User Interface Markup Language (UIML) Specification

Document Version 08 February 2002

** DRAFT **

Language Version 3.0

Send technical comments to uiml-editor@uiml.org.

Creating harmony between people and computers

User Interface Markup Language (UIML) Specification
Version: 3.0
Status: First draft
Release: 02/08/2002
Editors: Marc Abrams, abrams@uiml.org
Jim Helms, jhelms@harmonia.com

Harmonia, Inc.
P.O. Box 11282
Blacksburg, VA 24062-1282
U.S.A.
+1.540.951.5900

Latest UIML specification:

- HTML: http://www.uiml.org/specs/

This version:

- HTML: http://www.uiml.org/docs/uiml30-15Aug01.html
- PDF: http://www.uiml.org/docs/uiml30-15Aug01.pdf

Previous versions:

- UIML version 2.0a (January 17, 2000)
  - http://www.uiml.org/specs/docs/uiml20-17Jan00.pdf
- UIML version 2.0 (August 8, 1999)
  - http://www.uiml.org/specs/docs/uiml20-990801.pdf
  - http://www.uiml.org/specs/docs/uiml20-990801.html
- UIML version 1.0 (December 1997)
  - http://www.uiml.org/specs/docs/uiml_v10_ref.PDF

Abstract

UIML provides a highly device-independent method to describe a user interface. UIML factors any user interface description into six orthogonal pieces, answering six questions:

1. What are the parts comprising the UI?
2. What is the presentation (look/feel/sound) used for the parts?
3. What is the content (e.g., text, images, sounds) used in the UI?
4. What is the behavior of the UI (e.g., when someone clicks or says something)?
5. What is the mapping of the parts to UI controls in some toolkit (e.g., Java Swing classes or HTML tags)?
6. What is the API of the business logic that the UI is connected to?

UIML is a meta-language, which is augmented by a vocabulary of user interface parts, properties, and events defined outside this specification. In this way, UIML is independent of user interface metaphors (e.g., "graphical user interface", "dialogs").

UIML version 3 is a refinement of the previous versions of UIML, which were developed starting in 1997. Version 3 differs from UIML 2 in these ways:

- Support for dynamic user interfaces: A user interface is viewed as a virtual tree, whose initial content is specified by the <structure> element, and which can change during the lifetime of the user interface.
- Support for multimodal user interfaces: A user interface can contain multiple <interface> elements that are simultaneously used and kept synchronized. For example, a user interface might offer a voice UI and a screen-based UI (e.g., on a mobile phone), and the user can at any time use either mode of interaction (voice or display/keypad).
- Refinement of many language constructs, based on implementation experience with UIML 2. These items are described by proposals for changes to UIML 2, posted in the past on www.uiml.org.

It is Harmonia's intent that UIML 3.0 can be freely implemented by anyone. UIML 3.0 may be implemented without any license cost due to Harmonia, Inc.

Contents

1 Introduction to UIML 3.0

UI Markup Language 3 (UIML3) is a declarative, XML-compliant meta-language for describing user interfaces (UIs) that refines the UIML2 specification, released in 2000 [7]. The original UIML specification was released in 1998[5]. The philosophy used in developing the UIML specifications has been to design and gain experience implementing the language for a variety of devices to insure that the concepts in the language are sound and that the language is suited to real-world applications.

Among the motivations of UIML are the following:

- allow individuals to implement UIs for any device without learning languages and application programming interfaces (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, and
- allow extension to support UI technologies that are invented in the future.

The design objective of the UIML is to provide a <u>canonical</u> <u>representation</u> of <u>any</u> UI suitable for <u>mapping to</u> existing languages.

UIML is no more and no less than this. UIML provides a puzzle piece to be used in conjunction with other technologies, including UI design methodologies, design languages, authoring tools, transformation algorithms, and existing languages and standards (especially W3C specifications). UIML is not a silver bullet that replaces human decisions needed to create UIs.

Why is a canonical representation useful? Today, UIs are built using a variety of languages: XML variants (e.g., HTML, XHTML, VoiceXML,), JavaScript, Java, C++, etc. Each language differs in its syntax and its abstractions. For example, the syntax in HTML 4.0 to represent a button is "<button>", and in Java Swing "JButton b = new JButton;". The work on UIML asks the fundamental question, "Do we inherently need different syntaxes, or can one common syntax be used?" The benefit of using a single syntax is analogous to the benefit of XML: Software tools can be created for a single syntax (UIML), yet process UIs destined for any existing language. For example, a tool to author UIs can store the design in UIML, and then map UIML to target languages in use today (e.g., HTML, Java) or invented in the future. Progress in the field of UI design can move faster, because everyone can build tools that either map interface designs into UIML or map UIML out to existing languages. Tools can then be snapped together using UIML as a standard interchange language.

There is a second benefit of a canonical UI description. By using a single syntax to represent any UI, an interface is in a very malleable form. For example, one technique gaining popularity in the human computer interface community is transformation. With a canonical representation for any UI, someone that designs a transform algorithm can simply implement the algorithm to transform an input UIML document to a new output UIML document. Compare this approach to implementing the same transform algorithm only to transform HTML documents, then reimplementing the transform algorithm to only transform C++ interfaces, and so on.

In any language design, there is a fundamental tradeoff between creating something general versus special-purpose. UIML is for general-purpose use by people that implement UIs and people that build tools for authoring UIs. It is envisioned that UIML will be used with other languages with a more focused purpose, such as UI design languages. Ultimately most people may never write UIML directly – they may instead use a particular design language suited to a certain design methodology, and then use tools to transform the design into a UIML representation that is then mapped to various XML or programming languages.

Four key concepts underlie UIML:

> 1. <u>UIML is a meta-language</u>. To understand this, consider XML. XML does not define tags, such as <p>. Instead, one must add to XML a specification of the legal tags and their attributes, for example by creating a document type definition (DTD). Therefore the XML specification does not need to be modified as new tag sets are created, and a set of tools can be created to process XML independent of the tag sets that are used.
>
> UIML, while it is an XML schema, defines a small set of powerful tags, such as <part> to describe a part of a UI, or <property> to describe a property of a UI part. UIML tags are independent of any UI metaphor (e.g., graphical UIs), target platform (e.g., PC, phone), or target language to which UIML will be mapped (e.g., VoiceXML, HTML).
>
> To use UIML, one must add a *toolkit vocabulary* (roughly analogous to adding a DTD to an XML document). The vocabulary specifies a set of *classes* of parts, and *properties* of the classes. Different groups of people can define different vocabularies, depending on their needs. One group might define a vocabulary whose classes have a 1-to-1 correspondence to UI widgets in a particular target language (i.e., the classes might match those in the Java Swing API). Another group might define a vocabulary whose classes match abstractions used by a UI designer (e.g., *Title, Abstract, BodyText* for UIs to documents). UIML can be standardized once and tools can be developed for UIML, independently from the development of vocabularies.
>
> 2. <u>UIML "factors out" or separates the elements of a UI.</u> The design of UIML started with a clean sheet of paper and the question: what are the fundamental elements needed to describe any man-machine interaction. The separation in UIML identifies what parts comprise the UI, the presentation style for each part as a list of <property> elements, the content of each part (e.g., text, sounds, images) and binding of content to external resources (e.g., XML resources, or method calls in external objects), the behavior of parts

when a user interacts with the interface as a set of rules with conditions and actions, the connection of the UI to the outside world (e.g., to business logic), and the definition of the vocabulary of part classes. For a comparison of the separation in UIML to existing UI models, such as the Model View Controller, refer to Phanouriou [5].

3. <u>UIML views the structure of a UI, logically, as a tree of UI parts that changes over the lifetime of the interface.</u> There is an initial tree of parts, which is the UI initially presented to a user when the interface starts its lifetime. During the lifetime of the interface, the tree of parts may dynamically change shape by adding or deleting parts. For example, opening a new window containing buttons and labels in a graphical interface may correspond to adding a sub-tree of parts to the UIML tree. UIML provides elements to describe the initial tree structure (<structure>) and to dynamically modify the structure (<restructure>).

4. <u>UIML allows UI parts and part-trees to be packaged in templates.</u> Templates may then be reused in various interface designs. This provides a first class notion of reuse within UIML, which is missing from other XML UI languages, such as HTML and WML.

Due to these concepts, UIML is particularly useful for creating multiplatform, multimodal, multilingual, and dynamic UIs. Here are some examples:

- To create multiplatform UIs, one uses concept 1 to create a vocabulary of part classes (e.g., defining class *Button*), and then uses concept 2 to separately define the vocabulary by specifying a mapping of the classes to target languages (e.g., mapping UIML part class *Button* to class *java.awt.Button* for Java and to tag *<button>* for HTML 4.0). One can create a highly device-independent UI by creating a generic vocabulary that tries to eliminate bias toward particular UI metaphors and devices. (By "device" we mean PCs, various information appliances [e.g., handheld computers, desktop phones, cellular or PCS phones], or any other machine with which a human can interact.) In addition, because UIML describes the interface behavior as rules whose actions are applied to parts (concept 2), the rules can be mapped to code in the target languages (e.g., to lines of Java code or JavaScript code).

- To create multimodal UIs, one creates a multiplatform UI, and then annotates each part with its mode (e.g., which target platforms use that part), and the behavior section from concept 2 is used to keep the interface modes synchronized. For example, one might define a UIML part class *Prompt*, the mapping of *Prompt* parts to VoiceXML and HTML, and the behavior that synchronizes a VoiceXML and HTML UI to simultaneously prompt the user for input.

- To create multilingual UIs, one uses concept 2 to separate the content in each language from the rest of the UI.

- To create dynamic UIs – such as a Web page containing a table whose size and content comes from a database call made each time the page is loaded – can be achieved by binding the separated content to calls to methods in, say, Java beans (concept 2).

> Alternately, a behavior rule (concept 2) can specify that the page be restructured to dynamically add a table to the tree of interface parts (concept 3).

For further discussion of the motivation for and uses of UIML, please see Abrams et al [3] and [4].

1.1 UIML, an Open Specification

It is Harmonia's intent that UIML 3.0 can be freely implemented by anyone. UIML 3.0 may be implemented without any license cost due to Harmonia, Inc.

1.2 Relationship of UIML to XML, XSL, XForms, and CSS

UIML is compliant with the W3C XML 1.0 specification [1]. Appendix A contains the UIML 3.0 DTD.

When UIML is rendered to HTML, CSS style sheets or XSL formatting objects [8] can be used with the resultant HTML. In addition, XSLT [9] can be used to transform UIML to other XML-compliant markup languages. In some cases, Xforms Models [13] can be used to represent complex data objects being passed to or from the user interface.

1.3 Purpose of This Document

This document serves as the official language reference for UIML 3.0. It describes the syntax of the elements and their attributes, the structure of UIML documents, and usage examples. It also gives pointers to other reference documentation that may be helpful when developing applications using UIML.

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 http://www.uiml.org/discussion. A submission to a standards organization will occur after comments are received and this draft specification is finalized.

This document may be distributed freely, as long as all text and legal notices remain intact.

1.4 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.

Canonical Representation: A UI metaphor-independent enumeration of the parts, behaviors, content, and style of a user interface.

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. Examples include business logic (e.g., in the form of Enterprise Java Beans, Common Object Request Broker Architecture [CORBA] objects), databases, and any type of service that might run on a server (e.g., an a Lightweight Directory Access Protocol [LDAP] server). In a three-tier system architecture model, the application logic is the middle layer that mediates communication between the database and presentation layers.

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 it 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 [11] 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 [8], 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 that 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 2.1.

Render Time: This is the period of time before the interface is displayed to the user. During this time the rendering engine interprets the UIML and may make calls to the backend as specified in the UIML.

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.

Method: This spec uses the term "method" to generically represent any code entity (that uses a language other than UIML) that a rendering engine can invoke, and which may optionally return a value. Examples include functions, procedures, and methods in an object-oriented language, database queries, and directory accesses.

User Interface Lifetime: The period of time beginning when the interface is first displayed to the user and concluding when the interface is closed, exited, or otherwise terminated.

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 [10].

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.

1.5 Note about Harmonia, Inc. Rendering Engines

Rendering engines developed as part of LiquidUI™ versions 1.0d and 1.1a by Harmonia, Inc. were written for and have evolved from the UIML 2.0 specification. As such, the examples contained herein may or may not render correctly when using these versions.

2 Structure of a UIML Document

This section gives several examples of UIML documents. Further examples are available at [17].

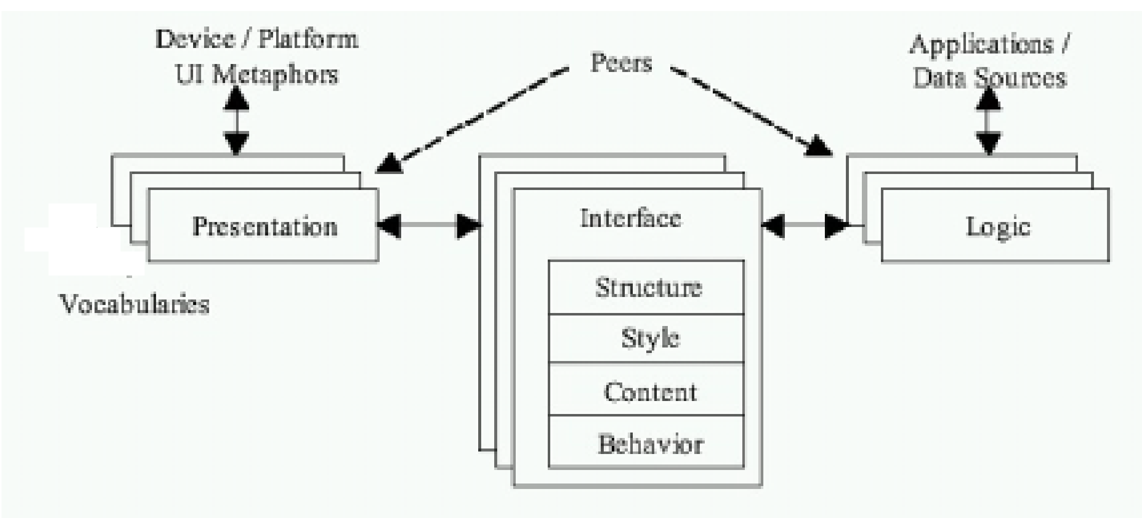
2.1 Overview

In UIML version 3.0, 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 interface 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. Some interface parts can receive input from the end-user. This is usually achieved through the use of interface artifacts like a scrollable selection list, pressable button, etc. Since the artifacts vary from device to device, the actual mapping (rendering) between an interface part and the associated artifact (widget) is done using either a *<presentation>* element or a special *<property>* element in the *<style>* element.

Defining a user interface in UIML answers the following five questions.

- What parts comprise the UI?
- What presentation style for each part? (rendering, font size, color, …)
- What content for each part? (text, sounds, image, …)
- What behavior do parts have?
- How to connect to outside world?(business logic, data sources, UI toolkit)

UIML is modeled by the Meta-Interface Model [5] pictured below.



2.1.1 Dynamic Interfaces Through a Virtual UI Tree

The interface portion of a UIML document defines a virtual tree of parts with their associated content, behavior, and style. This virtual tree is then rendered according to the specification of the presentation component and communicates with application logic via the logic definitions.

The virtual tree can be modified dynamically by repeating, deleting, replacing, or merging sub-trees and tree fragments in the main tree. This allows for a canonical description of a user interface through the lifetime of it interaction with the user.

2.1.2 Interface Behavior

UIML describes in a *<behavior>* element the actions that occur when an end-user interacts with a UI. The *<behavior>* element is built 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, *events* are triggered which cause some action to execute. In this version of the UIML specification, each condition is evaluated only when an event associated with the condition occurs. This simplifies the rendering of UIML by compilation to other languages.

Each action can do one or more of the following: (1) change a property of some part in the UI (2) invoke a function in a scripting language, (3) invoke a function or method from a backend object or (4) throw an event. 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 allow a developer to implement application logic that modifies a UI programmatically.

2.1.3 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 language evolves.

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 a simple set of generic tags. The UIML syntax also defines tag attributes 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 a vocabulary of toolkit components (to which each *<part>* element in a UIML document is mapped) and their property names

2.1.4 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 "-//Harmonia//DTD UIML 3.0 Draft//EN"
    "http://uiml.org/dtds/UIML3_0a.dtd">

<uiml>
  <interface>
    <structure>
      <part id="TopHello">
        <part id="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***>** element.

A VoiceXML rendering engine [12] using the above UIML code and the following *<peers>* 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 [11] Rendering engine using the above UIML code and the following <*peers*>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>
```

2.2 UIML Document Structure

A typical UIML 3.0 document is composed of these two parts:

1. A prolog identifying the XML language version, encoding, and the location of the UIML 3.0 document type definition (DTD):

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
  "-//Harmonia//DTD UIML 3.0 Draft//EN"
http://uiml.org/dtds/UIML3_0a.dtd">
```

Note: This prolog should appear in the beginning of every UIML file (even files containing only UIML templates [see Section 7.1]), 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/UIML3_0a.dtd'> ... </uiml>
```

   See Section 4.1 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 4.2.

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

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

Section 7.1 discusses the <*template*> element, and its use in building libraries of reusable UI components.

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

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

Section 5.3 discusses the <*interface*> element.

d) An optional element that describes the mapping of classes and names used in the UIML document to a UI toolkit and to the application logic:

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

Discussion of the <*peers*> element is deferred until Section 6.1, because the <*peers*> element normally just sources an external file.

White spaces, blank 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:

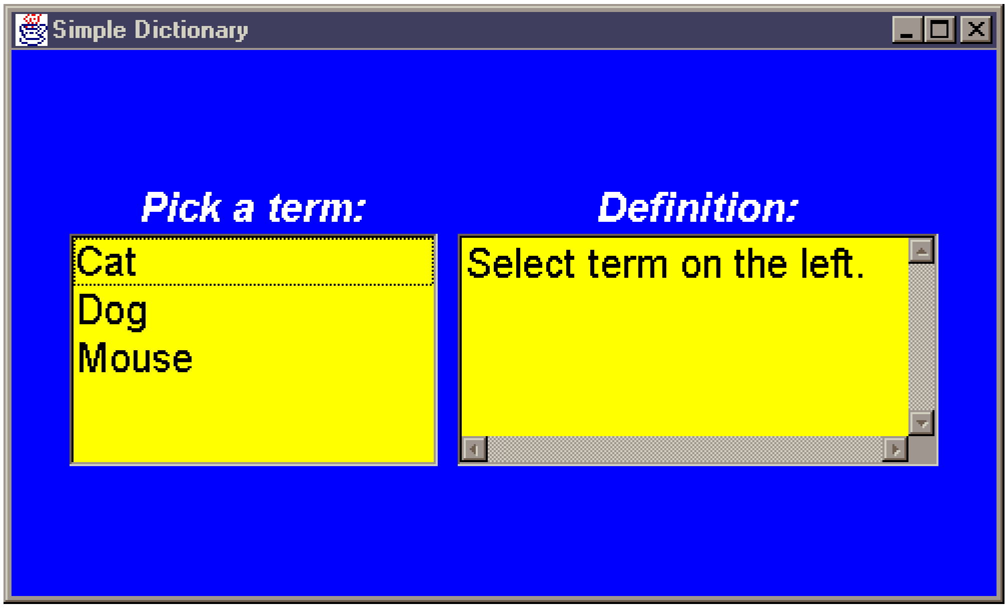
```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
    "-//Harmonia//DTD UIML 3.0 Draft//EN" "http://uiml.org/dtds/UIML3_0a.dtd">

<uiml xmlns='http://uiml.org/dtds/UIML3_0a.dtd'>
   <head>            ... </head>
   <template>        ... </template>
   <interface>       ... </interface>
   <peers>           ... </peers>
</uiml>
```

2.2.1 Second UIML Example

This section contains a simple example of a UIML document, which includes event handling.

The example below displays a single window representing a dictionary. The dictionary contains a list box in which an end-user can select a term (i.e. *Cat*, *Dog*, *Mouse*). The dictionary also contains a text area in which the definition of the currently selected term is displayed. For example, if *Cat* is selected on the left, a definition of a cat replaces the string “Select term on the left.” The style element in the UIML document that describes this interface uses the properties found in the Java AWT and Swing components.



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

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
    "-//Harmonia//DTD UIML 3.0a Draft//EN"
     http://uiml.org/dtds/UIML3_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 base="Java_1.3_Harmonia_1.0"/>
  </peers>
```

```
  <interface>
```

```
    <structure>
      <part   class="JFrame"   id="JFrame">
        <part class="JLabel"   id="TermLabel"/>
        <part class="List"     id="TermList" />
        <part class="JLabel"   id="DefnLabel"/>
        <part class="TextArea" id="DefnArea"/>
      </part>
    </structure>
```

```
    <style>
      <property  part-name="JFrame"    name="layout"
                                       >java.awt.GridBagLayout</property>
      <property  part-name="JFrame"    name="background" >blue</property>
      <property  part-name="JFrame"    name="location"   >100,100</property>
      <property  part-name="JFrame"    name="size"       >500,300</property>
      <property  part-name="JFrame"    name="title"      >Simple Dictionary</property>

      <property  part-class="JLabel"   name="foreground" >white</property>
      <property  part-class="JLabel"   name="gridx"      >RELATIVE</property>
      <property  part-class="JLabel"   name="gridy"      >RELATIVE</property>
      <property  part-class="JLabel"   name="font"
                                       >Helvetica-bolditalic-20</property>

      <property  part-name="TermLabel"  name="text"       >Pick a term:</property>

      <property  part-name="DefnLabel" name="text"       >Definition:</property>
      <property  part-name="DefnLabel" name="gridx"      >1</property>
      <property  part-name="DefnLabel" name="gridy"      >0</property>
      <property  part-name="DefnLabel" name="insets"     >0,10,0,0</property>

      <property  part-name="TermList"  name="background" >yellow</property>
      <property  part-name="TermList"  name="gridx"      >0</property>
      <property  part-name="TermList"  name="gridy"      >RELATIVE</property>
      <property  part-name="TermList"  name="fill"       >BOTH</property>
      <property  part-name="TermList"  name="font"       >Helvetica-20</property>
      <property  part-name="TermList"  name="content">
        <constant model="list">
          <constant id="Cat"   value="Cat"/>
          <constant id="Dog"   value="Dog"/>
          <constant id="Mouse" value="Mouse"/>
        </constant>
      </property>
```

```
      <property  part-name="DefnArea"  name="background" >yellow</property>
      <property  part-name="DefnArea"  name="gridx"      >1</property>
      <property  part-name="DefnArea"  name="gridy"      >RELATIVE</property>
      <property  part-name="DefnArea"  name="text"       >Select term on the 
left.</property>
      <property  part-name="DefnArea"  name="columns"    >20</property>
      <property  part-name="DefnArea"  name="rows"       >4</property>
      <property  part-name="DefnArea"  name="editable"   >false</property>
      <property  part-name="DefnArea"  name="insets"     >0,10,0,0</property>
      <property  part-name="DefnArea"  name="font"       >Helvetica-20</property>
</style>
```

```
    <behavior>

      <rule>
        <condition>
          <op name=”&&”>
            <event part-name="TermList" class="ItemListener.itemStateChanged"/>
            <op name=”==”>
              <property event-class=”ItemListener.itemStateChanged” name=”item”/>
              <constant value="0"/>
            </op>
          </op>
        </condition>
        <action>
          <property part-name="DefnArea" name="text"
            >Carnivourous, domesticated mammal that's fond of rats and mice</property>
        </action>
      </rule>

      <rule>
        <condition>
          <op name=”&&”>
            <event part-name="TermList" class="ItemListener.itemStateChanged"/>
            <op name=”==”>
              <property event-class=”ItemListener.itemStateChanged” name=”item”/>
              <constant value="1"/>
            </op>
          </op>
        </condition>
        <action>
          <property part-name="DefnArea" name="text"
            >Domestic animal related to a wolf that's fond of chasing cats</property>
        </action>
      </rule>

      <rule>
        <condition>
          <op name=”&&”>
            <event part-name="TermList" class="ItemListener.itemStateChanged"/>
            <op name=”==”>
              <property event-class=”ItemListener.itemStateChanged” name=”item”/>
              <constant value="2"/>
            </op>
          </op>
        </condition>
        <action>
          <property part-name="DefnArea" name="text"
            >Small rodent often seen running away from a cat</property>
        </action>
      </rule>

    </behavior>
```

```
  </interface>
</uiml>
```

The UIML document above starts with <?xml …>,<!DOCTYPE …>, and the <*uiml*> tag, which start every UIML document.

Next comes the <*peers*> element, enclosed by the first box. The <*presentation*> element inside *peers* contains *base="Java_1.3_Harmonia_1.0"*, which means that this UIML document uses the vocabulary defined in http://uiml.org/toolkits/Java_1.3_Harmonia_1.0.uiml. The vocabulary defines, among other things, the legal class names for <*part*> elements, the legal property values for each part class. Setting the base attribute to *Java_1.3_Harmonia_1.0* implies that most of the Java AWT and Swing class names (e.g., *JButton, JLabel*) can be used as part names in UIML, and names similar to AWT and Swing property method names can be used as UIML property names (e.g., *foreground* for a *JLabel*). Element `<presentation base="Java_1.3_Harmonia_1.0">` specifies that any renderer that implements vocabulary *Java_1.3_Harmonia_1.0* can render this UIML document. (In general, If a UIML document contains `<presentation base="x">` then that document can be rendered by any rendering engine that implements vocabulary "uiml.org/toolkits/x.uiml".)

The <*interface*> element comes next. The first element inside *interface* is *structure*, which appears in the second box. The <*structure*> element in this example describes a UI consisting of five parts. The first is named *JFrame*, and contains the other four parts, named *TermLabel, TermList*, *DefnLabel,* and *DefnArea.* The class names used to make the interface are *JFrame, JLabel, List, JLabel,* and *TextArea.* http://uiml.org/toolkits/Java_1.3_Harmonia_1.0.uiml defines these names as corresponding to the Java AWT and Swing classes javax.swing.JFrame, javax.swing.JLabel, java.awt.List, javax.swing.JLabel, and java.awt.TextArea. So whenever these names are used as the *class* for a <*part*> element, the default value of the *rendering* property for the parts defaults to the corresponding AWT or Swing class. Thus the <*part*> with *id* "DefnArea" will be rendered as a java.awt.TextArea.

The <*style*> element comes next. The <*style*> element in this example sets five properties on the part named *JFrame*: the layout manager to GridBagLayout, the frame background to blue, the upper left corner of the frame to appear 100 pixels down and 100 pixels to the right of the upper left corner of the screen, the frame dimensions to 500 pixels wide and 300 pixels high, and the frame title (the text that appears in the band at the top of the frame) to "Simple Dictionary".

The next four properties apply to all parts whose class name is *"JLabel"*. There are two such parts: *TermLabel* and *DefnLabel.* The foreground color, two properties associated with GridBagLayout (*gridx* and *gridy*), and the font are the same for all labels. The remaining properties in the <*style*> element are properties of individual *part*s in the interface.

The next box contains the <*behavior*> element. This consists of a sequence of rules, each containing a condition and an action. Each condition holds true when some event occurs. The <*event*> element in each condition names a Java event through its *class* attribute. Whoever defines the vocabulary for a UI toolkit defines the class names used for events. The vocabulary defined in *Java_1.3_Harmonia_1.0* uses the following convention for choosing UIML event class names: method names in Java AWT and Swing listener classes are used as UIML event names. For example, clicking items in an AWT List are handled by an instance of java.awt.event.ItemListener. Hence *Java_1.3_Harmonia_1.0* defines *ItemListener*'s methods, such as *itemStateChanged*, as UIML event class names.

Returning to the UIML document above, the first *condition* holds true when someone clicks on term number zero in the list or when someone clicks on *Cat*. Let's examine the first *condition* in detail:

```
<condition>
  <op name="&&">
    <event part-name="TermList" class="ItemListener.itemStateChanged"/>
    <op name="==">
      <property event-class="ItemListener.itemStateChanged" name="item"/>
      <constant value="0"/>
    </op>
  </op>
</condition>
```

The <*condition*> has as its child an <*op*> (for operand) element. The *name* attribute of <*op*> is "&&", or logical AND, which means that the <*op*> holds true when both of its children are true. The first child of <*op* name="&&"> is <*event*>, so this child is true when the event ItemListener.itemStateChange fires for the part named TermList. (Vocabulary *Java_1.3_Harmonia_1.0* uses as UIML event names *L.m*, where *L* is a Java listener [e.g., ItemListener], and *m* is a method in the listener [e.g., itemStateChange()]. See 6.2.2.3 for further information.) Put another way, the <*event*> is true when a user clicks on a term *Cat*, *Dog*, or *Mouse*. The second child of <*op* name="&&"> is another <*op*> element, this time <*op* name="==">. This inner <*op*> element is true when its children are equal. The first child is <*property*>, which evaluates to the property called *item* of an *ItemListener.itemStateChanged* event. (Vocabulary *Java_1.3_Harmonia_1.0* uses the method names in each Java event as property names for UIML events. The Java *itemStateChanged* method takes an *ItemEvent* as an argument, which in turn has a method named *getItem.* (Method getItem returns the index number of the item selected in the list, either 0, 1, or 2 in our Dictionary example.) Hence, the UIML event has a property named *item*. Therefore the inner <*op*> is true when the item number selected equals zero. In summary, the entire condition is true when the user clicks on an animal name, and the item clicked on is item zero, or *Cat*.

The action associated with clicking *Cat* is to change the content of part *DefnArea* to display the text string "Carnivourous, domesticated mammal…" – in other words, the definition of a cat pops up in the text area on the right of the UI.

Similar condition-action rules are given for *Dog* and *Mouse*.

2.3 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/UIML3_0a.dtd*". Note that this URI has not been activated yet.

Example

Here is an example that combines UIML and HTML vocabularies:

```
<uiml:uiml xmlns:uiml='http://uiml.org/dtds/UIML3_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/UIML3_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>
```

To learn more about XML name-spacing, refer to http://www.w3.org/2000/xmlns/ .

2.4 UIML Mime Type

The following mime type should be used for UIML documents:

```
text/uiml
```

Furthermore, the mime type could include the value of the *base* attribute in the *<presentation>* element (see Section 6.2), which identifies that any recipient software that processes the UIML document must implement the vocabulary identified in the *base* attribute. Here are some examples:

```
text/uiml/Java_1.3_Harmonia_1.0
```

```
text/uiml/Html_4.01frameset_Harmonia_0.1
```

3 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 3.0 DTD is given in 0.

Element	Purpose	Page
<action>	Perform an action if the condition of a rule is true	58
<behavior>	Specify rules for runtime behavior	49
<by-default>	Set of actions to be executed when <op> conditional is undefined	69
<call>	Call a function or method external to UIML document	58
<condition>	Specify a condition for a rule	54
<constant>	Define a constant value	47
<content>	Specify a set of constant values	45
<d-class>	Maps class names that can be used for parts and events to a UI toolkit	87
<d-component>	Maps a name used in a <call> element to application logic external to UIML document	87
<d-method>	Maps a method to a callable method or function in the API of the application logic	89
<d-param>	Defines a single formal parameter to a <d-method>	90
<d-property>	Maps a property name, for parts or events, to methods in a UI toolkit that get and set the property's value	88
<equal>	Compares the property value of an event with another value	54
<event>	Specify a UI or system event to be thrown or caught	55
<head>	A container for metadata information	28
<interface>	A container for all UIML elements that describe a user interface	32
<iterator>	A tag controlling the number of times a virtual tree contained in a *<repeat>* element is replicated.	63
<logic>	Describes mappings of names and classes used in *<call>* elements to application logic	85
<meta>	Define a piece of metadata as a name/value pair	29
<op>	Define a conditional expression or operation	56
<param>	Actual parameter used in a *<call>* element	69
<part>	Specifies a single abstract part of the user interface	34
<peers>	Describes mapping from class, property, event, and names used in <call> elements to identifiers in a UI toolkit and the application logic	71
<presentation>	Contains mappings of part and event classes, property names, and event names to a UI toolkit	72
<property>	Specify or retrieve a property for a *<part>* element or a class of <part> elements	35
<reference>	Reference to a constant or resource external to the UIML document	48
<repeat>	Groups parts which are repeated one of more times in a user interface	62

<*restructure*>	Modify the current virtual tree of parts	64
<*rule*>	A condition/action pair	54
<*script*>	A container for executable script code	91
<*structure*>	Defines the initial virtual tree organization (physical or temporal) of the parts comprising a user interface	32
<*style*>	Specify a set of style properties for the interface	35
<*template*>	A container for reusing <uiml> elements	92
<*uiml*>	Root element in a UIML document	28
<*when-true*>	Set of actions to execute when <op> condition is true	68
<*when-false*>	Set of actions to execute when <op> condition is false	69

4 The <uiml> and <head> Elements

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

4.1 The <*uiml*> Element

DTD

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

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/UIML3_0a.dtd'>...</uiml>
```

4.2 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.

4.2.1 The <*meta*> Element

DTD

```
<!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 an identifier for the meta-information; the *content* attribute gives 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>
```

5 Interface Description

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

5.1 Overview

The <*interface*> element contains four 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.

5.2 Attributes Common to Multiple Elements

Before explaining each of the elements introduced in Section 1, we first describe some attributes that are used in several of the elements.

5.2.1 The *id* and *class* Attributes

The <*part*>, <*event*>, and <*call*> elements in UIML may have an *id* and a *class* attribute.

The *id* attribute assigns a unique identifier to an element. No two elements can have the same id within the <u>same</u> UIML document.

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 [2] concept of class: a "class" specifies an object *type*, while the element's "id" 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 only associated with the element that specifies the value denoted in the style declaration for their *id* attribute.

5.2.2 The *source* and *how* Attributes

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

A *source* attribute can refer to two things:

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

  ```
  <constant id="Logo" source="http://uiml.org/images/UIMLLogo.jpg"/>
  ```

- *A URI to a resource that does contain UIML code.* The UIML code is inserted into the element that contains the *source*, as explained in Section 7.2. Inserting code has several uses, explained in section 7:

  - 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 id="Simple"> ... </style>
    <style id="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. Note that this URI is for demonstration only and is not truly active.

    ```
    <part id="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.

5.2.3 The *export* Attribute

The *export* attribute is used in the context of templates. See Section 7.4 for details.

5.3 The <*interface*> Element

DTD

```
<!ELEMENT interface (structure|style|content|behavior)*>
<!ATTLIST interface
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (append|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">
```

Description

All <uiml> elements that describe a user interface are contained in the <*interface*> element. The <*interface*> element describes a UI and a user's interaction with a UI, *not* the interaction of the UI and the backend application logic. The <*logic*> element is used to describe the UI/application logic interaction – see Section 6.3. 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 5.4), *style* (see Section 5.5), *content* (see Section 5.6), and *behavior* (see Section 5.7).

5.4 The <*structure*> Element

DTD

```
<!ELEMENT structure (part*)>
<!ATTLIST structure
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (append|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">
```

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 initially rendered. The relationship might be spatial (e.g., in a graphical UI) or temporal (e.g., in a voice interface).

The <*structure*> element defines the initial organization of the interface represented by the UIML document. This organization can be envisioned as a virtual tree of parts with each part's associated content, behavior, etc. attached to it.

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 rendering engine must select by id exactly one <*structure*> element and ignore all other <*structure*> elements. The id of the selected element is supplied by a mechanism outside the scope of this specification. The <*structure*> element whose id matches the supplied id is then used, and all other <*structure*> elements are ignored. If the supplied id does not match the id attribute of any *structure*, or if no id is supplied, then the last <*structure*> element appearing in the UIML document must be used.

Example

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

<structure id="SimpleUI">
   <part class="c1" id="n1"/>
</structure>

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

5.4.1.1 Dynamic Structure

The question remains as to how this initial virtual tree can be modified over the lifetime of the interface. Several "types" of dynamism exists in user interfaces. The three types that can be represented in UIML are described below:

- Content is dynamically supplied when the UI is rendered. This is handled by the <*reference*> element in section 5.6.2.

- The virtual tree of UI parts is modified during the lifetime of a UI. See the <*restructure*> element in section 5.7.11.
- The UI contains a sub-tree of parts that is repeated 1 or more times, where the number of times is determined at render time. This is the purpose of the <*repeat*> element.

5.4.2 The <*part*> Element

DTD

```
<!ELEMENT part (style?, content?, behavior?, part*,repeat*)>
<!ATTLIST part
          id          NMTOKEN                    #IMPLIED
          class       NMTOKEN                    #IMPLIED
          source      CDATA                      #IMPLIED
          where       (first|last|before|after)  "last"
          where-part  NMTOKEN                    #IMPLIED
          how         (append|cascade|replace)   "replace"
          export      (hidden|optional|required) "optional">
```

Description

Each <*part*> element represents either one instance of a class of UI widgets or 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 5.4). When the interface is rendered, only one structure is used (as discussed in “Description” under Section 5.4); thus, a part is always associated with a unique class.

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.

5.5 The <*style*> Element

DTD

```
<!ELEMENT style (property*)>
<!ATTLIST style
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (append|cascade|replace)    "replace"
          export (hidden|optional|required)  "optional">
```

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

The use of the style sheet helps achieve device independence. This is discussed in Section 5.5.2.

There must be at least one <*style*> element, and there may be more than one. There is often 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 7.2.

5.5.1 The <*property*> Element

DTD

```
<!ELEMENT property (#PCDATA|constant|property|reference|call|iterator)*>
<!ATTLIST property
          name        NMTOKEN                    #IMPLIED
          source      CDATA                      #IMPLIED
          how         (append|cascade|replace)   "replace"
          export      (hidden|optional|required) "optional"
          part-name   NMTOKEN                    #IMPLIED
          part-class  NMTOKEN                    #IMPLIED
          event-name  NMTOKEN                    #IMPLIED
          event-class NMTOKEN                    #IMPLIED >
```

Description

A property associates a name and value pair with a part or event (see Section 5.7.5). 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*.

5.5.1.1 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 a *<presentation>* element (see Section 6.2). A set of *<presentation>* elements for Java, HTML, WML, VoiceXML, and other toolkits are provided at http://uiml.org/toolkits. *<presentation>* elements, and hence property names, are created occasionally by experts on a target toolkit, like Java AWT, and are listed in a published location, such as http://uiml.org/toolkits. Many authors of UIML documents then reuse such <presentation> elements, referring to them in the base attribute or by a URI (e.g., http://uiml.org/toolkits/Java_1.3_Harmonia_1.0.uiml). A compiler or interpreter that renders UIML documents should also access the 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.

5.5.1.2 Semantics of <property> Element

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

- If the *<property>* element is a child element of a *<param>* (see Section 5.7.15), <op> (see Section 5.7.6), or another *<property>* element, then the semantics for that child *<property>* element are to *get* a single property's value.

- Otherwise the semantics are to *set* a value for a single property of an interface *part*, *event*, or *call*.

5.5.1.3 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 0 for more rules about strings and XML documents). Note that A UIML parser must preserve white space. A UIML rendering engine may ignore leading and trailing spaces when rendering text on certain UI toolkits.

```
<property part-name="p1" name="font">Helvetica-bold</property>
<property part-name="p1" name="title">Char: &amp;</property>
<property part-name="p1" name="content">
  <![CDATA[Character &]]>
</property>
```

- *A* <*reference*> element. In this case the property is set to the value of the <*reference*> element (see Section 5.6.2). In the following example[1], the value of *font* in the part with id *p1* is set to the value *Helvetica-bold*.

```
<property part-name="p1" name="font">
  <reference constant-name="font-name"/>
</property>
...
<content>
  <constant id="font-name">Helvetica-bold</constant>
</content>
```

In the following example, the user interface contains a Java button. When the button is pressed, the label of the button is set to the content of URL *referenceContent.xml*. It should be noted that content from a <*reference*> element is retrieved during the rendering of the UIML document, and **not** during the context from where the <*reference*> is defined. Thus, if the contents of referenceContent.xml were changed after the interfaces was initially rendered, but before the button was pressed the button would not reflect the changes. See Section 5.6.2 for more information on <*reference*>.

```
<interface>
  <structure>
    ...
    <part id="ok" class="JButton">
    <part id="label" class="JLabel">
    ...
  </structure>

  <behavior>
    <rule>
```

[1] From UIML2JAVAI/referenceURLinAction.uiml.

```
        <condition>
          <event part-name="ok" class="actionPerformed"/>
        </condition>
        <action>
          <property part-name="label" name="text">
            <reference url-name="referenceContent.xml"/>
          </property>
        </action>
      </rule>
    </behavior>

  <peers>
    <presentation base=   "Java_1.3_Harmonia_1.0"
                  source= "http://uiml.org/toolkits/
                          Java_1.3_Harmonia_1.0.uiml#vocab"/>
  </peers>
```

- *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 *p1* to whatever font *p2* currently has. The following UIML achieves this:

```
<property   part-name="p1" name="font">
  <property part-name="p2" name="font"/>
</property>
```

The nested *<property>* element gets the *font* of *p2*. The nested property does a *get* because it is nested in another *<property>* element, as explained in Section 5.5.1.2. That returned value then becomes the value of the *font* property in part *p1*.

It should be noted that defining a child *<property>* element of a nested *<property>* element is undefined, and does not provide any additional meaning.

- *A <call>* element. As explained in Section 5.7.8, a *call* is an invocation of code, such as calling a method in an object or a function in a script described in the *<logic>* element. In this case the property is set to the return value of the invocation. The following example[2] is an HTML user interface in which the content of a paragraph is set to the return value of method *random* in object *DoMath*. The call is performed at render time.

```
<uiml>
  <interface>
    <structure>
      ...
      <Pre id="pr1" content="Number 1: "/>
      <P id="ran1"/>
      ...
    </structure>

    <style>
      <property part-name="ran1" name="content">
        <call name="DoMath.random"/>
```

[2] From UIML2HTML/LocalExternalCall-RenderTime.uiml.

```
      </property>
      ...
    </style>

  <peers>
    <presentation how="replace"
                  source="HTML_3.2_Harmonia_1.0.uiml#vocab"
                  base=  "HTML_3.2_Harmonia_1.0"/>
    <logic>
      <d-component id="DoMath" maps-to="TestFunctionCalls">
        ...
        <d-method id="random"  maps-to="generateRandom"
                               return-type="int"/>
      </d-component>
    </logic>
  </peers>
</uiml>
```

The *<logic>* element (contained in the *<peers>* element of a UIML document) defines the code to which *DoMath.random* corresponds and how to invoke that code; see Section **6.3**.

5.5.1.4 Using *event-class* with <property> Elements

Just as <part> elements may have properties, so too may *<event>* elements have properties. There are some restrictions on using *<property>* with the *event-class* attribute. A *<property>* with the *event-class* attribute can only be used if the following apply:

- The *<property>* is a property-get operation (unless the *name* is *rendering*). Therefore the parent of *<property>* must be another *<property>*, *<op>*, or *<param>*
- The *<property>* has *<action>* as an ancestor.
- The ancestor *<rule>* of the *<property>* has in its *<condition>* child an *<event>* element.

The *<property>* naming an *event-class* always returns the value corresponding to the event occurrence named in the *<condition>*.

The three restrictions above arise because events normally represent transient events in a program, so it makes sense to query date associated with an event when the event occurs, but not later in the lifetime of the user interface.

Example

In the following UIML fragment, whenever a mouse click occurs for part *P*, method *doSomething* in object *O* is called with the x position of the mouse when clicked as an argument.

```
<rule>
  <condition>
    <event part-class=”P” class=”MouseListener.mouseClicked”/>
  </condition>
  <action>
    <call name=”O.doSomething”>
```

```
        <param><property event-class="MouseListener.mouseClicked"
                         name="X"/>
        </param>
      </call>
    </action>
  </rule>
```

5.5.1.5 Resolving Conflicting Property Values

A UIML document may contain more than one <*property*> element that sets the same property name for the same property. This is illustrated by the following example[3]:

```
<uiml>
  <interface id="myinterface">
    <structure>
      ...
        <part id="Button1" class="JButton">
          <style>
            <property name="text">Am I yellow?</property>
            <property name="background">blue</property>
          </style>
        </part>
      ...
    </structure>

    <style>
      <property part-name="Button1" name="background">orange</property>
      <property part-name="Button1" name="background">yellow</property>
    </style>
  </interface>
  <peers><presentation base="Java_1.3_Harmonia_1.0"/></peers>
</uiml>
```

In the example above, the background color of *Button1* is set in three <*property*> elements, to blue, orange, and yellow. A rendering engine to resolve such a conflict must follow the following semantic rule:

> Whenever a conflict arises between any two <*property*> elements in a UIML document, the <*property*> element that appears <u>last</u> in the canonical representation of the document must be used and the others must be ignored.

By the above rule, the button will have a background color of yellow in the preceding UIML document. Note that in canonical form, <*property*> elements whose grandparent is *interface* are appended to the end of the <*style*> element whose parent is the *part* or *parts* to which the properties refer. (The relative order of the <*property*> elements is preserved when they are moved.)

[3] From UIML2JAVAI/PropOrder0.uiml.

In the following example[4], the *Button1*, *Button2*, and *Button3* have background colors red, yellow, and green, respectively:

```
<uiml>
  <interface id="myinterface">
    <structure>
      ...
        <part id="Button1" class="JButton">
          <style>
            <property name="text">Am I red?</property>
            <property name="background">yellow</property>
          </style>
        </part>

        <part id="Button2" class="JButton">
          <style>
            <property name="text">Am I yellow?</property>
          </style>
        </part>

        <part id="Button3" class="JButton">
          <style>
            <property name="text">Am I green?</property>
          </style>
        </part>
      ...
    </structure>

    <style>
      <property part-class="JButton" name="background">green</property>
      <property part-name="Button1" name="background">red</property>
      <property part-name="Button2" name="background">yellow</property>
    </style>
  </interface>
  <peers><presentation base="Java_1.3_Harmonia_1.0"/></peers>
</uiml>
```

In canonical form, a *<property>* element with attribute *part-class* is copied to the *<style>* element of each *<part>* element in the *part-class*. Thus the canonical form of the preceding UIML becomes the following:

```
<uiml>
  <interface id="myinterface">
    <structure>
      ...
        <part id="Button1" class="JButton">
          <style>
            <property name="text">Am I red?</property>
            <property name="background">yellow</property>
            <property name="background">green</property>
            <property name="background">red </property>
          </style>
        </part>
```

[4] From UIML2JAVAI/PropOrder3.uiml.

```
          <part id="Button2" class="JButton">
            <style>
              <property name="text">Am I yellow?</property>
              <property name="background">green</property>
              <property name="background">yellow</property>
            </style>
          </part>

          <part id="Button3" class="JButton">
            <style>
              <property name="text">Am I green?</property>
              <property name="background">green</property>
            </style>
          </part>
        ...
      </structure>
    </interface>
    <peers><presentation base="Java_1.3_Harmonia_1.0"/></peers>
  </uiml>
```

Therefore the last <*property*> element for each <*part*> element is used, resulting in *Button1*, *Button2*, and *Button3* having background colors red, yellow, and green, respectively.

A rendering engine must evaluate all <*property*> elements for a given part in textual order and adhere to the conflict resolution policy described above when dealing with multiple conflicting <*property*> elements. Implementations of this policy are beyond the scope of this document, provided that the end result guarantees that the effect is equivalent to evaluating all conflicting <*property*> elements.

5.5.2 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 specify that it uses a vocabulary for Java AWT. This is done by a *presentation* element (exemplified earlier in Section 2.2.1):

```
<peers>
  <presentation base="JavaAWT_1.3_Harmonia_1.0"/>
</peers>
```

The above UIML fragment names base attribute that maps to a URI that defines the vocabulary. JavaAWT_1.3_Harmonia_1.0.uiml can be viewed as a black box by the UIML author. (It actually contains a <*presentation*> element, discussed in Section 6.2.) JavaAWT_1.3_Harmonia_1.0.uiml 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" id="submitButton"/>
</structure>
```

On the other hand, suppose we want to design a UIML document 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 want 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" id="submitButton"/>
</structure>
```

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

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

<style id="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*.

The above example is also very useful in a dynamic UI, where the binding of a part (e.g., *submitButton* to *Button* versus *JButton*) can change every time the UI is rendered. In the example below, each time the UIML document is rendered, the renderer executes the *<call>* element, and the return value of the *call* is either the string "Button" or "JButton". Therefore sometimes when the UI is rendered part *submitButton* will be an AWT *Button* and other times a Swing *JButton*. This might be useful if the UI is sometimes displayed on various devices, all of which implement AWT, and only some of which implement Swing.

```
<part class="AWTorSwingButton" id="submitButton">
  <style>
    <property name="rendering"><call name="X.getRendering"/></property>
  </style>
</part>
```

Another example of the use of *rendering* for a part-name or part-class is for UIs that contain the results of searches. The search result might be a table, and one column of the table might contain images on one search but text on another search. The choice of table column parts being images versus text would be determined by a *<call>* element to get the part rendering, similar to the one illustrated above.

Given this basic example, some variations are possible. First, the *style* element 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*. 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 menu item 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 map the name *ButtonOrMenuSelection* to a platform-specific name. In this case there would be <style> elements with two different ids:

```
<style id="Platform1">...</style>
<style id="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 id="Platform1">
  <property event-class="ButtonOrMenuSelected"
     name="rendering">ButtonSelected</property>
<style>

<style id="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 2.2.1 contains the following:

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

The *behavior* element describes what actions to take in response to various user interface events (see Section 5.7). 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 invocations of the *itemStateChanged* event in the Java

AWT class *ItemEvent*. If we were to modify the code in Section 2.2.1 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*.

5.5.2.1 Rules to Assign "rendering" Property

A UIML renderer must obey the following rules in assigning each *part* and *event* element a *rendering* property.

1. If a <property> element exists that contains attribute name="rendering" and one of the attributes part-class, event-class, part-name, or event-name use the <property> element value as the rendering.

2. Otherwise, the value of the rendering property is, the value of the class attribute for the <part> or <event> element that is found to be associated with this instance of <property> (this instance refers to the property in question). For example:

```
<part class="JButton" id="B1">
  <style>
    <property name="rendering"../>
```

The above defines that rendering of B1 is JButton. (The <presentation> element then defines the mapping of JButton to a UI widget in the toolkit, such as javax.swing.JButton.)

5.6 The <content> Element

DTD

```
<!ELEMENT content (constant*)>
<!ATTLIST content
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">
```

Description

A part in a UI can be associated with various content, such as words, characters, sounds, or 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 id="GUI">
   <part class="button" id="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>* elements 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 5.5] 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* elements 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 (either *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 id attribute of any *<content>* element, then the interface cannot be rendered.)

Example

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

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

```
      <reference constant-name="negativeLabel"/>
    </property>
  </style>

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

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

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

The last *<content>* element could also be shortened, by using the *source* attribute, discussed in Section 7.2, so that *EnglishSlang* inherited the *negativeLabel* from *English* as follows:

```
  <content id="EnglishSlang" source="English" how="cascade">
     <constant id="affirmativeLabel" >OK</constant>
  </content>
```

5.6.1 The *<constant>* Element

DTD

```
<!ELEMENT constant (constant)*>
<!ATTLIST constant
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (append|cascade|replace)    "replace"
          export (hidden|optional|required) "optional"
          model  CDATA                       #IMPLIED
          value  CDATA                       #IMPLIED>
```

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 an *id* 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. Note the following URI's are for demonstration purposes and may not be active.

```
<content>
   <constant id="Name" value="UIML"/>
   <constant id="Sound" source="http://uiml.org/uiml.wav"/>
   <constant id="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 5.7.3). For example:

```
<condition>
  <equal>
    <event part-name="inYear" class="filled" name="content"/>
    <constant value="2001"/>
  </equal>
</condition>
```

5.6.2 The <*reference*> Element

DTD

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

Description

The <*reference*> element references the value of the <*constant*> element specified by the *constant-name* attribute. Alternatively the <*reference*> element may specify a url-name attribute that contains a URI to an external document containing <*constant*> elements.

Example uses of the <*reference*> element are given in Sections 5.5.1.3 and 5.6.

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* element, and then the part's values can be initialized in the *style* element 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 2.2.1.

The semantics of a *<reference>* element is to replace the element with the *<constant>* element whose *id* attribute matches the *constant-name* attribute of the *<reference>* element. Or if the *url-name* attribute is specified, to replace the *<constant>* element contained within the document located by the URI given as the value of the *url*-name attribute. If no such element exists, then the UIML document cannot be rendered.

Implementations of UIML should retrieve content from *<reference>* elements during the context of rendering regardless of the context of from where the *<reference>* is being defined.

5.7 The *<behavior>* Element

DTD

```
<!ELEMENT behavior (rule*)>
<!ATTLIST behavior
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">
```

Description

The *behavior* element describes what happens when an end-user interacts with a user interface. For example, the *<behavior>* element might describe what happens when an end-user presses a button. The *<behavior>* element also describes when and how the user interface invokes methods (recall from Section 1.4 that a *method* refers to functions, procedures, database queries, and so on.)

The *<behavior>* element contains a sequence of rules. Each rule contains a condition and a list of actions. Whenever a condition holds, the associated action is performed. If the condition for more than one rule holds simultaneously, the algorithm in Appendix A is used to determine order of execution.

- UIML allows two types of conditions:
    - The first type of condition holds when an event fires (e.g., a button is pressed in the UI).
    - The second holds when an event fires and some logical expression of the data associated with the event evaluates to be true (e.g., a list selection is made and the selected item is "cat" – the first *<condition>* element in the dictionary example in Section 2.2.1).

(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 UI toolkits [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, and so on. The list of actions within a rule is executed in the order they appear in a UIML document, as described in Appendix A.

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 was the case for <*part*> elements, the UIML author chooses the *name* and *class* identifiers for events, and those names are mapped to an event in the underlying platform in the <*style*> and <*peers*> elements. For example, the end-user might define an event with class="selection," and the <*peers*> element for a Java AWT interface might map class "selection" to a java.awt.MouseEvent.

5.7.1 Examples of <*behavior*>, < *rule*>, <*condition*>, and <*action*> Elements

UIML allows the following actions in a behavior to be specified:

- *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>
      <!--A-->
      <event class="ButtonSelected" part-name="b1">
    </condition>
```

```
    <action>

      <!--1-->
      <property part-name="b1" name="color"/>blue</property>

      <!--2-->
      <property part-name="b2" name="color"/>
        <reference constant-name="green"/>
      </property>

      <!--3-->
      <property part-name="b2" name="color"/>
        <property part-name="b1" name="color"/>
      </property>

      <!--4-->
      <property part-name="b3" name="color"/><
        call name="serverObject.getColor"/></property>
```

```
    </action>
```

```
  </rule>
</behavior>
```

The <*behavior*> element above consists of one rule. The rule is executed whenever an event of class “ButtonSelected” for part “b1” fires. Let’s assume that “b1,” “b2,” and “b3” are buttons (established in <*part*> elements not shown), and “ButtonSelected” is a button click (established in a <*peers*> element not shown).

There is one <*action*> element associated with the rule. That action contains four elements that set properties in the interface, labeled 1 to 4 by comments.

1. The first action (labeled “<!--1-->”) sets property “color” of the button to blue when button b1 is clicked.
2. The second action sets the color of button b2 to a constant named “green” (and defined in the <*content*> element [see Section 5.6], not shown here).
3. The third action sets the color of button b2 to whatever color b1 currently has. Note that this action is executed after 1 and 2 above were executed, so button b2’s color is set to blue (because button b1’s color was set to blue in action 1 above).
4. The fourth action sets the color of button b3 to the return value of a call to something called “serverObject.getColor”. The <*d-component*> element in the <*logic*> element (discussed in Section 6.4.1) defines what “serverObject.getColor” is mapped to -- for example a method called “getColor” that takes no parameters in an object instance named “serverObject.” Note that no parameters are passed by the <*call*> element, so method getColor either must take no arguments or must have default values for all its formal arguments. Note that the return value of the call is converted to a character string, because the value of the <property part-name=“b3” name=“color”> element is a character string. Finally note that white space (spaces, tabs, line breaks) is significant in an XML document. Therefore the right angle bracket of the <property> tag must be immediately followed by the left angle bracket of the <call> tag, and similarly the right angle bracket of <call> must be immediately followed by the left angle bracket of the </property> tag.

- *Call a method*. The function or method call 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>
      <!--B-->
      <event class="ButtonSelected" part-name="b1">
    </condition>
```

```
    <action>

      <!--5-->
      <call name="m.storeData">
```

```
        <param>5</param>
        <param><reference constant-name="green"/></param>
      </call>

      <!--6-->
      <call name="m.storeColor">
        <param name="a3"><
          property part-name="b1" name="color"/
        ></param>
      </call>

      <!--7-->
      <call name="n.DisplayData">
        <param><call name="serverObject.getColor"/></param>
        <param><call name="q.getParam"><param>5</param></call></param>
      </call>

    </action>
```

```
  </rule>
</behavior>
```

The <*behavior*> element above consists of one rule. The rule is executed whenever an event of class "ButtonSelected" for part "b1" fires. Let's assume that "b1" is a button (established in a <*part*> element not shown), and "ButtonSelected" is a button click (established in a <*peers*> element not shown).

There is one <*action*> element associated with the rule. That action contains three elements that set properties in the interface, labeled 1 to 3 by comments.

1. The first action (labeled "<!--1-->") calls a method named "m.storeData" when button b1 is clicked. The method takes two arguments; the first is 5 and the second is the value of a constant named "green" (and defined in the <*content*> element [see Section 5.6], not shown here).

2. The second action is to call method "m.storeColor" with one parameter, whose value is the current color of button b1. The attribute name="a3" in the <*param*> element is used in the following situation: "m.storeColor" has more than one formal parameter, and one formal parameter is named "a3", and all other formal parameters have default values.

3. The third action calls method "n.DisplayData" with two parameters. The first is the return value of a call to method "getColor" in object "serverObject." The second is the return value of a call to method "getParam(5)" in object "g."

- *Fire an event*. An event can be fired from the <*action*> element. An <action> element may contain at most one <event> element, and this <event> element **must** appear as the last child of <action>.

```
<behavior>
  <rule>
```

```
    <condition>
      <!--C-->
      <event class="ButtonSelected" part-name="b1">
    </condition>
```

```
    <action>
      <!--8-->
      <!--executed when b1 is clicked -->
      <event class="ButtonSelected" part-name="b2"/>
    </action>
```

```
  </rule>

  <rule>
```

```
    <condition>
      <!--D-->
      <event class="ButtonSelected" part-name="b2">
    </condition>
```

```
    <action>
      <!--9-->
      <!--executed when b1 or b2 is clicked -->
      <call name="f1"><param>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 (event labeled “C”) and fire another event that will simulate the end-user pressing “b2” (action labeled “8”). Then the rendering engine will evaluate the condition for all the rules again (due to the algorithm in Appendix A), and the second rule will evaluate to true (event labeled “D”) and call f1(10) (action labeled “9”).

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).

5.7.2 The <*rule*> Element

DTD

```
<!ELEMENT rule (condition,action)?>
<!ATTLIST rule
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (append|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">
```

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 5.7.1 for an example and further explanation. Also, it is possible for multiple rules to be satisfied at any time; Appendix A defines how a rendering engine must handle this situation. See Section 5.7.1 for examples.

5.7.3 The <*condition*> Element

DTD

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

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 in the <*equal*> *or* <*op*> element evaluates to *true*. See Section 5.7.1 for examples.

5.7.4 The <*equal*> Element

DTD

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

Description

The <*equal*> element is a Boolean expression with value *true* or *false*. Every <*equal*> element must have exactly two children. Each child must be a <*constant*>, <*property*>, <*reference*>*, or* <*op*> element. The semantics of <*equal*> are as follows. Whenever the two children named in the <*equal*> element resolve to the same value then the <*equal*> element has value *true*. Otherwise the <*equal*> element has value *false*. Section 2.2.1 illustrates the <*equal*> element.

5.7.5 The <*event*> Element

DTD

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

Description

The <*event*> element is used in two contexts:

- As the child of a <*condition*> element. The parent *condition* is satisfied whenever the *event* occurs. (For example, see event labeled "A" in Section 5.7.1.)
- As the child of an <*action*> element. The event is fired. (For example, see event labeled "8" in 5.7.1.)

5.7.5.1 Exceptions as Events

<event> elements are used to represent spontaneous events that happen in the system. The most common example of these type pf events are user generated inputs like mouse clicks and keystrokes. However, user actions are not the only events that can be modeled by <event> elements of UIML; exceptions generated programmatically also fit into this model. Exceptions occur when something unexpected happens in the underlying system as the result of a <call> element's communication to that system (see section 5.7.8 for more details on the <call> element). In the example below, the exception *MyException* is thrown by some method call to the underlying backend. This code snippet would catch the exception and execute the action defined in the <action> element.

```
<uiml>
  …
  <rule>
    <condition>
      <event class="MyException"/>
    </condition>
    <action>
      …
    </action>
  </rule>
  …
</uiml>
```

Note that exceptions are treated differently from normal events in the following way; exceptions cannot be associated with any particular part. Rendering engines should produce a warning if the part-name attribute is associated with an <*event*> element that represents an exception.

<event> elements defining exceptions as shown above will result in the named exception being caught, no matter where it is thrown from.

5.7.5.2 Extracting Data from Events

Events often contain useful data that may be needed by other aspects of the user interface. For example, if the system detects a mouse click, it may be useful to extract the coordinates of the mouse click for display elsewhere in the interface. To facilitate this type of interaction, UIML recognizes <property> elements as children of <event> elements. These properties represent data wrapped up in the event and can be used to extract the data from the event. Data of this type can be accessed using the following <property> syntax:

<property event-class=”MyEvent” name=”MyEventData”/>

Since <event>’s cannot be named, this always refers to the last event of the specified class to occur.

5.7.5.3 Special Events

UIML facilitates initializing a page (similar in effect to an onLoad event in HTML) through the use of a special event-class named “init”. The init event class is used as a child of a <condition> element and the <action> associated with the condition containing the init <event> will be executed before the page is rendered. Thus users of UIML can use the init event for operations like initializing data objects in the backend or ensuring that pre-conditions to rendering are met before actual rendering.

5.7.6 The <*op*> Element

DTD

```
<!ELEMENT op (constant|property|reference|call|op|event)*>
<!ATTLIST op
          name   CDATA                     #REQUIRED>
```

Description

The <*op*> element allows multiple complex logic conditions to be expressed in UIML. In the previous examples simple conditions were used to control whether or not elements under <*action*> were executed. The simplicity of the previous examples allow for only one condition to hold true, usually this was reserved to see if a particular <*event*> had occurred. Even with the functionality introduced with the <*equal*> tag, an author could only evaluate two different conditions; furthermore the <*equal*> tag provided a limited logical condition, testing only if two values were equal. However with the <*op*> element, basic logical conditions (less than, greater than, equal, not equal, and, or…) may be expressed along with the ability to structure complex condition statements involving multiple values.

5.7.6.1 Semantics of <*op*>

The *name* attribute of <*op*> describes the conditional applied to the expression that you wish to create. The value of the *name* attribute can either be symbols representing operators or the written name of the operator itself. The following is a list of valid operators:

Name	**Operator**	**Definition**
equal	==	A == B Returns true if A equals B.
notequal	!=	A!=B Returns true if A does NOT equal B.
and	&&	A&&B Returns true if and only if both A and B are true.
or	||	A||B Returns true if either A or B is true.
lessthan	<	A<B Returns true if A is less than B.
greaterthan	>	A>B Returns true if A is greater than B.

The children of <*op*> are the operands for the conditional. The following is a list of valid children elements each of which have different semantics from each other.

1. The child is a <*constant*>*,* <*property*>, <*reference*>*, or* <*call*> element :
   The string value of the element is used as the operand for the conditional. The following example illustrates a conditional that compares if two values are equal to each other.

```
<condition>
  <op name="equal">
     <property name="value" part-name="city"/>
     <constant value="'Los Angeles'"/>
  </op>
 </condition>
```

2. The child is an <event> element:

   The operand for the conditional is the boolean value resulting from the evaluation of determining whether that specified event was caught. The following conditional hold true if the *event* "buttonClicked" was caught and the value of the <*property*> *element* was equal to the constant string "Los Angeles".

```
<condition>
  <op name="and">
    <event class="buttonClicked" part-name="city"/>
    <op name="equal">
      <property name="value" part-name="city"/>
      <constant value="Los Angeles"/>
    </op>
  </op>
</condition>
```

3. The child is an <op> element:

   The operand for the condition is the boolean value resulting from the evaluation of the <*op*> statement. Nested <*op*> statements allow expressions to contain multiple operators with multiple operands.

5.7.6.2 Resolution of Conditional Statements

The introduction of <*op*> element brings upon instances where certain actions may want to be defined as a result from the evaluation of the <*op*> element. Three new elements <*when-true*>, <*when-false*>, <*by-default*> have been introduced to define a set of actions when a conditional is found to be true, false, or undefined. The elements are found as children of <*action*>.

5.7.7 The <*action*> Element

DTD

```
<!ELEMENT action (((property|call|restructure)*, event?)|(when-
true?,when-false?,by-default?))>
```

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 a part (e.g., 1 to 4 in Section 5.7.1), a <*call*> element to invoke a method (e.g., 5 to 7 in Section 5.7.1), a <restructure> element to restructure an interface (Section 5.7.11), an <*event*> element to fire another event (e.g., 8 in Section 5.7.1), or a <when-true>, <when-false>, <by-default> element to determine a set course of actions depending on the value of the conditional expressed in the <*condition*> element (Sections 5.7.12, 5.7.13, 5.7.14). 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.

5.7.8 The <*call*> Element

DTD

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

Description

The <*call*> element is an abstraction of any type of invocation of code (that uses a language other than UIML). The code is referred to in this specification as a *method*, which in Section 1.4 is defined to include functions, procedures, and methods in an object-oriented language, database queries, and directory accesses.

5.7.8.1 Overview on <call>

UIML's philosophy on specifying function invocations is to allow the UIML author to freely choose a set of names for widgets, events, and functions referenced in the <interface> section. Each of these names is then mapped in the <peers> section to implementing entities (e.g., Swing user-interface components, methods in memory or remote object instances, entry points in remote procedures, functions in user scripts, etc.).

For Swing components, the hard work of creating the <peer> mappings has already been done – UIML authors need only import an existing body of predefined mappings, instead of writing an appropriate <peers> section. For object methods, the UIML author must write one <d-component> element for each object instance whose method or methods are to be invoked (note that <d-component> is a child of <logic>, which in turn is a child of <peers>).

The *<call>* element has one mandatory attribute, *name*.

A rendering engine executes a *<call>* element as follows:

1. The engine must locate the method to be invoked.
2. If the method to be invoked takes arguments, the rendering engine then matches the *<param>* elements in the *call* body to the method's formal arguments.
3. The rendering engine converts the argument values from character strings to the type of the formal parameters.
4. The rendering engine invokes the method.
5. If the method returns a value (attribute *return-type* in element d-*method* must have a non-null value -- see Section 6.4.4), then the value is converted to a character string.

Examples of the *<call>* element are shown in Section 5.5.1.3 and Section 5.7.1.

If a *<call>* element has as its grandparent a *<style>* element and is not the descendent of a *<restructure>* element, then the *call* must be evaluated at render time. Otherwise the *call* must be evaluated at runtime. Therefore if an *<action>* element describing what to do when a button is pressed in a UI contains a *call*, then the *call* is executed every time the button is pressed.

Invocation via Direct Specification

Suppose that the UIML document contained an <action> element to invoke method m1 in object back1, with actual parameters 5 and 10, as in the expression "back1.m1(5,10)". The <action> element might look like the following:

```
<behavior>
   <rule>
      ...
      <action>
         <call name="back1.m1">
            <param>5</param>
            <param>10</param>
         </call>
      </action>
```

```
   </rule>
</behavior>
```

The d-component element defining back1 might look like the following:

```
<peer>
   <logic>
      <d-component id="back1" maps-to="org.uiml.example.myClass">
        <d-method id="m1" maps-to="myFunction">
          <d-param id="p1"/>
          <d-param id="p2"/>
        </d-method>
      </d-component>
   </logic>
</peer>
```

The value of the *maps-to* attribute of the <d-component> element follows the syntax imposed by the language in which the rendering engine in use is implemented (Java, in this example). Here, the <d-component> element maps the name "back1" to the class "org.uiml.example.myClass", which in turn contains a method named "myFunction" to which the name "m1" is mapped. In order to make the invocation "myClass.myFunction(p1,p2)", the UIML <call> element should refer to the corresponding values of the *id* attributes of the <d-component> and <d-method> elements, hence "back1.m1". Given the <action> element above, the rendering engine then converts the expression "back1.m1(5,10)" into an evaluation of "myClass.myFunction(5,10)".

Note that the *id* attributes in the <d-param> elements are optional – they serve merely to document myFunction's parameter list.

5.7.8.2 Method Parameters and Return Values Types

Often, an object whose methods are to be invoked defines two or more methods which share the same name, but have different numbers and/or types of formal parameters, in which case parameter count and type information must be supplied so that the rendering engine can invoke the proper method. Within a <d-method> element, the number of <d-param> elements denotes the number of parameters taken by that particular method. Type information is introduced by means of the *type* attribute, as in the following:

```
<peer>
   <logic>
      <d-component id="back1" maps-to="org.uiml.example.myClass">
         <d-method id="m1" maps-to="myFunction">
           <d-param id="p1" type="int"/>       <!--p1 is an int-->
           <d-param id="p2" type="int"/>       <!--p2 is an int-->
         </d-method>
      </d-component>
   </logic>
</peer>
```

If the type attribute is omitted, the parameter type is assumed to be "string" – the value is then converted into the type required by the function to which the named method maps (i.e., "myFunction" in the example above). This type conversion is performed in accordance with the rules of the language in which the rendering engine is implemented.

By default, the return value from a method is ignored – it is not made available to the element enclosing the <call> from which the invocation is made. The return value can be made available by specifying the *return-type* attribute of the <d-method> element, as in the following:

```
<peer>
   <logic>
      <d-component id="back1" maps-to="org.uiml.example.myClass">
        <d-method id="m1" maps-to="myFunction"  return-type="int">
          <d-param id="p1" type="int"/>
          <d-param id="p2" type="int"/>
        </d-method>
      </d-component>
   </logic>
</peer>
```

5.7.8.3 Invoking Methods Upon External Objects in RMI, CORBA, LDAP, and EJB

So far, the examples shown have all made use of objects that are implicitly instantiated within and managed by the same rendering engine that manages the user interface. UIML itself does not require or even specify this particular behavior; however, additional details are typically needed when the object instance whose method is to be invoked resides outside the rendering engine's execution space. These details include the *class* of which the object is an instance and the object instance's *location*.

For example, consider the mechanics involved in making a Java RMI invocation upon an external object instance. The location of the Java virtual machine that hosts the external object instance (e.g., a hostname) must be specified; additionally, the name by which that instance is registered with the associated RMI registry must also be given. Once the object instance is found and a reference of type `java.lang.Object` to it is obtained, that reference must then be downcast into the proper object class.

The *maps-to* attribute of the <d-component> element is used to specify the external object's class as in the case where the object resides inside the rendering engine's execution space. The needed location information is supplied by means of the *location* attribute, as in the following example:

```
<peer>
   <logic>
      <d-component id="back1"
         maps-to="org.uiml.example.myClass"
         location="rmi://myHost.myCompany.com/Adder">

         <d-method id="m1" maps-to="myFunction"
```

```
              return-type="int">

              <d-param id="p1" type="int"/>
              <d-param id="p2" type="int"/>
           </d-method>
        </d-component>
     </logic>
  </peer>
```

As this example shows, the value of the *location* attribute is a standard URL, whose syntax varies with the URL's protocol, in this case, "`rmi`". This particular URL states that the remote object can be found on host "`myHost.myCompany.com`", and that the object is registered under the name "`Adder`". Once a reference to this object is obtained by the rendering engine, the reference is downcast to a reference to class "`org.uiml.example.myClass`".

A rendering engine implementation may support any protocol for the *location* attribute, as long as the protocol supports obtaining object references. The Java Renderer version 1.0b supports the protocols and corresponding URL formats shown below.

PROTOCOL	URL FORMAT
Rmi	rmi://hostname[:port]/registeredname
Iiop	iiop://hostname[:port]/registeredname
Ldap	ldap://hostname[:port]/entrypath/entryname
Ejb	ejb://hostname[:port]/jndi_name

5.7.9 The <*repeat*> Element

DTD

```
<!ELEMENT repeat (iterator,part*)>
<!ATTLIST repeat EMPTY>
```

Description

A <*repeat*> element must enclose one <*iterator*> element and a set of one or more <*part*> elements. The <*part*> elements denoted as children of the <*repeat*> element are repeated with their children a number of times designated by the <*iterator*> element. The <repeat> elements parent <*part*> element will not be repeated.

A <*repeat*> element has the following legal children, ordering of the children does not matter:

- Each <*repeat*> element must have one and only one <*iterator*> child. The <*iterator*> element denotes how many times the specified interface components will be repeated. If more than one <*iterator*> child is defined than the implementation must produce a warning and use the last <*iterator*> defined in textual order.
- Each <*repeat*> must have one or more <*part*> elements as children. These <*part*> elements represent the components to be repeated. If the components are named (i.e.

have a defined 'name' attribute), then each repetition of the <*part*> will have '_#' appended onto the part name where # is the integer representation of this iteration.

Nested repeats are allowed, meaning that a <repeat> can be a child of another <repeat>'s <*part*> element descendents (not just first level children). This allows for the dynamic construction of more complicated interfaces elements such as tables and static depth trees.

5.7.10 The <*iterator*> Element

DTD

```
<!ELEMENT iterator (#PCDATA|constant|property|call)>
<!ATTLIST iterator
          id     NMTOKEN                    #REQUIRED>
```

Description

The <*iterator*> element defines the number of times the interface components should be repeated. <*iterator*> elements can have only one child, but that child can be of four forms:

- A text string
- A <*call*> element
- A <*property*> element
- A <*constant*> element

The form of the child is irrelevant so long as it resolves to an integer value ***N***. This integer ***N*** is then used as the maximum number of iterations that the repeat will perform, counting from 1 to ***N***. Note that the step value for an <*iterator*> element is currently always one.

5.7.10.1 Using <*iterator*> in <*property*> and <*param*>

The <*iterator*> element can be used in <*property*> and <*param*> elements to provide an integer value representing the iteration number that is currently processing. In this way, the <*iterator*> element behaves very similarly to the <*property*> element.

It is important to note that an <*iterator*> is defined within the scope of the <*repeat*> it is a child of. Thus, no other <*iterator*> elements may have the same id if they are defined within a descendent of the current <*repeat*>. This also implies that an <*iterator*> whose <*repeat*> is an ancestor of another <*iterator*> can be accessed within the scope of the descendent <*iterator*>.

Example

```
<uiml>
  …
  <part class="JDialog">
    <repeat>
      <iterator id="i">10</iterator>
```

```
      <part class = "JCheckBox">
        <style>
          <property name="text"><iterator id="i"/></property>
        </style>
      </part>
    </repeat>
  </part>
  …
</uiml>
```

The example above demonstrates the two uses of the <*iterator*> element and would result in the appearance of a JDialog containing ten JCheckBoxes. The JCheckBoxes would be numbered 1 to 10.

5.7.11 The <*restructure*> Element

DTD

```
<!ELEMENT restructure (template)?>
<!ATTLIST restructure
          at-part      NMTOKEN                            #IMPLIED
          how          (union|cascade|replace|delete)     "replace"
          where        (first|last|before|after)          "last"
          where-part   NMTOKEN                            #IMPLIED
          source       CDATA                              #IMPLIED>
```

Description

The <*restructure*> element provides a way for the UI to change as a result of some condition being met. Most conditions include but are not limited to user's interactions. For example, using the Java AWT/Swing vocabulary for UIML, a UI containing a window with a button and a panel is described like this:

```
<structure>
  <part class="JFrame"    name="F">
    <part class="JButton" name="B"/>
    <part class="JPanel"  name="A"/>
  </part>
</structure>
```

Suppose when the initial UI is displayed, we wanted only the button to appear. When the user clicks the button, the panel appears. We would use the <*restructure*> element to define the necessary changes within the UI to remove the button and display the panel.

The semantics of UIML are changed to include the concept of a *virtual UI tree*. During the lifetime of a UI, the parts comprising the UI may change. (All parts that exist but are invisible to an end user are still part of the tree.) The parts present in the UI have a hierarchical relationship, therefore forming a tree. At any moment during the UI lifetime, one could enumerate the tree of parts that currently exist, and this is the *virtual UI tree*. Each node in this tree corresponds to a <part> element in the UI generated by UIML. We call the tree "virtual" because it may or may not be physically represented as a data structure on a computer, depending on how a rendering engine is implemented.

The *initial value* of the virtual UI tree is the content of the <structure> element in a UIML document. During the UI lifetime, the virtual UI tree can be modified by deleting nodes or adding nodes using the <restructure> tag. (The <restructure> tag is so-named because it modifies the <structure> section's representation in the virtual UI tree.) The <restructure> tag can only appear inside an <action> element in UIML.

5.7.11.1 Syntax of <*restructure*>

The syntax of <restructure> follows:

```
<restructure at-part="[part-name1]"
             how="append|cascade|replace|delete"
             where="first|last|before|after"
             where-part="[part-name2]"
             source="[template-location]">
```

The <restructure> element may not contain a body if one of the following holds:

- The *source* attribute is present
- *how="delete"* is present

Otherwise the <restructure> element must contain a body, and that body must contain exactly one <template> element, which must contain exactly one <part> element that matches the part specified in the at-part attribute.

5.7.11.2 Semantics of <*restructure*>

The semantics of <restructure> are to modify the virtual UI tree as follows:

how="delete":
Delete from the current virtual UI tree the sub-tree rooted at the part named in the "at-part" attribute. Also delete any properties or rules of the part. There can be no body for <restructure>. Attributes *where, where-part*, and *source* **cannot** be used.

how="replace":
Replaces the part specified by the "at-part" attribute with the parts defined as children of the template T, where T is defined as the child of the restructure.

how="append":
Appends parts defined as children of the template T, where T is defined as the child of the restructure. The way parts are appended to the structure is defined through use of the "where" and "where-part" attribute (see below).

how="cascade":
Cascades parts defined as children of the template T, where T is defined as the child of the restructure. The way parts are cascaded to the structure is defined through use of the "where" and "where-part" attribute (see below). The cascading behavior exhibits the same behavior as found in Section 7.2.3

The *where* attribute can only be used when the *source* attribute is present and *how="cascade"* or *how="append"* is present. The *where-part* attribute can be used only when *where="before"* or *where="after"* is used.

The attributes have the following semantics. Let the part element specified by *at-part* be <part name="P" ...>

If attribute *where="first"* is present: All children of part P in the <template> named in the *source* attribute must be inserted as children of part P *before* the existing children.

If attribute *where="last"* is present: All children of the part P in <template> named in the *source* attribute must be inserted as children of part P *after* the existing children.

If attribute *where="before"* and *where-part="[part-name]"* is present: All children of part P in the <template> named in the *source* attribute must be inserted as children of part P *before* the child of P with part name *part-name*, but after any children appearing before the child "part-name".

If attribute *where="after"* and *where-part="[part-name]"* is present: All children of part P in the <template> named in the *source* attribute must be inserted as children of part P *after* the child of P with part name *part-name,* but before any subsequent children of P.

5.7.11.3 Examples of <*restructure*>

Consider the button and panel introduced at the beginning of the proposal, where the panel contains three components: a label, a text field, and a check box. The UIML looks like this:

```
<structure>
  <part class="JFrame"    id="F">
    <part class="JButton" id="B"/>
    <part class="JPanel" id="A">
      <part class="JLabel" id="L1"/>
      <part class="JTextField" id="TF"/>
      <part class="JCheckbox" id="C"/>
    </part>
  </part>
</structure>
```

Append Examples

To add another label before L1, do this:

```
<restructure at-part="A" how="append" where="first">
```

```
  <template id="T1">
    <part>
      <part class="JLabel" id="L2"/>
    </part>
  </template>
</restructure>
```

The panel now contains the parts in this order: A_T1_L2, L1, TF, C. Note: the naming convention used for parts that are added from a template is discussed in Section 7.2. To add a label and a text area between TF and C, do this:

```
<restructure at-part="A" how="append" where="after" where-part="TF">
  <template id="T2">
    <part>
      <part class="JLabel" id="L3"/>
      <part class="JTextArea" id="TA"/>
    </part>
  </template>
</restructure>
```

(Alternately, one could have used *where="before" where-part="C"*.) The order of parts in the panel is now A_T1_L2, L1, TF, A_T2_L3, A_T2_TA, C.

As the UI is modified, the virtual UI tree changes. At any point in time, when a <restructure> is processed, the "where" attribute refers to the *current* virtual tree. So if we executed the above restructures and then the following, the resulting order of components in the panel would be A_T1_L2, L1, TF, A_T2_L3, A_T2_TA, A_T3_L4, C:

```
<restructure at-part="A" how="append" where="before" where-part="C">
  <template id="T3">
    <part>
      <part class="JLabel" id="L4"/>
    </part>
  </template>
</restructure>
```

The template tag surrounding the part tag provides a resolution mechanism for any name conflicts that might occur when using append. Consider the restructure tag below.

```
<restructure at-part="A" how="append">
  <template id="T4">
    <part>
      <part class="Label" id="L1"/>
    </part>
  </template>
</restructure>
```

There is already a part named L1 as a child of A, but the part L1 being added gets renamed to A.T4.L1, so now the panel A has the following children: A_T1_L2, L1, TF, A_T2_L3, A_T2_TA, A_T3_L4, C, A_T4_L1.

Replace Example

To replace the entire panel with a new panel containing just a label and a text field, do this:

```
<restructure at-part="A" how="replace">
  <template id="T5">
    <part>
      <part class="JLabel" is="L1"/>
      <part class="JTextField" is="TF"/>
    </part>
  </template>
</restructure>
```

The set of children of A in the virtual UI tree is replaced with the set of children listed in the restructure tag. So now the panel has only the following children: A_T5_L1, A_T5_TF.

Cascade Example

When cascade is used, parts in the template that have the same name as a child of the <part> specified by *at-part* are not used, but any other parts in the template are added as children of the <part>. For example:

```
<restructure at-part="A" how="cascade" where="last">
  <template id="T6">
    <part>
      <part class="JLabel" id="L1"/>
      <part class="JLabel" id="L5"/>
    </part>
  </template>
</restructure>
```

Since A already has a child named L1 (A_T5_L1), the L1 in template T6 is not used, but part L5 from the template is added as a child of A. So A's children are now A_T5_L1, A_T5_TF, A_T6_L5. Note that when checking for a part with the same name, only the last part of a fully-qualified name is checked, so A_T5_L1 matches L1.

Delete Example

To delete the panel, simply do this:

```
<restructure at-part="A" how="delete"/>
```

5.7.12 The <*when-true*> Element

DTD

```
<!ELEMENT when-true ((property|call)*,restructure?,op?,equal?,event?)>
```

Description

The <*when-true*> element defines a set of actions to be executed when the evaluation of a conditional expression defined by the element <*op*> results to be the Boolean value true.

5.7.13 The <*when-false*> Element

DTD

```
<!ELEMENT when-false ((property|call)*,restructure?,op?,equal?,event?)>
```

Description

The <*when-false*> element defines a set of actions to be executed when the evaluation of a conditional expression defined by the element <*op*> results to be the Boolean value false.

5.7.14 The <*by-default*> Element

DTD

```
<!ELEMENT by-default ((property|call)*,restructure?,op?,equal?,event?)>
```

Description

The <*by-default*> element defines a set of actions to be executed when the evaluation of a conditional expression defined by the element <*op*> results to be undeterminable or undefined.

5.7.15 The <*param*> Element

DTD

```
<!ELEMENT param (#PCDATA|
                 property|
                 reference|
                 call|
                 op|
                 event|
                 constant|
                 iterator)>
<!ATTLIST param
          name NMTOKEN #IMPLIED>
```

Description

Describes a single actual parameter of the method call specified by the parent <*call*> element. Note that the values of all parameters in UIML are character strings. See Section 6.4.5 for information on conversion of the arguments to the types required by the formal parameters of the method being called.

If the number of <*param*> elements equals the number of formal parameters in the method being called then the following hold:

- The *name* attribute is optional, and is ignored by the rendering engine if present.
- The order of <*param*> elements within the <*call*> element must match the order of the formal parameters in the method being called.

Otherwise there must be fewer <*param*> elements than formal parameters in the method being called, and the following holds:

- The *name* attribute is required on all *param* elements.
- The *name* attribute must be used by the rendering engine to match each <*param*> element to a formal parameter in the method being called.

A <*param*> element must have exactly one child

6 Peer Components

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

6.1 The *<peers>* Element

DTD

```
<!ELEMENT peers (presentation|logic)*>
<!ATTLIST peers
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (append|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">
```

Description

To facilitate extensibility, UIML includes a *<peers>* element that defines mappings from class, property, event, and call names used in a UIML document to entities external to the UIML document. The *<peers>* element has two child elements, for two types of mappings:

- The *<presentation>* element contains mappings of part and event classes, property names, and event names to a UI toolkit. This mapping defines a vocabulary to be used with a UIML document, such as a vocabulary of classes and names for VoiceXML. Normally a UIML author does not write a *<presentation>* element, but instead uses names in a UIML document that have been defined in the list of vocabularies at http://uiml.org/toolkits. Section 6.2.1 discusses vocabularies.
- The *<logic>* element maps names and classes used in *<call>* elements to application logic external to the UIML document. In large-scale software development, the *<logic>* element is defined once to represent the API of the application logic (typically as a *<template>* element), and then included in each UI for the project.

6.2 The <*presentation*> Element

DTD

```
<!ELEMENT presentation (d-class*)>
<!ATTLIST presentation
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (append|cascade|replace)    "replace"
          export (hidden|optional|required) "optional"
          base   CDATA                       #REQUIRED>
```

Description

Every UIML document uses a *vocabulary*. The vocabulary defines the legal class names that can be used for parts and events in a UIML document, as well as the legal property names. The formal definition of a vocabulary is done through a <*presentation*> element containing <*d-class*> elements (see Section 6.2.2). Each <*d-class*> element defines a legal class name.

At present, the list of standard vocabularies is posted on http://uiml.org/toolkits, in the form of a set of <presentation> templates that may be included into UIML documents.

In the remainder of this section, we first discuss (Section 6.2.1) UIML's use of a standard set of vocabulary names, which is all that most UIML authors need to know about the <*presentation*> element. Then (Section 6.2.2) we discuss how to define a new vocabulary using the children of the <*presentation*> element.

6.2.1 Naming an Existing Vocabulary in <*presentation*>

Normally, a UIML author uses an existing vocabulary. Therefore UIML requires a way to label each UIML document with the vocabulary used in that document. The labeling follows a convention, and authors of new UIML vocabularies must follow this convention.

There are two categories of vocabularies:

- Vocabularies with widespread use, whose definition is posted on uiml.org, in http://uiml.org/toolkits. Examples are vocabularies for Java 1.3 AWT and Swing, HTML 3.2, WML, and VoiceXML. We call these ***base*** vocabularies.
- Custom vocabularies that individuals, companies, or other organizations define. These may or may not be not posted on uiml.org. The vocabularies may be posted on web sites around the world of interest to specific communities, or posted in a company's internal network, or not posted at all. We call these ***custom*** vocabularies.

Typically a custom vocabulary extends a base vocabulary (e.g., a company creates custom Java UI classes that extend Swing classes). The unlikely exception is when someone creates a custom vocabulary from scratch that does not rely on any base vocabulary, yet a rendering engine for some base vocabulary can render that custom vocabulary. For example, someone might create a new UI toolkit for Java from scratch that does not use AWT or Swing, but design the classes so that a rendering engine for UIML using the Java 1.3 or Swing base vocabulary can still render the custom classes.

A UIML document must be labeled by a base vocabulary name, and may be labeled by a custom vocabulary name, in the manner described next.

6.2.1.1 Labeling Base Vocabularies with Attribute *base*

To easily determine the *base* language used within the UIML document, the *<presentation>* element requires an attribute named *base*, which identifies the base target language of the UIML document.

The syntax of values for the "base" attribute must follow this convention:

```
<vocab-name>_<vocab-version>_<author-of-vocab>_<author's-version-of-
vocab>
```

Several rules apply to *base*:

- The value of *base* is case insensitive.
- If the value of *base* is x, then the following URL must define x:

  http://uiml.org/toolkits/x.uiml.

  This URL must contain a <template> element, whose child is a <presentation> element, whose children define the vocabulary *x*. (Requests to post a new file should be sent to info@uiml.org.)

- Every rendering engine must implement one or more base vocabularies. If a rendering engine implements vocabulary "uiml.org/toolkits/x.uiml", then the rendering engine must be able to render any document that contains <presentation base="x"> (with or without the *source* attribute).
- For example, a rendering engine could be created to display UIs using the Java 1.3 Swing and AWT toolkits; in this case the rendering engine might recognize the vocabularies Java_1.3_Harmonia_1.0.uiml, JavaAWT_1.3_Harmonia_1.0.uiml, and JavaSwing_1.3_-Harmonia_1.0.uiml. Such a rendering engine must also render custom classes that a UIML author creates that extend Swing classes, by writing their own *<presentation>* element to extend the base vocabulary.
- Consider the following *<presentation>* element:

  ```
  <presentation
      source="MySwing_1.0_JoeProgrammer_0.1.uiml#vocab"
      base="Java_1.3_Harmonia_1.0"/>
  ```

- The *<presentation>* element says that any rendering engine that implements vocabulary uiml.org/toolkits/Java_1.3_Harmonia_1.0.uiml can render the UIML document containing the <*presentation*> element, even though the rendering engine was written without knowledge of what is in vocabulary MySwing_1.0_JoeProgrammer_0.1.uiml.

- (It is the responsibility of the UIML author to insure consistency between the values of the *source* and *base* attributes. For example, the UIML author should not define class names mapped to HTML tags in MySwing_1.0_JoeProgrammer_0.1.uiml and then set base to a vocabulary for Java [e.g., Java_1.3_Harmonia_1.0].)
- The following table gives the vocabulary that a rendering engine *actually* uses to render a UIML document. The table assumes that the <*presentation*> has the attribute 'base="x"'.

Does <presentation> contain *source* attribute (e.g., *source="y"*)?	**Does <presentation> have body?**	**Vocabulary is defined by this:**	**Example (see below)**
Yes	Doesn't matter	Combination of *y* and, if the body of <presentation> is not empty, anything in the body of <presentation>	#4
No	No	uiml.org/toolkits/x.uiml	#1,2,3
	Yes	uiml.org/toolkits/x.uiml augmented by the body of <presentation>	#5

Here are some examples of legal <*presentation*> elements and their meanings:

```
1. <presentation base="Java_1.3_Harmonia_1.0"/>
```

UIML document must be rendered by rendering engines implementing vocabulary uiml.org/toolkits/Java_1.3_Harmonia_1.0.uiml.

```
2. <presentation base="HTML_3.2_Harmonia_1.0"/>
```

UIML document must be rendered by rendering engines implementing vocabulary uiml.org/toolkits/HTML_3.2_Harmonia_1.0.uiml.

```
3. <presentation base="GenericJH_1.0_Harmonia_1.0"/>
```

UIML document must be rendered by rendering engines implementing vocabulary uiml.org/toolkits/GenericJH_1.0_Harmonia_1.0.uiml.

```
4. <presentation base="Java_1.3_Harmonia_0.8"
                 source="http://xyz.com/MySwing_1.0_xyz_0.1.uiml#vocab"/>
```

UIML document must be rendered by rendering engine implementing vocabulary uiml.org/toolkits/ Java_1.3_Harmonia_0.8.uiml, but the vocabulary used in this UIML document is the combination of Java_1.3_Harmonia_0.8 and the *presentation* defined in http://xyz.com/MySwing_1.0_xyz_0.1.uiml#vocab. Note that http://xyz.com/MySwing_1.0_xyz_0.1.uiml#vocab must contain a <*template*> element (see Section 7.1) whose id is *vocab.*

```
5. <presentation base="Java_1.3_Harmonia_0.8">
     <d-class name="MySuperCoolButton" used-in-tag="part" .../>
       ...
     </d-class>
   </presentation>
```

UIML document must be rendered by a rendering engine implementing vocabulary uiml.org/toolkits/Java_1.3_Harmonia_0.8.uiml, but the actual vocabulary used in this UIML document is Java_1.3_Harmonia_0.8 augmented by the part class *MySuperCoolButton* defined in the *d-class* element.

6.2.1.2 Labeling Custom Vocabularies with Attribute *source*

It is recommended that if a UIML author uses a custom vocabulary, that he/she creates a new UIML file containing a <template> element whose *id* attribute is "vocab", and whose body is a <presentation> element containing <d-class> elements defining the custom vocabulary. (Unless this recommended practice is followed, the custom vocabulary cannot be reused in multiple UIML documents.) Furthermore, the name of the UIML file should use the following syntax:

```
<custom-vocab-name>_<custom-vocab-version>_<your-organization's-
name>_<your-version-of-vocab>.uiml
```

For example, if you developed a library of classes that extends Java 1.3 Swing, and you call the library "MySwing", and the current version of MySwing is 1.0, and your name is JoeProgrammer, and this is the first UIML file you wrote to define the vocabulary (version 0.1), then you might name the custom vocabulary file *MySwing_1.0_JoeProgrammer_0.1.uiml*. Any UIML documents that use this custom vocabulary should then contain the following element:

```
<presentation
    source="MySwing_1.0_JoeProgrammer_0.1.uiml#vocab"
    base="Java_1.3_Harmonia_1.0"/>
```

6.2.1.3 Permitted Optimization for Rendering Engine

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., Java_1.3_Harmonia_1.0 in the example in Section 2.2.1). 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.

6.2.1.4 Multiple Presentation Elements

A UIML document may contain multiple *<presentation>* elements. However, each element must contain the *id* attribute. A rendering engine selects one of them based on information outside the UIML document (e.g., as a command line option to the rendering engine).

6.2.1.5 Suggested Use of Base Attribute in Authoring Tools

When an authoring tool for UIML opens an existing UIML document, the tool can quickly identify which vocabulary is used in the document from the *base* attribute, perhaps to display an appropriate palette of user interface widgets for further editing of the document (e.g., a palette of Java Swing objects if the file uses the Java 1.3 vocabulary).

6.2.2 Creating a New Vocabulary Using <*presentation*>

This section discusses two vocabulary files, for HTML and Java. Based on these examples, one can define a new vocabulary file. The full vocabularies for HTML and Java used here are available at the following URLs:

- http://uiml.org/toolkits/HTML_3.2_Harmonia_1.0.uiml
- http://uiml.org/toolkits/Java_1.3_Harmonia_1.0.uiml

6.2.2.1 Defining Legal Part Class Names Via <d-class>

HTML Case

To start with, suppose a UIML document contains the following:

```
<uiml>
  <interface>
    ...
      <part class="Button"/>
    ...
  </interface>
  <peers>
    <presentation base="HTML_3.2_Harmonia_1.0"/>
  </peers>
</uiml>
```

The mapping of the *Button* part to HTML is defined by uiml.org/toolkits/HTML_3.2_Harmonia_1.0.uiml, which maps any part of class *Button* to the HTML 3.2 <INPUT> tag:

```
<uiml>
  <template .../>
    <presentation .../>
      <d-class id="Button" used-in-tag="part" maps-type="tag"
                           maps-to="html:INPUT">
        ...
      </d-class>
    </presentation>
  </template>
</uiml>
```

UIML uses a set of elements that start with <*d-...*>. The "d-" prefix means that this element *defines* a class, property, or parameter name. Thus <*d-class*> defines a class name.

Java Case

In contrast, suppose a UIML document uses a Java JButton:

```
<uiml>
  <interface>
    ...
      <part class="JButton"/>
    ...
  </interface>
  <peers>
    <presentation base="Java_1.3_Harmonia_1.0"/>
  </peers>
</uiml>
```

The mapping of the *JButton* part to Java is defined by uiml.org/toolkits/Java_1.3_Harmonia_1.0.uiml, which maps any part of class *JButton* to the Swing *JButton* class:

```
<uiml>
  <template .../>
    <presentation .../>
      <d-class id="JButton" used-in-tag="part" maps-type="class"
                            maps-to="javax.swing.JButton">
        ...
      </d-class>
    </presentation>
  </template>
</uiml>
```

Note these differences between the HTML and Java <*d-class*> elements:

- The *maps-type* is *tag* for a mapping to a markup language, like HTML, and *class* for a mapping to an imperative object-oriented language like Java.
- The *maps-to* attribute lists a namespace and tag in that namespace ("html:INPUT" for HTML) or a string whose syntax is, at present, not defined in this spec ("javax.swing.JButton" for Java).

Next we will discuss what goes in the ... inside each <d-class> in the above <*presentation*> elements. This part answers the following questions:

- What property names can be used with the part class (e.g. the color of a button)?
- What events and event listeners are used with the part class (e.g. the event of clicking a button)?
- What properties exist for events (e.g. the X and Y positions for a mouse click event)?

These are discussed in turn below.

6.2.2.2 Defining Legal Property Names for <*part*> Classes via <*d-property*>

HTML Case

The UIML document below extends our previous example to give our HTML button a text label that says “Press me!”.

```
<uiml>
  <interface>
    ...
      <part class=”Button”>
        <style><property name=”VALUE”>Press me!</property></style>
      </part>
    ...
  </interface>
  <peers>
    <presentation base="HTML_3.2_Harmonia_1.0"/>
  </peers>
</uiml>
```

Now let’s look at what is required in the <*presentation*> element to map the UIML property *VALUE* to the corresponding *VALUE* attribute of the HTML <*INPUT*> tag. This is defined in the third <*d-property*> element below:

```
<uiml>
  <template .../>
    <presentation .../>
      <d-class      id="Button"  maps-type="tag" maps-to="html:INPUT">

        <d-property id="type" maps-type="attribute" maps-to="TYPE">
          <d-param  type="String">BUTTON</d-param>
        </d-property>

        <d-property id="name" maps-type="attribute" maps-to="NAME">
          <d-param  type="String"/>
        </d-property>

        <d-property id="value" maps-type="attribute" maps-to="VALUE">
          <d-param  type="String"/>
        </d-property>

        ...

      </d-class>
    </presentation>
  </template>
</uiml>
```

The three <*d-property*> elements above say that an HTML 3.2 button has three properties: its type (which is always *BUTTON*, and therefore cannot be set in a UIML document, its name, and its value (or the string text that appears in the button).

Java Case

Now consider the Java JButton. The UIML document below extends our previous example to give our Java Swing button a text label that says “Press me!”.

```
<uiml>
  <interface>
    ...
      <part class="JButton">
        <style><property name="text">Press me!</property></style>
      </part>
    ...
  </interface>
  <peers>
    <presentation base="Java_1.3_Harmonia_1.0"/>
  </peers>
</uiml>
```

Now let’s look at what is required in the *<presentation>* element to map the UIML property *text* to the proper Java set method for javax.swing.JButton. This is defined in the two *<d-property>* elements below:

```
<uiml>
  <template .../>
    <presentation .../>
      <d-class id="JButton" used-in-tag="part" maps-type="class"
                            maps-to="javax.swing.JButton">

        <d-property id="text"
                    maps-type="setMethod"
                    maps-to="setText">
          <d-param type="java.lang.String"/>
        </d-property>

        <d-property id="text"
                    return-type="java.lang.String"
                    maps-type="getMethod"
                    maps-to="getText"/>

        ...
      </d-class>
    </presentation>
  </template>
</uiml>
```

The *<d-property>* elements in the box above say that a property named *text* can be used with JButtons. In a *<property>* element that sets the property, the Java method *setText(java.lang.String)* should be used. To get the property, use *javax.swing.JButton.getText()*.

A *JButton* has many other properties – these are defined by additional *<d-property>* elements (e.g., to set color, font, icon) where the ellipsis appears in the *<presentation>* element in Java_1.3_Harmonia_1.0.uiml.

The box above could also contain <*d-method*> elements. This is useful for exposing methods in the Java class that are not properties (and hence should *not* be accessed in UIML <*property*> elements), but could be invoked via a <*call*> element. For example, java.awt.List contains a method *add(...)* to add another item to a list. An <*action*> in a UIML document might contain a <*call*> to the *add* method.

6.2.2.3 Defining Legal Events and Listeners for Part Classes Via <d-class>

HTML Case

The UIML document below extends our previous example to provide a behavior for an event called *onClick* for our HTML button.

```
<uiml>
  <interface>
    ...
      <part class="Button">
        ...
        <behavior>
          <rule>
            <condition><event name="onClick"/></condition>
            <action>...</action>
          </rule>
        </behavior>
      </part>
    ...
  </interface>
  <peers>
    <presentation base="HTML_3.2_Harmonia_1.0"/>
  </peers>
</uiml>
```

The <*behavior*> element in the box above causes the <*action*> element to be executed whenever an *onClick* event occurs for the button. The meaning of *onClick* is defined by additional lines in the <*presentation*> element:

```
<uiml>
  <template .../>
    <presentation .../>

      <d-class id="onClick" used-in-tag="event"
                            maps-type="attribute"
                            maps-to="onClick"/>

      <d-class id="Button"  used-in-tag="part"
                            maps-type="tag"
                            maps-to="html:INPUT">
        ...
        <event class="onClick"/>

      </d-class>
```

```
    </presentation>
  </template>
</uiml>
```

The above UIML says that one of the events in HTML is *OnClick*, and that a <part> whose class is *Button* can receive an *OnClick* event.

To summarize, when writing rules, the *class* attribute of an *event* element in a *condition* can be any of the names listed in an <event> element that is associated with the <part> in the <presentation> section. In the UIML example above, a *Button*'s events include *OnClick*, and therefore the following line was used in the UIML document given earlier:

```
<condition><event name="onClick"/></condition>
```

Java Case

Java provides a much richer UI toolkit than HTML, and so the information required in the *<presentation>* element is more complex.

There are two styles of events used by platforms:

- *Method 1:* A <part> defines how events for it are handled.
- *Method 2:* A <part>*does not* define how events for it are handled.

Method 1 is used in Java 1.0 and in HTML. In Java 1.0, a java.awt.Component had a method called *action(Event, Object)*. This method was called when an event occurred for the Component. In HTML, many tags (representing <part>s in UIML) can contain attributes denoting events. An example in HTML is <input type="Button" onClick="myfunction"/>. In this case the <part> defines the event handling (calling *myfunction()*). In UIML, this association between the event *onClick* and *myfunction* is made by a <rule>.

Method 2 is used in Java 1.1 and later. UIML adopts the Java model of requiring a *Listener* entity that defines how events are handled on behalf of a <part>. (If there exists a target language for UIML that uses Method 2 and also uses a concept entirely different than listeners, then UIML will need modification.) In this case, a *Listener* class defines how events are handled. In UIML, the association between the event *mouseClicked* and the actions to perform in response is made by a <rule>. The UIML user is not aware of the Listener or Event classes involved.

Thus UIML must contain a rich enough syntax to define the following:

- Class names representing events (such as "OnClick" for HTML, or "MouseEvent" for Java)
- Class names representing event listeners for Method 2 (such as MouseListener)
- Methods in event listeners (such as MouseListener.mouseClicked())
- A means to associate components, listeners, and events.
- Event property names (such as "X" and "Y" coordinates for a MouseEvent)

Next we show what these look like for clicks on a JButton.

The UIML document below extends our previous example to provide a behavior for an event called *ActionListener.actionPerformed* for our Java JButton.

```
<uiml>
  <interface>
    ...
      <part class="JButton">
        <behavior>
          <rule>
            <condition>
              <event class="ActionListener.actionPerformed">
            </condition>
            <action>
              ...
            </action>
          </rule>
        </behavior>
      </part>
    ...
  </interface>
  <peers>
    <presentation base="Java_1.3_Harmonia_1.0"/>
  </peers>
</uiml>
```

Here are the elements in *<presentation>* in Java_1.3_Harmonia_1.0.uiml to define the event *actionPerformed* for a button click:

```
<uiml>
  <template .../>
    <presentation .../>

<!-- ================= Define Event Classes ====================== -->
      <d-class     id="ActionEvent" used-in-tag="event"
                   maps-type="class"
                   maps-to="java.awt.event.ActionEvent">
        ...
      </d-class>
      ...

<!-- ================= Define Event Listener Classes ============== -->

      <d-class     id="ActionListener" used-in-tag="listener"
                   maps-type="class"
                   maps-to="java.awt.event.ActionListener">

        <d-method  id="actionPerformed" maps-to="actionPerformed">
          <d-param id="event" type="ActionEvent"></d-param>
        </d-method>

      </d-class>

<!-- ================= Define Part Classes ======================= -->
```

```
          <d-class     id="JButton" used-in-tag="part" maps-type="class"
                                    maps-to="javax.swing.JButton">

            <listener  class="java.awt.event.ActionListener"
                       attacher="addActionListener"/>

          </d-class>

        </presentation>
      </template>
    </uiml>
```

The section under the comment “Define Event Classes” defines a class named *ActionEvent*. The children of <d-class id="ActionEvent"> are discussed in Section 6.2.2.4. This is analogous to the <d-class id=”onClick”> element in the HTML <presentation> earlier.

The section under the comment “Define Event Listener Classes” uses the <d-class ... used-in-tag="listener"> element to define a Java event listener. This section has no analog in the HTML <presentation> earlier, because the HTML event model ("Method 1") does not use listeners.

The section under the comment “Define Part Classes” was shown in our earlier examples, and defines the *JButton* class. Here we add *<listener>* elements to list which listeners are used with a JButton. The *<listener>* element includes an "attacher" attribute that names the method that the JButton uses to attach the listener to itself. For example, the UIML above says that JButton has a method called "addActionListener" that is used to attach an ActionListener to a JButton.

To summarize, the above UIML says that one of the events in Java AWT/Swing is *ActionEvent*, one of the listeners is *ActionListener*, *ActionListener* has a method named *actionPerformed* that processes *ActionEvent*s, and a listener named ActionListener handles events for any <part> whose class is JButton.

Recall the following lines from the UIML document earlier:

```
<condition>
  <event class="ActionListener.actionPerformed">
</condition>
```

When writing rules, the *class* attribute of an *event* element in a *condition* uses a dotted notation consisting of the form <listener class name>.<method class name>. The <listener class name> is a class name defined by a <d-class used-in-tag="listener"> element, such as *ActionListener* in the UIML example above. The <method class name> is a method name defined in a <d-method> element that is a child of the <d-class used-in-tag="listener"> element, such as *actionPerformed* in the UIML above.

Whenever a JButton is clicked, the Java Virtual Machine calls the *actionPerformed* method. The *<rule>* above is therefore executed when *actionPerformed* is called on a JButton.

6.2.2.4 Defining Legal Event Property Names Via <d-class>

Let's consider our UIML document again. Suppose we want to display the x coordinate of the mouse pointer when it is clicked inside a button. The label on the button will be changed to display the x coordinate. The following UIML document accomplishes this:

```
<uiml>
  <interface>
    ...
      <part class="Button">
        <style><property name="VALUE">Press me!</property></style>
        <behavior>
          <rule>
            <condition>
              <event name="MouseListener.mouseClicked"/>
            </condition>
            <action>
              <property name="VALUE">
                <property event-class="MouseListener.mouseClicked"
                          name="X"/>
              </property>
            </action>
          </rule>
        </behavior>
      </part>
    ...
  </interface>
  <peers>
    <presentation base="Java_1.3_Harmonia_1.0"/>
  </peers>
</uiml>
```

Here are the elements in <*presentation*> in Java_1.3_Harmonia_1.0.uiml to define the property named *X* for a mouse click:

```
<uiml>
  <template .../>
    <presentation .../>

      <d-class id="MouseEvent"     used-in-tag="event"
                                   maps-type="class"
                                   maps-to="java.awt.event.MouseEvent">
        <d-method id="source"         maps-to="getSource"
                                      return-type="java.lang.Object"/>
        <d-method id="id"             maps-to="getID"
                                      return-type="int"/>
        <d-method id="clickCount"     maps-to="getClickCount"
                                      return-type="int"/>
        <d-method id="point"          maps-to="getPoint"
                                      return-type="int"/>
        <d-method id="X"              maps-to="getX"
                                      return-type="int"/>
        <d-method id="Y"              maps-to="getY"
                                      return-type="int"/>
        <d-method id="isPopupTrigger" maps-to="getIsPopupTrigger"
                                      return-type="boolean"/>
```

```
      </d-class>
      ...

    </presentation>
  </template>
</uiml>
```

6.3 The <*logic*> Element

DTD

```
<!ELEMENT logic (d-component*)>
<!ATTLIST logic
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (append|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">
```

Description

The <*logic*> element describes how the UI interacts with the underlying application 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 of <logic> Element

Here is an example of the <*logic*> element:

```
<logic>
 <d-component name="Counter"
              maps-to="com.harmonia.example.Counter2">
   <d-method  name="count" return-type="int" maps-to="count"/>
   <d-method  name="reset" return-type="int" maps-to="setCount">
     <d-param name="newVal" type="int"/>
   </d-method>
 </d-component>
</logic>
```

The fragment above says that there is an external object reached by name com.harmonia.example.Counter2. This object is given the name *Counter* in the UIML document. The component has two methods, named *count* and *reset* in the UIML document. These map, respectively, to *count* and *setCount* in com.harmonia.example.Counter2. Each returns a value of type *int*, where the meaning of *int* is whatever meaning ascribed by the language in which *com.harmonia.example.Counter2* is implemented. Finally, *count* takes no arguments when called, and *setCount* takes one argument, an *int*.

Example

The following UIML fragment describes the calling conventions for a variety of functions in external application logic and functions in scripts. Note that URLs given below are for example purposes only.

```
<logic>

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

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

    <d-method name="m2" returns-value=”true” maps-to="m2"/>

    <d-method name="master" returns-value=”true” maps-to="m3">
      <d-param name="p3"/>
    </d-method>

  </d-component>

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

  <d-component name="S1">

    <d-method name="m1" returns-value=”true” maps-to="Cube">
      <d-param name="i"/>

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

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


  </d-component>
```

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

</logic>
```

6.4 Subelements of <*presentation*> and <*logic*>

6.4.1 The <*d-component*> Element

DTD

```
<!ELEMENT d-component (d-method)*>
<!ATTLIST d-component
          id       NMTOKEN                     #REQUIRED
          source   CDATA                       #IMPLIED
          how      (append|cascade|replace)    "replace"
          export   (hidden|optional|required) "optional"
          maps-to  CDATA                       #IMPLIED
          location CDATA                       #IMPLIED>
```

Description

The <*d-component*> (a child of <logic> only) acts as a container for application methods (e.g., a class in an object oriented language). A d-*component* contains d-*methods*.

The *maps-to* attribute specifies the platform-specific type of the component or container that is being bound. The *location* attribute gives additional information (e.g., a URI) that is used by the rendering engine to locate the widget, event, or application class at runtime.

6.4.2 The <*d-class*> Element

DTD

```
<!ELEMENT d-class (d-method*, d-property*, event*, listener*)>
<!ATTLIST d-class
          id           NMTOKEN                     #REQUIRED
          source       CDATA                       #IMPLIED
          how          (append|cascade|replace)    "replace"
          export       (hidden|optional|required) "optional"
          maps-to      CDATA                       #REQUIRED
          maps-type    CDATA                       #REQUIRED
          used-in-tag  (event|listener|part)       #REQUIRED>
```

Description

The <*d-class*> (a child of <presentation> only) 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.

The *maps-to* attribute specifies the platform-specific type of the component or container that is being bound.

6.4.3 The <*d-property*> Element

DTD

```
<!ELEMENT d-property (d-method*,d-param*)>
<!ATTLIST d-property
          id          NMTOKEN                                #REQUIRED
          maps-type   (attribute|getMethod|setMethod|method) #REQUIRED
          maps-to     CDATA                                  #REQUIRED
          return-type CDATA                                  #IMPLIED>
```

Description

The <*d-property*> element specifies the mapping between the name appearing in a <*property*> element and the associated methods that assign or retrieve a value for the property.

Example

```
<peers>
  <presentation>
    <d-component name="button" maps-to="java.awt.Button">
      <d-property name="Color">
        <d-method returns-value="true" maps-to="getColor"/>
        </d-method>
        <d-method maps-to="setColor">
          <d-param name="color"/>
        </d-method>
      </d-property>
      ...
    </d-component>
  </presentation>
</peers>

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

6.4.4 The *<d-method>* Element

DTD

```
<!ELEMENT d-method (d-param*, script?)>
<!ATTLIST d-method
   id            NMTOKEN                      #REQUIRED
   source        CDATA                        #IMPLIED
   how           (append|cascade|replace)     "replace"
   export        (hidden|optional|required)   "optional"
   maps-to       CDATA                        #REQUIRED
   return-type   CDATA                        #IMPLIED>
```

Description

The *<d-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* gives the name of a 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 *<d-method>* element).

If the method described returns a value, the *returns-value* attribute should be specified and assigned the value `“true”.` The return value will be converted to a string before being used. Otherwise, the *returns-value* attribute defaults to `“false”`, and any value that might be returned by the method is discarded.

The *<d-method>* element supports three different execution models:

1. The method represents a remote (outside the rendering engine) executable code. This code executes outside the run-time context of the rendering engine and is treated as a black box. The rendering engine packages all the parameters, sends them to the server executing the code (which can be on the same machine or across the network), and waits for a reply. Here is an example:

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

2. The method represents a local script. This script is embedded inside the method and is executed within the run-time context of the rendering engine (i.e., it executes locally with respect to the rendering engine). If the *maps-to* attribute for the component is missing, this means that all the code is local. Here is an example:

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

6.4.5 The <*d-param*> Element

DTD

```
<!ELEMENT d-param (#PCDATA)>
<!ATTLIST d-param
          id   NMTOKEN #IMPLIED
          type CDATA   #IMPLIED>
```

Description

Describes a single formal parameter of the function described by the parent <*d-method*> element. Note that all parameters are character strings. The string value of a matching <d-param> element will be converted to a platform-specific data type specified by the *type* attribute, and that type is the type of the formal parameter of the function (e.g., java.lang.String). It is up to some intermediary to convert parameters from UIML character strings to other data types. For example, if we have

```
<d-param>37</d-param>
```

which is mapped to the parameter of function f(double) in this Java class

```
public class Demo {
    static void f(double);
}
```

then string “37” is converted by some intermediary to type double in Java.

Furthermore, if there is ambiguity in which function of the target language a parameter maps to, the rules of the target language are used to resolve the ambiguity. For example, suppose class Demo contains two functions *f* as follows:

```
public class Demo {
    static void f(double);
    static void f(float);
}
```

In this case the rules of Java would determine whether string “37” would be converted to a double or to a float. (Note: The semantics of Java are to use the method “f(float)”. See *Java Developer Connection(sm) (JDC) Tech Tips*, March 14, 2000, http://developer.java.sun.com/developer/TechTips/2000/tt0314.html for more information on this aspect of Java’s semantics.)

See Section 5.7.15 on the significance of parameter order.

6.4.6 The <*script*> Element

DTD

```
<!ELEMENT script (#PCDATA)>
<!ATTLIST script
          id     NMTOKEN                    #IMPLIED
          type   NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (append|cascade|replace)   "replace"
          export (hidden|optional|required) "optional">
```

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)

7 Reusable Interface Components

UIML *templates* enable interface implementers to design parts of their UI (or even the entire UI itself) as reusable components to be exploited by other UIs. 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. This is desirable since end-users tend to make fewer mistakes and are more efficient when presented with familiar UIs.

7.1 The <*template*> Element

DTD

```
<!ELEMENT template (behavior|d-class|d-
component|constant|content|interface|logic|part|peers|presentation|proper
ty|restructure|rule|script|structure|style)>
<!ATTLIST template
           id 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
- cascading style and other elements, in a manner analogous to the CSS specification **[2]**.

*Template*s work as follows. Most elements (see 5.2.2 for a list) can contain the *source* attribute; let such an element be *E*. The *source* attribute names a <*template*> element (either within the same document or in another document). The <*template*> element named must be 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 7.2.

Simple Example Using the *source* Attribute

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
  "-//Harmonia//DTD UIML 3.0 Draft//EN" http://uiml.org/dtds/UIML3_0a.dtd">
<uiml>
  <peers>
    <presentation base="Java_1.3_Harmonia_1.0"
      source="http://uiml.org/toolkits/Java_1.3_Harmonia_1.0.uiml#vocab"/>
  </peers>
  ...
</uiml>
```

The <*presentation*> element contains a *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/JavaAWT_1.3_Harmonia_1.0.uiml*" in turn contains the following:

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC
  "-//UIT//DTD UIML 3.0 Draft//EN" http://uiml.org/dtds/UIML3_0a.dtd">
<uiml>
  <template id="vocab">
    <presentation base="Java_1.3_Harmonia_1.0">
      <d-class id="Frame" used-in-tag="part"
                          maps-type="class"
                          maps-to="java.awt.Frame">
         ...
      </d-class>
      ...
    </presentation>
  <template>
</uiml>
```

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

Also note that the <!DOCTYPE …> element is included in the template file, just as it is in the sourcing UIML document. The <!DOCTYPE …> is required in the template, because tools that process UIML files may process template files separately from a document which sources the template, and such a tool may use a validating XML parser that requires the DTD. Including <!DOCTYPE> in all UIML files has a second benefit: it permits discovery of mismatches between the DTD version used in a template file and a sourcing file.

If the source attribute is not a URL but a relative path and file name or just a file name, then a rendering engine must first look for the source file relative to the directory in which the sourcing document was obtained. Thus, suppose that a UIML file *X* was located in the directory C:\uimlFiles. Any source attribute specifying a relative file name as its value would be resolved relative to C:\uimlFiles.

7.2 Rules for Templates

In the example of Section 7.1, 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 [13]). 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 branches are joined). The interface implementer has three choices on *how* to combine the <*template*> element with another element.

The methods are:

- replace,
- union, and
- cascade

When using the first choice, "*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). With "*union*," 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 to the list as well (see Figure 2). In both cases, the children of the <template> element are given a fully qualified id, to distinguish them from children of the sourcing element. This fully qualified id is constructed by identifying the child's location in the UIML tree. Thus the id is generated by starting with the `<uiml>` element and tracing "downward" through the tree. The original *id* of the element will be pre-pended with the *id* of every element above it in the UIML tree. (e.g., id = "*<interface id>__<structure id>__<grandparent id>__<parent id>__<original id>*"). This rule applies to any element irrespective of whether the *id* attribute is specified or not. This is because the DOM assigns an *id* value for elements for which an *id* is not specified. To avoid conflicts with the naming conventions other languages use, "__" has been chosen as delimiter *for id* appending.

The last choice is "*cascade*." This is similar to what happens in CSS [2]. 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 id), then the element on the main tree is retained (see Figure 3). See section 7.2.3 for a more detailed explanation of conflict resolution and deep cascading.

Note that the figures below use the old naming convention which separate pieces of the fully qualified name with '.' Instead of '_'.

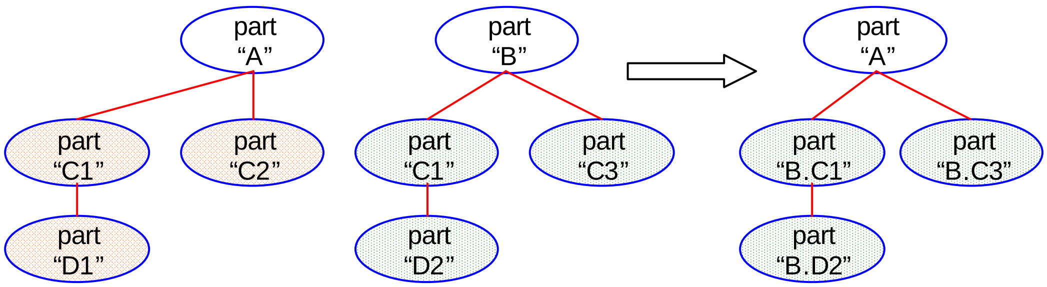


Figure 1: Part "A" sources part "B" using "replace"

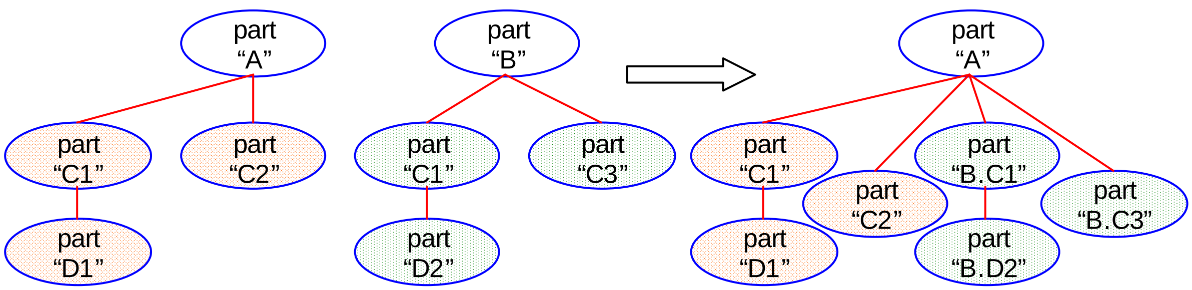


Figure 2: Part "A" sources part "B" using "union"

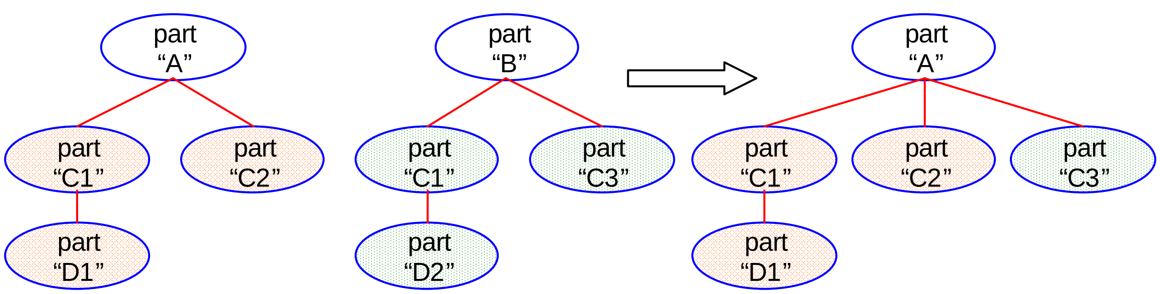


Figure 3: Part "A" sources part "B" using "cascade"

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

7.2.1 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> element. The element that sources the template can also include a default implementation of the element inside the template. If the template is for some reason inaccessible (e.g., network problems), then the rendering engine can ignore the template and still render 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

UIML enables the interface implementer to build a library of reusable interface components, and then include them as needed in new UIML documents. In the following UIML fragment, a dialog box defined in the <template> named DialogBox 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 id="DialogBox">
  <part id="TopLevel">
    <part id="CompanyLogo" class="ImageContainer"/>
    <part id="Message" class="Label"/>
    <part id="Accept" class="Button"/>
  </part>
</template>

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

      <!-- Default implementation -->

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

This example demonstrates how a template can be used to represent a complex re-usable interface object. Here the template represents what is commonly called an "OK Box". This is a simple dialog that contains a message and a button to hide or close the message box. The construct is complex in that it is made up of multiple <part> elements yet represents a commonly used interface object. This particular "OK Box" has been customized to hold the company logo as well as the message and button. Since the part sources the template using how="*replace*" the body of the sourcing element (represented in the example by <!--Default Implementation -->) will be removed and the body of the part in the template will become the body of the sourcing part.

7.2.2 Combine Using Union

Runtime behavior varies significantly from device to device. However, on the same device different platforms may share 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 "*union*" as the value for the *how* attribute, the UIML parser will append the list of common behavior rules to the <behavior> element in the main document.

Example

The following example shows how to reuse behavior rules:

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

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

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

This example demonstrates that by using templates UI designers can specify a set of operations common across platforms on a device or even possibly across devices. Here the template defines a set of rules for common desktop UI interactions, namely mouse movements and button clicks. The UI implementer then sources the template using how="union" to add the rules defined in the template to their custom <behavior> element. In this case, the represented UI running on a X-Windows platform would have a <behavior> element consisting of three rules: a rule for mouse movements, a rule for button clicks, and a rule for middle mouse button clicks. Similarly, the UI running on a MS-Windows platform would have a <behavior> element with a rule for mouse movements, a rule for button clicks, and a rule for window closing. Notice that the UIML author only had to define the rules for the common interactions once.

7.2.3 Combine Using Cascade

When using several UIML files to define a user interface, a UIML file may source other UIML files to include other parts not already within the interface defined in the current UIML file. The sourced file may in turn source more files for more extensions to the set of interface components, thereby increasing the richness of the presented interface. Through the use of the cascade value of the attribute *how*, this ability to include any other parts from external sources can be achieved.

Style is what dictates how an interface *looks* and *feels*. Many companies want the interfaces of their applications to share a common look and feel 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, and they want the menus to have a special structure (e.g., File, Edit, View, etc…), etc. UIML allows this and more. All the common style information can be specified in a template and then included in each of 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 id="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 id="MyStyle" source="#Graphical" how="cascade">
    <property name="content" part-name="myAbout"
      >About: Harmonia, Inc.</property>
  </style>
</interface>
```

The example above contains a template that defines the style for all parts of the class "ADialog". The 'rendering' property given in the template indicates that "Adialog" will be rendered as a 'Dialog', which is mapped to some interface widget in the target vocabulary. The rest of the properties defined in the template's <style> set the TitleColor property to "blue", set the TitleFont property to "Arial", and set the content of the ADialog to "About: UIT". When the template is sourced in the interface, each of these properties will be applied to the part element "myAbout". However, since the content property has also been set in the style section of the interface and the template was sourced using how="cascade", the value assigned to content in the template will be overwritten. Thus the result is that myAbout will be rendered as a "Dialog", the TitleColor property will be set to "blue", the TitleFont property will be set to "Arial", and the content will read "About: Harmonia, Inc."

Elements are said to be conflicting if one of two conditions is true:

- The source and target element has the same value for the *id* attribute.
- Neither the source nor the target element has an author-specified *id* and they both have the same value for their *class* attribute.

Under the rules for cascade, if an element in the sourcing document conflicts with an element of the target document the original element takes precedence over the target element. This allows UIML authors to overwrite values being sourced from the template.

However, the depth at which this comparison between elements takes place is important. In the UIML 2.0 specification, only the top-level elements in the sourced and sourcing file are compared. Any children of those elements are not compared. In this specification, the how="cascade" attribute is interpreted to propagate down the element-tree for which it was specified for. If there is a conflict between the top-level elements, any children of the sourcing element will also be compared. In turn, grandchildren of the sourcing element will also be compared if there is a conflict between the child elements and so on. These comparisons would continue till the conflicting elements are all resolved. The purpose of this is to compare elements from a bottom-up perspective rather than a top- down perspective.

7.3 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's attributes (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."

Unfortunately this presents the possibility that a template may directly or indirectly source itself, causing a sourcing cycle. It is an error if a template directly or indirectly sources itself. The following example demonstrates how a template can indirectly source itself:

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

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

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

In the example above, template "A" sources template "B" which in turn sources template "C". Template "C" then sources template "A", forming a sourcing cycle (A->B->C->A). Such a cycle cannot be resolved without special assumptions and must produce an error.

7.4 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 children can be optionally modified.

UIML allows the encapsulation of the elements inside a template by controlling what is visible and what is not with the *export* attribute. Setting the *export* attribute to “optional” replicates the default behavior, meaning that the element can be accessed (remains visible) by elements outside the <*template*> element. Any element inside the template with the *export* attribute set to “hidden” cannot have its properties changed outside the template, and a rendering engine must generate an error if an attempt is made to change a hidden property outside the template. Also, any element with the *export* attribute set to “required” must be assigned a value before the template can be rendered.

The semantics of <part … export=“hidden”> mean that *no* properties of the part listed in the template may be modified outside the template. The semantics of <part … export=“required”> mean that *all* properties given in the template of the part must be defined outside the template.

The semantics of <part><property name=“x” export=“hidden”/></part> mean that property *x* cannot be modified outside the template, regardless of whether the <*part*> element has an *export* attribute. The semantics of <part><property name=“x” export=“required”/></part> mean that property *x* must be defined outside the template, regardless of whether the <*part*> element has an *export* attribute.

Example

In the following template, a message box has three parts. It requires 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 id="MyDialog">
  <part id="TopLevel">
    <part id="MyLogo" class=”Logo” export="hidden"/>
    <part id="MyMessage" class=”Label”>
      <style>
        <property name="content" export="required"/>
      </style>
    </part>
    <part id="Ok" class=”OKButton” export="hidden"/>
  </part>
</template>
```

8 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 rendering engine can start rendering before an entire UIML document is received to reduce latency for an end-user in large UIML documents.

The DTD presented in Appendix A permits these combinations. Refer to the DTD for precise information on what organizations are legal.

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:

8.1 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
    "-//Harmonia//DTD UIML 3.0 Draft//EN"
    "http://uiml.org/dtds/UIML3_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>
```

8.2 UIML Template Mechanism

Using the *<template>* element a UIML document can be broken down into multiple pieces (as explained in Section 7.1). 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 7.4). 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 rendering engine 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] T. Bray, et al, Extensible Markup Language (XML), W3C Proposed Recommendation 10-February-1998, REC-xml-19980210, February 10, 1998, http://www.w3.org/TR/REC-xml.

[2] 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/.

[3] 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.

[4] Marc Abrams, Constantinos Phanouriou , *UIML: An XML Language for Building Device-Independent User Interfaces*, XML '99, Philadelphia, Dec. 1999.

[5] Constantinos Phanouriou, UIML: A Device-Independent User Interface Markup Language, http://scholar.lib.vt.edu/theses/available/etd-08122000-19510051/unrestricted/PhanouriouETD.pdf

[6] UIML1.0 specification, http://www.uiml.org/specs/uiml1/index.htm, 1997.

[7] UIML2.0 specification, http://www.uiml.org/specs/uiml2/DraftSpec.htm, 2000.

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

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

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

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

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

[13] W3C, XForms – The Next Generation of Web Forms, 31 January 2002, http://www.w3.org/MarkUp/Forms/

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

[15] XIML Forum, extensible Interface Markup Language (XIML), http://www.ximl.org/

[16] R. Cover, The XML Cover Pages Extensible User Interface Language (XUL), http://www.oasis-open.org/cover/xul.html, August, 2000.

[17] Harmonia, Inc., *UIML-Java rendering engine User Manual*, http://www.harmonia.com/products/java/manual.htm

[18] Harmonia, Inc., *UIML Example Gallery*, http://www.harmonia.com/products/java/gallery/index.htm; http://www.harmonia.com/products/html/examples.htm; http://www.harmonia.com/products/wml/examples.htm; and http://www.harmonia.com/products/voice/gallery.htm

Appendix A. UIML 3.0 Document Type Definition

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

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

    Developed by:

        Harmonia, Inc.

    Usage:

        <?xml version="1.0"?>
        <!DOCTYPE uiml PUBLIC "-//Harmonia//DTD UIML 3.0a Draft//EN"
        "http://uiml.org/dtds/UIML3_0a.dtd">
        NOTE: This URL has not yet been activated.

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

    Description:

        This DTD corresponds to the UIML 3.0a specification.

    Change History:


-->

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

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

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

<!--
    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
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">

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

<!ELEMENT interface (structure|style|content|behavior)*>
<!ATTLIST interface
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">

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

<!ELEMENT template (behavior|constant|content|d-class|d-
component|interface

|logic|part|peers|presentation|property|restructure|rule
                   |script|structure|style)>
<!ATTLIST template
          id NMTOKEN #IMPLIED>

<!-- Peer related elements -->

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

<!ELEMENT presentation (d-class*)>
<!ATTLIST presentation
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          base   CDATA                      #REQUIRED
          how    (union|cascade|replace)    "replace"
```

```
          export (hidden|optional|required) "optional">

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

<!ELEMENT logic (d-component*)>
<!ATTLIST logic
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">

<!--
    The 'd-component' element maps the name used in a <call> element to
    application logic external to the UIML document (e.g., a class in
an
    object oriented language or a function in a scripting langauge).
-->

<!ELEMENT d-component (d-method)*>
<!ATTLIST d-component
          id       NMTOKEN                    #REQUIRED
          source   CDATA                      #IMPLIED
          how      (union|cascade|replace)    "replace"
          export   (hidden|optional|required) "optional"
          maps-to  CDATA                      #IMPLIED
          location CDATA                      #IMPLIED>

<!--
    Maps class names that can be used for parts and events, as
    well as property and event data names, to UI toolkit.
-->

<!ELEMENT d-class (d-method*, d-property*, event*, listener*)>
<!ATTLIST d-class
          id          NMTOKEN                    #REQUIRED
          source      CDATA                      #IMPLIED
          how         (union|cascade|replace)    "replace"
          export      (hidden|optional|required) "optional"
          used-in-tag (event|listener|part)      #REQUIRED
          maps-type   (attribute|tag|class)      #REQUIRED
          maps-to     CDATA                      #REQUIRED>

<!--
   Maps a property name to methods in UI toolkit that get and
   set property’s value.
-->

<!ELEMENT d-property (d-method*, d-param*)>
<!ATTLIST d-property
          id            NMTOKEN
#REQUIRED
          maps-type     (attribute|getMethod|setMethod|method)
#REQUIRED
```

```
          maps-to       CDATA
#REQUIRED
          return-type   CDATA
#IMPLIED>

<!--
   Maps a method to a callable method or function in the API of
   the application logic.
-->

<!ELEMENT d-method (d-param*, script?)>
<!ATTLIST d-method
          id              NMTOKEN                     #REQUIRED
          source          CDATA                       #IMPLIED
          how             (union|cascade|replace)     "replace"
          export          (hidden|optional|required)  "optional"
          maps-to         CDATA                       #REQUIRED
          return-type     CDATA                       #IMPLIED>

<!--
    Defines a single formal parameter to a <d-method>.
-->

<!ELEMENT d-param (#PCDATA)>
<!ATTLIST d-param
          id   NMTOKEN #IMPLIED
          type CDATA   #IMPLIED>


<!--
    The 'script' element contains executable script code. The type
    specifies the scripting language (see HTML4.0).
-->

<!ELEMENT script (#PCDATA)>
<!ATTLIST script
          id     NMTOKEN                    #IMPLIED
          type   NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">


<!-- Interface related elements -->

<!--
    The 'structure' element describes the initial organization of the
    parts that comprise the user interface.
-->

<!ELEMENT structure (part*)>
<!ATTLIST structure
          id     NMTOKEN                    #IMPLIED
          source CDATA                      #IMPLIED
          how    (union|cascade|replace)    "replace"
          export (hidden|optional|required) "optional">
```

```
<!--
    Specifies a single abstract part of the user interface.
-->

<!ELEMENT part (style?, content?, behavior?, part*, repeat*)>
<!ATTLIST part
          id         NMTOKEN                     #IMPLIED
          class      NMTOKEN                     #IMPLIED
          source     CDATA                       #IMPLIED
          where      (first|last|before|after)   "last"
          where-part NMTOKEN                     #IMPLIED
          how        (union|cascade|replace)     "replace"
          export     (hidden|optional|required)  "optional">

<!--
    A 'repeat' element encapsulates a sub-tree of the overall interface
    virtual tree to be repeated 0 or more times. Each repeat MUST
    have one 'iterator' child.
-->

<!ELEMENT repeat (iterator,part*)>
<!ATTLIST repeat EMPTY>

<!--
    An 'iterator' defines how many times a sub-tree should be repeated
    in an interface and serves as an indicator of the current
iteration.
-->

<!ELEMENT iterator (#PCDATA|constant|property|call)>
<!ATTLIST iterator
          id     NMTOKEN                     #REQUIRED>

<!--
    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
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (union|cascade|replace)     "replace"
          export (hidden|optional|required)  "optional">

<!--
    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| op|
event| iterator)*>
<!ATTLIST property
          name        NMTOKEN                     #IMPLIED
          source      CDATA                       #IMPLIED
          how         (union|cascade|replace)     "replace"
          export      (hidden|optional|required)  "optional"
          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 #IMPLIED
          url-name       NMTOKEN #IMPLIED>

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

<!ELEMENT content (constant*)>
<!ATTLIST content
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (union|cascade|replace)     "replace"
          export (hidden|optional|required)  "optional">

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

<!ELEMENT constant (constant*)>
<!ATTLIST constant
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (union|cascade|replace)     "replace"
          export (hidden|optional|required)  "optional"
          model  CDATA                       #IMPLIED
          value  CDATA                       #IMPLIED>

<!--
    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
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (union|cascade|replace)     "replace"
          export (hidden|optional|required) "optional">


<!ELEMENT rule (condition,action)?>
<!ATTLIST rule
          id     NMTOKEN                     #IMPLIED
          source CDATA                       #IMPLIED
          how    (union|cascade|replace)     "replace"
          export (hidden|optional|required) "optional">

<!--
    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|op)>

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

<!ELEMENT op (constant|property|reference|call|op|event)*>
<!ATTLIST op
          name   CDATA                       #REQUIRED>

<!ELEMENT action (((property|call|restructure)*,event?)|(when-
true?,when-false?,by-default?))>

<!ELEMENT when-true ((property|call)*,restructure?,op?,equal?,event?)>

<!ELEMENT when-false ((property|call)*,restructure?,op?,equal?,event?)>

<!ELEMENT by-default ((property|call)*,restructure?,op?,equal?,event?)>

<!ELEMENT restructure (template)?>
<!ATTLIST restructure
          at-part NMTOKEN                           #IMPLIED
          how    (union|cascade|replace|delete) "replace"
          where  (first|last|before|after)      "last"
          where-part NMTOKEN                        #IMPLIED
          source  CDATA                             #IMPLIED>

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

<!--
    'event' denotes one of three things:
    (1) When a child of <condition> or <op>, denotes that when the
named
        event is fired, the condition should be evaluated.
    (2) When a child of <action>, denotes that the named event should
```

```
        be fired.
    (3) Inside <d-class>, denotes that the named event can occur for
        the part class named by the <d-class>.
-->

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

<!--
    'param' denotes a single actual parameter to a call-able routine.
-->

<!ELEMENT param (#PCDATA|property|reference|call|op|event|iterator)*>
<!ATTLIST param
          name NMTOKEN #IMPLIED>

<!--
    'listener' denotes that a name defined with d-class
    used-in-tag="listener" should be attached as a listener to the
    d-class, which contains this <listener> element.
-->

<!ELEMENT listener EMPTY>
<!ATTLIST listener
          class    NMTOKEN #IMPLIED
          attacher CDATA   #IMPLIED>
```

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 when a *condition* of a *<rule>* element is true, the associated *<action>* element is executed. UIML does not define any order on evaluation of *<condition>* elements.

```
// Non-deterministically choose a <rule> element from 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
```

Appendix C. Changes from UIML 2.0a Specification (1/17/2001)

Following are changes from the Document Type Definition given in the 17 Jan 2000 specification (named UIML2_0a.dtd). Elements not listed have not changed.

<*action* element:

<*action*> can now have two possible sets of children: 1. Any number of <*property*>, <*call*>, or<*restructure*> elements followed by one or zero <*event*> elements, or 2. A <*when-true*> element followed by a <*when-false*> element followed by a <*by-default*> element.

<*attribute*> element:

<*attribute*> is no longer a valid UIML element. It has been replaced by <*d-property*>

<*behavior*> element:

Attribute *name* has been replaced by attribute *id*.

<*by-default*> element:

New element in UIML 3.0.

<*component*> element:

<*component*> is no longer a valid UIML element. It has been replace by two elements <*d-class*> and <*d-component*> for use in <*presentation*> and <*logic*> respectively.

<*condition*> element:

<*op*> is now a valid child of <*condition*>

<*constant*> element:

Attribute *name* has been replaced by attribute *id*.

#PCDATA is no longer a valid child of <*constant*>. Instead the attribute *value* has been added.

Attribute *model* added

<*content*> element:

Attribute *name* has been replaced by attribute *id*.

<*d-class*>

New element in UIML 3.0.

<*d-component*> element:

New element in UIML 3.0.

<*d-method*>

New element in UIML 3.0.

<*d-param*> element:
New element in UIML 3.0.

<*d-property*> element:
New element in UIML 3.0.

<*equal*> element:
<*op*> is now a valid child of <*equal*>

<*interface*> element:
Attribute *name* has been replaced by attribute *id*.

<*listener*> element:
New element in UIML 3.0.

<*logic*> element:
<*component*> is no longer a valid child of <*logic*>. It has been replaced by <*d-component*>.
Attribute *name* has been replaced by attribute *id*.

<*method*> element:
<*method*> is not longer a valid UIML element. It has been replaced by <*d-method*>.

<*op*> element
New element in UIML 3.0.

<*param*> element:
This element has been split into two elements: <*param*>, which is used with the <*call*> element; and <*d-param*>, which is used with the <*d-method*> element.
#PCDATA, <*property*>, <*reference*>, <*op*>, <*event*>, and <*call*> are now valid children.

<*part*> element:
<*repeat*> is now a valid child of <*part*>.
Attribute *name* has been replaced by attribute *id*.
Attributes *where* and *where-part* have been added for use with <*restructure*>.

<*peers*> element:
Attribute *name* has been replaced with the attribute *id*.

<*presentation*> element:
<*component*> is no longer a valid child of <*presentation*>. It has been replaced by <*d-class*>.
Attribute *name* has been replaced by attribute *id*.

Attribute *base* has been added.

<*property*> element:
Attribute *name* has been replaced by attribute *id*.
<*op*>, <*event*>, and <*iterator*> are now valid children of <*property*>.

<*reference*> element:
Attribute *url-name* has been added

<*restructure*>
New element in UIML 3.0.

<*returns*> element:
<*return*> is no longer a valid UIIML element. Instead, use the *returns-value* attribute in element <*d-method*>.

This change was made because the <*returns*> element was redundant with the *type* attribute in the <*d-method*> element. The *type* attribute has been replaced by attribute *returns-value,* hence <*returns*> is replaced by *returns-value*.

<*rule*> element:
Attribute *name* has been replaced by attribute *id*.

<*script*> element:
Attribute *name* has been replaced by attribute *id*.

<*structure*> element:
Attribute *name* has been replaced by attribute *id*.

<*style*> element:
Attribute *name* has been replaced by attribute *id*.

<*system*> element:
<*system*> is no longer a valid UIML element.

<*template*> element:
<*component*> is no longer a valid child of <*template*>
<*d-class*>, <*d-component*>, and <*restructure*> are now valid children of <*template*>
Attribute *name* was replaced by attribute *id.*

<*uiml*> element:
The DTD has changed to allow any number of <*template*>,<*interface*>, and <*peers*> children in any order.
<*when-false*> element
New element in UIML 3.0.

<*when-true*> element:

New element in UIML 3.0.

General changes in the DTD

append is no longer a valid value for the *source* attribute present on many UIML elements. It has been replaced with *union.*