User Interface Markup Language (UIML) Draft Specification

Document Version 17 January 2000

Language Version 2.0a

Editor (uiml-editor@uiml.org):

Constantinos Phanouriou
Universal Interface Technologies, Inc., phanouriou@universalit.com

Disclaimer: *This document is subject to change without notice.*

1 Introduction to UIML 2.0a

The User Interface Markup Language (UIML) is the result of starting with a clean sheet of paper and creating 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. UIML 2 is a declarative, XML-compliant language that originated with the UIML 1.0 specification, created in 1997 [5].

To create a user interface (UI), one writes a UIML document, which includes presentation style appropriate for devices on which the UI will be deployed. UIML is then automatically mapped to a language used by the target device, such as HTML, WML, VoiceXML, C++ (with an API such as MFC), Java (with an API such as Swing), and so on.

Among the goals of UIML are the following:

- allow individuals to implement UIs for any device without learning languages and APIs specific to the device,
- reduce the time to develop UIs for a family of devices,
- provide a natural separation between UI code and application logic code,
- allow non-programmers to implement UIs,
- permit rapid prototyping of UIs,
- simplify internationalization and localization,
- allow efficient download of UIs over networks to client machines,
- facilitate enforcement of security, and
- allow extension to support UI technologies that are invented in the future.

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

1.1 Relationship of UIML to XML, XSL, and CSS

UIML is compliant with the W3C XML 1.0 specification [1]. Appendix A contains the UIML 2.0a DTD.

When UIML is compiled to HTML, CSS style sheets or XSL formatting objects [6] can be used with the resultant HTML. In addition, XSLT [7] can be used to transform UIML to other XML-compliant markup languages.

1.2 Purpose of this Document

This document serves as the official language reference for UIML 2.0a. 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.

UIML is intended to be an open, standardized language, which may be freely implemented without any licensing costs. The goal of this document is to elicit feedback from the wider community. Comments are encouraged; please send them to uiml-editor@uiml.org or participate in discussion on the listserv uiml-language@uiml.org. A submission to a standards organization will occur after comments are received and this draft specification is finalized.

1.3 Terminology

Certain terminology used in the specification is made precise through the definitions below.

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

End user: The person that uses the application's UI.

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

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

UI Toolkit: A toolkit is the markup language or software library upon which an application's UI runs. Note that we use the word "toolkit" in a more general sense than its traditional use. We use I to mean both markup languages that are capable of representing UIs (e.g., Wireless Markup Language [WML], HTML, and VoiceXML) as well as APIs for imperative programming languages (e.g., Java AWT, Java Swing, Microsoft Foundation Classes).

Platform: A platform is a combination of a device, operating system (OS), and a UI 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 [9] renderer.

Rendering: Rendering is the process of converting a UIML document into a form that can be displayed (e.g., through sight or sound) to an end user, and with which an end user can interact. Rendering can be accomplished in two ways:

1. By *compiling* UIML into another language (e.g., WML, Java), which allows display and interaction of the UI described in UIML. Compilation might be accomplished by XSL [6], or by a program written in a traditional programming language.
2. By *interpreting* UIML, meaning that a program reads UIML and makes calls to an API which displays the UI and allows interaction. Interpretation is the same process that a Web browser uses when presented with an HTML document.

Rendering engine: Software that performs the actual process or rendering a UIML document.

UI Widget: UIML describes how to combine UI widgets. The UI toolkit with which the UI is implemented provides primitive building blocks, which we call widgets. The term “widget” is traditionally used in conjunction with a graphical UI. However we use it in a more general sense, to mean presentation elements of any UI paradigm.

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, VoiceXML) 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 3.1.

Runtime: This is the period of time during which the UI is displayed (e.g., through sight or sound) to an end user, and the end user can interact with the UI.

Other terms: 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].

Ellipses (. . .) indicate where attribute values or content have been omitted. Many of the examples given below are UIML fragments and additional code maybe needed to render them.

URLs given inside the code segments in this document are for demonstration only and may not actually exist.

2 Document Status

Versions of this document available online are listed below:

- Latest version of UIML2 spec: http://www.uiml.org/docs/uiml20
- This version of the document:
  - HTML: http://www.uiml.org/docs/uiml20a-17Jan00.html
  - PDF: http://www.uiml.org/docs/uiml20a-17Jan00.pdf
- Previous version:
  - HTML: http://www.uiml.org/docs/uiml20-990801.html
  - PDF: http://www.uiml.org/docs/uiml20-990801.pdf

2.1 Editor

Constantinos Phanouriou, Universal Interface Technologies, Inc., PO Box 20746, Roanoke, VA 24018, uiml-editor@uiml.org.

2.2 Copyright Notice

2.3 UIML License Cost

UIML 2.0a may be freely implemented without any license cost.

2.4 Errata

The list of known errors in this specification is available at the following location:

http://www.uiml.org/docs/uiml20a-errata.html

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

2.5 Comments

Comments regarding this document can be submitted uiml-editor@uiml.org.

3 Structure of a UIML Document

3.1 Overview

In UIML version 2.0a, a UI is a set of interface elements with which the end 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 UI. The parts may be organized differently for different categories of end users and different families of devices. Each interface part has *content* (e.g., text, sounds, images) used to communicate information to the end user. Interface parts can also receive information from the end 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.

3.1.1 Interface Behavior

UIML describes in a *behavior* element what actions are to occur as an end user interacts with a UI. The *behavior* element is based on rule-based languages. Each rule contains a condition and a sequence of actions. Whenever a condition is true, the associated actions are executed.

Whenever an end user interacts with a UI, the UI generates *events*. In this version of the UIML specification, each condition can occur only when an event occurs. This simplifies the rendering of UIML by compilation to other languages.

There are three types of actions: (1) a property of some part in the UI changes, (2) a function in a scripting language is invoked, or (3) a function or method in the application logic is invoked. In cases (2) and (3), UIML gives a *syntax* for *describing* the calling convention, but does not specify an implementation of how the call is performed (e.g., RPC, RMI, CORBA).

Finally, a UIML document provides sufficient information to create an implementation in which the application logic modifies the UI programmatically.

3.1.2 Philosophy Behind UIML's Tags

UIML can be viewed as a meta-language or an extensible language, analogous to XML. XML does not contain tags specific to a particular purpose (e.g., HTML's <H1> or <IMG>). 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 an extensible language can be standardized once, rather than requiring periodic standardization committee meetings to add new tags as the applications evolve.

Analogously, UIML does not contain tags specific to a particular UI toolkit (e.g., <WINDOW> or <MENU>). UIML captures the elements that are common to any UI through 28 generic elements. 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 in UIML. Thus UIML only

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

Thus a UIML author needs more than this document, which specifies the UIML language. You also need one document for each UI toolkit (e.g., Java Swing, Microsoft Foundation Classes, WML) to which you wish to map UIML. The toolkit-specific document enumerates for a particular toolkit a vocabulary of toolkit components (to which each *part* element in a UIML document is mapped) and their property names.

3.1.3 First UIML Example: Hello World

Here is the famous “Hello World” example in UIML. It simply generates a UI that contains the words "Hello World!".

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//UIT//DTD UIML 2.0a Draft//EN"
    "http://uiml.org/dtds/UIML2_0a.dtd">

<uiml>
  <interface>
    <structure>
      <part name="TopHello">
        <part name="hello" class="helloC"/>
      </part>
    </structure>
    <style>
      <property
        part-name="TopHello" name="rendering">Container</property>
      <property
        part-name="TopHello" name="content">Hello</property>
      <property
        part-class="helloC"  name="rendering">String</property>
      <property
        part-name="hello"    name="content">Hello World!</property>
    </style>
  </interface>
  <peers> ... </peers>
</uiml>
```

To complete this example, we must provide something for the `<peers>…</peers>` element. A VoiceXML [10] renderer given the above UIML code and the following peer element

```
<peers>
  <presentation name="VoiceXML">
    <component name="Container" maps-to="vxml:form"/>
    <component name="String" maps-to="vxml:block">
      <attribute name="content" maps-to="PCDATA"/>
    </component>
  </presentation>
</peers>
```

would output the following VoiceXML code:

```
<?xml version="1.0"?>
<vxml>
  <form>
    <block>Hello World!</block>
  </form>
</vxml>
```

A WML [9] renderer given the above UIML code and the following peer element

```
<peers>
  <presentation name="WML">
    <component name="Container" maps-to="wml:card">
      <attribute name="content" maps-to="wml:card.title"/>
    </component>
    <component name="String" maps-to="wml:p">
      <attribute name="content" maps-to="PCDATA"/>
    </component>
  </presentation>
</peers>
```

would output the following WML code:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.0//EN"
  "http://www.wapforum.org/DTD/wml.xml">

<wml>
  <card title="Hello">
    <p>Hello World!</p>
  </card>
</wml>
```

3.2 UIML Document Structure

A typical UIML 2.0a document is composed of these two parts:

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

   ```
   <?xml version="1.0"?>
   <!DOCTYPE uiml PUBLIC
     "-//UIT//DTD UIML 2.0a Draft//EN" http://uiml.org/dtds/UIML2_0a.dtd">
   ```

Note: This prolog should begin every UIML document, but for ease of readability some of the examples given in this document omit it.

2. The root element in the document, which is the *uiml* tag:

   ```
   <uiml xmlns='http://uiml.org/dtds/UIML2_0a.dtd'> ... </uiml>
   ```

See Section 5.1 for more information on the root element *uiml*. The *uiml* element contains four child elements:

a) An optional header element giving metadata about the document:

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

The *head* element is discussed in Section 5.2.

b) An optional UI description, which describes the parts comprising the UI, and their structure, content, style, and behavior:

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

Section 6.3 discusses the *interface* element.

c) An optional element that describes the mapping from each property and event name used elsewhere in the UIML document to a UI toolkit and to the application logic:

```
<peers> ... </peers>
```

Discussion of the *peers* element is deferred until Section 7.1, because the *peers* element normally just names an external file.

d) An optional element that allows reuse of fragments of UIML:

```
<template> ... </template>
```

Section 8.1 discusses the *template* element, and its use in building libraries of reusable UI components.

White space (spaces, new lines, tabs, and XML comments) may appear before or after each of the above tags (provided that the XML formatting rules are not violated).

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

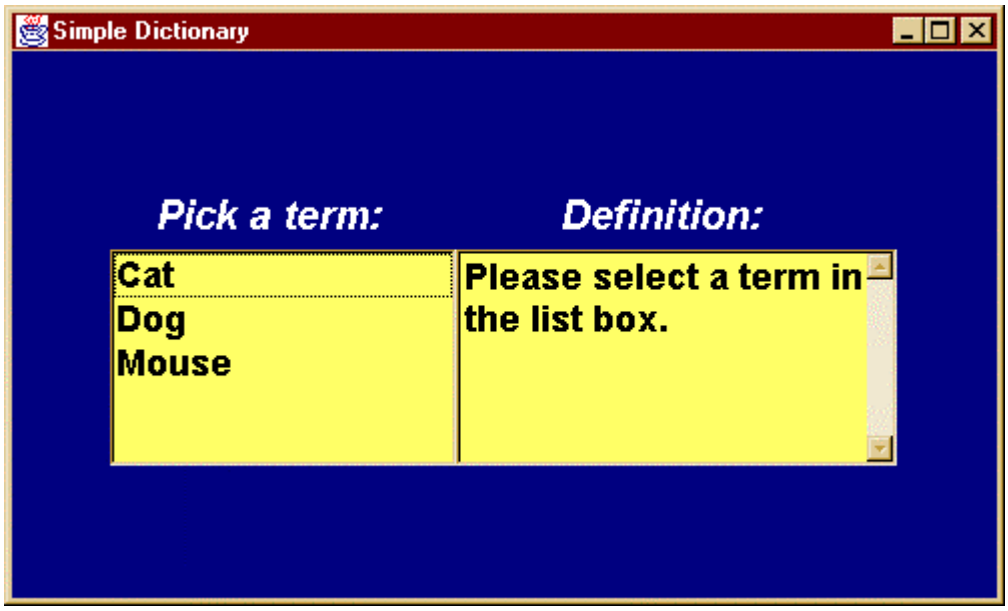
```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
    "-//UIT//DTD UIML 2.0a Draft//EN" "http://www.uiml.org/dtds/UIML2_0a.dtd">

<uiml xmlns='http://uiml.org/dtds/UIML2_0a.dtd'>
   <head>           ... </head>
   <interface>      ... </interface>
   <peers>          ... </peers>
   <template>       ... </template>
</uiml>
```

The four elements *head, interface, peers,* and *template* may appear in any order.

3.3 Second UIML Example

This section contains a 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 an end 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.



Boxes are overlaid on the UIML document to make reading easier by identifying major elements.

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
    "-//UIT//DTD UIML 2.0a Draft//EN"
    "http://uiml.org/dtds/UIML2_0a.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>
```

```
  <peers>
    <presentation name="java"
      source="http://uiml.org/toolkits/Java20AWT.ui"
      how="replace"/>
  </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"
           how="cascade">
      <property  part-class="Frame"    name="layout"     >gridBagLayout</property>
      <property  part-class="Frame"    name="xplace"     >relative</property>
      <property  part-class="Frame"    name="yplace"     >relative</property>
      <property  part-class="Frame"    name="background" >blue</property>
      <property  part-class="Frame"    name="font-style" >bold</property>
      <property  part-class="Frame"    name="location"   >100,100</property>
      <property  part-class="Frame"    name="size"       >500,300</property>

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

      <property  part-class="List"      name="background" >gray</property>

      <property  part-class="TextArea"  name="background" >gray</property>

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

      <property  part-name="Terms"     name="xplace"    >0</property>
      <property  part-name="Terms"     name="alignment" >north</property>
      <property  part-name="Terms"     name="fill"      >both</property>
```

```
      <property  part-name="DefnLabel" name="alignment" >center</property>
      <property  part-name="DefnLabel" name="xplace"    >1</property>
      <property  part-name="DefnLabel" name="yplace"    >0</property>

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

      <property event-class="LSelected" name="rendering" >ItemEvent</property>
    </style>
```

```
    <behavior>

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

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

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

    </behavior>
```

```
  </interface>

</uiml>
```

3.4 UIML Namespace

UIML is design to work with existing standards. This includes other markup languages that specify platform-dependent formatting (i.e., HTML for text, JSGF for voice, etc.). XML Namespaces remove the problem of recognition and collisions between elements and attributes of two or more markup vocabularies in the same file. All UIML elements and attributes are inside the "*uiml*" namespace, identified by the URI "*http://uiml.org/dtds/UIML2_0a.dtd*".

Example

Here is an example that combines UIML and HTML vocabularies:

```
<uiml:uiml xmlns:uiml='http://uiml.org/dtds/UIML2_0a.dtd'>
   <uiml:interface>
     <uiml:structure>
       <uiml:part uiml:name="A"/>
     </uiml:structure>

     <uiml:style>
       <uiml:property uiml:name="content" uiml:part-name="A">
         <html:em xmlns:html='http://www.w3.org/TR/REC-html40'
           >Emphasis</html:em>
       </uiml:property>
     </uiml:style>
   </uiml:interface>
</uiml:uiml>
```

The above code can be simplified by making uiml the default namespace as follow:

```
<uiml xmlns='http://uiml.org/dtds/UIML2_0a.dtd'>
   <interface>
     <structure>
       <part name="A"/>
     </structure>

     <style>
       <property name="content" part-name="A">
         <html:em xmlns:html='http://www.w3.org/TR/REC-html40'
           >Emphasis</html:em>
       </property>
     </style>
   </interface>
</uiml>
```

3.5 UIML Mime Type

The following mime type should be used for UIML documents:

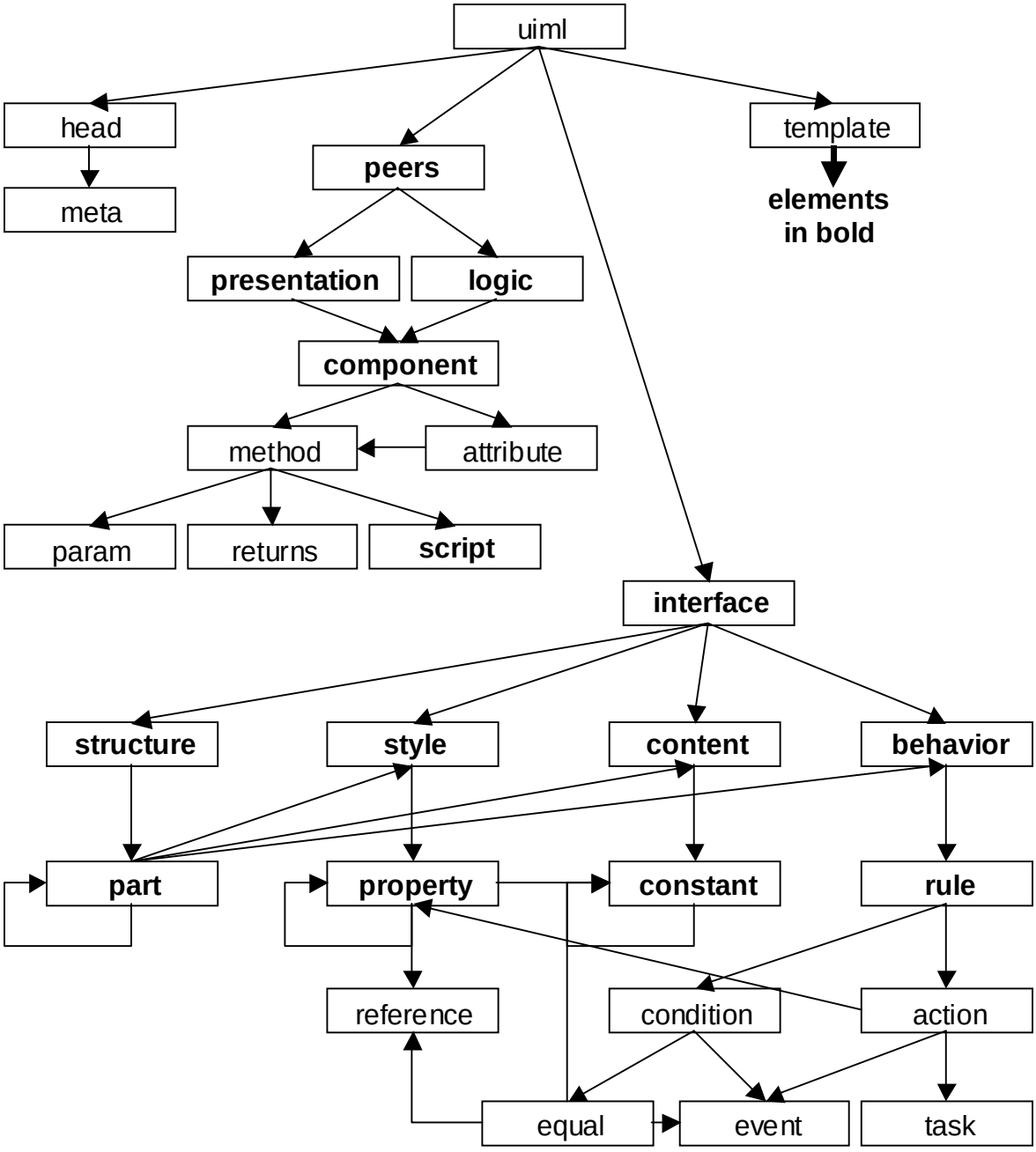
```
text/uiml
```

4 Table of UIML Elements

The table below is both an overview of all elements in UIML, and an index to where they are discussed in the remainder of this document. The UIML 2.0a DTD is given in Appendix A.

Element	Purpose	Page
<action>	Perform an action if the condition of a rule is true	38
<attribute>	A toolkit property	45
<behavior>	Specify rules for runtime behavior	34
<call>	Call a function or method external to UIML document	39
<component>	Specify a presentation or application logic peer	44
<condition>	Specify a condition for a rule	37
<constant>	Define a constant value	32
<content>	Specify a set of constant values	31
<equal>	Compares the value of an event with another value	37
<event>	Specify a runtime UI event	38
<head>	A container for metadata information	19
<interface>	A container for the interface description	22
<logic>	A container for computation components	43
<meta>	Define a piece of metadata as a name/value pair	20
<method>	An executable method	45
<param>	A parameter to a method	39
<part>	Specify a single interface part	24
<peers>	Describes mapping from property and event names to a UI toolkit and the application logic	41
<presentation>	A container for presentation components	41
<property>	Specify an interface property	26
<reference>	Reference a constant	33
<returns>	The return value of a method	47
<rule>	A condition/action pair	37
<script>	A container for executable script code	47
<structure>	Specify an interface physical structure	23
<style>	Specify a set of style properties for the interface	25
<template>	A container for reusing UIML elements	48
<uiml>	Top-level element in each UIML document	19

The elements of the UIML 2.0a DTD are represented below in the form of a diagram. Elements shown in **bold** are legal children of the *template* element.


uiml
head
meta
peers
presentation
logic
component
method
attribute
param
returns
script
template
elements
in bold
interface
structure
style
content
behavior
part
property
constant
rule
reference
condition
action
equal
event
task

5 The *uiml* and *head* elements

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

5.1 The *uiml* Element

DTD

```
<!ELEMENT uiml (head?, peers?, interface?, template*)>
```

Description

The *uiml* element is the root element in a UIML document. All other elements are contained in the *uiml* element. The *uiml* element appears as follow:

```
<uiml>...</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.

When multiple markup vocabularies are used within the same UIML file, then the *uiml* namespace must be specified as follow:

```
<uiml xmlns='http://uiml.org/dtds/UIML2_0a.dtd'>...</uiml>
```

5.2 The *head* Element

DTD

```
<!ELEMENT head (meta)*>
```

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 UI.

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

5.2.1 The *meta* Element

DTD

```
<!ELEMENT meta EMPTY>
<!ELEMENT meta EMPTY>
<!ATTLIST meta
   name      NMTOKEN  #REQUIRED
   content   CDATA    #REQUIRED>
```

Description

The *meta* element has the same semantics as the meta element in HTML. It describes a single piece of metadata about the current UIML document. This may includes author information, date of creation, etc.

The *name* attribute specifies the meta-information and the *content* attribute its content.

Example

```
<head>
  <meta name="Author" content="UIML Editor"/>
  <meta name="Date" content="November 1, 2001"/>
  <meta name="Description" content=
    "This is an example of how to use the meta tag in UIML.
    The content of the meta tag can include white space."/>
</head>
```

6 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.

6.1 Overview

The *interface* element contains four elements: *structure*, *style*, *content*, and *behavior:*

```
<interface>
   <structure> </structure>
   <style>     </style>
   <content>   </content>
   <behavior>  </behavior>
</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 images associated with interface parts to facilitate internationalization or customization of UIs to various user groups (e.g., by job role).

The *behavior* element defines what UI events should be acted on and what should be done.

6.2 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.

6.2.1 The *name* and *class* Attributes

The *part*, *event*, and *call* 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. However, elements of two different types may have the same name.

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.

6.2.2 The *source* Attribute

Certain UIML elements (*behavior*, *component*, *constant*, *content*, *interface*, *logic*, *part*, *peers*, *presentation*, *property*, *rule*, *script*, *structure*, and *style*) may contain an *source* attribute. Like HTML, the *source* attribute specifies a link from the UIML document to a Web resource named by a URI. However, the reason for using a link in UIML differs from HTML.

An *source* attribute can refer to two things:

- *A URI that does not contain UIML code*. In this case, the file can be textual (e.g. HTML) or binary (e.g., JPEG). This case is analogous to the IMG tag in HTML; the URI specifies

```
<constant name="Logo" source="http://uiml.org/icons/UIML_Logo.jpg"/>
```

- *A URI that does contain UIML code.* The UIML code is inserted into the element that contains the *source*, as explained in Section 8.2. Inserting code has several uses, explained in section 8:

  - Splitting a UI definition into several UIML documents
  - Creating a library of reusable UI components
  - Achieving the cascading behavior of CSS style sheets

  The URI may either be an element in the same document as the *source* appears, or in a different document:

  - *URI names the same document*. The two elements must either have the same tag or the URI must name a *template* element.

```
<style name="Simple"> ... </style>
<style name="Complex" source="#Simple" how="cascade"> ...
</style>
```

  - *URI names another document*. Again, the two elements must either have the same tag or the URI must name a *template* element.

```
<part name="Dialog"
   source="http://uiml.org/templates/Dialog.uiml#SimpleDialog"
   how="replace"
/>
```

    A *how* attribute of *cascade* achieves behavior similar to cascading in CSS, while *replace* allows a UIML document to be split into multiple files.

6.3 The *interface* Element

DTD

```
<!ELEMENT interface (structure|style|content|behavior)*>
<!ATTLIST interface
   %SourceAttributes;>
```

Description

All UIML elements that describe the interface are contained in the *interface* element. The *interface* element describes a UI, *not* the interaction of the UI and the application logic. The *logic* element is used to describe the UI/application logic interaction – see Section 7.3. A UIML interface may be as simple as a single string, or as complex as hundreds of interface elements that employ various interface technologies (e.g., voice, graphics, and 3D).

An *interface* is composed of four elements: *structure* (see Section 6.4), *style* (see Section 6.5), *content* (see Section 6.6), and *behavior* (see Section 6.7).

6.4 The *structure* Element

DTD

```
<!ELEMENT structure (part*)>
<!ATTLIST structure
   %SourceAttributes;>
```

Description

An application program can have a UI with one or more *organizations* associated with it. 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 UI) 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 an end 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 over a river is a structure that consists of many parts (e.g., steel beam, bolts), a UI 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*.

If a UIML document contains more than one *structure* element, then a UIML renderer must select by name exactly one *structure* element and ignore all other *structure* elements. The name of the selected element is 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.

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>
```

6.4.1 The *part* Element

DTD

```
<!ELEMENT part (style?, content?, behavior?, part*)>
<!ATTLIST part
   %SourceAttributes;
   class    NMTOKEN  #IMPLIED>
```

Description

Each *part* element corresponds either to one UI widget or to nothing (*null*). (It is sometimes useful to associate a part with nothing; for example a part might be needed for a large screen UI, but is omitted from a small device screen. In the former case, the part corresponds to a UI widget, and in the later case the part corresponds to nothing.)

Parts may be nested to represent a hierarchical relationship of parts. Let *a* and *b* denote two *part elements.* If part *b* is nested inside part *a*, and both *a* and *b* correspond to UI widgets (i.e., neither *a* nor *b* correspond to *null*), then *b*'s UI widget must be "contained in" *a*'s widget, where "contained in" is defined in terms of the UI toolkit. If the UI toolkit does not define nesting, then nesting part *b* in part *a* in a UIML document is equivalent to a UIML document in which the parts are not nested.

For example, the Java Swing toolkit has a notion of containers and components. Containers contain other containers or components, forming a hierarchy. Or, in a voice-based language, 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 6.4). When the interface is rendered, only one structure is used (as discussed in "Description" under Section 6.4); thus, a part is always associated with a unique class.

UIML allows the style, content, and behavior information associated with a particular part to be specified within the part itself. Usually, this information is specified in the corresponding *style*, *content*, and *behavior* elements.

6.5 The *style* Element

DTD

```
<!ELEMENT style (property*)>
<!ATTLIST style
   %SourceAttributes;>
```

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.

For example, the following fragment will make all parts with class="c1" use the Comic font, and the single part named "n1" have size 100 by 200:

```
<style name="Graphical">
  <property part-class="c1"  name="font"        >Comic</property>
  <property part-name="n1"   name="size"        >100,200</property>
</style>
```

However, unlike CSS and XSL, the style sheet is used to achieve device independence. This is discussed in Section 6.5.2.

There must be at least one *style* element, and there may be more than one. There is normally 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.

Style sheets may also use the mechanism for cascading described in Section 8.2.

6.5.1 The *property* Element

DTD

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

<!ATTLIST property
   %SourceAttributes;
   part-name      NMTOKEN #IMPLIED
   part-class     NMTOKEN #IMPLIED
   event-name     NMTOKEN #IMPLIED
   event-class    NMTOKEN #IMPLIED
   call-name      NMTOKEN #IMPLIED
   call-class     NMTOKEN #IMPLIED>
```

Description

A property associates a name and value pair with a part, event (see Section 6.7.4), or call (see Section 6.7.6). For example, a UI part named "button" might be associated with a property name "color" and value "blue". The *property* element provides the syntax to make the association between the name *color* and value *blue* with the part *button*.

6.5.1.1 Where property names are defined

Property names are *not* defined by the UIML specification. This is a powerful concept, because it permits UIML to be extensible: one can define whatever property names are appropriate for a particular device. For example, a "color" might be a useful property name for a device with a screen, while "loudness" might be appropriate for a voice-based device.

Property names instead are defined by the *peer* element (see Section 7). Normally the person that creates a UIML document does not define the property names. Instead, someone defines a set of properties one time, for example for the Java AWT UI toolkit, and the *peer* element simply specifies a URI that defines those property names. The compiler or interpreter that renders UIML should also access this URI to map property names in the UIML document to the desired UI toolkit.

Thus to use UIML one needs both a copy of this specification and a document defining the property names used in a particular *peer* element.

6.5.1.2 Semantics of property element

The semantics of a *property* element are as follows:

- If the *property* element is contained in a *param* or another *property* element, then the semantics are to get (return to the element containing the *property* element) a single property's value.
- Otherwise the semantics are set a value for a single property of an interface *part*, *event*, or *call*.

6.5.1.3 Legal values for property elements

The value for each *property* element can be one of the following:

- *A text string*. In this case the property has no children, and its body is set to the character sequence. If the string contains the ampersand character (&) or the left angle bracket (<), then they must be escaped using either numeric character references or the strings "&" and "<" respectively (see [1] for more rules about strings and XML documents). Note that A UIML parser must preserve white space. A UIML renderer may ignore leading and trailing spaces when rendering text on certain widgets.

```
<property part-name="part1" name="font">Helvetica-bold</property>
<property part-name="part1" name="title">Char: &amp;</property>
<property part-name="part1" name="content"
    ><![CDATA[Character &]]></property>
```

- *A reference element*. In this case the property is set to the value stored in a *constant* element that the *reference* element is associated with. In the following example, the value of *font* in the part with name *part1* is set to the value of the constant with name *font-name*.

```
<property part-name="part1" name="font">
  <reference constant-name="font-name"/>
</property>
...
<content>
  <constant name="font-name">Helvetica-bold</constant>
</content>
```

- *Another property element*. The value of one property can be set to the value of another property. For example, suppose we want to set the font of part *part1* to whatever font *part2* currently has. The following UIML achieves this:

```
<property part-name="part1" name="font">
  <property part-name="part2" name="font"/>
</property>
```

The nested *property* element gets the *font* of *part2*. The nested property does a get because it is nested in another *property* element, as explained in Section 6.5.1.2. That returned value then becomes the value of the *font* property in part *part1*.

- *A call element*. As explained in Section 6.7.6, a *call* is an invocation of code, such as calling a function or method in a script, in the application logic, or external to the application. In this case the property is set to the return value of the invocation.

```
<property part-name="part1" name="font">
  <call name="getFont"/>
</property>
```

  The *logic* element (contained in the *peers* element of a UIML document) defines the code to which *getFont* corresponds and how to invoke that code; see Section 7.3.

Resolving Conflicting Property Values

Consider the following example:

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

<style>
  <property part-name="n1" name="color">Blue</property>
  <property part-class="c1" name="color">Green</property>
</style>
```

Both property elements are assigning a value for the color of the same part. However, a property for a part can have only one value at any given time. To remove any ambiguity, two (or more) *property* elements cannot have the same *part-name*. Also two *property* elements cannot have the same *part-class*. However, two property elements can point to the same *part* (one with *part-name* and another with *part-class*). When there is a conflict, the property that specifies a *part-name* overrides the property that specifies a *part-class*. The same applies for property elements that assign values to events and methods.

6.5.2 Using Properties to Achieve Platform Independence

One of the powerful aspects of UIML is the ability to design a UIML document that can be mapped to multiple platforms. This is achieved by a special property called *rendering*.

To illustrate the use of *rendering*, let's look at an example. Suppose we were going to create a UI specifically for Java AWT. First our UIML document would need to introduce a vocabulary of UI widgets for Java AWT. This is done by the *peers* section of the dictionary example in Section 3.3:

```
<peers>
  <presentation name="java"
    source="http://uiml.org/toolkits/Java20AWT.ui" how="replace"/>
</peers>
```

The above UIML fragment refers to an external Web resource that defines the vocabulary. That file can be viewed as a black box by the UIML author. (It actually contains a *presentation* element, discussed in Section 7.2.) The Web resource uses all Java AWT class names as UI widget names: Button, List, and so on. The UIML author can then directly use these names as class names when defining parts:

```
<structure>
  <part class="Button" name="submitButton"/>
</structure>
```

On the other hand, suppose we wanted to design a UIML file that could be mapped either to Java AWT or to Java Swing. And suppose the Web resource named in the *peers* element introduced all the Swing class names as vocabulary to use in the UIML document. Now if we wanted to map the *submitButton* either to an AWT *Button* or to a Swing *JButton*, then we could not make

submitButton's class *Button*. Instead, UIML permits the introduction of a psuedo-name chosen by the UIML author. Suppose we choose as our class name *AWTorSwingButton*. Our UIML fragment above then becomes this:

```
<structure>
  <part class="AWTorSwingButton" name="submitButton"/>
</structure>
```

Now comes the key idea. The *style* section is used to map *AWTorSwingButton* to either Button or JButton:

```
<style name="AWT-specific">
  <property part-class="AWTorSwingButton" name="rendering"
  >Button</property>
</style>

<style name="Swing-specific">
  <property part-class="AWTorSwingButton" name="rendering"
  >JButton</property>
</style>
```

If the rendering engine is invoked with style name *AWT-specific*, then the *submitButton* will map to an AWT button; otherwise if *Swing-specific* is used, then the *submitButton* maps to *JButton*.

Given this basic example, some variations are possible. First, the *style* section can specify the *rendering* property not only for *part-class*, but also *part-name*. In this case, the rendering specified only applies to the part with the specified *part-name*.

Second, the *rendering* property can also be specified for *event-class*, *event-name*, *call-class*, and *call-name*. Let's consider *event-class*. 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 the name *ButtonOrMenuSelection* to a platform-specific name. In this case there would be style elements with two different names:

```
<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 event-class="ButtonOrMenuSelected"
     name="rendering">ButtonSelected</property>
<style>

<style name="Platform2">
```

```
  <property event-class="ButtonOrMenuSelected"
     name="rendering">MenuSelected</property>
<style>
```

(The values *ButtonSelected* and *MenuSelected* are part of the vocabulary of the target platform, defined in the *peers* element.)

As a second example, the dictionary example of Section 3.3 contains the following:

```
<style>
   <property event-class="LSelected"
     name="rendering" >ItemEvent</property>
</style>
...
<behavior>
  ...
        <event part-name="Terms" class="LSelected"
               name="item-selected"/>
  ...
</behavior>
```

The *behavior* section describes what actions to take in response to various user interface events (see Section 6.7). The *event* element refers to an event of class *LSelected*, named to represent a list selection of one of the animals in the dictionary list. The *style* element specifies that all events with class *LSelected* are mapped to the Java AWT class *ItemEvent.* If we were to modify the code in Section 3.3 to map to another platform, we could then map *LSelected* to something else in another toolkit by specifying a different *rendering* property for event-class *LSelected.*

Finally, *call* elements can also be given a *rendering* attribute. This would allow a *call* action to map to different function calls, allowing a single user interface to be used with different application logic or scripting.

Rules to assign "rendering" property: A UIML renderer must obey the following rules in assigning each *part, event,* and *call* element a *rendering* property.

- If a *property* element exists with name *rendering* and *part-class, event-class,* or *call-class,* use the *property* element value as the rendering.
- Otherwise, if a *property* element exists with name *rendering* and *part-name, event-name,* or *call-name,* use the *property* element value as the rendering.
- Otherwise, use as the *rendering* property value the value of the *part-class, event-class,* or *call-class* attribute.

6.6 The *content* Element

DTD

```
<!ELEMENT content (constant*)>
<!ATTLIST content
   %SourceAttributes;>
```

Description

A part in a UI can be associated with various content, such as words, characters, sounds, images. UIML permits separation of the content from the structure in a UI. Separation is useful when different content should be displayed under different circumstances. For example, a UI might display the content in English or French. Or a UI might use different words for an expert versus a novice user, or different icons for a color-blind user. UIML can express this.

Normally one would set the content associated with a UI part through the *property* element:

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

<style>
  <property part-name="affirmativeChoice" name="label">Yes</property>
</style>
```

In the UIML fragment above, the button label is hard-wired to the string "Yes". Suppose we wanted to internationalize the interface. In this case UIML allows the value of a property to be what a programmer would think of as a variable reference using the *reference* element:

```
<style>
  <property part-name="affirmativeChoice" name="label">
    <reference constant-name="affirmativeLabel"/>
  </property>
</style>
```

The *reference* element refers to a *constant-name*, which is defined in the *content* element in a UIML document. The important concept is that there may be multiple *content* elements in a UIML document, each with a different name. When the interface is rendered, one of the *content* element is specified, and the *content* elements inside are then used to satisfy the *reference* elements.

This is illustrated in the following example. The UI contains two parts. The class name “button” suggests that each part be rendered as a button in a graphical UI. (The *style* element [Section 6.5] actually determines how the class called “button” is rendered – it may be rendered as radio buttons or a voice response.) The button labels are references to *constant-name* "affirmativeLabel" and "negativeLabel". There are three alternative definitions of these *constant-name*s, corresponding to three languages: English, German, or slang English. Thus three *content* sections are defined, one for each language. Within each *content* element one or

more *constant* elements are used to provide the actual literal string that appears in the UI (e.g., “Yes” for English but “OK” for slang English).

When the interface is rendered, a mechanism outside the scope of this specification supplies a content name (eithr *English*, *German*, or *EnglishSlang)*. The *content* element whose name matches the supplied name is then used, and all other *content* elements are ignored. This then determines whether the value of the label property for the "affirmativeChoice" button is "Yes", "Ja", or "OK." (If the supplied name does not match the name attribute of any *content* element, then the interface cannot be rendered.)

Example

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

<style>
  <property part-name="affirmativeChoice" name="label">
    <reference constant-name="affirmativeLabel"/>
  </property>
  <property part-name="nagativeChoice" name="label">
    <reference constant-name="negativeLabel"/>
  </property>
</style>

<content name="English">
   <constant name="affirmativeLabel" >Yes</property>
   <constant name="negativeLabel"    >No</property>
</content>

<content name="German">
   <constant name="affirmativeLabel" >Ja</property>
   <constant name="negativeLabel"    >Nein</property>
</content>

<content name="EnglishSlang">
   <constant name="affirmativeLabel" >OK</property>
   <constant name="negativeLabel"    >No</property>
</content>
```

The last *content* element could also be shortened, by using the *source* attribute, discussed in Section 8.2, so that *EnglishSlang* inherited the *negativeLabel* from *English* as follows:

```
<content name=”EnglishSlang” source=”English” how="cascade">
   <property part=”affirmativeChoice” pname=”label”>OK</property>
</content>
```

6.6.1 The *constant* Element

DTD

```
<!ELEMENT constant (#PCDATA|constant)*>
<!ATTLIST constant
   %SourceAttributes;>
```

Description

Constant elements contain the actual text strings, sounds, and images associated with UI parts from the *part* element. Each constant element is identified by a *name* attribute and is referenced by the *reference* element.

Example

The following example shows how to create constant elements that point to a string, a sound clip, and an image. Similarly, you can create constants that point to video clips, binary files, and other objects.

```
<content>
   <constant name="Name">UIML</constant>
   <constant name="Sound" source="http://uiml.org/uiml.wav"/>
   <constant name="Image" source="http://uiml.org/uiml.jpg"/>
</content>
```

The *constant* element can also be used to represent literal strings used inside the *condition* element (see Section 6.7.2). For example:

```
<condition>
  <equal>
    <event part-name="inYear" class="filled" name="content"/>
    <constant>2000</constant>
  </equal>
</condition>
```

6.6.2 The *reference* Element

DTD

```
<!ELEMENT reference EMPTY>
<!ATTLIST reference
   constant-name  NMTOKEN #REQUIRED>
```

Description

The *reference* element references the value of the *constant* element specified by the *constant-name* attribute.

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 anywhere the string is required (e.g., as values of a property) the *reference* element can be used. Thus, if we can modify the text in the *constant* element, the change propagates to all the places in the UIML document that is referenced.

- 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 end 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, and then the part's values can be initialized in the *style* section using a *property* element containing a *reference* element as its value. In the *behavior* 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 3.3.

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

6.7 The *behavior* Element

DTD

```
<!ELEMENT behavior (rule*)>
<!ATTLIST behavior
   %SourceAttributes;>
```

Description

The *behavior* of the interface when the end user interacts with it (e.g., what happens when an end user presses a button) is described by enumerating a set of *conditions* and associated *actions*. This is motivated by rule-based systems. Whenever a condition is true, the associated action is performed.

- UIML allows two types of conditions. The first is true when an event occurs (e.g., a button is pressed in the UI). The second is true when an event occurs and the value of some data associated with the event is equal to a certain value (e.g., a list selection is made and the selected item is "cat" – the first *condition* element in the dictionary example in Section 3.3.

  (UIML does not allow other conditions, to avoid implementations that are computationally expensive [e.g., continuous polling to determine when a condition holds] or impossible with simple platforms [e.g., WML]).

- Actions can be internal to the UIML document -- specifying a change in a property's value -- or external -- invoking a method in a script, program, or object.

A unique aspect of UIML is that events are also described in a device-independent fashion, by giving each event a *name* and identifying the *class* to which it belongs. As we discussed for parts, the UI implementor uses instance and class names of his/her choice for events, and those names are mapped to an event in the underlying platform in the *style* element. For example, the end user might use the class "selection," and the style element for a graphical UI maps "selection" to a "mouse click" event.

In UIML you can specify the following behavior:

- *Assign a value to a part's property*. The value can be any of the following: a constant value, a reference to a constant, the value of property, or the return value of a call.

```
<behavior>
  <rule>
    <condition>
      <event class="ButtonSelected" part-name="b1">
    </condition>
    <action>
      <property part-name="b1" name="color"/>blue</property>

      <property part-name="b2" name="color"/>
        <reference constant-name="green"/>
      </property>

      <property part-name="b2" name="color"/>
        <property part-name="b1" name="color"/>
      </property>

      <property part-name="b3" name="color"/>
        <call name="getColor"/>
      </property>
    </action>
  </rule>
</behavior>
```

Actions are executed sequentially thus the property "color" of part "b2" will finally get the value "blue." The method "getColor" is an abstract method and can be mapped to either a local script or external logic method in the style element. Also, the method in this example does not take any argument (see next case for an example with arguments).

- *Call an external function or method*. The function or method call (see Section 6.7.6) can take any number of arguments. Each argument to the call can be any of the following: a constant value, a reference to a constant, the value of property, or the return value of another call.

```
<behavior>
  <rule>
    <condition>
      <event class="ButtonSelected" part-name="b1">
    </condition>
    <action>
      <call name="storeData">
```

```
        <param name="a1">5</param> <!--arg is constant→
        <param name="a2">          <!--arg is reference to constant-->
          <reference constant-name="green"/>
        </param>
      </call>

      <call name="storeColor">
        <param name="a3">          <!--arg is property's value-->
          <property part-name="b1" name="color"/>
        </param>
      </call>

      <call name="DisplayData">
        <param name="a4">          <!--arg is return value -->
          <call name="getColor"/>
        </param>
        <param name="a5">
          <call name="getParam">
            <param name="a6">5</param>
          </call>
        </param>
      </call>
    </action>
  </rule>
</behavior>
```

- *Fire an event*. An event can be fired from the *action* element. The event must be the last element.

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

  <rule>
    <condition>
      <event class="ButtonSelected" part-name="b2">
    </condition>
    <action>
      <!--executed when b1 or b2 is clicked -->
      <call name="f1"><param name="a1">10</param><call/>
    </action>
  </rule>
</behavior>
```

Assume that both “b1” and “b2” are rendered as buttons and “ButtonSelected” is mapped to the event that is fired when a button is pressed. Whenever the end user clicks button “b1” then the first rule will evaluate to true and fire another event that will simulate the end user pressing “b2”.

Then the renderer will evaluate the condition for all the rules again, and the second rule will evaluate to true and call f1(10).

This feature must be used with care, to avoid creating an infinite loop (e.g., if the second *action* element was "`<event class="ButtonSelected" part-name="b1"/>`" to simulate a click on button b1, instead of the *call* element). Infinite loops are very difficult to detect ahead of time, especially when the UIML code is broken into multiple files.

6.7.1 The *rule* Element

DTD

```
<!ELEMENT rule (condition,action)?>
<!ATTLIST rule
   %SourceAttributes;>
```

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 any elements inside the *action* element are executed sequentially (i.e., property assignment, external function or method call, or event firing). See Section 6.7 for an example and further explanation. Also, it is possible for multiple rules to be satisfied at any time, see Appendix B for an algorithm on rule selection.

6.7.2 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*. The only Boolean expression defined in the DTD for UIML 2.0a is the equality operator, described by the *equal* element. See Section 6.7 for an example and further explanation.

6.7.3 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*, 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* and the *action* is executed. Otherwise the *equal* element has value *false* and the *action* is not executed.

6.7.4 The *event* Element

DTD

```
<!ELEMENT event EMPTY>
<!ATTLIST event
   part-name   NMTOKEN #IMPLIED
   part-class  NMTOKEN #IMPLIED
   name        NMTOKEN #IMPLIED
   class       NMTOKEN #IMPLIED>
```

Description

The *event* element is used in three contexts:

- As the child of a *condition* element. The parent *condition* is satisfied whenever the *event* occurs.
- As the child of an *equal* element. The grand-parent *condition* is satisfied whenever the *event* occurs and contains a value that is equal to the value of the sibling element (i.e., *property*, *constant*, or *reference*).
- As the child of an *action* element. The event is fired.

6.7.5 The *action* Element

DTD

```
<!ELEMENT action ((property|call)*, event?)>
```

Description

The *action* element contains one or more elements that are executed in the order they appear in the UIML document. Each element can be either a *property* element to set a property of an element, a *call* element, which invokes code (e.g., a function or method), or an *event* element, which fires another event. The *event* element, if present, must be the last element inside the *action*. As result of this, you can only fire one *event* within the action *element*. See Section 6.7 for an example and further explanation.

6.7.6 The *call* Element

DTD

```
<!ELEMENT call (param*)>
<!ATTLIST call
   name        NMTOKEN #IMPLIED
   class       NMTOKEN #IMPLIED>
```

Description

The *call* element is an abstraction of any type of invocation of code. The code could be a script defined within the UIML document; otherwise the code is external to the UIML document. For example, the call might represent invocation of a function or method defined in one of the following:

- A scripting language, either within the UIML document or outside the UIML document
- The application logic
- A remote procedure
- Anything else that the rendering engine supports as connection between UIML and external entities, such as databases and directory services.

Each *call* has a *name* and *class* (just like *part*) and a *rendering* property (in the *style* element) that maps it to a *method* (in the *peers*/*logic* element). This *method* can be the execution of a local script, a call to a remote method, or a combination of the two (see Section 7.4.3).

Examples of the *call* element are shown in Section 6.5.1.3 and at the start of Section 6.7.

6.7.7 The *param* Element

DTD

```
<!ELEMENT param (#PCDATA|constant|method|property|reference)?>
<!ATTLIST param
    name   NMTOKEN  #IMPLIED>
```

Description

Describes a single parameter of the call described by the parent *call* 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 *call* element is significant if the *name* attribute is missing from any element (parameters are matched by position) and not significant if the *name* attribute is present in all elements (parameters are matched by name).

7 Peer Components

UIML can be viewed as a *meta*- or extensible language, analogous to XML. UIML does not contain tags specific to a particular UI toolkit (e.g., <*WINDOW*> or <*MENU*>). Instead, it uses a set of generic tags (e.g., <*part*>, <*property*>).

As discussed earlier, UIML captures the elements that are common to any UI: an enumeration of the UI parts, events that occur for those parts, presentation style, content, and interconnection to application logic. The UIML author specifies instance and class names of their own choice for interface parts, events, and methods. These names are mnemonics for the interface implementor. The UIML document specifies a mapping from those names to names that are vocabulary specific to a particular target platform. For example, if the target is Java AWT, the vocabulary might consist of the "*java.awt.*" and "*java.awt.event.*" class names, such as "Frame," "Menu," and "Button." If the target is WML, the vocabulary might be tag names, such as "card," "select," and "input." The vocabulary of target platforms is not a part of UIML. That vocabulary only appears in UIML as the value of attributes in UIML. Thus UIML only needs to be standardized once, and different constituencies of end users can define vocabularies that are suitable for various toolkits independently of UIML. In addition to creating vocabularies for particular toolkits (e.g., Java AWT), a vocabulary for generic classes of toolkits (e.g., mapping to any graphical UI) could be devised. Or new vocabularies can be defined as new devices and UI technologies are created in the future.

To illustrate, the interface implementor might write this in UIML:

```
<part name="Line" class="MenuItemOrIcon">
```

The name MenuItemOrIcon is a mnemonic because the implementor is thinking that this class of parts could be rendered as a menu item or as an icon in two different presentations. But they could have used any name in place of MenuItemOrIcon. In one presentation of the interface using Java AWT, the *style* element may then map this to a java.awt.MenuItem as follows:

```
<style>
  <property part-class="MenuItemOrIcon"
     name="rendering">MenuItem<property/>
  ...
</style>
```

A UIML document also includes an element, called *peers*, that enumerates the mapping from property values to specific tags or objects in the target platform:

```
<peers>
  <presentation name="Java-AWT">
    <component name="MenuItem" maps-to="java.awt.MenuItem">
    ...
  </presentation>
  ...
<peers>
```

To summarize, UIML uses three levels of names for interface parts and events. The first is chosen by the UIML author. The second name is in the *style* element and maps the mnemonic to an abstract widget name (e.g., MenuItem). The second level allows a mapping from one abstract set of names (e.g., BigWindow) to multiple platforms (e.g., MFC or Java Swing) without modifying the rest of the interface description. Finally, the third name in the *peers* element is part of a toolkit-specific vocabulary and maps the abstract widget name to a name of a widget from the target platform (e.g., *java.awt.TextField*).

7.1 The *peers* Element

DTD

```
<!ELEMENT peers (presentation|logic)*>
<!ATTLIST peers
   %SourceAttributes;>
```

Description

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

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

```
<peers>
  <presentation name="Java"
    source="http://uiml.org/toolkits/Java20Swing.ui"/>
  <presentation name="wml"
    source="http://uiml.org/toolkits/wml.ui"/>
  ...
  <logic name="Java"
    source="http://uiml.org/apps/CalendarApp.logic"/>
  <logic name="Scripts"
    source="http://uiml.org/apps/scripts/CalendarApp.logic"/>
</peers>
```

7.2 The *presentation* Element

DTD

```
<!ELEMENT presentation (component*)>
<!ATTLIST presentation
   %SourceAttributes;>
```

Description

The *presentation* element provides information about a single UI toolkit. It describes the different widgets (that are used to render parts) and events (that are generated during the course of application execution).

It is possible to have multiple UIML *presentation* elements for the same UI toolkit. UI designers can create their own UI vocabulary and then map it to the underlying toolkit. See Section 7 for more comments on this perspective.

The *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 *presentation* 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 presentation files for the toolkits that it supports (e.g., Java20Swing.ui in the example above). Alternatively, the *presentation* 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.

Example

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

```
<presentation name="JavaAWT">
  <component name="Button" maps-to="java.awt.Button">
    <attribute name="content">
      <method type="input" maps-to="getLabel"/>
      <method type="output" maps-to="setLabel"/>
    </attribute>

    <attribute name="color">
      <method type="input" maps-to="getColor"/>
    </attribute>
  </component>

  <component name="mouseOver" maps-to="java.awt.event.MouseOver"/>
</presentation>
```

7.3 The *logic* Element

DTD

```
<!ELEMENT logic (component*)>
<!ATTLIST logic
   %SourceAttributes;>
```

Description

The *logic* element describes how the UI 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 UI, or by some combination of these (e.g., to check for validity of data entered by an end user into a UI and then object methods are called), or in other ways.

Thus, the *logic* element acts as the glue between a UI described in UIML and other code. It describes the calling conventions for methods in application logic that the UI 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.

Example

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

```
<logic>

  <component name="back1" maps-to="org.uiml.example.myClass">

    <method name="m1" maps-to="myfunction">
      <param name="p1"/>
      <param name="p2"/>
    </method>

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

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

  </component>

  <component name="back2" maps-to="org.uiml.example.myClass1">
    <method name="m3" maps-to="m9">
      <param name="p4"/>
    </method>
```

```
    </component>

    <component name="S1">

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

      <script type="application/ecmascript"><![CDATA[

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

    </component>

    <component name="S2" maps-to="http://somewhere/vb"/>
      <method name="m101" maps-to="f2">
        <param name="p5"/>
      </method>
    </component>

  </logic>
```

7.4 Common Elements

7.4.1 The *component* Element

DTD

```
<!ELEMENT component (attribute|method)*>
<!ATTLIST component
   %SourceAttributes;
   maps-to    CDATA  #IMPLIED>
```

Description

The *component* element binds a name used in the rendering *property* of a *part* or an *event* element elsewhere in the interface to a component that is part of the presentation toolkit. A *component* can also act as a container for application methods (e.g., a class in an object oriented languages). A *component* may contain *attributes* and *methods*.

The *maps-to* attribute specifies the name that is being bound. This name is used by the renderer to locate the widget, event, or application class at runtime.

7.4.2 The *attribute* Element

DTD

```
<!ELEMENT attribute (method*)>
<!ATTLIST attribute
   name        NMTOKEN  #REQUIRED>
```

Description

The *attribute* element specifies the mapping between a style property and the associated methods that assign or retrieve a value for the property.

Example

```
<peers>
  <presentation>
    <component name="button" maps-to="java.awt.Button">
      <attribute name="Color">
        <method type="input" maps-to="getColor"/>
        <method type="output" maps-to="setColor"/>
      </attribute>
      ...
    </component>
  </presentation>
</peers>

<interface>
  ...
  <style>
    <property name="Color" part-name="bElem">Blue</property>
  </style>
  ...
</interface>
```

7.4.3 The *method* Element

DTD

```
<!ENTITY % TypeOptions
   "type      (input|output|inout|none) 'inout'">

<!ELEMENT method (param*, returns?, script?)>
<!ATTLIST method
   %SourceAttributes;
   maps-to    CDATA   #IMPLIED
   %TypeOptions;>
```

Description

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

The *maps-to* attribute specifies the name that is being bound. The value of *maps-to* points the name of an actual method that can be executed. The method can represent a toolkit method (if it is inside a *presentation* element), an application method (if it is inside a *logic* element), or scripting code (with scripting nested inside the *method* element).

A *method* can be any of four *type*s: *input*, *output*, *inout*, and *none*. The *input* and *output* types are usually associated with get and set methods for attributes. A method of type *input* is expected to return a value that is then used to assign a value to the parent attribute. A method of type *output* is expected to take one argument that is the value of the parent attribute. A method of type *inout* can take any number of arguments and/or return a value. This is the default type and what usually logic methods are. Finally, a method of type *none* usually takes no arguments and does not return anything but does maintenance operations (e.g., initialization, refresh, etc.).

The *method* element supports three different execution models:

1. The method represents a remote (outside the renderer) executable code. This code executes outside the sandbox of the renderer and is treated like a black box. The renderer will package all the parameters, send them to the server executing the code (which can be on the same machine or across the network), and wait for a reply. Here is an example:

```
<component name="Math" maps-to="myClass.Math.CommonRoutines">
  <method name="findMean" maps-to="calcMean">
    <param name="a"/>
    <param name="b"/>
    <returns name="result"/>
  </method>
</component>
```

2. The method represents a local script. This script is embedded inside the method and is executed within the sandbox of the renderer. If the *maps-to* attribute for the component is missing, this means that all the code is local. Here is an example:

```
<component name="Math">
  <method name="findMean" maps-to="calcMean">
    <param name="a"/>
    <param name="b"/>
    <returns name="result"/>
    <script type="text/javascript">
      <![CDATA[
        calcMean(int a, int b) {
          return (a+b)/2;
        }
      ]]>
    </script>
  </method>
</component>
```

3. The method represents a combination of the above. This is useful if you want to do some error checking locally before calling a remote method or manipulate the result after it is returned. *The semantics of how to do this are under revision*.

7.4.4 The *param* Element

DTD

See Section 6.7.7.

Description

Describes a single formal parameter of the function described by the parent *method* 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 *method* element is significant. This order must correspond to the order in which the parameters were originally declared in the external application.

7.4.5 The *returns* Element

DTD

```
<!ELEMENT returns EMPTY>
<!ATTLIST returns
    name   NMTOKEN  #IMPLIED>
```

Description

The parent of a *returns* element is a *method* element. The *method* 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.

7.4.6 The *script* Element

DTD

```
<!ELEMENT script (#PCDATA)>
<!ATTLIST script
   %SourceAttributes;
   type    NMTOKEN #IMPLIED>
```

Description

The *script* element contains a program written in the scripting language identified by the *type* attribute (this is similar to the script element in HTML 4.0).

8 Reusable Interface Components

UIML *templates* enables interface implementors to design parts or to make their entire UI reusable as a component in another UI. For example, many UIs for electronic commerce applications include a credit-card entry form. If such a form is described in UIML as a template, then it can be reused multiple times either within the same UI or across other UIs. This reduces the amount of UIML code needed to develop a UI and also ensures a consistent presentation across enterprise-wide UIs. End users tend to make fewer mistakes and are more efficient when presented with familiar UIs.

8.1 The *template* Element

DTD

```
<!ENTITY % SourceModes "(append|cascade|replace) 'replace'">

<!ENTITY % SourceAttributes
   "name     NMTOKEN      #IMPLIED
    source     CDATA        #IMPLIED
    how      %SourceModes; ">

<!ENTITY % SourceElements
   "(behavior|component|constant|content|interface|logic
   |part|peers|presentation|property|rule|script|structure|style)">

<!ELEMENT template %SourceElements;>
<!ATTLIST template
   name   NMTOKEN #IMPLIED>
```

Description

The *template* element permits several handy shortcuts when writing UIML. It allows

- one fragment of UIML to be inserted in multiple places in a UIML document,
- one UIML document to include a UIML fragment from another document, and
- style and other elements to be cascaded, in a manner analogous to the CSS specification [3].

*Template*s work as follows. Most elements (see *SourceElements* list) can contain the *source* attribute; call such an element *E*. The *source* attribute names a *template* element (either within the same document or in another document). The *template* named must contain an element of the same type as the element *E* (i.e., have the same tag name). The *source* attribute causes the body of the element inside the *template* to be combined with the body of *E*. The rules to control how this combining is done are explained later in Section 8.2.

Simple Example Using the *source* Attribute

```
<uiml>
  <peers>
```

```
    <presentation name="java"
      source="http://uiml.org/toolkits/Java20AWT.ui#Java_AWT"
      how="replace"
    />
  </peers>
  ...
</uiml>
```

The *presentation* element box contains an *source* attribute that names a URL. The effect of this is to insert as the body of the *presentation* element a fragment that is named by the URL. The URL "*http://uiml.org/toolkits/Java20AWT.ui*" in turn contains the following:

```
<uiml>
  <template name="Java_AWT">
    <presentation>
      <component name="Frame" maps-to="java.awt.Frame">
         ...
      </component>
      <component .../>
      ...
    </presentation>
  <template>
</uiml>
```

Note that the "*#Java_AWT*" portion of the URL refers to the *template* element with a *name* attribute of "*Java_AWT*". In the case where only one template exist in the file, then this name can be omitted. If the name is omitted and multiple templates exist, then the first one is used.

8.2 Rules for Templates

In the example of Section 8.1, the element containing the *source* attribute (*E*) has no body. Therefore the body of the fragment is inserted as the child of *E*. However, it is also possible for *E* to have a body. In this case, a set of rules must be specified on how to combine the bodies of the two elements.

For example, consider this UIML file:

```
<interface>
    <structure>
      <part class="label" name="l1">
      ...
    </structure>

    <style source="file://phone.ui#model_508">
      <property name="position" part-name="label">2</property>
    </style>
  <interface>
```

Next suppose that file "phone.ui" contains the following:

```
  <template name="model_508">
    <style>
```

```
      <property name="font_style" part-name="label">bold</property>
      <property name="position"   part-name="label">1</property>
    </style>
  <template>
```

The *style* element in the main document already has a body and both *style* elements have a property named *position*, which one should be used?

A template element is like a separate branch on the UIML tree (think of a DOM tree [11]). A template branch can be joined with the main UIML tree anywhere there is a similar branch (i.e., the first and only child of template must have the same tag name as the element on the UIML tree where the joined is made). The interface implementor has three choices on *how* to combine the *template* element with another element. The first choice is "*replace*." All the children of the element on the main tree that sources the template are deleted, and in their place all the children of the template element are added (see Figure 1). The second choice is "*append*." All the children of the element on the main tree that sources the template are kept, and all the children of the template element are added then to the list too (see Figure 2). In both cases, the names of the children of the template element are appended with the name given to the template before they are added (e.g., name = "templateName.originalName"). The last choice is "*cascade*." This is similar to what happens in CSS. The children from the template are added to the element on the main tree. If there is a conflict (e.g., two elements with the same name), then the element on the main tree is retained (see Figure 3).

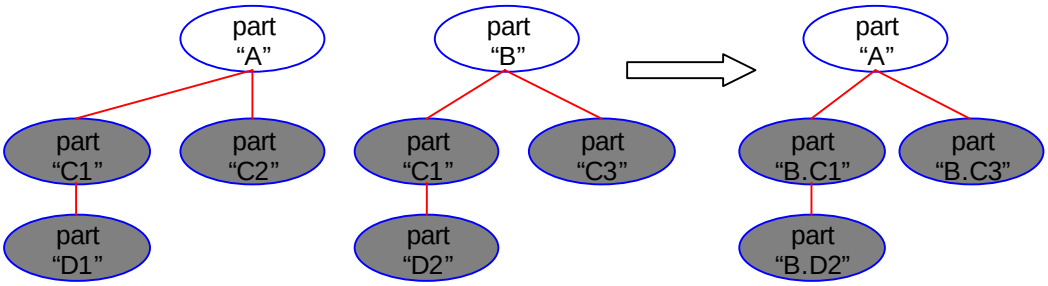


Figure 1: Part "A" sources part "B" using "replace"

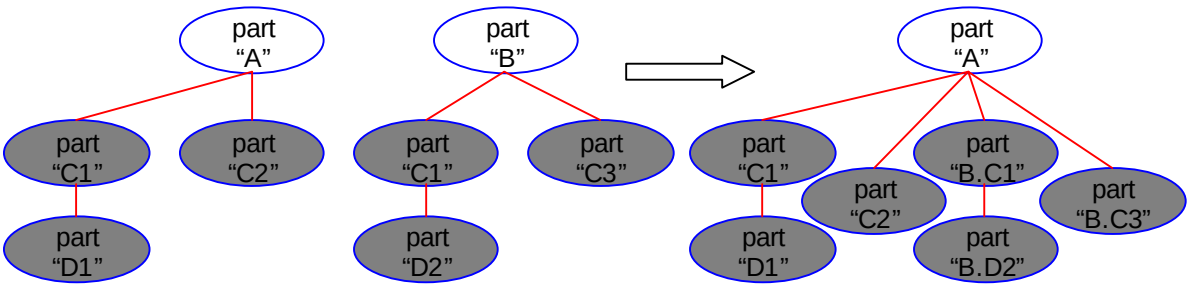


Figure 2: Part "A" sources part "B" using "append"

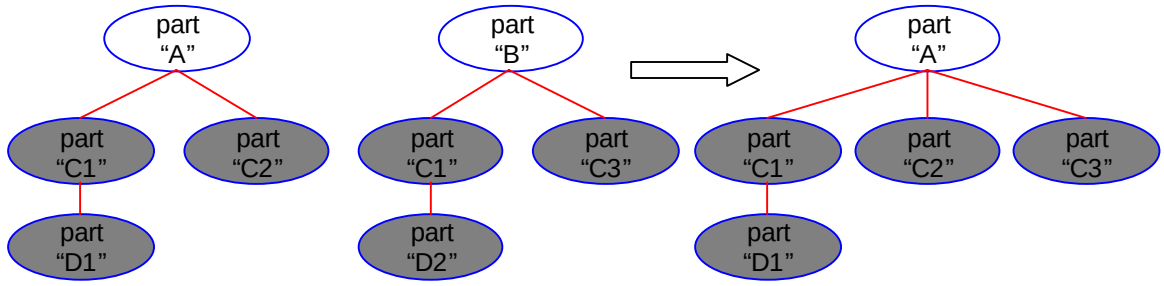


Figure 3: Part “A” sources part “B” using “cascade”

Below are common usage examples of templates that demonstrate the different rules:

8.2.1 Combine Using Replace

Interface parts can be reused by placing them inside a template and then sourcing that template at the appropriate places in the interface structure. The part that sources the template can also include a default implementation of the part inside the template. If the template is for some reason inaccessible (e.g., network problems), then the renderer can ignore the template and still renderer the part. Using “*replace*” as the value for the *how* attribute, the UIML parser will delete the default implementation and add the implementation from the template.

Example

With UIML, one could build a library of reusable interface components, and then include them as needed in a new UIML documents. 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.

```
<template name="DialogBox">
  <part name="TopLevel">
    <part name="CompanyLogo" class="ImageContainer"/>
    <part name="Message" class="Label"/>
    <part name="Accept" class="Button"/>
  </part>
</template>

...
<interface>
  <structure>
    ...
    <part name="FileNotFoundBox" class="DialogBox"
      source="#DialogBox.ui" how="replace">

      <!-- Default implementation -->

    </part>
   ...
```

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

8.2.2 Combine Using Append

Runtime behavior varies significantly from device to device. However, on the same device different platforms are sharing the same behavior. For example, both MS-Windows and X-Windows have events like mouse movement and button clicks. It is therefore convenient, when describing the behavior of similar platforms, to specify the common behavior (rules) in a template and source the template in the behavior for each platform. Using "*append*" as the value for the *how* attribute, the UIML parser will append the list of common behavior rules to the behavior in the main document.

Example

The following example shows how to reuse behavior rules:

```
<template name="GUI_Rules">
  <behavior>
    <rule> <!-- Mouse Movement --> </rule>
    <rule> <!-- Button Click --> </rule>
  </behavior>
</template>

<interface>
  ...
  <behavior name="X-Windows" source="#GUI_Rules" how="append">
    <rule> <!-- Middle Mouse Click --> </rule>
  </behavior>

  <behavior name="MS-Windows" source="#GUI_Rules" how="append">
    <rule> <!-- Window Closing --> </rule>
  </behavior>
</interface>
```

8.2.3 Combine Using Cascade

Style is what dictates how an interface *looks* and *feels*. Many companies want the interfaces to their applications to look the same when presented on the same platform. For example, they want all the "about" dialogs to show their company logo and copyright statement, they want the name of their company to be in a special font and color everywhere it appears, they want the menus to have a special structure (e.g., File, Edit, View, etc…), etc. UIML allows all these and more. All these common style information can be specified in a template and then included in all the interface descriptions. Using "*cascade*" as the value for the *how* attribute, will include the common style information but will also give the ability to customize certain properties. Any local property with the same name will override the property in the template.

Example

The following example demonstrates how to use common style properties and customize them:

```
<template name="Graphical">
  <style>
    <property name="TitleColor" part-class="ADialog">Blue</property>
    <property name="TitleFont" part-class="ADialog">Arial</property>
    <property name="rendering" part-class="ADialog">Dialog</property>
    <property name="content" part-class="ADialog">About: UIT</property>
  </behavior>
</template>

<interface>
  ...
  <style name="MyStyle" source="#Graphical" how="cascade">
    <property name="content" part-name="myAbout"
      >Universal Interface Technologies</property>
  </style>
</interface>
```

8.3 Multiple Inclusions

Elements inside a template can source elements inside other templates. This allows a hierarchical inclusion of UIML templates. This is useful when describing the peer components to a language with an object hierarchy. For example, the Java AWT classes are organized in a hierarchy with each child class inheriting the parent class attribute (thus avoiding redefining the attributes for each class). For example the "Window" inherits its layout attributes from "Container," which inherits its formatting attributes from "Component."

It is an error if a template indirectly sources itself. Note that a template cannot directly source itself. The following example demonstrates how a template can indirectly source itself:

```
<template name="A">
  <part name="a1" source="#B"/>
</template>

<template name="B">
  <part name="b1" source="#C"/>
</template>

<template name="C">
  <part name="c1" source="#A"/>
</template>
```

8.4 The *export* Attribute

DTD

```
<!ENTITY % ExportOptions
   "export     (hidden|optional|required)  'optional'">
```

Description

By default all elements that appear inside a *template* element are visible (can be accessed) from the elements at the location it is included and their data can be optionally modified. UIML allows the encapsulation of what is inside a template by controlling what is visible and what is not with the *export* attribute. Any element inside the template with the *export* attribute set to "hidden" is not visible and thus generates an error if another element outside the template tries to modify it. Also, any element with the *export* attribute set to "required" must be assigned a value before the template can be rendered.

Example

In the following template, a message box has three parts. It required that the content of one part be assigned a value but hides the other two parts. Now, assume that this template is rendered as a dialog window, with a logo image, a label, and an "Ok" button. The UI that sources this template must provide the content for the label (and thus display a custom message), but cannot modify the logo or the "Ok" button.

```
<template name="MyDialog">
  <part name="TopLevel">
    <part name="MyLogo" export="hidden"/>
    <part name="MyMessage"/>
      <property name="content" export="required"/>
    </part>
    <part name="Ok" export="hidden"/>
  </part>
</template>
```

9 Alternative Organizations of a UIML document

Until now, UIML documents shown have followed a rigid format: appearing in the *uiml* element is first the optional *head* element, followed by the *peers* element, and then the *interface* element. Alternative document organizations are possible:

- The *content*, *style*, and *behavior* 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 an end user in large UIML documents.

The DTD in the Appendix A 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.

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

9.1 Normal XML Mechanism

XML allows file inclusion as illustrated below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
    "-//UIT//DTD UIML 2.0a Draft//EN"
    "http://uiml.org/dtds/UIML2_0a.dtd">
<!ENTITY peers       SYSTEM "http://uiml.org/peers.ui">
<!ENTITY parts       SYSTEM "parts.ui">
<!ENTITY style       SYSTEM "style.ui">
<!ENTITY content     SYSTEM "content.ui">
<!ENTITY behavior    SYSTEM "behavior.ui">

<uiml>
  &peers;
  <interface>&parts;&style;&content;&behavior;</interface>
</uiml>
```

9.2 UIML Template Mechanism

Using the *template* element a UIML document can be broken down into multiple pieces (as explained in Section 8.1). The major difference between the normal XML mechanism and UIML templates is that templates provide more control on what information is visible to the main document (see Section 8.4). For example, a template may encapsulate the implementation of a dialog box and export only the content property of the input widget. Also, a smart renderer

may delay the loading and parsing of templates until that part of the code is reached, whereas in the XML mechanism all the inclusions must be done during parsing.

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) for Version 17Jan00*, http://www.uiml.org/specs/UIML17Jan00SpecExamples.pdf.

[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] J. Clark and S. Deach, eds, *Extensible Style Language (XSL)*, W3C Proposed Recommendation, 12 January 2000. http://www.w3.org/TR/xsl.

[7] J. Clark, *XSL Transformations (XSLT)*, W3C Recommendation 16 November 1999, http://www.w3.org/TR/xslt.

[8] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, March 1997.

[9] Wireless Markup Language (WML), Wireless Application Protocol, June 16, 1999, http://www.wapforum.org/

[10] Voice Extensible Markup Language (VoiceXML), VoiceXML Forum, August 17, 1999, http://www.voicexmlforum.org/

[11] Document Object Model (DOM), W3C, http://www.w3.org/

Appendix A. UIML 2.0a Document Type Definition

```
<?xml version="1.0" encoding="ISO-8859-1"?>

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

    Developed by:

        Universal Interface Technologies, Inc.

    Authors:

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

    Usage:

        <?xml version="1.0"?>
        <!DOCTYPE uiml PUBLIC
            "-//UIT//DTD UIML 2.0a Draft//EN"
            "http://uiml.org/dtds/UIML2_0a.dtd">

        <uiml>
          <head> ... </head>
          <peers> ... </peers>
          <template> ... </template>
          <interface> ... </interface>
        </uiml>

    Description:

        This DTD corresponds to the UIML 2.0a specification,
        which may be found at the following URL:

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

    Change History:
        16 Jan 2000 - M Abrams (abrams@uiml.org)
                    - Changed "href" attribute back to old name, "source"
                    - Changed "task" tag back to old name, "call"
        08 Oct 1999 - C Phanouriou (phanouri@universalit.com)
                    - Updated DTD to UIML spec version "2.0a"
                    - Major changes and tag renaming
                    - Added support for templates and peer components
        31 Jul 1999 - A Batongbacal (alanlb@universalit.com)
                    - Updated DTD to UIML spec version "2.0"
        24 Jul 1999 - M Abrams (abrams@uiml.org)
                    - updated to revised language
        15 Jul 1999 - C Phanouriou (phanouri@universalit.com)
                    - first draft
-->

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

<!-- Template related attributes -->

<!ENTITY % SourceModes "(append|cascade|replace) 'replace'">
```

```
<!-- Export options for elements inside a template -->
<!ENTITY % ExportOptions
   "export     (hidden|optional|required)  'optional'">

<!ENTITY % SourceAttributes
   "name      NMTOKEN      #IMPLIED
    source    CDATA        #IMPLIED
    how      %SourceModes;
    %ExportOptions;">

<!-- Elements that can be inside a template -->
<!ENTITY % SourceElements
   "(behavior|component|constant|content|interface|logic
   |part|peers|presentation|property|rule|script|structure|style)">


<!-- Type options for methods -->
<!ENTITY % TypeOptions
   "type      (input|output|inout|none) 'inout'">


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

<!--
    'uiml' is the root element of a UIML document.
-->

<!ELEMENT uiml (head?, peers?, template*, interface?)>

<!--
    The 'head' element is meant to contain metadata about the UIML
    document.  You can specify metadata using the meta tag,
    this is similar to the head/meta from HTML.
-->

<!ELEMENT head (meta)*>
<!ELEMENT meta EMPTY>
<!ATTLIST meta
   name      NMTOKEN  #REQUIRED
   content   CDATA    #REQUIRED>

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

<!ELEMENT peers (presentation|logic)*>
<!ATTLIST peers
   %SourceAttributes;>

<!--
    The 'template' element allows reuse of UIML elements.
    When an element appears inside a template element it can
    sourced by another element with the same tag.
-->

<!ELEMENT template %SourceElements;>
<!ATTLIST template
   name   NMTOKEN #IMPLIED>

<!--
    The 'interface' element describes a user interface in terms of
```

```
    presentation cues, component structure and behavior specifications.
-->

<!ELEMENT interface (structure|style|content|behavior)*>
<!ATTLIST interface
   %SourceAttributes;>

<!-- Peer related elements -->

<!--
    The 'presentation' element specifies the mapping between
    abstract interface parts and platform dependent widgets.
-->

<!ELEMENT presentation (component*)>
<!ATTLIST presentation
   %SourceAttributes;>

<!--
    The 'logic' element specifies the connection between the interface
    and the backend application, including support for scripting.
-->

<!ELEMENT logic (component*)>
<!ATTLIST logic
   %SourceAttributes;>

<!--
    The 'component' element represents components either in the backend
    (e.g., a class in an object oriented language) or components from
    a renderable toolkit (e.g., a widget)
-->
<!ELEMENT component (attribute|method)*>
<!ATTLIST component
   %SourceAttributes;
   maps-to    CDATA  #IMPLIED>

<!--
   The 'attribute' element
-->
<!ELEMENT attribute (method*)>
<!ATTLIST attribute
   name        NMTOKEN  #REQUIRED>

<!--

-->
<!ELEMENT method (param*, returns?, script?)>
<!ATTLIST method
   %SourceAttributes;
   maps-to    CDATA   #IMPLIED
   %TypeOptions;>

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

<!ELEMENT param EMPTY>
<!ATTLIST param
    name   NMTOKEN  #IMPLIED>


<!--
```

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

<!ELEMENT returns EMPTY>
<!ATTLIST returns
    name   NMTOKEN  #IMPLIED>

<!--
    The 'script' element contains data passed to an embedded scripting
    engine. The type specifies the scripting language (see HTML4.0)
-->

<!ELEMENT script (#PCDATA)>
<!ATTLIST script
   %SourceAttributes;
   type    NMTOKEN #IMPLIED>



<!-- Interface related elements -->

<!--
    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
   %SourceAttributes;>

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

<!ELEMENT part (style?, content?, behavior?, part*)>
<!ATTLIST part
   %SourceAttributes;
   class    NMTOKEN  #IMPLIED>

<!--
    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
   %SourceAttributes;>

<!--
    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|constant|property|reference|call)*>
```

```
<!ATTLIST property
   %SourceAttributes;
   part-name      NMTOKEN #IMPLIED
   part-class     NMTOKEN #IMPLIED
   event-name     NMTOKEN #IMPLIED
   event-class    NMTOKEN #IMPLIED
   call-name      NMTOKEN #IMPLIED
   call-class     NMTOKEN #IMPLIED>


<!--
    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
   constant-name  NMTOKEN #REQUIRED>

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

<!ELEMENT content (constant*)>
<!ATTLIST content
   %SourceAttributes;>

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

<!ELEMENT constant (#PCDATA|constant)*>
<!ATTLIST constant
   %SourceAttributes;>

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

<!ELEMENT behavior (rule*)>
<!ATTLIST behavior
   %SourceAttributes;>


<!ELEMENT rule (condition,action)?>
<!ATTLIST rule
   %SourceAttributes;>

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

<!ELEMENT action ((property|call)*, event?)>
```

```
<!ELEMENT call (param*)>
<!ATTLIST call
   name        NMTOKEN #IMPLIED
   class       NMTOKEN #IMPLIED>

<!ELEMENT event EMPTY>
<!ATTLIST event
   part-name   NMTOKEN #IMPLIED
   part-class  NMTOKEN #IMPLIED
   name        NMTOKEN #IMPLIED
   class       NMTOKEN #IMPLIED>


<!-- Syntax under revision -->
<!ELEMENT system EMPTY>
```

Appendix B. Behavior Rule Selection Algorithm

The *behavior* element contains one or more *rule* elements. Sometimes the *condition* for more than one *rule* may be satisfied at the same time. A UIML rendering engine must render UIML in such a way that *rule* elements are selected for execution according to the algorithm below. For each *rule* element selected, the elements inside the *action* element are executed sequentially.

```
// Scan each rule element sequentially
// (as they appear in the UIML file)
foreach (rule inside behavior) do

  // Evaluate the condition of the rule
  if eval(rule.condition) == TRUE then

    // A condition is found that evaluates to true
    // Scan action elements sequentially
    foreach (element inside action) do

      // If the element is a property
      if (element instanceof property) then
        do property assignment

      // If the element is a method
      else if (element instanceof method) then
        do method call

      // If the element is an event
      // This must be the last element in the action
      else if (element instanceof method) then
        do event firing
        RETURN

    end foreach

    // End when a rule is found and its actions are executed
    RETURN
  endif
end foreach
```