

User Interface Markup Language (UIML) Draft Specification

Document Version 1-August-1999

Language Version 2.0

Proposed by Universal Interface Technologies, Inc.

Editor: Constantinos Phanouriou (uiml-editor@uiml.org)

© Copyright Universal Interface Technologies, Inc., 1999

Disclaimer: This document is subject to change without notice.

Contents

1	<i>Introduction to UIML 2.0</i>	4
1.1	Overview	4
1.1.1	Relationship of UIML to XML	5
1.1.2	Purpose of this Document.....	5
1.2	Terminology	5
1.3	Goals of UIML	6
2	<i>Document Status</i>	8
2.1	Editor	8
2.2	Copyright Notice	8
2.3	Errata	8
2.4	Comments	9
3	<i>Structure of a UIML Document</i>	10
3.1	Example of a UIML Document	11
4	<i>The uiml and head elements</i>	14
4.1	The <i>uiml</i> Element	14
4.2	The <i>head</i> Element	14
5	<i>Interface Description</i>	16
5.1	Overview	16
5.2	Attributes Common to Multiple Elements	16
5.2.1	The <i>name</i> and <i>class</i> Attributes	16
5.2.2	The <i>source</i> Attribute	17
5.3	The <i>interface</i> Element	19
5.4	The <i>structure</i> Element	20
5.4.1	The <i>part</i> Element.....	21
5.5	The <i>content</i> Element	22
5.5.1	The <i>property</i> Element	24
5.5.2	The <i>reference</i> Element.....	26
5.6	The <i>style</i> Element	27
5.6.1	The <i>property</i> Element	29
5.7	The <i>action-list</i> Element	29
5.7.1	The <i>rule</i> Element	36
5.7.2	The <i>condition</i> Element.....	37
5.7.3	The <i>equal</i> Element.....	38
5.7.4	The <i>reference</i> Element.....	39
5.7.5	The <i>event</i> Element	39
5.7.6	The <i>action</i> Element	40
5.7.7	The <i>property</i> Element.....	41
5.7.8	The <i>call</i> Element	41
5.7.9	The <i>param</i> Element	42
5.7.10	The <i>edit</i> Element.....	42

UIML 2.0 Language Reference

6	<i>Application Logic</i>	45
6.1	The <i>logic</i> Element	45
6.2	The <i>object</i> Element	46
6.2.1	The <i>function</i> Element.....	48
6.2.2	The <i>param</i> Element	49
6.2.3	The <i>returns</i> Element	49
6.2.4	The <i>script</i> Element	50
7	<i>Mapping Property and Event Names to Toolkits</i>	51
7.1	The <i>toolkit-peers</i> Element	51
7.2	The <i>toolkit</i> Element	52
7.2.1	The <i>widget</i> Element.....	53
7.2.2	The <i>property-method</i> Element.....	53
7.2.3	The <i>event</i> Element.....	54
8	<i>Alternative Organizations for UIML Files</i>	56
8.1	Splitting a UIML Document into Multiple Files	56
	<i>References</i>	58
	<i>Appendix – UIML 2.0 Document Type Definition</i>	59

1 Introduction to UIML 2.0

The User Interface Markup Language (UIML) is a declarative language for describing user interfaces in a highly device-independent manner. By “device” we mean PCs, various information appliances (e.g., handheld computers, desktop phones, cellular or PCS phones), or any other machine that a human can interact with. The use of UIML for the user interface facilitates a clean separation between the interface and the application logic.

UIML is intended to be an open, standardized language. Submission to a standards organization will occur after comments are received and this draft specification is finalized.

For a discussion of the motivation for and uses of UIML, please see Abrams et al [4].

This specification is a revision of the original UIML 1.0 specification, originally created in 1997 [5].

1.1 Overview

In UIML version 2.0, a user interface is a set of interface elements with which the user interacts. Each interface element is called a *part*; just as an automobile or a computer is composed of a variety of parts, so is a user interface. The parts may be organized differently for different categories of users and different families of devices. Each interface part has *content* (e.g., text, sounds, images) used to communicate information to the user. Interface parts can also receive information from the user using interface artifacts (e.g., a scrollable selection list) from the underlying device. Since the artifacts vary from device to device, the actual mapping (rendering) between an interface part and the associated artifact (widget) is done using a *style* element.

As a user interacts with a user interface, the user interface generates *events*. Events can be handled within the UIML document, by scripts in some scripting language, or by the application logic. UIML supports a variety of modes of interface/backend communication. For example, the user interface can call the backend, or the backend can modify the user interface programmatically.

UIML can be viewed as a meta-language or extensible language, analogous to XML. XML does not contain tags specific to a particular purpose (e.g., HTML’s <H1> or). Instead, XML is combined with a document type definition (DTD) to specify what tags are legal in a particular markup language that is XML-compliant. The advantage is that UIML can be standardized once, rather than requiring periodic standardization committee meetings to add new tags as the language evolves.

Analogously, UIML does not contain tags specific to a particular user interface toolkit (e.g., <WINDOW> or <MENU>). UIML captures the elements that are common to any user interface: an enumeration of the user interface parts, events that occur for those parts, presentation style, content (text, images, sounds), and interconnection logic between parts or with one or more backend applications. UIML syntax also defines language elements that map these elements to a particular toolkit. However, the vocabulary of particular toolkits (e.g., a window or a card) is not part of UIML, because the vocabulary appears as the value of attributes

UIML 2.0 Language Reference

in UIML. Thus UIML only needs to be standardized once, and different constituencies of users can define vocabularies that are suitable for various toolkits independently of UIML.

Thus to use UIML, you need more than this document, which specifies the UIML language. You also need one document for each toolkit (e.g., Java Swing, Microsoft Foundation Classes) to which you wish to map UIML. The toolkit-specific document enumerates what names (e.g., of properties) are used for a particular toolkit.

1.1.1 Relationship of UIML to XML

UIML is an XML language. This document uses XML 1.0 W3C specification [1], but future versions of the language will track the future XML specifications. Appendix A contains the UIML 2.0 DTD.

1.1.2 Purpose of this Document

This document serves as the official language reference for UIML 2.0. It describes the syntax of the elements and their attributes, the structure of UIML documents, and usage examples. It also gives pointers to other reference documentation that may be helpful when developing applications using UIML.

Comments are solicited on this specification. Feedback on this draft will be used in further refinements of the language. Please send comments to uiml-editor@uiml.org.

This document is intended for UIML user interface developers. This version of the UIML specification may be distributed freely, as long as all text and legal notices remain intact.

1.2 Terminology

Certain terminology is used in the specification:

Application: When we speak of building a user interface, the user interface along with the underlying logic that implements the functionality visible through the interface is called the application.

Application Logic: Code that is part of the application but not part of the user interface.

Device: A device is a physical object with which an end user interacts using a user interface, such as a PC, a handheld or palm computer, a cell phone, an ordinary desktop voice telephone, or a pager.

User Interface Toolkit: A toolkit is the markup language or software library upon which an application's user interface runs. Note that we use the word "toolkit" in a more general sense than its traditional use, because we include markup languages that are capable of representing user interfaces. Examples of toolkits include Java AWT, Java Swing, Microsoft Foundation Classes, Wireless Markup Language (WML), HTML, and SpeechML.

Platform: A platform is a combination of a device, operating system (OS), and a user interface toolkit. An example of a platform is a PC running Windows NT on which applications use the Java Swing toolkit. Another example is a cellular phone running a manufacturer-specific OS and a WML [6] renderer.

Rendering: Rendering is the process of converting a UIML document into a form with which an end user can interact. In the same sense that a Web browser renders HTML by creating a visual representation of a document from HTML, a UIML renderer creates a functional user interface from UIML. In the case of UIML, rendering might be accomplished by a program that interprets UIML while the user interacts with the interface (analogous to what a Web browser does for HTML), or UIML might be translated to another language (e.g., via XSL [7] into WML) and then the other language rendered (e.g., via a WML renderer).

UI Widget: The user interface toolkit with which the user interface is implemented provides primitive building blocks, which we call widgets. The term “widget” is traditionally used in conjunction with a graphical user interface, but we use it in a more general sense. UIML describes how to combine UI widgets.

For example, a widget might be a component in the Microsoft Foundation Classes or Java Swing toolkits, or a card or a text field in a WML document. In some toolkits, a widget name is a class name (e.g., the `java.awt.Button` class in the Java AWT toolkit, or the `CWindow` class in Microsoft Foundation Classes). If the toolkit is a markup language (e.g., WML, HTML, SpeechML) then a widget name may be a tag name (e.g., “CARD” or “TEXT” for WML). The definition of names is outside the scope of this specification, as explained in Section 1.1.

Runtime: This is the period of time during which the user interface is available for the user to interact with.

The following are terms and conventions used throughout this specification.

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 [8].

1.3 Goals of UIML

UIML is designed to do the following:

- provide a natural separation between user interface code and non-interface code,
- be usable by non-programmers,
- reduce the time to develop user interfaces for multiple device families,
- facilitate rapid prototyping on many platforms,
- facilitate internationalization and localization,

UIML 2.0 Language Reference

- allow efficient download of user interfaces over networks to Web browsers,
- facilitate security, and
- be extensible to support future technologies.

2 Document Status

Version of this document are available online in the following formats:

- Latest version of UIML 2.0 spec: <http://www.uiml.org/docs/uiml20>
- This version of the document:
 - HTML: <http://www.uiml.org/docs/uiml20-990801>
 - PDF: <http://www.uiml.org/docs/uiml20-990801.pdf>

2.1 Editor

Constantinos Phanouriou, Universal Interface Technologies, Inc.
uiml-editor@uiml.org

2.2 Copyright Notice

© Copyright Universal Interface technologies, Inc., 1999. All rights reserved.

Permission to use, copy, and distribute the contents of this document, but not to excerpt it, modify it, or create derivative works, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on *ALL* copies of the document, or portions thereof, that you use:

1. A link to or statement of the URL www.uiml.org/docs/uiml20.
2. The pre-existing copyright notice of the original author. If no such notice exists, a notice of the form: "© Copyright Universal Interface technologies, Inc., 1999. All rights reserved."

COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE.

2.3 Errata

The list of known errors in this specification is available at:
<http://www.uiml.org/docs/UIML20-errata.html>

Please report errors in this document to uiml-editor@uiml.org.

2.4 Comments

Comments regarding this document can be submitted to the editor of this document in the manner published at <http://www.uiml.org> by sending email to uiml-editor@uiml.org.

3 Structure of a UIML Document

A typical UIML 2.0 document is composed of these parts:

1. A prolog of two lines identifying the XML language version and encoding and the location of the UIML2.0 document type definition (DTD):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE uiml PUBLIC
  "-//Universal Interface Technologies//UIML 2.0//EN"
  "http://uiml.org/dtd/UIML20.dtd">
```

2. The root element in the document, which is the *uiml* tag. The *uiml* tag also contain version information:

```
<uiml version="2.0"> ... </uiml>
```

See Section 4.1 for more information on the root element *uiml*. The *uiml* element contains five subelements:

- a) A header element giving metadata about the document:

```
<head> ... </head>
```

The *head* element is discussed in Section 4.2.

- b) An element that describes the mapping from property and event names used in the UIML document to primitives in the toolkits on which the interface will be deployed:

```
<toolkit-peers> ... </toolkit-peers>
```

Discussion of the *toolkit-peers* element is deferred until much later in Section 7, because this element usually just names an external file that contains the mappings for each toolkit. Thus a reader can normally skip reading Section 7.

- c) An interface description, which contains information about the structure, content, style, and actions of the interface:

```
<interface> ... </interface>
```

Section 5.3 discusses the *interface* element.

- d) An element that describes how the user interface interacts with the underlying logic that implements the functionality manifested through the interface.

```
<logic> ... </logic>
```

See Section 6 for a discussion of the *logic* element.

White space (spaces, newlines, tabs, and comments) may appear before or after each of the above tags.

To summarize, here is a skeleton of a UIML document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE uiml PUBLIC
  "-//Universal Interface Technologies//UIML 2.0//EN"
  "http://www.uiml.org/dtds/uiml20.dtd">

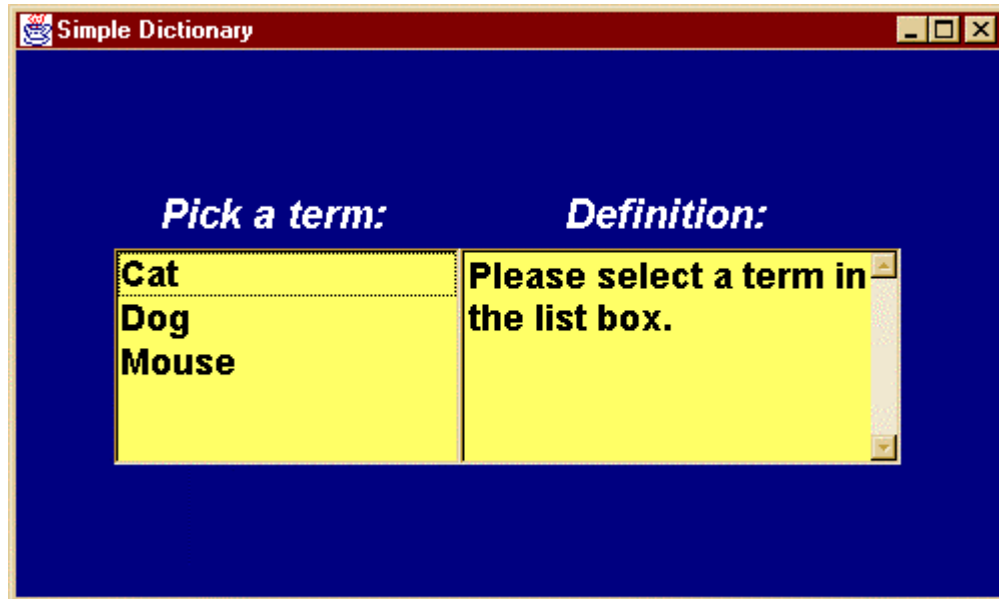
<uiml>          ... </uiml>
  <head>        ... </head>
  <toolkit-peers> ... </toolkit-peers>
  <interface>   ... </interface>
  <logic>       ... </logic>
</uiml>
```

Section 5, 7, and 6 describe the elements that are contained in the *uiml* element.

3.1 Example of a UIML Document

This section contains one simple example of a UIML document. For further examples, please see [2].

The example below displays a single window representing a dictionary. The dictionary contains of a list box in which a user can click on a term (i.e., dog, cat, mouse). The dictionary also contains a text area in which the definition of the currently selected term is displayed. The style sheet maps the interface to the Java AWT toolkit.



UIML 2.0 Language Reference

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE uiml PUBLIC
  "-//Universal Interface Technologies//UIML 2.0//EN"
  "http://www.uiml.org/dtds/uiml20.dtd">

<!-- This is Dictionary.ui.
  Displays one window on the screen containing a list of animals
  and a textbox. Clicking an animal's name displays a definition in the
  textbox. -->

<uiml>

  <toolkit-peers>
    <toolkit name="java" source="http://uiml.org/toolkits/Java20AWT.ui"/>
  </toolkit-peers>

  <interface>

    <structure>
      <part class="Frame"      name="ListBoxes">
        <part class="Label"   name="IntroLabel"/>
        <part class="List"    name="Terms"/>
        <part class="Label"   name="DefnLabel"/>
        <part class="TextArea" name="Defn"/>
      </part>
    </structure>

    <style source="http://uiml.org/ui/JavaAWTRenderings.ui#AWT"
      mode="merge-children">
      <property class="Frame"      pname="layout"      >gridBagLayout</property>
      <property class="Frame"      pname="xplace"      >relative</property>
      <property class="Frame"      pname="yplace"      >relative</property>
      <property class="Frame"      pname="background"  >blue</property>
      <property class="Frame"      pname="font-style"  >bold</property>
      <property class="Frame"      pname="location"    >100,100</property>
      <property class="Frame"      pname="size"        >500,300</property>

      <property class="Label"      pname="font-size"   >20</property>
      <property class="Label"      pname="foreground"  >white</property>
      <property class="Label"      pname="font-style"  >boldItalic</property>

      <property class="List"       pname="background"  >gray</property>

      <property class="TextArea"    pname="background"  >gray</property>

      <property part="ListBoxes"   pname="content"     >Simple Dictionary</property>
      <property part="IntroLabel"  pname="content"     >Pick a term:</property>
      <property part="Terms"       pname="content">
      <constant name="Cat"      >Cat</constant>
      <constant name="Dog"      >Dog</constant>
      <constant name="Mouse"    >Mouse</constant>
      </property>
      <property part="DefnLabel"   pname="content">Defn:</property>
      <property part="Defn"       pname="content">Please select a term...</property>

      <property part="Terms"      pname="xplace"      >0</property>
      <property part="Terms"      pname="alignment"   >north</property>
      <property part="Terms"      pname="fill"        >both</property>

      <property part="DefnLabel"   pname="alignment"   >center</property>
      <property part="DefnLabel"   pname="xplace"      >1</property>
      <property part="DefnLabel"   pname="yplace"      >0</property>
    </style>
  </interface>
</uiml>
```

UIML 2.0 Language Reference

```
<property part="Defn" pname="xplace" >1</property>
<property part="Defn" pname="columns" >20</property>
<property part="Defn" pname="rows" >4</property>
<property part="Defn" pname="scrollbars">vertical-only</property>
<property part="Defn" pname="editable" >false</property>
</style>

<action-list>
  <rule>
    <condition>
      <equal>
        <event part="Terms" class="LSelected" pname="item-selected"/>
        <reference name="Cat"/>
      </equal>
    </condition>
    <action>
      <property part="Defn" pname="content">Carnivorous, domesticated mammal that's
fond of rats and mice</property>
    </action>
  </rule>

  <rule>
    <condition>
      <equal>
        <event part="Terms" class="LSelected" pname="item-selected"/>
        <reference name="Dog"/>
      </equal>
    </condition>
    <action>
      <property part="Defn" pname="content">Domestic animal related to a wolf that's
fond of chasing cats</property>
    </action>
  </rule>

  <rule>
    <condition>
      <equal>
        <event part="Terms" class="LSelected" pname="item-selected"/>
        <reference name="Mouse"/>
      </equal>
    </condition>
    <action>
      <property part="Defn" pname="content">Small rodent often seen running away from a
cat</property>
    </action>
  </rule>

</action-list>

</interface>

</uiml>
```

4 The *uiml* and *head* elements

Whenever a new element is introduced in the remainder of the document, we first give the appropriate DTD fragment.

4.1 The *uiml* Element

DTD

```
<!ENTITY % AppId                "ID">
<!ELEMENT uiml (head?, toolkit-peers, interface, logic?)>
<!ATTLIST uiml
    name      %AppId; #IMPLIED
    version   CDATA   #FIXED   "2.0">
```

Description

The *uiml* element is the most fundamental element in a UIML document. It provides the basic structure of a UIML application. All other elements are contained in the *uiml* element. The *uiml* element appears as follows:

```
<uiml version="2.0">...</uiml>
```

Usually, one *uiml* element equates to one file, in much the same way that there is one HTML element per file when developing HTML-based applications. However, other arrangements are possible. For example, the *uiml* element might be retrieved from a database or the elements contained within the *uiml* element might be stored in multiple files.

Attributes

<i>name</i> (see Section 5.2.1)	The <i>name</i> attribute is optional and assigns a name to the application whose interfaces are described within the <i>uiml</i> element. The chief reason for using the name attribute is to set properties associated with the application. In particular, there is an attribute "exists" for any application, whose initial value is "true". If the attribute's value is set to "false" by any <i>action-list</i> element, then the application will terminate execution and the user interface will cease to exist.
<i>version</i>	Must be the string "2.0". Normally this is omitted.

4.2 The *head* Element

DTD

```
<!ELEMENT head EMPTY>
```

Description

The *head* element contains metadata about the current UIML document. Elements in the head element are not considered part of the interface, and have no effect on the rendering or operation of the user interface.

UIML authoring tools should use the *head* element to store information about the document (e.g., author, date, version, etc...) and other proprietary information.

5 Interface Description

This section describes the elements that go inside the *interface* element, their attributes, and their syntax. Examples are provided to help show common usage of each element.

5.1 Overview

The *interface* element contains four subelements: *structure*, *style*, *content*, and *action-list*:

```
<interface>
  <structure> </structure>
  <style>     </style>
  <content>   </content>
  <action-list> </action-list>
</interface>
```

The *structure* element enumerates a set of interface parts and their organization for various platforms.

The *style* element defines the values of various properties associated with interface parts (analogous to style sheets for HTML).

The *content* element gives the words, sounds, and image associated with interface parts to facilitate internationalization or customization of user interfaces to various user groups (e.g., by job role).

The *action-list* element defines what user interface events should be acted on and what should be done.

5.2 Attributes Common to Multiple Elements

Before explaining each of the elements introduced in Section 3, we first describe some attributes that are used in several of the elements.

5.2.1 The *name* and *class* Attributes

Most elements in UIML may have a *name* and a *class* attribute.

The *name* attribute assigns a unique identifier to that element. No two elements of the same type (e.g., no two *part* elements) can have the same name. (An exception is the *property*, *content*, and *action-list* elements. There may be multiple *property* elements in a document with the same *name* attribute. Semantically, this is equivalent to concatenating the children of the elements into one *property* element, preserving the relative order of the children from the original document. An analogous statement applies to the *content* and *action-list* elements.) However, elements of two different types may have the same name. Thus there may be a *part* as well as a *content* element that both have the "Label2" for their *name* attributes. However, two *part* elements may not both have "Label2" for their *name* attribute.

The *class* attribute assigns a class name to an element. Any number of elements may be assigned the same class name.

The use of the attribute *class* is based on the CSS [3] concept of class: a class specifies an object *type*, while the element's "name" uniquely identifies an *instance* of that type. A style associated with all instances of a class is associated with all elements that specify the same value for their *class* attribute; a style associated with a specific instance of a class is associated with any elements that specify the same value for their *name* attribute.

5.2.2 The *source* Attribute

The *source* attribute is used to include another element or the content of a URI into a point within a UIML document. There are two variants, depending on the value of the *mode* attribute that is used with *source*:

- *mode*="merge-children"
- *mode*="replace"

An element *e* uses *mode*="merge-children" to populate its children. In contrast, *e* uses the *mode*="replace" attribute to replace *e* itself.

5.2.2.1 Mode="merge-children"

One UIML element can inherit attributes from another element using the *source* and *mode*="merge-children" attributes. Syntactically, the *mode*="merge-children" relationship is expressed like this:

```
<style name="BasicStyle"> ... </style>

<style name="FancyStyle" source="BasicStyle" mode="merge-children">
  ...
</style>
```

The *mode*="merge-children" attribute has the following semantics: At the time that element *y* in the user interface is rendered, the UIML document is represented as an XML tree. The tree is modified as follows. For each node in the tree with name *y* that contains the *source*="x" and *mode*="merge-children" attributes, the rendering process takes the children of *x* and adds all children of *y*, replacing any child of *x* that has the same name as a child of *y*. After all such nodes have been processed in this manner, the tree is rendered. This achieves the semantics of cascaded styles sheets in the CSS specification [3].

The following UIML elements can have the *mode*="merge-children" attribute: *content*, *style*, *action-list*. A *content* element can only name another *content* element in a *source* attribute and not, say, a *style* element; a *style* element can only name another *style* element; and so on.

If the UIML document contains multiple *style* sections that have the same *name* attribute, then the *mode="merge-children"* attribute must be present only in the first such element that appears in the document. An analogous statement applies to the *content* and *action-list* elements.

Example

Sometimes the value of a user interface component should come from some resource external to the UIML document. The following UIML fragment illustrates how the value of a constant can be set to a Universal Resource Identifier (URI). The body of the *constant* element is set to whatever *http://a.com/file.txt* contains.

```
<constant name="InitialTextValue"
          source="http://a.com/file.txt"
          mode="merge-children"/>
```

Section 5.5 further exemplifies the use of *mode="merge-children"*.

See Section 8.1 for additional information.

5.2.2.2 Mode="replace"

Any UIML element that allows a *name* attribute and a body can contain a *source* and *mode="replace"* attribute: *action-list*, *constant* (if body is not empty), *content*, *function* (if body not empty), *interface*, *object* (if body not empty), *param* (if body not empty), *part* (if body is not empty), *property-method* (if body is not empty), *structure*, *style*, *toolkit*, *uiml*, and *widget* (if body is not empty).

The default value of the *mode* attribute is *replace*. Thus in the examples in this document that have the *source* attribute but no *mode* attribute, the meaning is *replace*.

If an element *e* contains a *mode="replace"* attribute, then

- *e* must not contain a body;
- *e* must either contain no other attributes, or contain just the *name*, *source*, and *mode="replace"* attributes;
- the value of the *source* attribute must be a URI; and
- the URI must contain a single element, and that element must be of type *e*.

The semantics are as follows. Consider an element `<e name="n" source="uri"/>`. Element *e* is replaced by the element in *uri*, but the *name* attribute of the replacing element is changed to be *n*.

Example 1

UIML 2.0 Language Reference

In the following UIML fragment the *toolkit* element is replaced by the content of <http://uiml.org/toolkits/Java20AWT.ui>. That URI must contain as its root document exactly one element of type *toolkit*.

```
<toolkit name="java-awt"  
        source="http://uiml.org/toolkits/Java20AWT.ui"/>
```

The URI might contain this:

```
<toolkit name="java-awt">  
  <widget name="Button" maps-to="java.awt.Button">...</widget>  
  <event name="mouseOver" maps-to="java.awt.event.MouseOver"/>  
  ...  
</toolkit>
```

Example 2

One could build a library of reusable interface components, and then include them as needed in a new UIML document. In the following UIML fragment, a dialog box in file `DialogBox.ui` is inserted into the UIML document in place of the following *part* element. Note that the dialog box can then be customized elsewhere in the UIML document by setting various properties (including the content) of the dialog box.

```
<part name="FileNotFoundBox" class="DialogBox"  
      source="DialogBox.ui"/>
```

See Section 8.1 for additional information.

5.3 The *interface* Element

DTD

```
<!ENTITY % InterfaceName      "NMTOKEN">  
<!ENTITY % URIorNameReference "CDATA">  
<!ENTITY % SourceModes       "(replace|merge-children)">  
  
<!ELEMENT interface (structure|(style|content|action-list)*)+>  
<!ATTLIST interface  
  name      %InterfaceName;      #IMPLIED  
  source    %URIorNameReference; #IMPLIED  
  mode      %SourceModes;        "replace">
```

Description

All UIML elements that describe the interface are contained in the *interface* element. (The *interface* element describes a user interface, *not* the interaction of the user interface and the application logic. The *logic* element is used to define the interaction – see Section 6.1.) A UIML

interface may be as simple as a single string, or as complex as several hundred interface elements that employ various interface technologies (e.g., voice, graphics, and 3D).

An *interface* is composed of four elements: *structure* (see Section 5.4), *content* (see Section 5.5), *style* (see Section 5.6), and *action-list* (see Section 5.7).

Attributes

<i>name</i> (see Section 5.2.1)	The <i>name</i> attribute for the <i>interface</i> element is optional and currently does not have any semantics.
<i>source</i>	Optional. If present, the body of the <i>interface</i> element must be empty. <i>Source's</i> value is a URI of a UIML fragment that is inserted as the child of the <i>interface</i> element.
<i>mode</i> (see Section 5.2.2)	Either "merge-children" or "replace".

5.4 The *structure* Element

DTD

```

<!ENTITY % URIorNameReference      "CDATA">
<!ENTITY % SourceModes             "(replace|merge-children)">

<!ELEMENT structure (part+)>
<!ATTLIST structure
      name      %StructureName;      "default"
      source   %URIorNameReference;  #IMPLIED
      mode     %SourceModes;         "replace">
    
```

Description

An application program will have user interfaces with one or more *organizations* associated with them. By “organization,” we mean the set of UI widgets that are present in the interface, and the relationship of those widgets to each other when the interface is rendered. The relationship might be spatial (e.g., in a graphical user interface) or temporal (e.g., in a voice interface).

For example, there may be one interface organization for a desktop PC, and another organization for a voice interface. The two interfaces may be radically different in terms of which UI widgets are present. For example the voice interface may have fewer widgets, allowing a user to select only a subset of the operations available in the PC interface. In addition, the two interfaces may be organized differently. The voice interface might be a hierarchy of menus, implementing the paradigm of a voice activated response system. Meanwhile the PC interface might be in the form of a wizard and consist of a sequence of dialog boxes. Thus a UIML document needs to enumerate which interface parts are present in each version of the interface, and how those parts are organized (e.g., hierarchically). This is the purpose of the *structure* element. Just as a bridge

UIML 2.0 Language Reference

over a river is a structure that consists of many parts (e.g., steel beam, bolts), a user interface consists of a structure (its organization) and many parts (e.g., widgets).

All interface descriptions must include at least one structure description.

There may be more than one *structure* element, each representing a different organization of the interface. (Thus in the PC and voice interface example above, there are two *structure* elements.) Each *structure* element is given a unique name.

When the interface is rendered, a structure name can be supplied by a mechanism outside the scope of this specification. The structure element whose name matches the supplied name is then used, and all other structure elements are ignored. If the supplied name does not match the name attribute of any structure, then the interface cannot be rendered.

Attributes

<i>name</i> (see Section 5.2.1)	Optional. A unique identifier for a structure. We recommend that the structure name should be descriptive of the purpose of the interface (e.g., "voiceUI" or "complexUI" vs. "simpleUI").
<i>source</i>	Optional. If present, the body of the <i>structure</i> element must be empty. <i>Source</i> 's value is a URI of a UIML fragment that is inserted as the child of the <i>structure</i> element.
<i>mode</i> (see Section 5.2.2)	Either "merge-children" or "replace".

Example

```
<structure name="default">
  <part class="c1" name="n1"/>
  <part class="c2" name="n2"/>
</structure>

<structure name="ComplexUI">
  <part class="c2" name="n3">
    <part class="c1" name="n2"/>
  </part>
</structure>

<structure name="SimpleUI">
  <part class="c1" name="n1"/>
</structure>
```

5.4.1 The *part* Element

DTD

UIML 2.0 Language Reference

```
<!ENTITY % PartId "ID">
<!ENTITY % ClassName "NMTOKEN">
<!ENTITY % URIorNameReference "CDATA">
<!ENTITY % SourceModes "(replace|merge-children)">

<!ELEMENT part (part|style|content|action-list)*>
<!ATTLIST part
  name %PartId; #REQUIRED
  class %ClassName; #IMPLIED
  source %URIorNameReference; #IMPLIED
  mode %SourceModes; "replace">
```

Description

Each *part* element corresponds to one UI widget.

Parts may be nested to represent a hierarchical relationship of parts. For example, the Java Swing toolkit has a notion of containers and components. Containers contain other containers or components, forming a hierarchy. Or, in SpeechML, the oral equivalent of menus can be nested, again forming a hierarchy.

Each part must be associated with a single class. However, if multiple *structure* elements exist, then a part can be associated with a different class in each structure (see example in Section 5.4). When the interface is rendered, only one structure is used (as discussed in “Description” under Section 5.4); thus, a part is always associated with a unique class.

Attribute

<i>name</i> (see Section 5.2.1)	A unique identifier for a part.
<i>class</i> (see Section 5.2.1)	Specifies which class this part belongs to.
<i>source</i>	Optional. If present, the body of the <i>part</i> element must be empty. <i>Source</i> 's value is a URI of a UIML fragment that is inserted as the child of the <i>part</i> element.
<i>mode</i> (see Section 5.2.2)	Either "merge-children" or "replace".

5.5 The *content* Element

DTD

UIML 2.0 Language Reference

```
<!ENTITY % StyleName          "NMTOKEN">
<!ENTITY % URIorNameReference "CDATA">
<!ENTITY % SourceModes        "(replace|merge-children)">

<!ELEMENT content (constant+)>
<!ATTLIST content
  name      %StyleName;          "default"
  source    %URIorNameReference; #IMPLIED
  mode      %SourceModes;       "replace">
```

Description

A user interface can present various contents: words, characters, sounds, and images. For example, a button in a graphical user interface has a text label, such as “press me,” or may result in a sound being played when pressed to signal an error. A voice interface may speak the words “Say five,” whereas a phone interface may speak the words “Press or say five.”

It may be desirable to have alternative text strings, sounds, and images, for example to internationalize interfaces or to provide user interfaces tailored to experts and beginners. For this reason, the *content* element exists. Inside a *content* element are grouped a sequence of *property* elements that contain the actual text strings, sounds, and images associated with user interface parts from the *part* element.

Each *content* element in a UIML document has a name. When the interface is rendered, a mechanism outside the scope of this specification supplies a *content* name. The *content* element whose name matches the supplied name is then used, and all other *content* elements are ignored. If the supplied name does not match the name attribute of any *content* element, then the interface cannot be rendered.

Attributes

<i>name</i> (see Section 5.2.1)	A unique identifier for the <i>content</i> element. The name for the <i>content</i> element should be descriptive of the type of vocabulary used (e.g., expertVocabulary vs. noviceVocabulary or English vs. Greek).
<i>source</i> (see Section 5.2.2)	A name of another <i>content</i> element containing the <i>source</i> and <i>mode="merge-children"</i> attributes or a URI whose target is another <i>content</i> element. Allows this <i>content</i> element to include the children (i.e., <i>property</i> elements) of another <i>content</i> element, which is named by the <i>source</i> attribute.
<i>mode</i> (see Section 5.2.2)	Either <i>replace</i> or <i>merge-children</i> .

Example

In the following example, the user interface contains two parts. The class name “button” suggests that each part be rendered as a button in a graphical user interface. (The *style* element [Section 5.6] actually determines how the class called “button” is rendered – it may be rendered as radio buttons or a voice response.) The button labels are text in one of three languages: English, German, or a slang English. Thus three *content* sections are defined, one for each language. Within each *content* element one or more *property* elements are used to provide the actual literal string that appears in the user interface (e.g., “Yes” for English but “OK” for slang English). Finally, note the use of *source* with the EnglishSlang *content* element: All *property* elements from the English *content* element are included but the “affirmativeChoice” is overridden.

```
<structure name="GUI">
  <part class="button" name="affirmativeChoice"/>
  <part class="button" name="negativeChoice"/>
</structure>

<content name="English">
  <property part="affirmativeChoice" pname="label">Yes</property>
  <property part="negativeChoice"    pname="label">No</property>
</content>

<content name="German">
  <property part="affirmativeChoice" pname="label">Ja</property>
  <property part="negativeChoice"    pname="label">Nein</property>
</content>

<content name="EnglishSlang" source="English" mode="merge-children">
  <property part="affirmativeChoice" pname="label">OK</property>
</content>
```

In the following example, an image and an alternative text string are provided as the content associated with a user interface part (analogous to an HTML tag).

```
<structure name="GUI">
  <part class="pretty-picture" pname="company-logo"/>
  ...
</structure>

<content name="default">
  <property part="company-logo"
    pname="image">http://www.uiml.org/fig.jpg</property>
  <property part="company-logo"
    pname="alt-text">Picture of company logo</property>
</content>
```

5.5.1 The *property* Element

DTD

UIML 2.0 Language Reference

```
<!ENTITY % PropertyName          "NMTOKEN">
<!ENTITY % PartReference         "IDREF">
<!ENTITY % ClassReference        "NMTOKEN">

<!ELEMENT property (#PCDATA|reference|property|constant)*>
<!ATTLIST property
  pname      %PropertyName;      #REQUIRED
  part       %PartReference;      #IMPLIED
  class      %ClassReference;     #IMPLIED
  operation  (set|get)            "set">
```

Description

UIML is designed to be mapped to any device and user interface toolkit, even those not yet invented. Thus property names are not a part of the UIML specification. Instead, UIML is extensible in the sense that one can define whatever property names are needed specific for a particular device and then use those names in any UIML document. Property names are specified by the *toolkit* element (see Section 7.2), which normally contains a *source* attribute whose value is the URI of a document defining the property names used for a particular toolkit. Thus to use UIML one needs both a copy of this specification and a document defining the property names used in a particular *toolkit* element.

Given that property names are not defined in this specification, the semantics of a *property* element are to set a value for a single property of an interface part or event, or get (return to the parent element) a single property's value. The value for each *property* element is one of the following:

1. *A single UIML element that returns a value.* In this case the *property* element has one child, which is the element contained in the *property* element. The semantics of this case is that the UIML element contained in the property element is evaluated, and the value returned is the value to which the property is set. In the following example, the *label* property of part *p* is set to whatever the *label* property is for part *q*:

```
<property part="p" pname="label">
  <property part="q" pname="label" operation="get"/>
</property>
```

2. *Any character sequence not containing the symbol "<".* In this case the *property* element has no children, and its body is set to the character sequence. In the following example, the *label* property of part *p* is set to the character string "property part="q"":

```
<property part="p" pname="label">Welcome, &quot;pilgrim&quot;</property>
```

3. *<![CDATA[...]]>, where "... is any character sequence not containing "]">.* In this case the *property* element has no children, and its body is set to the character sequence "...". CDATA is part of XML, and specifies that the enclosed character string is to be inserted as the body of the parent element and not parsed for XML tags. In the following

UIML 2.0 Language Reference

example, the *label* property of part p is set to the character string '<property part="q" pname="label" operation="get"/>':

```
<property part="p" pname="label"><![CDATA[<property part="q"
pname="label" operation="get"/>]]></property>
```

(The above UIML fragment is actually a single line, but may be reproduced in this document as two lines.)

In the cases 2 and 3 above, the character sequence can be any valid XML code. This allows property values with international characters. As with any XML document, new lines and white space within a *property* element are preserved by the parser. Thus the following UIML fragment will set the label to a string that starts with a return character and ends with a return followed by blank spaces:

```
<property part="p" pname="label">
  Welcome, &quot; pilgrim&quot;
</property>
```

Attributes

<i>part</i>	Value must be the <i>name</i> attribute of some <i>part</i> element in the <i>structure</i> section. Associates this property with that part name. Each <i>property</i> element must have either a <i>part</i> or a <i>class</i> , but not both.
<i>class</i> (see Section 5.2.1)	Value must be the <i>class</i> attribute of some <i>part</i> element in the <i>structure</i> section. The class attribute associates this property with all parts from the <i>structure</i> element with that class name. Each <i>property</i> element must have either a <i>part</i> or a <i>class</i> , but not both.
<i>pname</i> (see Section 5.2.1)	Specifies the name of a property.
<i>operation</i>	Value must be "set" (denoting that <i>property</i> element is setting the property's value) or "get" (denoting that <i>property</i> element is returning the property's value). If omitted, the default is "set".

5.5.2 The *reference* Element

DTD

```
<!ENTITY % ConstantReference      "NMTOKEN" >
<!ELEMENT reference EMPTY>
<!ATTLIST reference
      name %ConstantReference; #REQUIRED>
```

Description

Informally, a *reference* may be thought of as a property-get operation, where the "property" to be returned is the body of a *constant* element defined in a *content* element whose *name* attribute matches the *name* attribute of the *reference* element.

There are several uses for references:

- The same text string might be used in two or more places in a UIML document. In this case a *constant* element can be defined containing the string, and in each place where the string is required (e.g., as values of properties) a *reference* element can be used so that the text string only appears once in the UIML document.
- Often an interface part is initialized to contain several text strings, and when an event later occurs for the part, an *equal* element tests to see which text string the user selected in triggering the event. (For example, lists and choices in Java AWT contain multiple text items.) In this case, a *constant* element can be defined in the *content* section, then the part's values can be initialized in the *style* section using a set *property* element containing a *reference* element as its value. In the *action-list* element the *rule* element handling events for the part can test whether the item selected corresponded to the *constant* element by using a *reference* element. An example of this appears in Section 3.1.

The semantics of a *reference* element is to replace the element with the *constant* element whose *name* attribute matches the name attribute of the *reference* element. If no such element exists, then the UIML document contains an error.

Attributes

<i>name</i> (see Section 5.2.1)	Required. The name must match that of a <i>name</i> attribute in some <i>constant</i> element.
------------------------------------	--

5.6 The *style* Element

DTD

```
<!ENTITY % StyleName          "NMTOKEN">
<!ENTITY % URIorNameReference "CDATA">
<!ENTITY % SourceModes       "(replace|merge-children)">

<!ELEMENT style (property+)>
<!ATTLIST style
  name      %StyleName;          "default"
  source    %URIorNameReference; #IMPLIED
  mode      %SourceModes;       "replace">
```

UIML 2.0 Language Reference

Description

The *style* element contains a list of properties and values that are used to render the interface. Like the CSS and XSL specifications, UIML properties specify attributes of how the interface will be rendered on various devices, such as fonts, colors, layout, and so on.

However, the *style* element in UIML documents normally contain *property* elements that set two properties that are at the heart of achieving device-independence: *rendering* and *event*. (*Property* elements are described in Section 5.6.1.)

- The *rendering* property specifies the name of a widget in the target user interface toolkit to which the interface part is mapped. This allows an interface part to be rendered as a `java.awt.Button` in Java by a device supporting Java and as the `<INPUT>` tag in HTML by a Web browser.
- The *event* property specifies the name of an event for a widget in the target toolkit. This allows a UIML author to use generic event names in the *action-list* element, then map them to toolkit-specific events in the *style* element. For example, the *event* property might map an event named “Selection” in the *action-list* element to the *OnClick* event in WML’s `OPTION` tag or a *MouseDown* event in a GUI toolkit.

If an interface part is required for a particular toolkit, but the UIML document contains no *rendering* property for that part, then the interface cannot be rendered. This is also true for *event* properties.

There must be at least one *style* element, and there may be more than one. There is at least one *style* element for each toolkit to which the UIML document will be mapped. For a given toolkit, there may be multiple *style* elements serving a variety of purposes: to generate different interface presentations for accessibility, to support a family of similar but not identical devices (e.g., phones that differ in the number of characters that their displays support), to support different target audiences (e.g., children versus adults), and so on.

Attributes

<i>name</i> (see Section 5.2.1)	The name for <i>style</i> element should be descriptive of its intended use (e.g., “PC800x600”, “Speech”, “NTSCforChild”).
<i>source</i> (see Section 5.2.2)	A name of another <i>style</i> element containing the <i>source</i> and <i>mode="merge-children"</i> attributes or a URL whose target is another <i>style</i> element. Allows this <i>style</i> element to include the children (i.e., <i>property</i> elements) of another <i>style</i> element, which is named by the <i>source</i> attribute. This achieves cascading of styles.
<i>mode</i> (see Section 5.2.2)	Either <i>replace</i> or <i>merge-children</i> .

Example

```
<style name="Graphical" source="Simple" mode="merge-children">
  <property class="c1"      pname="font"      >Comic</property>
  <property part="n1"      pname="size"      >100</property>
  <property part="n1"      pname="rendering" >java.awt.Button</property>
  <property part="event1"  pname="event"   >mouseOver</property>
</style>

<style name="Simple">
  <property class="c1"  pname="color" >white</property>
  <property part="n1"  pname="color" >white</property>
</style>
```

5.6.1 The *property* Element

DTD

```
<!ENTITY % PropertyName      "NMTOKEN" >
<!ENTITY % PartReference     "IDREF" >
<!ENTITY % ClassReference    "NMTOKEN" >

<!ELEMENT property ( #PCDATA | reference | property | constant ) * >
<!ATTLIST property
  pname      %PropertyName;      #REQUIRED
  part       %PartReference;     #IMPLIED
  class      %ClassReference;    #IMPLIED
  operation  ( set | get )       "set" >
```

Description

The *property* element specifies a value for a single style property of an interface part. The name of the style property (in the *set* or *get* attribute) resolves to a toolkit-specific name through the *toolkit-peers* element (see Section 7).

See Section 5.5.1 for further information.

Attributes

See the "Attributes" section in 5.5.1.

5.7 The *action-list* Element

DTD

UIML 2.0 Language Reference

```
<!ENTITY % StyleName          "NMTOKEN">
<!ENTITY % URIorNameReference "CDATA">
<!ENTITY % SourceModes        "(replace|merge-children)">

<!ELEMENT action-list (rule+)>
<!ATTLIST action-list
  name      %StyleName;          "default"
  source    %URIorNameReference; #IMPLIED
  mode      %SourceModes;       "replace">
```

Description

The *action-list* element lists what actions can happen while the user interface is executing. The *action-list* element specifies what actions to take when an event is *fired* while the user interface is executing. The *action-list* element is responsible for the following behavior:

1. Assign a value to a part's property. The value can be any of the following:
 - a. a constant value
 - b. the value of a property of some part
 - c. the return value of a call to a function in a script or in external application logic
2. Call a function in a script or external application logic. Each argument to the function can be any of the following:
 - a. a constant
 - b. the value of a property of some part
 - c. the return value of a function call
3. Fire one or more other events
4. Load another UIML document and update the user interface based on the content of that document.

The *actions-list* element contains one or more *rules*. Each rule contains a *condition* and a corresponding *action* element. The condition is either the name of an event or a Boolean expression. The semantics is that whenever the event fires or the Boolean expression in the condition evaluates to *true*, then the *action* element is executed. (The only Boolean expression defined in the DTD for UIML2.0 is equality of an event and a constant.) The use of the terms *rule*, *condition*, and *action* are motivated by condition-action pairs used in rule-based languages. The use of rules is reflective of the viewpoint that markup languages are declarative languages, and in that spirit it is appropriate to declare that whenever certain conditions are true during the lifetime of the interface, certain actions will be executed.

The *action* element contains one or more subelements that are executed in the order given in the UIML document. Each subelement can be either a *property* element to set a property of an element or a *call* element, which calls a function in a script or in external application logic.

Attributes

<i>name</i> (see Section 5.2.1)	The name for the <i>action-list</i> element should be descriptive of the type of platform the actions are used with (e.g., “Java” or “voice”). A mechanism outside of the scope of this specification can provide a name, and <i>action-list</i> elements matching that name are used, and all other <i>action-list</i> elements are ignored.
<i>source</i> (see Section 5.2.2)	A name of another <i>action-list</i> element containing the <i>source</i> and <i>mode="merge-children"</i> attributes or a URL whose target is another <i>action-list</i> element. Allows this <i>action-list</i> element to include the children (i.e., <i>event</i> elements) of another <i>action-list</i> element, which is named by the <i>source</i> attribute.
<i>mode</i> (see Section 5.2.2)	Either <i>replace</i> or <i>merge-children</i> .

Example1 (Case 1a)

Note: These examples use the elements rule, condition, equal, event, action, and call. These are discussed in subsequent sections.

The following UIML fragment will change the color of a user interface part to blue each time an event called “buttonEvent” is clicked. The *rule* element contains a *condition* element. The *condition* element says that whenever the event called *ButtonSelected* fires for Button *b1*, execute the list of elements in the body of the *action* element. The *action* element contains one subelement, which sets the color property of button *b1* to the color *blue*.

```

<structure name="default">
  ...
  <part class="Button" name="b1"/>
  ...
</structure>

<action-list>
  ...
  <rule>
    <condition>
      <event class="ButtonSelected" part="b1">
    </condition>
    <action>
      <property part="b1" pname="color"/>blue</property>
    </action>
  </rule>
  ...
</action-list>

```

Example 2 (Case 1b)

UIML 2.0 Language Reference

This is a variation of Example 1. Suppose that whenever button b1 is clicked, we want the font size of b1's label to be changed to whatever the current font size is of the label on button b2. The UIML required expresses the idea that "b1's font size = b2's font size".

```
<action-list>
  ...
  <rule>
    <condition>
      <event class="ButtonSelected" part="b1">
    </condition>
    <action>
      <property part="b1" pname="font-size">
        <property part="b2" pname="font-size" operation="get"/>
      </property>
    </action>
  </rule>
  ...
</action-list>
```

The element `<property part="b2" name="font-size" operation="get"/>` in the above example returns the value of part b2's font-size attribute. By enclosing this in the corresponding property element for part b1, part b1's font size is set accordingly.

The default value of attribute *operation* is "set", which is why our previous examples of the *property* tag had no *operation* attribute. So an equivalent way to write the above UIML fragment is this:

```
<property part="b1" pname="font-size" operation="set">
  <property part="b2" pname="font-size" operation="get"/>
</property>
```

Example 3 (Case 1b)

Besides properties, an *event* element can also set the content of an interface part. The following example sets the text label on button b1 to the text label on button b2.

```
<action-list name="ex1">
  <rule>
    <condition><event class="buttonEvent" part="b1"/></condition>
    <action>
      <property part="b1" pname="content">
        <property part="b2" pname="content" operation="get"/>
      </property>
    </action>
  </rule>
</action-list>
```

Example 4 (Case 1b)

UIML 2.0 Language Reference

So far we considered a simple interface part, a button, which usually only has one property in most toolkits (e.g., representing the button being pressed). Events often have multiple properties. For example, the event generated by pushing a mouse button down could have properties for the x and y coordinates of the pointer when the mouse was pressed. The following UIML fragment displays the mouse's x-coordinate in a text area when the button is pressed. This UIML fragment illustrates the use of the *name* attribute (set to "m1") for an *event* element (a mouse event), which is later referenced in a *property* element within the corresponding *action* element. (The scope of an event names is limited to the *rule* element in which the *event* element occurs that defines the name.)

```
<action-list>
  <rule>
    <condition>
      <event class="MouseDown" part="Frame1" name="m1">
    </condition>
    <action>
      <property part="textarea" pname="value">
        <property event="m1" pname="x" operation="get"/>
      </property>
    </action>
  </rule>
</action-list>
```

In the example above, whenever the frame component receives a mouse down event, the x coordinate of the mouse event is displayed in the interface part called *textarea*.

Example 5 (Case 2a)

Besides setting properties and content, an *action* element can also contain a *call* element, which invokes a function in the application logic. The following example calls a function identified as *App1.back2.m3*, which is the method named *m3* in the object named *back2* in the external program named *App1*. The function has one parameter, which is the number five. The meaning of such a dotted string is defined in the *object* element, which is contained in the *logic* element (see Section 6.1). Each parameter in a function call has a name (e.g., *p* in the example below), which is defined in the *object* element.

```
<action-list name="ex1">
  <rule>
    <condition>
      <event class="ButtonSelected" part="b1">
    </condition>
    <action>
      <call function="App1.back2.m3">
        <param name="p">5</param>
      </call>
    </action>
  </rule>
</action-list>
```

Example 6 (Case 2b)

The value of a parameter passed to a function does not have to be fixed (e.g., “5” in the previous example). Instead, it can be the value of a property of some interface part. For example, suppose whenever *App1.back2.m3* was called we wanted its parameter to be the current value of the width of user interface part *n1*. This can be accomplished by modifying the previous example as follows:

```
<action>
  <call function="App1.back2.m3">
    <param name="p4">
      <property part="n1" pname="width" operation="get"/>
    </param>
  </call>
</action>
```

Alternately, the parameter to *App1.back2.m3* might be the content of a user interface part. Perhaps the interface contains a part named “textfield” that is a field in which the application user types his or her name. That name can be used as the parameter to *App1.back2.me* as follows:

```
<action>
  <call function="App1.back2.m3">
    <param name="p4">
      <property part="textfield" pname="content" operation="get"/>
    </param>
  </call>
</action>
```

Example 7 (Case 1c and 2c)

Besides setting properties and calling a function, an event can also set an interface part property to the return value of a function. The following *action* element sets the color property of a button named *b1* to the return value of a function call *App1.back1.m3(S1.func())* when the button is pressed:

```
<rule>
  <condition>
    <event class="ButtonSelected" part="b1">
  </condition>
  <action>
    <property part="b1" pname="color">
      <call function="App1.back1.m3">
        <param name="p3">
          <call function="S1.func"/>
        </param>
      </call>
    </property>
  </action>
</rule>
```

Example 8 (Case 3)

Whenever an *event* element is the child of an *action* element, then executing the *action* element fires the child *event*. The following example illustrates how one event can fire another event. Whenever button *b1* is clicked, the action taken is the same as if button *b2* was clicked.

```
<structure name="default">
  ...
  <part class="Button" name="b1"/>
  <part class="Button" name="b2"/>
  ...
</structure>

<action-list name="ex1">

  <rule>
    <condition>
      <event class="ButtonSelected" part="b1"/>
    </condition>
    <action>
      <!-- executed when b1 is clicked -->
      <event class="ButtonSelected" part="b2"/>
    </action>
  </rule>

  <rule>
    <condition>
      <event class="ButtonSelected" part="b2"/>
    </condition>
    <action>
      <!-- executed whenever b1 or b2 is clicked -->
      ...
    </action>
  </rule>

</action-list>
```

It may seem at first that this code creates an infinite loop, with button *b1* pressing button *b2*, which presses *b1* again, and so forth. This is not the case, because pressing button *b1* doesn't press button *b2*. Rather, pressing button *b1* fires the *event* associated with button *b2*.

Example 9

This example illustrates a *condition* element that is a Boolean expression testing for equality of two values. In this example the user interface contains a list box named *Colors* whose content is the colors *red* and blue. The *action* element is executed if the user clicks on the color *red*.

```
<structure>
  ...
  <part class="List"      name="Colors"/>
  ...
```

```
</structure>

<action-list>
  <rule>
    <condition>
      <equal>
        <event part="Colors" class="LSelected"
              pname="item-selected"/>
        <constant>red</constant>
      </equal>
    </condition>
    <action>
      <!-- Execute some action if user clicks on red. -->
    </action>
  </rule>
</action-list>
```

The *item-selected* property of the *Lselected* event returns the text of the value the user clicked on in the list (e.g., "red"). The *constant* element returns the text "red". The *equals* element compares the two strings, and finds that they are identical, which causes the *condition* element to be satisfied. This then causes the *action* element's children to be executed.

Example 10

This example simply shows that the body of an *action* element can be empty, in which case an event is ignored.

```
<action-list>
  <rule>
    <condition>
      <event class="ButtonSelect">
    </condition>
    <action>
      <!-- ignore all ButtonSelect events -->
    </action>
  </rule>
</action-list>
```

5.7.1 The *rule* Element

DTD

```
<!ELEMENT rule (condition,action)>
```

Description

The *rule* element defines a binding between a *condition* element and an *action* element. Whenever the *condition* element within the rule is satisfied, then the property changes, function calls, and event firings listed in the children of the *action* element are performed. See Section 5.7 for an example and further explanation.

Attributes

None.

5.7.2 The *condition* Element

DTD

```
<!ELEMENT condition (equal|event)>
```

Description

The *condition* element contains as a child either an *event* element or a Boolean expression. The *action* element associated with this *condition* by the parent *rule* element is executed whenever either the event named in the *event* element fires or the Boolean expression evaluates to *true*. See Section 5.7 for an example and further explanation.

The following rules address the situation when more than one condition might match an event. The rules eliminate the possibility that multiple conditions may be simultaneously satisfied (assuming that in the platform events are fired sequentially):

- A UIML document contains an error if it contains two *condition* elements whose *event* elements both have a *class* and a *part* attribute, and the *class* attribute of both *event* elements is identical, and the *part* attribute of both event elements is identical.

For example, the UIML document containing the following fragment contains this error:

```
<rule>  
  <condition><event class="ButtonSelect" part="p"/></condition>  
  <action>...</action>  
</rule>  
...  
<rule>  
  <condition><event class="ButtonSelect" part="p"/></condition>  
  <action>...</action>  
</rule>
```

- A UIML document contains an error if it contains two *condition* elements whose *event* elements both have a *class* attribute and do not have a *part* attribute, and the *class* attribute of both *event* elements is identical.

For example, the UIML document containing the following fragment contains an error:

```
<rule>  
  <condition><event class="ButtonSelect"/></condition>  
  <action>...</action>  
</rule>  
...  
<rule>
```

UIML 2.0 Language Reference

```
<condition><event class="ButtonSelect"/></condition>
<action>...</action>
</rule>
```

- Suppose a UIML document contains two *condition* elements whose *event* elements both have a *class* attribute and only one *event* element has a *part* attribute (say with value *p*), and the *class* attribute of both *event* elements is identical (say the value is *c*). Whenever an event of class *c* occurs for part *p*, only the *condition* element containing the *part* attribute will be evaluated.

Consider this example:

```
<rule>
  <condition><event class="ButtonSelect"/></condition>
  <action><property part="x" pname="prop">0</property></action>
</rule>

<rule>
  <condition><event class="ButtonSelect" part="b1"/></condition>
  <action><property part="x" pname="prop">1</property></action>
</rule>
```

Whenever button *b1* is clicked, property *prop* is set to 1 (and not to 0).

Attributes

None.

5.7.3 The *equal* Element

DTD

```
<!ELEMENT equal (event,(constant|property|reference))>
```

Description

The *equal* element is a Boolean expression with value *true* or *false*. Every *equal* element must have exactly two children. One must be an *event* element with a *property* attribute. The other must be a *constant*, *property* (with attribute *operation*="get"), or *reference* element. The semantics of *equal* are as follows. Whenever (a) the event named in the *event* element fires and (b) the value of the element property named in the *event* tag equals the result of evaluating the *constant*, *property*, or *reference* element, then the *equal* element has value *true*. Otherwise the *equal* element has value *false*.

Attributes

None.

5.7.4 The *reference* Element

See Section 5.5.2.

5.7.5 The *event* Element

DTD

```

<!ENTITY % EventName           "NMTOKEN" >
<!ENTITY % PartReference       "IDREF" >
<!ENTITY % PropertyName        "NMTOKEN" >
<!ENTITY % ClassReference      "NMTOKEN" >
<!ENTITY % NameReference       "NMTOKEN" >

<!ELEMENT event EMPTY>
<!ATTLIST event
    name      %EventName;      #IMPLIED
    part      %PartReference;   #IMPLIED
    pname     %PropertyName;    #IMPLIED
    class     %ClassReference;  #REQUIRED
    maps-to   %NameReference;   #IMPLIED>

```

Description

The *event* element is used in three contexts:

- As the child of a *condition* element. In this case the *class* attribute is mandatory, the *part* attribute is optional, and all other attributes are illegal. The semantics are that the parent *condition* is satisfied whenever an event occurs matching the class and optional part specified.
- As the child of an *equal* element. In this case the *class*, *property*, and *name* elements are mandatory, and all other attributes are illegal. The semantics is that the *event* element returns a value to the parent *equal* element. That value is the value of the property specified for the event which occurred for the named part.
- As the child of an *action* element. In this case the *class* and *part* attributes are mandatory. The semantics are that an event of the named class is fired for the named part.

Event Names

One of the powerful aspects of UIML is the naming of events. In a conventional language (e.g., Javascript) events have names reflective of the interface components to which they correspond (e.g., *OnClick* for a button). However one UIML document may be mapped to several different platforms. An interface part *p* might be a button on platform 1 or a menuitem on platform 2. Therefore the *event* element for part *p* specifies a *class* attribute that can be set to whatever the UIML author wishes (e.g., *ButtonOrMenuSelection*). The *style* element in the UIML document then maps that name to a platform-specific name. In this case there would be style elements with two different names:

UIML 2.0 Language Reference

```
<style name="Platform1">...</style>
<style name="Platform2">...</style>
```

The *style* element then maps the generic name (e.g., *ButtonOrMenuSelected*) to a button selection in platform 1 and a menu item selection in platform 2 using the *rendering* property:

```
<style name="Platform1">
  <property class="ButtonOrMenuSelected" pname="rendering">ButtonSelected</property>
</style>
```

Recall that the values of *rendering* properties are defined in the *toolkit* element (which as mentioned is not normally written by a UIML author).

Attributes

<i>name</i> (see Section 5.2.1)	Optional. Each <i>event</i> is uniquely referenced, within an <i>interface</i> description, by the <i>name</i> attribute. The event may have properties (e.g., the x and y position of the pointer for a mouse event), and the <i>name</i> attribute of can be given to a <i>property</i> element to use the event properties.
<i>part</i>	Optional. Name of an interface element that this <i>event</i> element corresponds to.
<i>pname</i>	Optional. Name of an event property that is returned by this element.
<i>class</i> (see Section 5.2.1)	Mandatory. The <i>class</i> parameter resolves the <i>event</i> at rendering time to a UI event through the <i>style</i> element.
<i>maps-to</i>	Not used. (Used only when <i>event</i> element is contained in a <i>toolkit</i> element – see Section 7.2)

5.7.6 The *action* Element

DTD

```
<!ELEMENT action (property|event|edit)+>
```

Description

If the *condition* element associated with the *action* element is satisfied, then each child of the *action* element is executed in an in-order traversal of the current XML document tree rooted at the *action* element.

Attributes

None.

5.7.7 The *property* Element

DTD

```

<!ENTITY % PropertyName          "NMTOKEN">
<!ENTITY % PartReference         "IDREF">
<!ENTITY % ClassReference        "NMTOKEN">

<!ELEMENT property (#PCDATA|reference|property|constant)*>
<!ATTLIST property
  pname      %PropertyName;      #REQUIRED
  part       %PartReference;     #IMPLIED
  class      %ClassReference;    #IMPLIED
  operation  (set|get)           "set">

```

Description

When a *property* element is a child of an *action* element, it must have the attribute *operation="set"* (or the *operation* attribute must be absent, in which case the default value is "set").

The semantics of *property* in this case are as follows. The grandparent of the *property* element is a *rule* element. The *rule* element must contain a *condition* element as a child. If that *condition* element is satisfied, then the *property* element is executed, which results in setting the named property of the named part to the *property* element's value.

See Section 5.5.1 for further information.

Attributes

See Section 5.5.1.

5.7.8 The *call* Element

DTD

```

<!ENTITY % FunctionName          "NMTOKEN">

<!ELEMENT call (param)*>
<!ATTLIST call
  function  %FunctionName;      #REQUIRED

```

Description

Invokes a function in a script or the external application logic. See Section 5.7 for an example and further explanation.

Attributes

<i>function</i>	The name of a function to call. The syntax of the <i>function</i> attribute value is “Application.Object.Function”, “Application.Function”, or “Script.Function”. The name must be defined by an <i>application</i> or <i>script</i> element within the <i>logic</i> element.
------------------------	---

5.7.9 The *param* Element

DTD

```
<!ENTITY % Name "NMTOKEN">
<!ENTITY % URIorNameReference "CDATA">
<!ENTITY % SourceModes "(replace|merge-children)">

<!ELEMENT param (property|call)*>
<!ATTLIST param
  name %Name; #REQUIRED
  source %URIorNameReference; #IMPLIED
  mode %SourceModes; "replace">
```

The *param* element specifies the parameters to a function called by the *call* element. The body of the *param* element is the value associated with the parameter whose name is given by the *name* attribute.

See Section 5.7 for an example and further explanation.

Attributes

<i>name</i> (see Section 5.2.1)	The name of the parameter. Note that parameters are not matched by position but by name.
---	--

5.7.10 The *edit* Element

DTD

```
<!ENTITY % NameReference "NMTOKEN">
<!ENTITY % URIorNameReference "CDATA">
<!ENTITY % SourceModes "(replace|merge-children)">

<!ELEMENT edit EMPTY>
<!ATTLIST edit
  element %NameReference; #REQUIRED
  name %NameReference; #REQUIRED
  source %URIorNameReference; #IMPLIED
  mode %SourceModes; "replace">
```

Description

The action can dynamically modify the UIML document using the *edit* element. For example, the following UIML fragment loads a URI containing a new user interface part (e.g., a window) when *DisplayButton* is clicked.

The UIML fragment contains a *structure* element named *PC*. That structure contains a window called *MainWindow*, which in turn contains a button called *DisplayButton*. The *action* element performed when the button is clicked does two things. First it performs the *edit* element, which causes a new window to pop up on the screen, and then it performs a *property* element that causes *MainWindow* to disappear from view.

```
<structure name="PC">
  <part class="Frame" name="MainWindow">
    <part class="Button" name="DisplayButton">
      <action-list>
        <rules>
          <condition>
            <event class="BSelected"/>
          </condition>
          <action>
            <edit element="structure" name="PC"
                  source="file://SecondaryWindow.ui"
                  mode="merge-children"/>
            <property part="MainWindow" pname="visible">false</property>
          </action>
        </rules>
      </action-list>
    </part>
  </part>
  ...
</structure>
```

The *edit* element modifies the current representation of the UIML document as an XML tree. The *element* and *name* attributes identify what node of the tree is to be modified. In this case it is the entire *structure* element named *PC* (which includes the *edit* tag itself as a descendant!). The remaining attributes, *source* and *mode="merge-children"*, defines how the node is to be modified: in this case, a new child will be added to the *structure* element. (The alternative to the *mode="merge-children"* attribute is *mode="replace"*, in which case the attribute's value is a URI and the root tree specified by the URI replaces the node.) The new child added is the tree contained in file *SecondaryWindow.ui*. Suppose that *SecondaryWindow.ui* contains the following:

```
<part class="Frame" name="SecondaryWindow">
  ...
</part>
```

After completion of performing the *edit* element, the UIML document has the following structure:

UIML 2.0 Language Reference

```
<structure name="PC">
  <part class="Frame" name="MainWindow">
    ...
  </part>
  <part class="Frame" name="SecondaryWindow">
    ...
  </part>
</structure>
```

Note that the semantics of *add* are to insert the new child after the previous children of the *structure* element.

Attributes

<i>element</i>	Mandatory. Value is any element type (if attribute <i>mode</i> ="merge-children" appears) or any element type that is defined to have a non-empty body (if attribute <i>mode</i> ="replace" appears).
<i>name</i> (see Section 5.2.1)	Mandatory. Value is the <i>name</i> attribute of an element specified by the <i>element</i> attribute.
<i>source</i> (see Section 5.2.2)	Name of URI.
<i>mode</i> (see Section 5.2.2)	Either "merge-children" or "replace".

See Section 8.1 for additional information.

6 Application Logic

The *logic* element describes how the user interface interacts with the underlying logic that implements the functionality manifested through the interface. The underlying logic might be implemented by middleware in a three tier application, or it might be implemented by scripts in some scripting language, or it might be implemented by a set of objects whose methods are invoked as the end user interacts with the user interface, by some combination of these (e.g., to check for validity of data entered by an end user into a user interface and then object methods are called), or in other ways.

Thus the *logic* element acts as the glue between a user interface described in UIML and other code. The *logic* element also permits scripting in a language of the author's choice to be used with a user interface.

6.1 The *logic* Element

DTD

```
<!ELEMENT logic (object)+>
```

Description

The *logic* element describes the calling conventions for functions in application logic that the user interface invokes. Examples of such functions include objects in languages such as C++ or Java, CORBA objects, programs, legacy systems, server-side scripts, databases, and scripts defined in various scripting languages.

Attributes

None.

Example

The following UIML fragment describes the calling conventions for a variety of functions in external application logic and functions in scripts.

```
<logic>
  <object name="back1" type="application/java"
    maps-to="org.uiml.example.myClass">
    <function name="m1" maps-to="myfunction">
      <param name="p1"/>
      <param name="p2"/>
    </function>
  </object>
</logic>
```

```
<function name="m2" maps-to="m2">
  <returns name="r1"/>
</function>

<function name="master" maps-to="m3">
  <param name="p3"/>
  <returns name="r2"/>
</function>

</object>

<object name="back2" maps-to="org.uiml.example.myClass1">
  <function name="m3" maps-to="m9">
    <param name="p4"/>
  </function>
</object>

<object name="S1" type="application/ecmascript">

  <function name="m1" maps-to="Cube">
    <param name="i"/>
    <returns name="result"/>
  </function>

  <script><![CDATA[
Cube(int i) {
  return i*i*i;
}
]]></script>

</object>

<object name="S2" type="application/x-javascript"
  maps-to="http://somewhere/vb"/>
  <function name="m101" maps-to="f2">
    <param name="p5"/>
  </function>

</logic>
```

6.2 The *object* Element

DTD

UIML 2.0 Language Reference

```

<!ENTITY % Name                "NMTOKEN">
<!ENTITY % MIMETYPE            "CDATA">
<!ENTITY % ObjectReference     "CDATA">
<!ENTITY % URIorNameReference  "CDATA">
<!ENTITY % SourceModes         "(replace|merge-children)">

<!ELEMENT object (function*, script?)>
<!ATTLIST object
  name      %Name;                #REQUIRED
  type      %MIMETYPE;            #REQUIRED
  maps-to   %ObjectReference;     #IMPLIED
  source    %URIorNameReference;  #IMPLIED
  mode      %SourceModes;         "replace">

```

Description

The *object* element binds a name used in *call* elements elsewhere in the interface to an object that is part of the application logic. The object contains functions invoked by the *call* elements.

The *object* element's attributes specify the name that is being bound, a MIME type of the language in which the object is written, and a way to locate the object. The object location is specified in exactly one of the following ways:

- *In the maps-to attribute of the object element:* In this case the value of the attribute is either a URI of the location of the object or a class name.
- *By a child element of object that is called script:* In this case the object must be a script in some scripting language that contains one or more functions. The script itself appears inside the *script* element.

Attributes

<i>name</i> (see Section 5.2.1)	An identifier used in <i>call</i> elements to refer to an object that implements the functions described in child <i>function</i> elements.
<i>type</i>	A MIME type designating the language in which functions in the body of the <i>object</i> element are implemented. The names are defined outside the scope of this specification.
<i>maps-to</i>	Contains a URI or class name. This name is used to locate the object that implements the functions described in child <i>function</i> elements. For example, if <i>maps-to</i> is a URI, it might be the location of a function in a scripting language or a Java RMI registry. If the <i>maps-to</i> attribute is a class name (e.g., the UIML document and the needed objects are packaged into a stand-alone program), then it might use the dotted notation used in Java.
<i>source</i>	Optional, but only allowed if <i>object</i> has a body. If present, the body of the <i>object</i> element must be empty. <i>Source</i> 's value is a URI of a UIML fragment that is inserted as the child of the <i>object</i>

	element.
mode (see Section 5.2.2)	Optional, but only allowed if <i>object</i> has a body. Either "merge-children" or "replace".

6.2.1 The *function* Element

DTD

```

<!ENTITY % FunctionName          "NMTOKEN">
<!ENTITY % URIorNameReference    "CDATA">
<!ENTITY % SourceModes          "(replace|merge-children)">

<!ELEMENT function (param*, returns?)>
<!ATTLIST function
    name      %FunctionName;          #REQUIRED
    maps-to   %FunctionName;          #REQUIRED
    source    %URIorNameReference;    #IMPLIED
    mode      %SourceModes;           "replace">

```

Description

The *function* element describes a function in the external application logic in terms of its optional formal parameters and optional return value.

Attributes

name (see Section 5.2.1)	An identifier used in the <i>interface</i> element to refer to the function.
maps-to	Contains the function name that is used in the application logic.
source	Optional, but only allowed if <i>function</i> has a body. If present, the body of the <i>function</i> element must be empty. <i>Source</i> 's value is a URI of a UIML fragment that is inserted as the child of the <i>function</i> element.
mode (see Section 5.2.2)	Optional, but only allowed if <i>function</i> has a body. Either "merge-children" or "replace".

6.2.2 The *param* Element

DTD

```

<!ENTITY % Name "NMTOKEN">
<!ENTITY % URIorNameReference "CDATA">
<!ENTITY % SourceModes "(replace|merge-children)">

<!ELEMENT param (property|call)*>
<!ATTLIST param
  name %Name; #REQUIRED
  source %URIorNameReference; #IMPLIED
  mode %SourceModes; "replace">

```

Description

Describes a single formal parameter of the function described by the parent *function* element. Note that all parameters are character strings. It is up to some intermediary to convert parameters from character strings to other data types (e.g., integer or Boolean) if required.

The order of *param* elements within the *function* element is significant. This order must correspond to the order in which the parameters were originally declared in the external application.

The matching of parameters with the *call* element (elsewhere in the user interface) to the parameters in the *function* element is done by name and not by position (thus they can be listed in any order in the *call* element).

Attributes

<i>name</i> (see Section 5.2.1)	A named used for the parameter in <i>call</i> elements.
<i>source</i>	Optional, but only allowed if <i>param</i> has a body. If present, the body of the <i>param</i> element must be empty. <i>Source's</i> value is a URI of a UIML fragment that is inserted as the child of the <i>param</i> element.
<i>mode</i> (see Section 5.2.2)	Optional, but only allowed if <i>param</i> has a body. Either "merge-children" or "replace".

6.2.3 The *returns* Element

DTD

UIML 2.0 Language Reference

```
<!ENTITY % Name "NMTOKEN" >

<!ELEMENT returns EMPTY>
<!ATTLIST returns
    name %Name; #IMPLIED>
```

Description

The parent of a *returns* element is a *function* element. The *function* element describes some function, say *f*. The *returns* element if present declares that *f* does indeed return a value, and the *returns* element defines a name for the return value of *f*. All return values are character strings.

Attributes

<i>name</i> (see Section 5.2.1)	Identifier used in the <i>interface</i> element to refer to the return value of the function corresponding to the <i>function</i> element that is the parent of the <i>returns</i> element.
---------------------------------	---

6.2.4 The *script* Element

DTD

```
<!ELEMENT script (#PCDATA)>
```

Description

The *script* element contains a program written in the scripting language identified by the parent *object* element's *type* attribute.

Attributes

None.

7 Mapping Property and Event Names to Toolkits

7.1 The *toolkit-peers* Element

DTD

```
<!ELEMENT toolkit-peers (toolkit+)>
```

Description

In UIML, all the device and toolkit information is isolated in the *toolkit-peers* element. This information is used by a UIML rendering engine to resolve all the names from the *style* and *event* elements into actual widgets, functions, and events.

Normally a UIML author does not write *toolkit* elements, but simply includes existing ones like this:

```
<toolkit-peers>
  <toolkit name="java"
    source="http://uiml.org/toolkits/Java20Swing.ui"/>
  <toolkit name="wml" source="http://uiml.org/toolkits/wml.ui"/>
</toolkit-peers>
```

The *toolkit-peers* element assists in the creation of programs that generate renderers, so that renderers do not have to be hand-crafted for each toolkit. It also provides the authoritative definition of the vocabulary used in UIML for each toolkit.

An implementation of a rendering engine may omit reading the *toolkit* element to reduce the execution time of and mitigate the effect of network delays upon rendering time. Instead, the engine might cache copies of the toolkit files for the toolkits that it supports (e.g., *Java20Swing.ui* in the example above). Alternatively, the *toolkit* element's information might be hard-wired into the rendering engine, so that the engine does not even have to spend time reading and processing the information.

Attributes

None.

Example

The following example illustrates what goes into a *toolkit* element. As stated earlier, a UIML author normally does not write *toolkit* elements.

```
<toolkit-peers>
```

UIML 2.0 Language Reference

```
<toolkit name="java">
  <widget name="Button" maps-to="java.awt.Button">
    <property-method name="content" set-method="setText"/>
    <property-method name="color" set-method="setColor"
      get-method="getColor"/>
  </widget>
  <event name="mouseOver" maps-to="java.awt.event.MouseOver"/>
</toolkit>
<toolkit name="wml">
</toolkit>
</toolkit-peers>
```

7.2 The *toolkit* Element

DTD

```
<!ENTITY % ToolkitName "NMTOKEN">
<!ENTITY % URIorNameReference "CDATA">
<!ENTITY % SourceModes "(replace|merge-children)">

<!ELEMENT toolkit (widget|event)*>
<!ATTLIST toolkit
  name %ToolkitName; #REQUIRED
  source %URIorNameReference; #IMPLIED
  mode %SourceModes; "replace">
```

Description

The *toolkit* element provides information about a single user interface toolkit (defined in Section 1.2). It describes the different widgets (which are used to render parts) and events (that are generated during the course of application execution).

It is possible to have multiple UIML *toolkit* elements for the same user interface toolkit. UI designers can create their own UI vocabulary and then map it to the underlying toolkit. See Section 1 for more comments on this perspective.

Attributes

<i>name</i> (see Section 5.2.1)	Each <i>toolkit</i> element is identified by name. The name should be descriptive of the user interface toolkit (e.g., Java, WML, VoxML).
<i>source</i>	Optional. If present, the body of the <i>toolkit</i> element must be empty. <i>Source</i> 's value is a URI of a UIML fragment that is inserted as the child of the <i>toolkit</i> element.
<i>mode</i> (see Section 5.2.2)	Either "merge-children" or "replace".

7.2.1 The *widget* Element

DTD

```

<!ENTITY % NameReference      "NMTOKEN">
<!ENTITY % URIorNameReference "CDATA">
<!ENTITY % SourceModes       "(replace|merge-children)">

<!ELEMENT widget (property-method)*>
<!ATTLIST widget
  name      %NameReference;      #REQUIRED
  maps-to   %NameReference;      #REQUIRED
  source    %URIorNameReference; #IMPLIED
  mode      %SourceModes;        "replace">
    
```

The *widget* element describes a single UI widget.

Attributes

<i>name</i>	A <i>name</i> that is used in a <i>property</i> element in a <i>style</i> element with ‘pname=“rendering”’.
<i>maps-to</i>	The attribute <i>maps-to</i> names the <i>toolkit</i> class that implements <i>widget</i> (if <i>toolkit</i> is a library of interface components), or possibly the tag name that <i>widget</i> maps to (if <i>toolkit</i> is another markup language).
<i>source</i>	Optional, but only allowed if <i>widget</i> has a body. If present, the body of the <i>widget</i> element must be empty. <i>Source</i> 's value is a URI of a UIML fragment that is inserted as the child of the <i>widget</i> element.
<i>mode</i> (see Section 5.2.2)	Optional, but only allowed if <i>widget</i> has a body. Either "merge-children" or "replace".

7.2.2 The *property-method* Element

DTD

```

<!ENTITY % NameReference      "NMTOKEN">
<!ENTITY % FunctionName       "NMTOKEN">
<!ENTITY % URIorNameReference "CDATA">
<!ENTITY % SourceModes       "(replace|merge-children)">

<!ELEMENT property-method (#PCDATA|property|call)*>
<!ATTLIST property-method
  name      %NameReference;      #IMPLIED
  set-method %FunctionName;      #IMPLIED
  get-method %FunctionName;      #IMPLIED
  source    %URIorNameReference; #IMPLIED
  mode      %SourceModes;        "replace">
    
```

Description

If the *toolkit* is a markup language, then the *set-method* and *get-method* attributes of the *property-method* typically map to tag names in the markup language (e.g., CARD in WML, or INPUT in HTML). If the *toolkit* is a predefined library of interface components (e.g., the Microsoft Foundation Classes or Java Swing class libraries), then the *set* and *get* attributes map to class methods or function names in the *toolkit's* application programming interface.

Attributes

<i>name</i> (see Section 5.2.1)	The <i>name</i> that is used in the <i>style</i> element to render the UI element property.
<i>set-method</i> and <i>get-method</i>	Name of a <i>set</i> and a <i>get</i> method in the toolkit, respectively. Read-only properties have only the <i>get-method</i> attribute defined.
<i>source</i>	Optional, but only allowed if <i>property-method</i> has a body. If present, the body of the <i>property-method</i> element must be empty. <i>Source's</i> value is a URI of a UIML fragment that is inserted as the child of the <i>property-method</i> element.
<i>mode</i> (see Section 5.2.2)	Optional, but only allowed if <i>property-method</i> has a body. Either "merge-children" or "replace".

7.2.3 The *event* Element

DTD

```

<!ENTITY % EventName          "NMTOKEN">
<!ENTITY % PartReference      "IDREF">
<!ENTITY % PropertyName      "NMTOKEN">
<!ENTITY % ClassReference     "NMTOKEN">
<!ENTITY % NameReference     "NMTOKEN">

<!ELEMENT event EMPTY>
<!ATTLIST event
  name      %EventName;      #IMPLIED
  part      %PartReference;  #IMPLIED
  pname     %PropertyName;   #IMPLIED
  class     %ClassReference; #REQUIRED
  maps-to   %NameReference;  #IMPLIED>

```

Description

When appearing as a child of a *toolkit* element, the *event* element specifies the mapping from an arbitrary name used within the *style* element in the *interface* element to the name of a *toolkit* event type. For example, the event class "Selected" might be mapped to the Java toolkit class "java.awt.event.ActionEvent".

Attributes

<i>name</i> (see Section 5.2.1)	The <i>name</i> that is used in the <i>style</i> element to render the UI event.
<i>class</i> (see Section 5.2.1)	Not used.
<i>maps-to</i>	Specifies the toolkit event to which <i>name</i> maps.
<i>pname</i>	Not used.
<i>part</i>	Not used.

8 Alternative Organizations for UIML Files

Until now, UIML documents shown have followed a rigid format: appearing in the *uiml* element is first the *toolkit-peer* element, then the *interface*, then the *logic*. Alternative document organizations are possible:

- Elements can be ordered differently if desired.
- The *content*, *style*, and *action-list* elements can be embedded within the *part* element. This makes it easier to write UIML, because all information about an interface part is centralized where the *part* is defined.
- The UIML document can be split into multiple documents, with different documents loaded only when an event triggers loading.
- A renderer can start rendering before an entire UIML document is received to reduce latency for a user in large UIML documents.

The DTD in the Appendix permits these combinations. Refer to the DTD for precise information on what organizations are legal, and to the examples document [2] for some illustrations of alternate organizations.

8.1 Splitting a UIML Document into Multiple Files

Often it is desirable to put UIML fragments into separate files, and then include one file within another. This can be accomplished in four ways in UIML.

8.1.1.1 Normal XML mechanism

XML allows file inclusion as illustrated below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE uiml PUBLIC
    "-//Universal Interface Technologies//UIML 2.0//EN"
    "http://uiml.org/dtd/UIML20.dtd">

<!ENTITY toolkit      SYSTEM "http://a.com/toolkit-java.ui">
<!ENTITY parts        SYSTEM "parts.ui">
<!ENTITY style        SYSTEM "style.ui">
<!ENTITY content      SYSTEM "content.ui">
<!ENTITY action-list  SYSTEM "action-list.ui">

<uiml>
  &toolkit;
  <interface>&parts;&style;&content;&action-list;</interface>
</uiml>
```

8.1.1.2 Source attribute with Mode="merge-children"

Section 5.2.2 explains and exemplifies how to use the *source* attribute and *mode* attribute of "merge-children" to populate the children of an element with the content of a URI (which could be a UIML fragment).

8.1.1.3 Source attribute with Mode="replace"

Section 5.2.2.2 explains and exemplifies how to use the *source* attribute and *mode* attribute of "replace" to replace an element with the content of a URI (which could be a UIML fragment).

8.1.1.4 Edit element

Section 5.7.10 explains and exemplifies how to use the *edit* element, which can add a new child to an element in the UIML document, or replace an element in the UIML document. The new child is the content of a URI, which should contain a UIML fragment.

References

- [1] Extensible Markup Language (XML), W3C Proposed Recommendation 10-February-1998, REC-xml-19980210, T. Bray, et al, February 10, 1998, <http://www.w3.org/TR/REC-xml>.
- [2] *Examples of User Interface Markup Language (UIML) Version 2.0*, http://www.uiml.org/specs/UIML_Examples_1e.pdf, 1 August 1999.
- [3] B. Bos, H. W. Lie, C. Lilley, I. Jacobs, Cascading Style Sheets, level 2, CSS2 Specification. W3C Recommendation 12-May-1998, <http://www.w3.org/TR/REC-CSS2/>.
- [4] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, Jonathan E. Shuster, "UIML: An Appliance-Independent XML User Interface Language," 8th International World Wide Web Conference, Toronto, May 1999, <http://www8.org/w8-papers/5b-hypertext-media/uiml/uiml.html>. Also appeared in *Computer Networks*, Vol. 31, pp. 1695-1708.
- [5] UIML1.0 specification, <http://www.uiml.org/docs/UIML1-spec.html>, 1997.
- [6] Wireless Application Protocol Forum, Wireless Application Protocol Wireless Markup Language Specification, April 1998, <http://www.wapforum.org/docs/technical/wml-30-apr-98.pdf>.
- [7] J. Clark and S. Deach, eds, Extensible Style Language (XSL), W3C Proposed Recommendation, 18 August 1998. <http://www.w3.org/TR/WD-xsl>.
- [8] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, <http://www.ietf.org/rfc/rfc2119.txt>, March 1997.

Appendix – UIML 2.0 Document Type Definition

```
<?xml version="1.0" encoding="US-ASCII"?>

<!--
  User Interface Markup Language (UIML)
  =====

  Developed by:

      Universal Interface Technologies, Inc.

  Authors:

      Constantinos Phanouriou (constantinos.phanouriou@universalit.com)
      Alan L. Batongbacal
      Marc Abrams (abrams@universalit.com)

  Usage:

      <?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
      <!DOCTYPE uiml PUBLIC
        "-//Universal Interface Technologies//UIML 2.0//EN"
        "http://www.uiml.org/dtds/uiml20.dtd">

      <uiml>
        <head> ... </head>
        <toolkit-peers> ... </toolkit-peers>
        <interface> ... </interface>
        <logic> ... </logic>
      </uiml>

  Description:

      This DTD describes the implementation of UIML as of 01 August 1999.
      Additional information may be found at the following URL:

          http://www.uiml.org/docs/uiml20/

  Change History:

      31 Jul 1997 - A Batongbacal (alanlb@universalit.com)
                  - further language revisions
      24 Jul 1999 - M Abrams (abrams@universalit.com)
                  - updated to revised language
      15 Jul 1999 - C Phanouriou (constantinos.phanouriou@universalit.com)
                  - first draft

-->

<!-- ===== Entities ===== -->

<!ENTITY % AppId                " ID">
<!ENTITY % AppName              "NMTOKEN">
<!ENTITY % ClassName            "NMTOKEN">
<!ENTITY % ClassReference       "NMTOKEN">
<!ENTITY % ConstantName         "NMTOKEN">
<!ENTITY % ConstantReference    "NMTOKEN">
<!ENTITY % EventName            "NMTOKEN">
<!ENTITY % FunctionName         "NMTOKEN">
<!ENTITY % InterfaceName        "NMTOKEN">
```

UIML 2.0 Language Reference

```
<!ENTITY % MIMEType          "CDATA">
<!ENTITY % Name              "NMTOKEN">
<!ENTITY % NameReference     "NMTOKEN">
<!ENTITY % ObjectReference   "CDATA">
<!ENTITY % PartId            "ID">
<!ENTITY % PartReference     "IDREF">
<!ENTITY % PropertyName      "NMTOKEN">
<!ENTITY % ScriptName        "NMTOKEN">
<!ENTITY % SourceModes       "(replace|merge-children)">
<!ENTITY % StructureName     "NMTOKEN">
<!ENTITY % StyleName         "NMTOKEN">
<!ENTITY % ToolkitName       "NMTOKEN">
<!ENTITY % URI                "CDATA">
<!ENTITY % URIorNameReference "CDATA">

<!-- ===== Content Models ===== -->

<!--
  'uiml' is the root element of a UIML document.  The current
  language version is 2.0.
-->

<!ELEMENT uiml (head?, toolkit-peers, interface, logic?)*>
<!ATTLIST uiml
  name      %AppId; #IMPLIED
  version   CDATA   #FIXED   "2.0">

<!--
  The 'head' element is meant to contain metadata about the UIML
  document.  Its content is currently undefined.
-->

<!ELEMENT head EMPTY>

<!--
  The 'toolkit-peers' element contains information that defines
  how a UIML interface component is mapped to the target platform's
  rendering technology.
-->

<!ELEMENT toolkit-peers (toolkit+)*>

<!ELEMENT toolkit (widget|event)*>
<!ATTLIST toolkit
  name      %ToolkitName;          #REQUIRED
  source    %URIorNameReference;  #IMPLIED
  mode      %SourceModes;          "replace">

<!ELEMENT widget (property-method)*>
<!ATTLIST widget
  name      %NameReference;        #REQUIRED
  maps-to   %NameReference;        #REQUIRED
  source    %URIorNameReference;  #IMPLIED
  mode      %SourceModes;          "replace">

<!ELEMENT property-method (#PCDATA|property|call)*>
<!ATTLIST property-method
  name      %NameReference;        #IMPLIED
  set-method %FunctionName;        #IMPLIED
  get-method %FunctionName;        #IMPLIED
  source    %URIorNameReference;  #IMPLIED
  mode      %SourceModes;          "replace">
```

UIML 2.0 Language Reference

```
<!ELEMENT call (param)*>
<!ATTLIST call
    function %FunctionName; #REQUIRED>

<!--
    The 'interface' element describes a user interface in terms of
    presentation cues, component structure and action specifications.
-->

<!ELEMENT interface (structure|(style|content|action-list)*)+>
<!ATTLIST interface
    name %InterfaceName; #IMPLIED
    source %URIOrNameReference; #IMPLIED
    mode %SourceModes; "replace">

<!--
    The 'structure' element describes the initial logical relationships
    between the components (i.e., the "part"s) that comprise the user
    interface.
-->

<!ELEMENT structure (part+)>
<!ATTLIST structure
    name %StructureName; "default"
    source %URIOrNameReference; #IMPLIED
    mode %SourceModes; "replace">

<!--
    A 'part' element describes a conceptually complete component of the
    user interface.
-->

<!ELEMENT part (part|style|content|action-list)*>
<!ATTLIST part
    name %PartId; #REQUIRED
    class %ClassName; #IMPLIED
    source %URIOrNameReference; #IMPLIED
    mode %SourceModes; "replace">

<!--
    A 'style' element is composed of one or more 'property' elements,
    each of which specifies how a particular aspect of an interface
    component's presentation is to be presented.
-->

<!ELEMENT style (property+)>
<!ATTLIST style
    name %StyleName; "default"
    source %URIOrNameReference; #IMPLIED
    mode %SourceModes; "replace">

<!--
    A 'property' element is typically used to set a specified
    property for some interface component (or alternatively,
    a class of interface components), using the element's
    character data content as the value. If the 'operation'
    attribute is given as "get", the element is equivalent to
    a property-get operation, the value of which may be "returned"
    as the content for an enclosing 'property' element.
-->

<!ELEMENT property (#PCDATA|reference|property|constant)*>
<!ATTLIST property
```

UIML 2.0 Language Reference

```
    pname      %PropertyName;    #REQUIRED
    part       %PartReference;    #IMPLIED
    class      %ClassReference;   #IMPLIED
    operation  (set|get)          "set">

<!--
  A 'reference' may be thought of as a property-get operation,
  where the "property" to be read is a 'constant' element defined
  in the UIML document's 'content' section.
-->

<!ELEMENT reference EMPTY>
<!ATTLIST reference
  name %ConstantReference; #REQUIRED>

<!--
  The 'content' element is composed of one or more 'constant'
  elements, each of which specifies some fixed value.
-->

<!ELEMENT content (constant+)>
<!ATTLIST content
  name %StyleName;            "default"
  source %URIOrNameReference; #IMPLIED
  mode %SourceModes;         "replace">

<!--
  'Constant' elements may be hierarchically structured.
-->

<!ELEMENT constant (#PCDATA|constant)*>
<!ATTLIST constant
  name %ConstantName;        #REQUIRED
  source %URIOrNameReference; #IMPLIED
  mode %SourceModes;         "replace">

<!--
  The 'action-list' element gives one or more "rule"s that
  specifies what 'action' is to be taken whenever an associated
  'condition' becomes TRUE.
-->

<!ELEMENT action-list (rule+)>
<!ATTLIST action-list
  name %StyleName;            "default"
  source %URIOrNameReference; #IMPLIED
  mode %SourceModes;         "replace">

<!ELEMENT rule (condition,action)>

<!--
  At the moment, "rule"s may be associated with two types of
  conditions: (1) whenever some expression is equal to some other
  expression; and (2) whenever some event is triggered and caught.
-->

<!ELEMENT condition (equal|event)>

<!ELEMENT action (property|event|edit)+>

<!ELEMENT equal (event,(constant|property|reference))>

<!ELEMENT event EMPTY>
```

UIML 2.0 Language Reference

```
<!ATTLIST event
    name      %EventName;      #IMPLIED
    part      %PartReference;   #IMPLIED
    pname     %PropertyName;    #IMPLIED
    class     %ClassReference;  #REQUIRED
    maps-to   %NameReference;   #IMPLIED>

<!ELEMENT edit EMPTY>
<!ATTLIST edit
    element %NameReference;      #REQUIRED
    name    %NameReference;      #REQUIRED
    source  %URIOrNameReference; #IMPLIED
    mode    %SourceModes;        "replace">

<!--
    The 'logic' element describes the application upon which the
    user interface described by the UIML document is layered, and
    how the two are to be connected.
-->

<!ELEMENT logic (object)+>

<!--
    An 'object' element describes a single object which is part of
    the application logic.
-->

<!ELEMENT object (function*, script?)>
<!ATTLIST object
    name      %Name;              #REQUIRED
    type      %MIMEType;          #REQUIRED
    maps-to   %ObjectReference;   #IMPLIED
    source    %URIOrNameReference; #IMPLIED
    mode      %SourceModes;        "replace">

<!--
    'Function' describes a function that is part of the application logic.
    A function can have any number of formal parameters and a return value.
-->

<!ELEMENT function (param*, returns?)>
<!ATTLIST function
    name      %FunctionName;      #REQUIRED
    maps-to   %FunctionName;      #REQUIRED
    source    %URIOrNameReference; #IMPLIED
    mode      %SourceModes;        "replace">

<!--
    'Param' denotes a single formal or actual parameter to a function.
-->

<!ELEMENT param (property|call)*>
<!ATTLIST param
    name      %Name;              #REQUIRED
    source    %URIOrNameReference; #IMPLIED
    mode      %SourceModes;        "replace">

<!--
    The 'returns' element marks the return value of a callable function.
-->
```

UIML 2.0 Language Reference

```
<!ELEMENT returns EMPTY>
<!ATTLIST returns
    name %Name; #IMPLIED>

<!--
    The 'script' element contains data passed to an embedded scripting
    engine.
-->

<!ELEMENT script (#PCDATA)>
```