XHTML™ Events Module

An updated events syntax for XML-based markup languages

W3C Working Draft 21 December 1999

This version:
http://www.w3.org/TR/1999/WD-xhtml-events-19991221
(Postscript version, PDF version, ZIP archive, or Gzip'd TAR archive)
Latest version:
http://www.w3.org/TR/xhtml-events
Previous version:
None
Editors:
Ted Wugofski, Gateway
Patrick Schmitz, Microsoft
Shane P. McCarron, Applied Testing and Technology

Abstract

This specification defines the XHTML Event Module, a module that provides XML languages with the ability to represent in syntax the semantics of the Document Object Model (DOM) Level 2 event interfaces [DOM2] [p.29] .

The DOM specifies an event model that provides the following features:

- the event system is generic,
- a means is provided for registering event handlers,
- events may be routed through a tree structure, and
- context information for each event is available.

In addition, the DOM provides an event flow architecture that describes how events are captured, bubbled, and canceled. In summary, event flow is the process through which an event originates from the DOM implementation and is passed into the document object model. The methods of event capture and event bubbling, along with various event listener registration techniques, allow the event to then be handled in a number of ways. It can be handled locally at the target `Node` level or centrally from a `Node` higher in the document tree.

The XHTML Event Module contains an `event-target`, `event-listener`, and an `event` element. The `event-target` element is used to attach an event handler to an element. The `event-listener` element is used to represent the DOM event handler. The `event` element is used to represent the DOM event.

The design of the XHTML Event Module is such that it can be used together with XHTML modules [XMOD [p.29]] and SMIL modules [SMOD [p.29]].

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. The latest status of this document series is maintained at the W3C.

This is the first public working draft of the of the XHTML Event Module specification. It is guaranteed to change; anyone implementing it should realize that we will not allow ourselves to be restricted by experimental implementations when deciding whether to change the specifications.

This specification is a Working Draft of the HTML Working Group for review by W3C members and other interested parties. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress".

Publication as a Working Draft does not imply endorsement by the W3C membership, nor of members of the HTML, SYMM, nor DOM working groups. This is still a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite W3C Working Draft as other than "work in progress."

This document has been produced as part of the W3C HTML Activity and SYMM Activity). The authors of this document are members of the HTML Working Group and the SYMM Working Group.

This document is for public review. Comments on the normative aspects of this document or the integration with XHTML should be sent to the public mailing list www-html@w3.org. Comments regarding the integration with SMIL should be sent to the public mailing list www-smil@w3.org.

A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

Contents

1. Overview of the DOM Event Model

The Document Object Model (DOM) Level 2 specifies an event model that provides the following features:

- the event system is generic
- a means is provided for registering event handlers
- events may be routed through a tree structure
- context information for each event is available

The DOM Level 2 specification [DOM2 [p.29]] further specifies three classes of events: user interface events, user interface logical events, and mutation events. In addition to these three events, the XHTML Event Module specifies a trigger event.

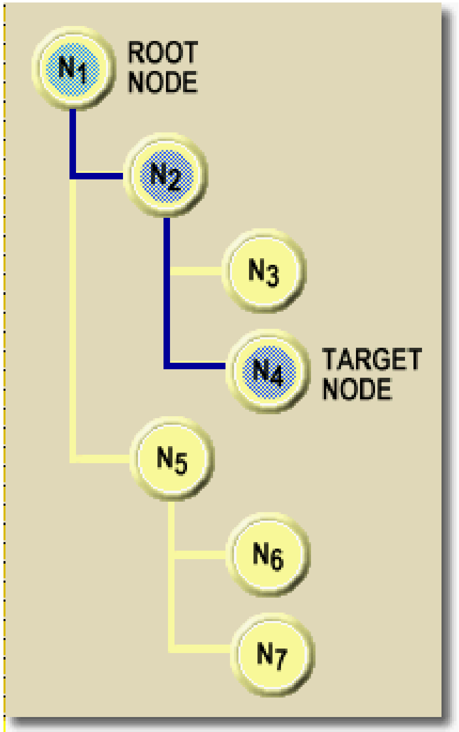
UI Events
: User interface events. These events are generated by user interaction through an external device (such as a mouse, keyboard, or remote control).

UI Logical Events
: Device independent user interface events such as focus change messages or element triggering notifications.

Mutation Events
: Mutation events are caused by any action which modifies the structure of the document.

Trigger Events
: Trigger events are artificial events that are not associated with the user interface or mutations in the document. They are typically created (or asserted) through scripts or inserted through the document object model.

The [DOM2 [p.29]] provides an event flow architecture that describes how events are captured, bubbled, and canceled. In summary, event flow is the process through which an event originates from the DOM implementation and is passed into the document object model. The methods of event capture and event bubbling, along with various event listener registration techniques, allow the event to then be handled in a number of ways. It can be handled locally at the target `Node` level or centrally from a `Node` higher in the document tree.



Each event has a `Node` toward which the event is directed by the DOM implementation. This `Node` is the event target. In Figure 1, the event target is the node N_4. Figure 1 also shows how the event traverses the nodes of the document tree. The event flows from the root node to the target node passing through nodes N_1, N_2, and then to the target node N_4. After the event is processed by the target node, it flows back to the root node N_1.

Event capture is the process by which an ancestor of the event's target can register to intercept events of a given type before they are received by the event's target. In Figure 1, nodes N_1 and N_2, as ancestors of the target node, can capture the event prior to the event reaching N_4.

Event bubbling is the process by which an ancestor of the event's target can register to handle the events of a given type after they are received by the event's target. In Figure 1, nodes N_1 and N_2, as ancestors of the target node, can handle the event after it has reached N_4.

For any given event type, capturing is either enabled or it is disabled. If any ancestor to the target node enables capturing, all ancestors of the target node (who handle the appropriate event type) will capture the event. Likewise, if any ancestor to the target node disables capturing, all ancestors of the target node (who handle the appropriate event type) can no longer capture the event.

Similarly, for any given event type, bubbling is either enabled or disabled.

2. Event Module Elements

This section is normative.

This working draft proposes the `event` [p.7] and `eventlistener` [p.8] elements to support the features of the DOM Event Model.

The event element and the eventlistener element are named after the corresponding DOM interfaces. To avoid ambiguity in the text, when the element is referenced, the word element is used, as in:

- `event` element
- `eventlistener` element

If the DOM implementation is being referenced, only the event and eventlistener term is used, as in event and eventlistener.

There is some question whether the `event` element is actually needed. The working group is still exploring whether the semantics of the `event` element may be collapsed into the `eventlistener` element, thus reducing the syntax. In addition, there is some concern that since the semantics of the `event` element and `eventlistener` element are not absolutely the same as the DOM classes of the same name, that perhaps these elements should have different names.

2.1 The event Element

```
<!ENTITY % event-content "EMPTY" >
<!ELEMENT event (%event-content;)*
<!ATTLIST event
   id              ID         #IMPLIED
   type            NMTOKEN    #REQUIRED
   active          %Boolean   #IMPLIED
   cancelable      %Boolean   #IMPLIED
   bubbles         %Boolean   #IMPLIED
>
```

The `event` element provides a means for content authors to declare a trigger [p.5] event within a document or presentation. A trigger event may activated immediately, or it may be activated through some other means. The `event` element has the following attributes:

id

The `id` attribute is a document-unique identifier for the element. The value of this identifier is often used to manipulate the element through a DOM interface. Note that the `id` attribute and the `type [p.8]` attribute are not the same. The `id` value specifies a particular instance of an event. The `type [p.8]` value specifies a class of events that may be instantiated by several `event` elements within a document.

type
: The `type` of the event. The `type` attribute is used for associating an event handler with an event.

active
: The `active` attribute is true when the all of the conditions for activating the event are true. Content authors use this attribute to declare an event without it actually occurring and then activate the event through a DOM interface or through synchronization. The default value for `active` is true, which means that the event is immediately activated.

cancelable
: Some events are specified as cancelable. Cancellation is accomplished through the `eventlistener` [p.8] element's `prevent-default` [p.9] attribute or through the DOM Event interface's `preventDefault()` method. The default value for the `cancelable` attribute is false, which means that the event cannot be canceled.

bubbles
: Some events are specified as bubbling, which means that they follow the bubbling rules of the DOM event flow. The default value for the `bubbles` attribute is true, which means that the event will bubble upward after being dispatched to the target `Node`.

The target node for the event corresponding to an `event` element is the `event` element itself.

2.2 The eventlistener Element

```
<!ENTITY % eventlistener-content "EMPTY" >
<!ELEMENT eventlistener (%eventlistener-content;)*
<!ATTLIST eventlistener
   id               ID         #IMPLIED
   type             NMTOKEN    #REQUIRED
   register-with    IDREF      #IMPLIED
   trigger-once     %Boolean   #IMPLIED
   use-capture      %Boolean   #IMPLIED
   prevent-capture  %Boolean   #IMPLIED
   prevent-bubble   %Boolean   #IMPLIED
   prevent-default  %Boolean   #IMPLIED
>
```

The `eventlistener` element provides a means for handling events of a particular type. The `eventlistener` element has the following attributes:

id
: The `id` attribute is a document-unique identifier. The value of this identifier is often used to manipulate the element through a DOM interface.

type
: The `type` attribute specifies the event type for which the content author is registering. The `type` attribute has the following syntax:

```
Eventhandler-types ::= ((EventSource ".") ? Event-type )*
Event-source ::= Id-value
Event-type ::= NMTOKEN
```

The `type` attribute provides a means of having an `eventlistener` register for multiple event types.

register-with
: The `register-with` attribute specifies the target `Node` for this `eventlistener` element. The default value for the `register-with` attribute is "empty", which means that the `eventlistener` element registers with its parent `Node`. Otherwise, `register-with` is the value of desired target `Node` element's `id` attribute.

trigger-once
: The `trigger-once` attribute specifies whether the `eventlistener` element should be removed from its target `Node` after the `eventlistener` has processed the event. The default value for the `trigger-once` attribute is false, which means that the `eventlistener` remains registered with the it's target `Node`.

use-capture
: If true, the `use-capture` attribute indicates that the content author wishes to initiate capture. After initiating capture, all events of the specified type will be dispatched to the registered `eventlistener` before being dispatched to any elements beneath the `eventlistener` element's target `Node` in the tree. Events which are bubbling upward through the tree will not trigger an `eventlistener` designated to use capture. The default value for the `use-capture` attribute is false, which means that event flow proceeds normally.

prevent-bubble
: The `prevent-bubble` attribute is used to end the bubbling phase of the event flow. If this attribute is true for any `eventlistener` elements registered on the same target `Node` during bubbling, the bubbling phase will cease at that level and the event will not be propagated upward within the tree. The default value for `prevent-bubble` attribute is false: events will continue to bubble.

prevent-capture
: The `prevent-capture` attribute is used to end the capturing phase of event flow. If this attribute is true for any `eventlistener` elements registered on the same target `Node` during capturing, the capturing phase will cease at that level and the event will not be propagated any further down. The default value for `prevent-capture` attribute is false; events will continue through the capture phase.

prevent-default
: If an event is cancelable, the `prevent-default` attribute is used to signify that the event is to be canceled. If, during any stage of event flow, the `prevent-default` attribute is true, the event is canceled and any default action associated with the event will not occur. Setting this attribute to true for a non-cancelable event has no effect. The default value for the `prevent-default` attribute is false: events will not be canceled.

2.3. The Event Module Namespace

The Event Module will use the "http://www.w3.org/2000/event" namespace.

3. Using the Event Module in XHTML

This section is informative.

The Event Module may be integrated into XHTML to add extensibility to the event handling already present through a variety of properties. This section is informative: it is provided as a way of explaining how the Event Module may be used with XHTML.

3.1. Integrating the Event Module into XHTML

The first step of using the Event Module is to determine how it integrates with the other modules already in XHTML. One possible integration is to:

1. add the `eventlistener [p.8]` element to the content model for any existing XHTML element that supports intrinsic events
2. add the XHTML `script` element to the content model for the `eventlistener [p.8]` element

Adding the `eventlistener [p.8]` element to existing XHTML content models integrates the Event Module into XHTML; adding the `script` element to the `eventlistener [p.8]` element's content model integrates XHTML into the Event Module.

When an `eventlistener [p.8]` element contains a `script` element, the `script` element content is evaluated when the `eventlistener [p.8]` element processes an event. In addition, the DOM 2 Event object that corresponds to the event is available to the enclosed `script` element content through a "theEvent" object.

3.2. Registering an Event Handler

Content authors that wish to register an event handler for an element use the `eventlistener [p.8]` element.

```
Example 3.1

<img id="b" ...>
  <eventlistener id="a" type="(onclick)">
    <script>
        ... some scripting code ...
    </script>
  </eventlistener>
</img>
```

In Example 3.1, an `eventlistener [p.8]` element is registered with the `img` element. This `eventlistener [p.8]` element will be triggered when an `onclick` event targets or bubbles through the `img` element.

If the content author wishes to be more specific when declaring an `eventlistener [p.8]` element, they may add the target `Node`'s identifier to the `type` [p.8] attribute's value.

```
Example 3.2

<img id="b" ...>
  <eventlistener id="a" type="id(b)(onclick)">
    <script>
        ... some scripting code ...
    </script>
  </eventlistener>
</img>
```

In Example 3.2, the `eventlistener [p.8]` element is still registered with the `img` element. In this example, however, the `eventlistener [p.8]` will only be triggered when an `onclick` event has the `img` element as its target `Node`.

3.3. Event Bubbling

As specified in [DOM2 [p.29]], if bubbling is not prevented at the target `Node`, the event will follow the target `Node`'s parent chain upward, checking for any `eventlistener [p.8]` elements registered on each successive `Node`. Therefore, the following example provides the same behavior as that in Example 3.1.

```
Example 3.3

<div class="top">
  <eventlistener type="(onclick)">
    ...
  </eventlistener>
  <img id="start" ... />
</div>
```

In Example 3.3, if the bubbling of the `onclick` event is not prevented by the `img` element, it will bubble up to the parent `Node` (the `div` element) and it's registered `eventlistener [p.8]` . Where we see a difference in behavior is if we introduce a second object to the tree.

```
Example 3.4

<div class="top">
  <eventlistener type="(onclick)">
    ...
  </eventlistener>
  <img id="start" ... />
  <img id="stop" ... />
</div>
```

In Example 3.4, we can no longer distinguish, declaratively, between events bubbling from "start" image and events bubbling from "stop" image.

If the content author wishes to distinguish between these two types of events, they could write:

```
Example 3.5


<div class="top">
  <eventlistener type="id(start)(onclick)">
    ...
  </eventlistener>
  <eventlistener type="id(stop)(onclick)">
    ...
  </eventlistener>
  <img id="start" ... />
  <img id="stop" ... />
</div>
```

In Example 3.5, the first `eventlistener` [p.8] specifies the target `Node` with the "start" identifier and the second `eventlistener` [p.8] element specifies the target `Node` with the "stop" identifier.

Note: the author could have achieved a similar behavior by simply encapsulating the `eventlistener` [p.8] elements within their respective `img` elements.

3.4. Responding Once to an Event

In the previous examples, the `eventlistener` [p.8] element would respond each and everytime the corresponding event passed through its registered `Node`. Content authors may use the `trigger-once` [p.9] attribute to control whether an `eventlistener` [p.8] unregisters from its `Node` after it has been triggered.

```
Example 3.6


<div>
  <img id="cow" ... />
  <img id="cat" ... />
  <eventlistener id="cow-click" type="id(cow)(onclick)" trigger-once=true >
      ...
  </eventlistener>
  <eventlistener id="cat-click" type="id(cat)(onclick)" trigger-once=true >
      ...
  </eventlistener>
</div>
```

In Example 3.6, the `onclick` event is generated each and everytime that the user clicks on the "cow" image. The "cow-click" event handler only responds to the first instance of the `onclick` event. The same behavior is provided by the "cat-click" event handler for onclick events targeting the "cat" image.

3.5. Handling Multiple Events

Content authors may also register an `eventlistener [p.8]` that responds to multiple events.

```
Example 3.7


<eventlistener type="(onclick)(onstopkey)">
  ...
</eventlistener>
<img id="b" ... />
<img id="c" .../>
```

In Example 3.7, the `eventlistener [p.8]` element is listening for two events: an `onclick` event and an `onstopkey` event. In response to either of these events, the `eventlistener [p.8]` is triggered and the script is executed.

3.6. Preventing the Bubbling of Events

The DOM event model specifies that events continue to bubble up the document tree until an event handler cancels bubbling. Content authors may declaratively control bubbling through the `prevent-bubble [p.9]` attribute. For example:

```
Example 3.8

<div>
  <eventlistener id="b" type="(onclick)" ...>
    ...
  </eventlistener>
  <img id="logo" ... />
  <div>
    <eventlistener id="a" type="(onclick)" prevent-bubble="true">
      ...
    </eventlistener>
    <img id="stop" ... />
  </div>
</div>
```

In Example 3.8, we have two event handers for the `onclick` event. If the user clicks on the "stop" image, it is handled by event handler "a". Since event handler "a" cancels the bubbling action, it is not handled by event handler "b".

Content authors may procedurally control bubbling through script that accesses the DOM:

```
Example 3.9

<div>
  <eventlistener id="b" type="(onclick)" ...>
    ...
  </eventlistener>
  <img id="logo" ... />
```

```
    <div>
      <eventlistener id="a" type="(onclick)">
        <script>
          ... some scripting code ...
           theEvent.preventBubble();
          ... some scripting code ...
        </script>
      </eventlistener>
      <img id="stop" ... />
    </div>
  </div>
```

In Example 3.9, the `preventBubble()` method is called, ending the bubbling phase of event flow.

3.7. Capturing an Event

Content authors that wish to capture an event use the `eventlistener` [p.8] element:

```
Example 3.10

<eventlistener type="(onclick)(onstopkey)" use-capture="true" >
   ...
</eventlistener>
<img id="stop" ... />
...
</html>
```

In Example 3.10, the `eventlistener` [p.8] is specified as a peer to the "stop" image. Following the rules for event capturing outlined in [DOM2 [p.29]], if a user clicks on the object, the `onclick` event is dispatched to the target `Node` and any registered event listeners are triggered.

Unlike previous examples, the `use-capture` [p.9] attribute is "true", therefore the `eventlistener` [p.8] element intercepts the message before it reaches the target `Node`. This permits event handling to be centralized and isolated at a higher location in the document hierarchy.

After the `eventlistener` [p.8] has captured the event, it can let the event continue to propagate down the hierarchy to the target `Node`, or it can stop propagation by setting the `prevent-capture` [p.9] attribute to "true".

```
Example 3.11

<eventlistener type="(onclick)" use-capture="true" prevent-capture="true" >
   ...
</eventlistener>
<img id="b" ... />
```

In Example 3.11, the `eventlistener [p.8]` element captures the event and, by canceling the propagation of the event, effectively hides the event from `img` element. This behavior can also be accomplished through script:

```
Example 3.12

<eventlistener type="(onclick)" use-capture="true" >
   <script>
   ... some scripting code ...
       theEvent.preventCapture();
   ... some scripting code ...
   </script>
</eventlistener>
<img id="b" ... />
```

In Example 3.12, the event node's `preventCapture()` method is used to end the capturing phase of event flow and overrule the declarative syntax.

3.8. Declaring an Immediate Event

Content authors can specify an event that is immediately dispatched by setting the `active [p.8]` attribute to true. For example:

```
Example 3.13

<eventlistener type="(foo)">
   ...
</eventlistener>
<img id="b" ... >
   <event type="foo" />
</img>
```

In Example 3.13, the `eventlistener [p.8]` element is associated with a "foo" event. At some point in processing the document, the `img` element is processed and the event is immediately dispatched (since the `img` element contains an `event` element), triggering the `eventlistener [p.8]` element .

While this provides a simple syntax for declaring an event within a document, its real value may be that it provides a framework for the insertion of immediate events into the document through the DOM. For example, consider this reduction of the previous example:

```
Example 3.14

<eventlistener type="(foo)">
   <script>
      ... some scripting code ...
   </script>
</eventlistener>
<img id="b" ... />
```

In Example 3.14, we have removed the explicit declaration of the event (the `event` element was removed). The object might be a media player and the media player might insert the "foo" event through the DOM. Furthermore, the `event` [p.7] element and the `eventlistener` [p.8] element may both be inserted into the document. In this way, the event (represented by the `event` [p.7] element) and its associated behavior (represented by the `eventlistener` [p.8] element) can both be delivered in real-time.

3.9. Dispatching an Event from within the Document

Events can be dispatched by inserting an `event` [p.7] element through the Document Object Model, or by declaring an `event` [p.7] element from within the document. For example:

```
Example 3.15


<img id="b" ...>
  <event type="foo" active=false />
  <eventlistener type="(foo)">
    ...
  </eventlistener>
</img>
```

In this example, we declare an `event` [p.7] element that is a "foo" event. The `active` [p.8] attribute for this `event` [p.7] element is false, which means that while we have declared the event, it is not currently available to the event flow architecture. If the `active` [p.8] attribute is set to true (perhaps through a DOM interface), the appropriate `eventlistener` [p.8] will receive the event.

4. Using the Event Module in SMIL

This section is informative.

The Event Module may be integrated into SMIL to add event handling. This section is informative; it is provided as a way of explaining how the Event Module may be used with SMIL. Much of this syntax may evolve if and when SMIL adopts a user event model.

4.1. Integrating the Event Module into SMIL

The first step of using the Event Module is to determine how it integrates with the other modules already in SMIL. One possible integration is to:

1. add the `eventlistener` [p.8] element to the content model of the timing structure elements
2. add the `eventlistener` [p.8] element to the content model of the media object elements
3. add timing properties to the `eventlistener` [p.8] element

Adding the `eventlistener` [p.8] element to the various SMIL element content models provides a means of integrating the Event Module into the SMIL language; adding the timing properties to the `eventlistener` [p.8] element provides a means of scheduling and synchronizing event handlers.

4.2. Registering an Event Handler

Content authors that wish to register an event handler use the `eventlistener` [p.8] element:

```
Example 4.1

<par>
   <eventlistener type="(onclick)" />
   <img id="c" ... />
</par>
...
</smil>
```

In Example 4.1, an `eventlistener` [p.8] element is specified as a peer to the `img` element.

Following the rules for event bubbling outlined in [DOM2 [p.29]], if a user clicks on the image, the `onclick` event is dispatched to the target `Node` (image "c") and any event handlers registered to this `Node` are triggered. If bubbling is not canceled, the event will follow the target `Node`'s parent chain upward, checking for any event handlers registered on each successive `Node`.

In Example 4.1, the `onclick` event would bubble to the `par` element, which has registered the `eventlistener [p.8]` element.

Content authors may also associate an event handler with a region:

```
Example 4.2

<head>
  <layout>
    <region id="a" top="5" />
    <eventlistener register-with="a" type="(onclick)" />
  </layout>
</head>
<body>
  <text region="a" src="text.html" dur="10s" />
</body>
</smil>
```

In Example 4.2, the `eventlistener [p.8]` element is bound to region "a" using the `register-with [p.9]` [p.9] attribute (`region` is an empty element).

4.3. Responding to an Event

Registering an `eventlistener [p.8]` element is of little value unless there is a means of responding to an event. In SMIL, content authors may control other media elements and timelines by referring to the `eventlistener [p.8]` element.

```
Example 4.3

<par>
   <img id="c" ... />
   <eventlistener id="a" type="(onclick)" />
   <audio id="b" begin="id(a)(trigger)" ... />
</par>
...
</smil>
```

In Example 4.3, the `eventlistener [p.8]` element is specified as a peer to the `audio` element and the `img` element. The `audio` element begins playing when the `eventlistener [p.8]` element is triggered. In practical terms, this means that the `audio` object will begin playing when the `img` has been clicked.

4.4. Synchronizing Event Handlers

Adding temporal properties to the `eventlistener [p.8]` element provides a means for synchronizing event handlers. Content authors can enable and disable `eventlistener [p.8]` elements over time and change `eventlistener [p.8]` elements depending on time. For example:

```
Example 4.4

<par>
   <eventlistener id="a" type="(onclick)" begin="2s"/>
   <audio id="b" begin="id(a)(trigger)" ... />
   <img id="c" ... />
</par>
```

In Example 4.4, the `eventlistener` [p.8] element is not registered during the first 2 seconds of displaying the image. Using SMIL, we can then have different `eventlistener` [p.8] elements based on the current position in the timeline. For example:

```
Example 4.5

<par>
   <eventlistener id="a" type="(onclick)" end="2s"/>
   <eventlistener id="d" type="(onclick)" begin="2s"/>
   <audio id="b" begin="id(a)(trigger)" end="id(e)(begin)"... />
   <audio id="e" begin="id(b)(trigger)" ... />
   <img id="b" ... />
</par>
```

In Example 4.5, there are two `eventlistener` [p.8] elements. The first `eventlistener` [p.8] element, "a", is active during the first 2 seconds of the timeline. The second `eventlistener` [p.8] element, "d" is active starting at 2 seconds and ending when the timeline is complete. There are two audio files that conditionally play depending on which `eventlistener` [p.8] element is triggered.

4.5. Handling Multiple Events

Content authors may also register an `eventlistener` [p.8] that responds to multiple events.

```
Example 4.6

<par>
   <eventlistener id="a" type="(onclick)(onstopkey)" end="2s"/>
   <audio id="b" end="id(a)(trigger)"... />
   <img id="c" ... />
</par>
...
</smil>
```

In Example 4.6, the `eventlistener` [p.8] element is listening for two events: an `onclick` event and an `onstopkey` event. In response to either of these events, the `eventlistener` [p.8] element is triggered, subsequently causing the audio object to stop playing.

The previous examples illustrated `eventlistener` [p.8] elements that responded to a particular type of event for all of their descendants. In some situations, content authors may wish to handle specific events for specific children. For example:

```
Example 4.7

<par>
   <eventlistener id="a" type="id(stop)(onclick), onstopkey" end="2s"/>
   <audio id="b" end="id(a)(trigger)"... />
   <img id="stop" ... />
   <img id="play" ... />
</par>
...
</smil>
```

In Example 4.7, the `eventlistener [p.8]` element responds to `onclick` events targeting the "stop" image but not `onclick` events targeting the "play" image.

Event handlers are implicitly associated with an element when an element refers to an event with that element's id:

```
Example 4.8

<par>
   <audio id="b" end="id(stop)(onclick)"... />
   <img id="stop" ... />
   <img id="play" ... />
</par>
...
</smil>
```

In Example 4.8, the `audio` element refers to an `onclick` event for the element with an id of "stop".

4.6. Responding Once to an Event

Content authors may use the `trigger-once [p.9]` attribute to control whether an `eventlistener [p.8]` unregisters from its target `Node` after it has been triggered.

```
Example 4.9

<par>
   <img id="cow" ... />
   <img id="cat" ... />
   <eventlistener id="cow-click" type="id(cow)(onclick)" trigger-once=true />
   <eventlistener id="cat-click" type="id(cat)(onclick)" trigger-once=true />
   <par begin="id(cow-click)(trigger)" ... />
      <audio id="moo" ... />
      <video id="cow-jump" ... />
   </par>
   <par begin="id(cat-click)(trigger)" ... />
      <audio id="meow" ... />
      <video id="cat-fiddle" ... />
   </par>
</par>
...
</smil>
```

In Example 4.9, the `onclick` event is generated each and every time that the user selects the "cow" image. The event handler "cow-click", however, only responds to the first instance of the `onclick` event, ignoring subsequent `onclick` events. The same behavior is provided by the "cat-click" event handler for `onclick` events targeting the "cat" image.

4.7. Preventing the Bubbling of an Event

The [DOM2 [p.29]] event model specifies that events continue to bubble up the document tree until an event handler cancels bubbling. Content authors may declaratively control bubbling through the `prevent-bubble [p.9]` attribute. For example:

```
Example 4.10

<par>
  <eventlistener id="c" type="(onclick)" ... />
  <img id="logo" ... />
  <par>
    <eventlistener id="a" type="(onclick)" prevent-bubble="true" />
    <audio id="b" end="id(a)(trigger)" ... />
    <img id="stop" ... />
  </par>
</par>
...
</smil
```

In Example 4.10, we have two `eventlistener [p.8]` elements for the `onclick` event. If the user clicks on the "stop" image, it is handled by event handler "a". Since event handler "a" cancels the bubbling action, it is not handled by event handler "c".

4.8. Capturing an Event

Content authors that wish to capture an event use the `eventlistener [p.8]` element:

```
Example 4.11

<par>
   <eventlistener use-capture="true" type="(onclick)" />
   <img id="c" ... />
</par>
...
</smil>
```

In Example 4.11, the `eventlistener [p.8]` element is specified as a peer to image element "c". Following the rules for event capturing outlined in [DOM2 [p.29]], if a user clicks on the image, the `onclick` event is dispatched to the target `Node` (image "c") and any registered event handlers are triggered.

Unlike the previous example, the `use-capture [p.9]` attribute is "true", therefore the `eventlistener [p.8]` element intercepts the message from the descendent target `Node`. This permits event handling to be centralized and isolated at a higher location in the document hierarchy.

After the `eventlistener` [p.8] element has captured the event, it can let the event continue to propagate down the hierarchy to the target `Node`, or it can stop propagation by setting the `prevent-capture` [p.9] attribute to "true".

```
Example 4.12

<par>
   <eventlistener use-capture="true" prevent-capture="true" type="(onclick)" />
   <img id="c" ... />
</par>
```

In Example 4.12, the `eventlistener` [p.8] element captures the event and, by canceling the propagation of the event, effectively hides the event from image element "c".

4.9. Declaring an Immediate Event

Content authors can specify an event that fires immediately upon activation of the enclosing timeline:

```
Example 4.13

<par>
   <eventlistener id="a" type="(onintroend)" />
   <seq>
      <audio id="b" ... />
      <event type="onintroend" />
      <video id="d" ... />
   </seq>
   <img id="e" begin="id(a)(trigger)" .../>
   ...
</par>
```

In Example 4.13, we have an `audio` element that plays and then a `video` element that plays. At the end of the `audio` element, an `onintroend` event is generated which triggers the display of an `img` element.

While this provides a simple syntax for declaring an event within a document, its real value may be that it provides a framework for the insertion of immediate events into the document through the DOM. For example, consider this reduction of the previous example:

```
Example 4.14

<par>
   <eventlistener id="a" type="(onintroend)" />
   <seq>
      <audio id="b" ... />
      <video id="d" ... />
   </seq>
   <img id="e" begin="id(a)(trigger)" .../>
   ...
</par>
```

In Example 4.14, we have removed the explicit declaration of the event. Rather, the audio player may generate an `onintroend` event and insert the corresponding `event [p.7]` element into the document using a DOM interface, thereby triggering the `eventlistener [p.8]` element.

4.10. Synchronizing Events

The `event [p.7]` element provides a means for declaring events that are timed or synchronized with other element timelines. For example:

```
Example 4.15

<par>
   <eventhandler id="a" type="(foo)" />
   <audio id="b" ... />
   <event type="foo" begin="id(b)2s" />
   <video id="d" ... />
   <img id="e" begin="id(a)(trigger)" .../>
   ...
</par>
```

In Example 4.15, the `event [p.7]` element is synchronized with the `audio` element "b": two seconds into playing the audio, the "foo" event is dispatched and caught by the `eventlistener [p.8]` . This causes the `img` element to be rendered.

As in the previous section, this provides a framework for inserting timed events through a DOM interface.

```
Example 4.16

<par>
   <eventhandler id="a" type="(foo)" />
   <audio id="b" ... />
   <video id="d" ... />
   <img id="e" begin="id(a)(trigger)" .../>
   ...
</par>
```

In Example 4.16, the `event [p.7]` element is no longer declared, but an external process could insert an `event [p.7]` element while the document is rendering and have the "foo" event triggered at the appropriate time. This is important in a streaming architecture when triggered events might be streamed prior to the actual time at which they should fire.

4.11. Synchronizing on Multiple Events

As previously shown, multiple events may be handled by the same `eventlistener [p.8]` element. For example:

```
Example 4.17

<par>
   <eventlistener id="a" type="(foo)(onclick)" />
   <audio id="b" ... />
   <event id="c" type="foo" begin="id(b)10s" />
   <video id="d" ... />
   <img id="e" begin="id(a)(trigger)" .../>
   ...
</par>
```

In Example 4.17, the `eventlistener` [p.8] element responds to an `onclick` event and a "foo" event (the "foo" event occurs 10 seconds into the audio). Therefore, the `eventlistener` [p.8] element will be triggered when the user clicks on a child of the `par` element or after 10 seconds, whichever occurs first.

Appendix A. A Comparison with BECSS

A recent working draft [BECSS] [p.29] defines a means of declaring standalone behaviors that can be attached to HTML or XML elements without modification of the document type definition (DTD).

1. While not explicitly stated, the requirements for the BECSS solution appear to be:
2. Scripts can be defined external to a document
3. External scripts can be reused across multiple documents
4. External scripts can be combined for use by a document
5. The documents DTD does not need to be modified
6. A means of extending the properties and behaviors of an element without requiring new DOM interfaces

The requirements for the XHTML Event Module defined in this draft were:

- To expose the DOM event model to an XML document
- To allow for new event types without modification to the DOM or the DTD.
- To only require XML in the implementation.
- To be able to integrate with HTML, SMIL, and other languages

The BECSS solution is considerably more expressive since it provides a mechanism for extending the properties and behaviors of an element (albeit this is benefit is not available to languages other than ECMAScript). There was no requirement on the XHTML Event Module for such capability.

The BECSS solution also provides a means of reusing event "sheets" across multiple documents (Requirements 1-4, above [p.27]). External definition of events could be accomplished by linking, but reuse is limited until a selector syntax is introduced (adding complexity). There was no requirement on the XHTML Event Module for such capability.

The BECSS solution does require user agents to implement CSS in addition to XML. This limits its usefulness in languages that do require CSS (such as SMIL) and small platforms such as that proposed by the Wireless Applications Protocol Forum [REF]. The XHTML Event Module focused on ensuring that event handlers and events could participate in the SMIL time model and its intrinsic timeline behaviors.

In conclusion, while there is some overlap in functionality (they both expose the event model to programmers), the functionalities are not identical. Each solution has benefits (BECSS exposes custom properties and XHTML events simply integrate with XML), and each has its deficiencies (BECSS is complexity and requires CSS and XHTML event have no selector syntax). More importantly, these two solutions can be simultaneously supported if so desired.

Appendix B. References

[BECSS] Behavioral Extensions to CSS, Apparao V., et.al., http://www.w3.org/TR/becss. This document is a work in progress.

[DOM2] Document Object Model (DOM) Level 2 Specification, Wood L., et.al., http://www.w3.org/TR/WD-DOM-Level-2 . This document is a work in progress.

[SMOD] SMIL Modules

[XMOD] XHTML Modules

[WAIEVT] http://www.w3.org/WAI/PF/Group/DOM/Events-19990813