whether successful or not, should be taken, in document order. These controls are those listed in the form's `elements` DOM attribute, and excludes certain controls as specified in the section describing the repetition model.

   Note: The original list of ways in which a control can be successful is defined in HTML4 section 17.3.2. This specification extends this list in places, for example `output` elements are defined to never be successful.

   "}

2. **Step two: Build a form data set**

   A **form data set** is a sequence of *control-name, index, current-value* triples constructed from the controls identified in the first step.

   It is constructed by iterating over the form controls listed in step one, taking note of the form control names as they are seen. With each control, if it is the first time that control's name has been seen, then the control is assigned an index of 0. Otherwise, if the control name was associated with an earlier control, then the index assigned is exactly one more than the last control with that name. Even unsuccessful controls are so numbered. However, only *successful controls* are added to the form data set. A successful control with more than one value is added multiple times, one for each value (each time with the same form control name and form

control index).

> For example, the following form:
>
> ```
> <form>
>   <label> Name: <input type="text" name="username"/> </labe
>   Lottery numbers:
>      <input type="integer" name="number" min="1" max="49"/>
>      <input type="integer" name="number" min="1" max="49"/>
>      <input type="integer" name="number" min="1" max="49"/>
>      <input type="integer" name="number" min="1" max="49"/>
>      <input type="integer" name="number" min="1" max="49"/>
>  
>  
>   <label>
>    Games:
>    <select name="type" multiple="multiple">
>     <option value="Thunderbolt"> Thunderbolt </option>
>     <option value="Lightning"> Lightning </option>
>    </select>
>   </label>
>  
>  
>   <input type="submit" value="Send">
>  
> </form>
> ```
>
> ...if filled in with the name "Erwin" and the numbers 20, 30 and 40 with the first and last number fields left blank, and all the values in the select list selected, would generate the following form data set:
>
> 1. username, 0, "Erwin"
> 2. number, 1, "20"
> 3. number, 2, "30"
> 4. number, 3, "40"
> 5. type, 0, "Thunderbolt"
> 6. type, 0, "Lightning"

The form data set also includes a list of which repetition blocks are involved in the submission.

For each control in the form data set, the control and the control's ancestors are examined, up to but not including the first node that is a common ancestor of the control and the form, or is the form itself. For each element so examined, if it is a repetition block that is not an orphan repetition block and whose template does have an ID, and that repetition

block has not yet been added to the list of repetition blocks, it is added.

3. **Step three: Encode the form data set**

   The form data set is then encoded according to the content type specified by the `method` and `enctype` attribute of the `form` element. If `method` is `get`, then `enctype` is treated as if it had the value `application/x-www-form-urlencoded`, whatever its real value. The possible values of `enctype` defined by this specification are:

   `application/x-www-form-urlencoded`
   : Described below.

   `multipart/form-data`
   : Described in [HTML4], section 17.13.4. Note that this submission method discards the index and repetition block parts of the form data set.

   `application/x-www-form+xml`
   : Described below.

   Other values may be defined by other specifications.

   During this step, the form data set is examined to ensure all the characters are representable in the submission character encoding.

4. **Step four: Dispatch `submit` events.**

   The `submit` event is then submitted as described in [HTML4]. If it is canceled, then the submission processing stops at this point.

5. **Step five: Submit the encoded form data set**

   Finally, the encoded data is sent to the processing agent designated by the `action` attribute using the protocol method specified by the `method` attribute.

   For HTTP GET submissions, the encoded form data is used as the query value. For HTTP POST submissions, the encoded data is used as the entity body.

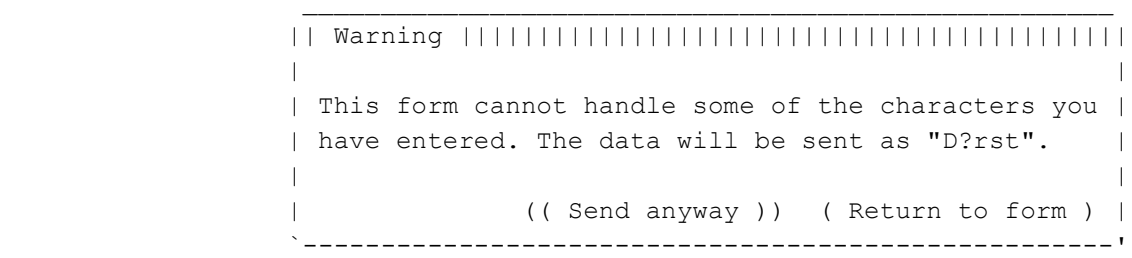
   User agents should handle responses to the submission. For example if the `action` denotes an HTTP resource, and the `method` is `post` and the remote server replies with a `200 OK` response, then the returned document should be displayed to the user as if the user had navigated to that document by following a link to it.

5.1. Handling characters outside the submission character set

The different form data set encoding types each define how to find the character encoding to use to submit the data.

Sometimes, the form submission character set used is not able to represent all the character present in the form submission.

If the form data set contains characters that are outside the submission character set, the user agent should inform the user that his submission will be changed, for example using a dialog in the form:



If the submission is not canceled, the user agent MUST replace each character that is not in the submission character set with one or more replacement characters.

For each such missing character, UAs must either transliterate the character to a UA-defined human-recognizable representation (for example transliterating U+263A to the three-character string ":-)" in US-ASCII, or U+2126 to the byte 0xD9 in ISO-8859-7), or, for characters where a dedicated transliteration is not known to the UA, replace the character with either U+FFFD, "?", or some other single character representing the same semantic as U+FFFD.

Note that a string containing the codepoint's value itself (for example the six-character string "U+263A" or the seven-character string "☺") is not considered to be human readable and must not be used as a transliteration. (This is to discourage servers from attempting to mechanically convert such codepoints back into Unicode characters, as there is no way to distinguish such characters from identical literal strings entered by the user.)

5.2. `application/x-www-form-urlencoded`

This is the default content type. Forms submitted with this content type must be encoded as follows:

1. The submission character encoding is selected from the form's `accept-charset` attribute. If the attribute is not specified, then the client should use either the page's character encoding, or UTF-8. Character encodings that are not supersets of US-ASCII must not be used (this includes UTF-16 and EBCDIC).

2. If the form contains an input control of type `hidden` with the name `_charset_`, it is forced to appear in the form data set, with the value equal to the name of the submission character encoding used.

3. Control names and values are escaped. Space characters are replaced by `+', and then non-alphanumeric characters are encoded in the submission character encoding and each resulting byte is replaced by `%HH', a percent sign and two hexadecimal digits representing the value of the byte.

4. The control names/values are listed in the order they appear in the form data set. The name is separated from the value by `=' and name/value pairs are separated from each other by `&'.

Note that the index and repetition block parts of the form data set are not used.

5.3. `application/x-www-form+xml`: XML submission

This section defines the expected behaviour for step 3, "Step three: Encode the form data set", of the submission algorithm described above, for the form content type `application/x-www-form+xml`. The rest of the form submission process progresses as described above.

The message entity is an XML 1.1 document, encoded as UTF-8, which has a root element named "submission", with no prefix, defining a default namespace `data:,formData`. UA may include an XML declaration but this is not necessary as it would be redundant (the MIME type and version must not be changed from the above values, and submission documents have no document type and therefore are always standalone.).

Note that the form's `accept-charset` attribute is ignored for this encoding type.

First, for each repetition block in the form data set, an element `repeat` is inserted, with an attribute `template` equal to the ID of the template, and an attribute `index` equal to the index of the repetition block. The element is empty.

Servers are generally expected to ignore `repeat` elements; they are primarily included so that form data can be round-tripped using the `data` attribute on the form element.

Then, for each successful control that is not a file upload control, in the order that the controls are to be found in the original document, an element `field` is inserted, with an attribute `name` having the name of the form control, an attribute `index` having the index described above in the definition of the form data set, and with the element content being the current value of the form control. Form controls with multiple values result in multiple `field` elements being inserted into

the output, one for each value, all with the same index.

File controls are submitted using a `file` element instead of a `field` element. The `file` element has four attributes, `name`, `index`, `filename`, and `type`. The `name` attribute contains the name of the file control. The `index` attribute contains the index in the control's entry in the form data set. The `filename` attribute is optional and may contain the name of the file. The `type` attribute is not optional and must contain the MIME type of the file. The contents of the file are base64 encoded and then included literally as content directly inside the `file` element. As base64 data is whitespace-clean, UAs may introduce whitespace into the `file` element to ensure the submitted data has reasonable line lengths. This is, however, completely optional. (It is primarily intended to make it possible to write readable examples of submission output.)

UAs may use either CDATA blocks, entities, or both in escaping the contents of attributes and elements, as appropriate. The resulting XML must be a well-formed XML instance. The only mention of namespaces in the submission document must be the declaration of the default namespace on the root element.

Whitespace may be inserted around elements that are children of the `submission` element in order to make the submitted data easier to scan by eye. However, this is optional. Processors should not be affected by such whitespace, or whitespace inside `file` elements, when reading the submitted data back from the XML instance. (Whitespace inside `field` elements is significant, however.)

> The following example illustrates `application/x-www-form+xml` encoding. Suppose we have the following form:
>
> ```
> <form action="http://example.com/cgi/handle"
>       enctype="application/x-www-form+xml"
>       method="post">
>  
>   <label> What is your name? <input type="text" name="submit-name"/
>   <label> What files are you sending? <input type="file" name="file
>   <label> When were they written? <input type="date" name="stamp"/>
>   <input type="submit" value="Send">
>  
> </form>
> ```
>
> If the user enters "Larry" in the text input, selects the text file "file1.txt", and picks an arbitrary date, the user agent might send back the following data:
>
> ```
> Content-Type: application/x-www-form+xml
>
> <submission xmlns="data:,formData">
>  <field name="submit-name" index="0">Larry</field>
> ```

```
 <file name="files" index="0" filename="file1.txt" type="text/plain
  Y29udGVudHMgb2YgZmlsZTEudHh0
 </file>
 <field name="stamp" index="0">1979-04-13</field>
</submission>
```

If the user selected a second (image) file "file2.png", and changes the date, the user agent might construct the entity as follows:

```
Content-Type: application/x-www-form+xml

<submission xmlns="data:,formData">
 <field name="submit-name" index="0">Larry</field>
 <file name="files" index="0" filename="file1.txt" type="text/plain
  Y29udGVudHMgb2YgZmlsZTEudHh0
 </file>
 <file name="files" index="0" filename="file2.png" type="image/png"
  iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAMAAAAoyzS7AAAABGdBTUEAAK
  /INwWK6QAAABl0RVh0U29mdHdhcmUAQWRvYmUgSW1hZ2VSZWFkeXHJZTwA
  AAAGUExURQD/AAAAG8DfkMAAAAMSURBVHjaYmAACDAAAAIAAU9tWeEAAA
  AASUVORK5CYII=
 </file>
 <field name="stamp" index="0">1979-12-27</field>
</submission>
```

Note how the content of the plain text attached file is base64-encoded, despite being a plain text file. This preserves the integrity of the file in cases where the MIME type is incorrect. It also means that files with malformed content, for example a file encoded as UTF-8 with stray continuation bytes, will be transmitted faithfully instead of being re-encoded by the UA.

This example illustrates this encoding for the case with two form controls with the same name. Suppose we have the following form:

```
<form enctype="application/x-www-form+xml" method="post">
 <p>
  Enter your new password twice:
  <input type="password" name="password"/>
  <input type="password" name="password"/>
  <input type="submit" value="Send">
 </p>
</form>
```

If the user enters "perfect" and "prefect", the user agent might send back the following data:

```
Content-Type: application/x-www-form+xml

<submission xmlns="data:,formData">
 <field name="password" index="0">perfect</field>
 <field name="password" index="1">prefect</field>
```

```
</submission>
```

Recall the example for repetition blocks. If it was immediately submitted, the output would be an XML file equivalent to:

```
Content-Type: application/x-www-form+xml

<submission xmlns="data:,formData">
 <repeat template="row" index="0"/>
 <field name="name_0" index="0">John Smith</field>
 <field name="count_0" index="0">1</field>
</submission>
```

6. Seeding a form with initial values

If a `form` has a `data` attribute, it must be a URI that points to a well-formed XML file containing a `formdata` element in the `data:,formData` namespace.

UAs must process this file if it has an XML MIME type, if it is a well-formed XML file, and if the root element is the right root element in the right namespace. If any of these conditions are not met, UAs must act as if the attribute was not specified, although they may report the error to the user. UAs are expected to correctly handle namespaces, so the file may use prefixes, etc.

If the UA processes the file, it must use the following algorithm to fill the form.

1. The form must be reset to its initial values as specified in the markup.
2. Child text nodes, CDATA blocks, comments, and PIs of the root element of the specified file must be ignored.
3. `repeat` elements in the `data:,formData` namespace that are children of the root element, have a non-empty `template` attribute and an `index` attribute that contains only one or more digits in the range 0-9 with an optional leading minus sign, have no other non-namespaced attributes, and have no content, must be processed as follows:

   If the `template` attribute specifies an element that is not a repetition template, then the element is ignored.

   If the `template` attribute specifies a repetition template and that template already has a repetition block with the index specified by the `index` attribute, then the element is ignored.

   Otherwise, the specified template's `addRepetitionBlockByIndex()` method is called, with a null first argument and the index specified by the `repeat`

element's `index` attribute as the second.

4. `field` elements in the `data:,formData` namespace that are children of the root element, have a non-empty `name` attribute and an `index` attribute that contains only one or more digits in the range 0-9, have no other non-namespaced attributes, and have either nothing or only text and CDATA nodes as children, must be used to initialize fields, as follows.

   First, the form control that the field references must be identified. This is done by walking the list of form controls associated with the form until one is found that has a name exactly equal to the name given in the `field` element's `name` attribute, skipping as many such matches as is specified in the `index` attribute.

   If the identified form control is a file upload control, a push button control, or an image control, then the `field` element is now skipped.

   Next, if the identified form control is not a multiple-valued control (a multiple-valued control is one that can generate more than one value on submission, such as a `<select multiple="multiple">`), or if it is a multiple-valued control but it is the first time the control has been identified by a `field` element in this data file that was not ignored, then it is set to the given value (the contents of the `field` element), removing any previous values (even if these values were the result of processing previous `field` elements in the same data file). Otherwise, this is a subsequent value for a multiple-valued control, and the given value (the contents of the `field` element) should be *added* to the list of values that the element has selected.

   If the element cannot be given the value specified, the `field` element is ignored and the control's value is left unchanged.

   If the element is a multiple-valued control and the control already has the given value selected, but it can be given the value again, then that occurs. For example, in the following case:

   ```
   <select name="select" multiple="multiple">
    <option>test</option>
    <option>test</option>
    <option>test</option>
   </select>
   ```

   ...if the data file contained two instances of:

   ```
   <field name="select" index="0">test</select>
   ```

   ...then the first two `option` elements would end up selected, and the last

would not. This would be the case irrespective of which `option` elements had their `selected` attribute set in the markup.

Note: The `option` elements are never directly matched by `field` elements; it is the `select` element in this case that is matched (twice). This is why the two `field` elements select subsequent values in the control.

If the element is a multiple-valued control and the control already has the given value selected and it *cannot* be given the value again, then the field is ignored.

5. All other elements in the file must be ignored.
6. A `formchange` event is then fired on all the form controls of the form.

Note: Note that file upload controls cannot be repopulated. However, `output` control can be populated. This can be used, for example, for localizing a form by including the structure in one file and the strings in another. (The semantics of this practice are somewhat dubious, however. It is only mentioned because XForms advocates claim this as a feature.)

7. Extensions to the HTML Level 2 DOM interfaces

Unless otherwise specified, these interfaces have the same semantics as defined in [DOM2HTML].

```
interface HTMLFormElement : HTMLElement {
  readonly attribute HTMLCollection  elements;
  readonly attribute long            length;
           attribute DOMString       name;
           attribute DOMString       acceptCharset;
           attribute DOMString       action;
           attribute DOMString       enctype;
           attribute DOMString       method;
           attribute DOMString       target;
  void               submit();
  void               reset();

  // new in this specification:
  const    unsigned short            ERROR_TYPE_MISMATCH     = 1;
  const    unsigned short            ERROR_RANGE_UNDERFLOW   = 2;
  const    unsigned short            ERROR_RANGE_OVERFLOW    = 4;
```

```
  const    unsigned short           ERROR_TOO_LONG          = 8;
  const    unsigned short           ERROR_PATTERN_MISMATCH  = 16;
  const    unsigned short           ERROR_REQUIRED          = 32;
  const    unsigned short           ERROR_USER_DEFINED      = 32768;

           attribute DOMString      accept;
  readonly attribute HTMLCollection templateElements;
  void              validate();
};

interface HTMLSelectElement : HTMLElement {
  readonly attribute DOMString       type;
           attribute long            selectedIndex;
           attribute DOMString       value;
           attribute unsigned long   length;
                                        // raises(DOMException) on sett

  readonly attribute HTMLFormElement form;
  readonly attribute HTMLOptionsCollection options;
           attribute boolean         disabled;
           attribute boolean         multiple;
           attribute DOMString       name;
           attribute long            size;
           attribute long            tabIndex;
  void              add(in HTMLElement element,
                        in HTMLElement before)
                                        raises(DOMException);
  void              remove(in long index);
  void              blur();
  void              focus();

  // new in this specification:
           attribute boolean         editable;
           attribute DOMString       pattern;
           attribute boolean         required;
           attribute boolean         autocomplete;
           attribute DOMString       inputmode;
           attribute HTMLCollection  selectedOptions;
  void              reset();

  readonly attribute long            validity;
  void              validate();
  void              markValid();
  void              markInvalid();
};

interface HTMLOptGroupElement : HTMLElement {
           attribute boolean         disabled;
           attribute DOMString       label;
};

interface HTMLOptionElement : HTMLElement {
  readonly attribute HTMLFormElement form;
           attribute boolean         defaultSelected;
```

```
  readonly attribute DOMString       text;
  readonly attribute long            index;
           attribute boolean         disabled;
           attribute DOMString       label;
           attribute boolean         selected;
           attribute DOMString       value;
};

interface HTMLInputElement : HTMLElement {
           attribute DOMString       defaultValue;
           attribute boolean         defaultChecked;
  readonly attribute HTMLFormElement form;
           attribute DOMString       accept;
           attribute DOMString       accessKey;
           attribute DOMString       align;
           attribute DOMString       alt;
           attribute boolean         checked;
           attribute boolean         disabled;
           attribute long            maxLength;
           attribute DOMString       name;
           attribute boolean         readOnly;
           attribute unsigned long   size;
           attribute DOMString       src;
           attribute long            tabIndex;
           attribute DOMString       type;
           attribute DOMString       useMap;
           attribute DOMString       value;
  void               blur();
  void               focus();
  void               select();
  void               click();

  // new in this specification:
           attribute long            min;
           attribute long            max;
           attribute DOMString       pattern;
           attribute boolean         required;
           attribute boolean         autocomplete;
           attribute DOMString       inputmode;
  readonly attribute RepetitionElement template;
  void               reset();

  readonly attribute long            validity;
  void               validate();
  void               markValid();
  void               markInvalid();
};

interface HTMLTextAreaElement : HTMLElement {
           attribute DOMString       defaultValue;
  readonly attribute HTMLFormElement form;
           attribute DOMString       accessKey;
           attribute long            cols;
           attribute boolean         disabled;
```

```
           attribute DOMString       name;
           attribute boolean         readOnly;
           attribute long            rows;
           attribute long            tabIndex;
  readonly attribute DOMString       type;
           attribute DOMString       value;
  void              blur();
  void              focus();
  void              select();

  // new in this specification:
           attribute DOMString       wrap;
           attribute DOMString       pattern;
           attribute boolean         required;
           attribute boolean         autocomplete;
           attribute DOMString       inputmode;
  void              reset();

  readonly attribute long            validity;
  void              validate();
  void              markValid();
  void              markInvalid();
};

interface HTMLButtonElement : HTMLElement {
  readonly attribute HTMLFormElement form;
           attribute DOMString       accessKey;
           attribute boolean         disabled;
           attribute DOMString       name;
           attribute long            tabIndex;
  readonly attribute DOMString       type;
           attribute DOMString       value;

  // new in this specification:
  readonly attribute RepetitionElement template;
};

interface HTMLLabelElement : HTMLElement {
  readonly attribute HTMLFormElement form;
           attribute DOMString       accessKey;
           attribute DOMString       htmlFor;
};

interface HTMLFieldSetElement : HTMLElement {
  readonly attribute HTMLFormElement form;

  // new in this specification
           attribute boolean         disabled;
};

interface HTMLLegendElement : HTMLElement {
  readonly attribute HTMLFormElement form;
           attribute DOMString       accessKey;
           attribute DOMString       align;
```

```
};

// new in this specification
interface HTMLOutputElement : HTMLElement {
           attribute DOMString       defaultValue;
  readonly attribute HTMLFormElement form;
           attribute DOMString       name;
           attribute DOMString       value;
  void               reset();
};

// new in this specification
interface RepetitionElement {
  const              unsigned short  REPETITION_NONE = 0;
  const              unsigned short  REPETITION_TEMPLATE = 1;
  const              unsigned short  REPETITION_BLOCK = 2;

           attribute unsigned short  repetitionType;
           attribute long            repetitionIndex;
  readonly attribute Element         repetitionTemplate;
  readonly attribute HTMLCollection  repetitionBlocks;
  void               addRepetitionBlock(in Node refNode);
  void               addRepetitionBlockByIndex(in Node refNode, in long
  void               moveRepetitionBlock(in long distance);
  void               removeRepetitionBlock();
}

// new in this specification
interface FormDocument {
  Document           load(in DOMString action, in DOMString method,
                          in DOMString enctype, in DOMString content);
}
```

7.1. Additions specific to the `HTMLFormElement` interface

The new `accept` attribute reflects the `form` element's `accept` attribute and its addition here merely address an oversight in DOM2.

The `templateElements` attribute contains the list of form controls associated with this form that form part of repetition templates that the form itself is not also a part of. It is defined in more detail in the section on the repetition model.

The `validate()` method invokes the `validate()` method of all the elements in the form's `elements` list whose interfaces have a `validate()` method defined.

7.2. Additions specific to the `HTMLSelectElement` interface

The `editable` DOM attribute reflects the `editable` content attribute in the same way as the `multiple` DOM attribute reflects the `multiple` content attribute.

The `selectedOptions` attribute provides a readonly list of the descendant

`HTMLOptionElement` nodes that currently have their `selected` attribute set to a true value (a subset of the controls listed in the `options` attribute). The list is returned live, so changing the options selected (by the user or by script) will change the list. The order of the list should be consistent with the order of the `options` list.

7.3. The `HTMLOutputElement` interface

This interface is added for the new `output` element. Its attributes work analogously to those on other controls. The semantics of the `value` and `defaultValue` DOM attributes are described in the section describing the `output` element.

7.4. Validation APIs

The `validity` attribute returns whether the form control is currently valid. Its value is a bit field giving the errors that currently apply ot the control, the sum of the relevant `ERROR_*` constants defined on the `HTMLFormElement` interface. These have the following meanings:

ERROR_TYPE_MISMATCH
: The data entered does not match the type of the control. For example, if the UA allows uninterpreted arbitrary text entry for `expdate` fields, and the user has entered `SEP02`, then this error code would be used. This code is also used when the selected file in a file upload control does not have an appropriate MIME type.

ERROR_RANGE_UNDERFLOW
: The numeric, date, or time value of a field with a `min` attribute is lower than the min.

ERROR_RANGE_OVERFLOW
: The numeric, date, or time value of a field with a `max` attribute is higher than the max.

ERROR_TOO_LONG
: The value of a field with a `maxlength` attribute is longer than the attribute allows.

ERROR_PATTERN_MISMATCH
: The value of the field with a `pattern` attribute doesn't match the pattern.

ERROR_REQUIRED
: The field has the `required` attribute set but has no value.

ERROR_USER_DEFINED
: The field was marked invalid from script. See the definitions of the `markValid()` and `markInvalid()` methods.

When the definitions above refer to elements that have an attribute set on them,

they do not refer to elements on which that attribute is defined not to apply. For example, the `ERROR_REQUIRED` code cannot be set on an `<input type="checkbox">` element, even if that element has the `required` attribute set, since `required` doesn't apply to check boxes.

The `validate()` method, present on several of the form control interfaces, causes an `formerror` event to be fired on that control, unless the `validity` of the control is zero. ***Recall that this is automatically done during form submission.***

The `markValid()` and `markInvalid()` methods set and reset (respectively) the `ERROR_USER_DEFINED` bit on the `validity` attribute. Even attributes that are empty and not required can be marked invalid like this, and would abort form submission if so marked.

7.5. New DOM attributes for new content attributes

The new `pattern`, `required`, `autocomplete`, `inputmode`, `min`, `max`, `wrap`, and `disabled` attributes simply reflect the current value of their relevant attribute.

7.6. Resetting form controls

In addition to the `reset()` method on the form interface, this specification introduces the `reset()` method on the form conrol interfaces to reset just the relevant control to its initial value.

7.7. Repetition interfaces

The `RepetitionElement` interface should be implemented by all elements.

If the element is a repetition template, its **`repetitionType`** DOM attribute must return `REPETITION_TEMPLATE`. Otherwise, if the element is a repetition block, it must return `REPETITION_BLOCK`. Otherwise, it is a normal element, and that attribute should return `REPETITION_NONE`.

The **`repetitionIndex`** attribute must return the current value of the index of the repetition template or block. If the element is a normal element, it must return zero. Setting this attribute must not affect the `repeat` content attribute.

The **`repetitionTemplate`** attribute is null unless the element is a repetition block, in which case it points to the block's template. If the block is an orphan repetition black then it returns null.

The **`repetitionBlocks`** attribute is null unless the element is a repetition template, in which case it points to a list of elements (an `HTMLCollection`, although the name of that interface is a misnomer since there is nothing HTML-

specific about it). The list consists of all the repetition blocks that have this element as their template. The list is live.

The `addRepetitionBlock()`, `addRepetitionBlockByIndex()`, `moveRepetitionBlock()` and `removeRepetitionBlock()` methods are defined in the section on the repetition model.

The `template` DOM attribute on the `HTMLInputElement` and `HTMLButtonElement` interfaces represents the repetition template that the `template` content attribute refers to. If the content attribute points to a non-existent element or an element that is not a repetition template, the DOM attribute returns null. This DOM attribute is readonly in this version of this specification.

7.8. Loading remote documents

The `FormDocument` interface can be obtained using binding-specific casting methods on the `document` object.

The `load` method on the `FormDocument` interface returns a `Document` interface, and then queues the specified resource to be loaded into that document. When the document has finished loading, a `load` event fires on the object. If a failure occurs during loading, an `error` event fires instead.

This method has four arguments:

action
: The URI to lead.

method
: The method (e.g. for HTTP actions, 'GET', 'POST') used to load the URI.

enctype
: The Content-Type of the content, if any. For HTTP requests with entity bodies, this is the value of the submitted Content-Type header.

content
: The entity body of submission requests, if required.

Implementations may limit which hosts, ports, and schemes can be accessed using this method. For example, it is highly recommended that the SMTP port not be allowed, since otherwise it can be used to relay spam on the behalf of the unwitting user. Similarly, cross-domain scripting restrictions are fully expected to apply.

The `load` method is asynchronous, and is guarenteed to not finish loading the document or signal an error before the running script either completes or yields to the user (e.g. by calling `window.alert()`). Thus, the following code is guarenteed to hook in the event handlers before the document has either

finished loading or signalled an error:

```
var d = document.load('http://example.org/search', 'post',
                      'application/xml', '');
d.addEventListenerNS('http://www.w3.org/2001/xml-events', 'load',
                      function () { alert('loaded!'); },
                      false, null);
d.addEventListenerNS('http://www.w3.org/2001/xml-events', 'error',
                      function () { alert('loaded!'); },
                      false, null);
```

For more control over loading remote documents, see DOM3 Load and Save [DOM3LS].

8. Styling form controls

The CSS working group is expected to develop a language designed, amongst other things, for the advanced styling of form controls. In the meantime, technologies such as [HTC] and [XBL] can be used as guides for what is expected.

UAs, in the absence of such advanced styling information, may render form controls described in this draft as they wish. It is recommended that form controls remain faithful to the look and feel of the system's global user interface, though.

Note that [CSS21] explicitly does not define how CSS applies to form controls.

8.1. Relation to the CSS3 User Interface module

[CSS3UI] introduces a number of pseudo-classes for form controls. Their relationship to the form controls described in this specification is described here.

:enabled
: Matches form control elements that do not have the `disabled` attribute set.

:disabled
: Matches form control elements that do have the `disabled` attribute set.

:checked
: Matches radio and check box form control elements that are `checked`.

:indeterminate
: Matches no HTML form control elements.

:default
: Matches the button (if any) that will be selected if the user presses the enter key (or some equivalent behaviour on less typical systems).

:valid
: Matches form control elements that would not have the `formerror` event fired at them if the form was submitted.

:invalid
: Matches form control elements that would have the `formerror` event fired at them if the form was submitted.

:in-range
: Matches numeric, date-related, or time-related form control elements when the current value is type-correct, greater than or equal to the minimum (if any), and less than or equal to the maximum (if any).

:out-of-range
: Matches numeric, date-related, or time-related form control elements when the current value is type-correct, but either less than the minimum or greater than the maximum.

:required
: Matches form control elements that have the `required` attribute set.

:optional
: Matches form control elements that do not have the `required` attribute set.

:read-only
: Matches form control elements that have the `readonly` attribute set.

:read-write
: Matches form control elements that do not have the `readonly` attribute set (including `password` fields, although technically they should be called "writeonly").

When the definitions above refer to elements that have an attribute set on them, they do not refer to elements on which that attribute is defined not to apply. For example, the `:read-only` attribute cannot apply to a `<input type="radio">` element, even if that element has the `readonly` attribute set, since `readonly` doesn't apply to radio buttons.

A. XHTML module definition

The Forms Extensions Module provides all of the forms features found in HTML 4.0, plus the extensions described above. Specifically, the Forms Extensions Module supports:

Elements	Attributes	Minimal Content Model

Elements	Attributes	Minimal Content Model
form	Common, accept (ContentTypes), accept-charset (Charsets), action (URI), method ("get"* | "post"), enctype (ContentType)	(Heading | List | Block - form)*
input	Common, accept (ContentTypes), accesskey (Character), alt (Text), autocomplete ("true"* | "false"), checked ("checked"), disabled ("disabled"), form (IDREF), help (URI), inputmode (CDATA), maxlength (Number), min (CDATA), max (CDATA), name (CDATA), pattern (CDATA), readonly ("readonly"), required ("required"), size (Number), src (URI), tabindex (Number), template (IDREF), type ("text"* | "password" | "checkbox" | "radio" | "button" | "submit" | "reset" | "add" | "remove" | "file" | "hidden" | "image" | "datetime" | "date" | "expdate" | "time" | "number" | "integer" | "email" | "uri"), value (CDATA),	EMPTY
select	Common, autocomplete ("true"* | "false"), disabled ("disabled"), editable ("editable"), form (IDREF), help (URI), inputmode (CDATA), multiple ("multiple"), name (CDATA), pattern (CDATA), required ("required"), size (Number), tabindex (Number)	(optgroup | option)*
optgroup	Common, disabled ("disabled"), label* (Text)	option*
option	Common, disabled ("disabled"), label (Text), selected ("selected"), value (CDATA)	PCDATA
textarea	Common, accesskey (Character), autocomplete ("true"* | "false"), cols (Number), disabled ("disabled"), form (IDREF), help (URI), inputmode (CDATA), name (CDATA), readonly ("readonly"), required ("required"), rows (Number), tabindex (Number), wrap ("soft"* | "hard")	PCDATA
output	Common, form (IDREF), name (CDATA)	PCDATA
button	Common, accesskey (Character), disabled ("disabled"), form (IDREF), help (URI), name (CDATA), tabindex (Number), template (IDREF), type ("button" | "submit"* | "reset" | "add" | "remove"), value (CDATA)	(PCDATA | Heading | List | Block - Form | Inline - Formctrl)*
fieldset	Common, disabled ("disabled"), form (IDREF), help (URI),	(PCDATA | legend | Flow)*

Elements	Attributes	Minimal Content Model
legend	Common, accesskey (Character)	(PCDATA | Inline)*
label	Common, accesskey (Character), for (IDREF)	(PCDATA | Inline - label)*

This module defines two content sets:

Form
: form | fieldset

Formctrl
: input | select | textarea | output | button | label

When this module is used, it adds the `Form` content set to the `Block` content set and it adds the `Formctrl` content set to the `Inline` content set as these are defined in the Text Module.

All XHTML elements (all elements in the `http://www.w3.org/1999/xhtml` namespace) may have the `repeat` attribute specified. Similarly, the global attribute `repeat` in the `http://www.w3.org/1999/xhtml` namespace may be specified on any non XHTML element.

The `form` element may be placed inside XHTML `head` elements when it is empty.

The `oninput` attribute is added to all the elements that have an `onchange` attribute in the XHTML Intrinsic Events module. The `onformchange`, `onforminput` and `onformerror` attributes are added to all form control elements (including `fieldset`).

The Forms Extensions Module is a superset of the Forms and Basic Forms modules. These modules may not be used together in a single document type. Note that the content models in this module differ from those of the XHTML1 Forms module in some subtle ways (for example, the `select` element may be empty).

B. Attribute summary

The `input` element takes a large number of attributes that do not always apply. The following table summarizes which attributes apply to which input types.

type	text	password	checkbox	radio	button	submit	reset	add	re r r (
accept	-	-	-	-	-	-	-	-	
accesskey	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
alt	-	-	-	-	-	-	-	-	
autocomplete	Yes	Yes	Yes	Yes	-	-	-	-	
checked	-	-	Yes	Yes	-	-	-	-	
disabled	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
form	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
help	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
inputmode	Yes	Yes	-	-	-	-	-	-	
maxlength	Yes	Yes	-	-	-	-	-	-	
min	-	-	-	-	-	-	-	-	
max	-	-	-	-	-	-	-	-	
name	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
pattern	Yes	Yes	-	-	-	-	-	-	
readonly	Yes	Yes	-	-	-	-	-	-	
required	Yes	Yes	-	Yes	-	-	-	-	
size	Yes	Yes	-	-	-	-	-	-	
src	-	-	-	-	-	-	-	-	
tabindex	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
template	-	-	-	-	-	-	-	Yes	
value	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	

References

[CHARMOD]

Character Model for the World Wide Web 1.0, M. Dürst, F. Yergeau, R. Ishida, M. Wolf, T. Texin. W3C, August 2003. The latest version of the Character Model specification is available at http://www.w3.org/TR/charmod/

[CSS21]

CSS 2.1 Specification, B. Bos, T. Çelik, I. Hickson, H. Lie. W3C, September 2003. The latest version of the CSS 2.1 specification is available at http://www.w3.org/TR/CSS21

[CSS3UI]

CSS3 Basic User Interface Module, T. Çelik. W3C, July 2003. The latest version of the CSS3 UI module is available at http://www.w3.org/TR/css3-ui

[CSS3CONTENT]

CSS3 Generated and Replaced Content Module, I. Hickson. W3C, May 2003. The latest version of the CSS3 Generated and Replaced Content module is available at http://www.w3.org/TR/css3-content

[DOM3CORE]

Document Object Model (DOM) Level 3 Core Specification, A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne. W3C, November 2003. The latest version of the DOM Level 3 Core specification is available at http://www.w3.org/TR/DOM-Level-3-Core/

[DOM3EVENTS]

Document Object Model (DOM) Level 3 Events Specification, P. Le Hégaret, T. Pixley. W3C, November 2003. The latest version of the DOM Level 3 Events specification is available at http://www.w3.org/TR/DOM-Level-3-Events/

[DOM2HTML]

Document Object Model (DOM) Level 2 HTML Specification, J. Stenback, P. Le Hégaret, A. Le Hors. W3C, January 2003. The latest version of the DOM Level 2 HTML specification is available at http://www.w3.org/TR/DOM-Level-2-HTML/

[DOM3LS]

Document Object Model (DOM) Level 3 Load and Save Specification, J. Stenback, A. Heninger. W3C, November 2003. The latest version of the DOM Level 3 Load and Save specification is available at http://www.w3.org/TR/DOM-Level-3-LS/

[ECMA262]

ECMAScript Language Specification, Third Edition. ECMA, December 1999. This version of the ECMAScript Language is available at http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM

[HTC]

HTML Components, Chris Wilson. Microsoft, September 1998. The HTML Components submission is available at http://www.w3.org/TR/1998/NOTE-HTMLComponents-19981023

[HTML4]

HTML 4.01 Specification, D. Raggett, A. Le Hors, I. Jacobs. W3C,

December 1999. The latest version of the HTML4 specification is available at http://www.w3.org/TR/html4

[ISO8601]

ISO8602:2000 Data elements and interchange formats -- Information interchange -- Representation of dates and times. ISO, December 2000. ISO8601 is available for purchase at http://www.iso.ch/

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels, S. Bradner. IETF, March 1997. RFC2119 is available at http://www.ietf.org/rfc/rfc2119

[RFC2396]

Uniform Resource Identifiers (URI): Generic Syntax, T. Berners-Lee, R. Fielding, L. Masinter. IETF, August 1998. RFC2396 is available at http://www.ietf.org/rfc/rfc2396

[RFC3106]

ECML v1.1: Field Specifications for E-Commerce, D. Eastlake, T Goldstein. IETF, April 2001. RFC 3106 is available at http://www.ietf.org/rfc/rfc3106

[RFC822]

Standard for the Format of ARPA Internet Text Messages, David H. Crocker. IETF, August 1982. RFC822 is available at http://www.ietf.org/rfc/rfc822

[XBL]

XML Binding Language, David Hyatt. Mozilla, February 2001. The XBL submission is available at http://www.w3.org/TR/2001/NOTE-xbl-20010223/

[XML]

Extensible Markup Language (XML) 1.0 (Second Edition), T Bray, J Paoli, C. M. Sperberg-McQueen, E. Maler. W3C, October 2000. The latest version of the XML specification is available at http://www.w3.org/TR/REC-xml/

[XHTML1]

XHTML™ 1.1 - Module-based XHTML, M. Altheim, S. McCarron. W3C, May 2001. The latest version of the XHTML 1.1 specification is available at http://www.w3.org/TR/xhtml11

[XForms]

XForms 1.0, M. Dubinko, L. Klotz, R. Merrick, T. Raman. W3C, October 2003. The latest version of the XForms specification is available at http://www.w3.org/TR/xforms

Acknowledgements

Thanks to Håkon Wium Lie, Maciej Stachowiak, David Hyatt, Peter N Stark, Jason Kersey, Neil Rashbrook, Brendan Eich, Bert Bos, and John Keiser for their substantial comments.

Thanks also to Anne van Kesteren, Michael Daskalov and Rigo Wenning for their comments, and to the #mozilla crew, the #elektra crew, and the #mrt crew for their ideas and support.

Thanks to the XForms working group for unintentionally giving the incentive to develop this specification.