Platform-Independent Accessibility API: Accessible Document Object Model

Andres Gonzalez
Adobe Systems Inc.
345 Park Avenue
San Jose CA 95110 USA.
+1-408-536-6224
andgonza@adobe.com

Loretta Guarino Reid
Adobe Systems Inc.
345 Park Avenue
San Jose CA 95110 USA.
+1-408-536-2166
lguarino@adobe.com

ABSTRACT
This paper addresses the problem of supporting accessibility in applications that run in multiple operating environments. It analyzes the commonalities of existing platform-specific Accessibility APIs, and defines a platform-independent accessibility API, the Accessible DOM.

The Accessible DOM encompasses the features of existing APIs and overcomes the limitations of existing APIs to express dynamic, complex document contents.

The Accessible DOM can be used to support existing and future platform-specific accessibility APIs. It will also allow the development of platform-independent accessibility clients.

Categories and Subject Descriptors
D.4.m [Operating Systems]: Miscellaneous – Accessibility APIs; H.1.2 [User/Machine Systems]: Human factors; H.5.2 [User Interfaces]: Graphical user interfaces (GUI), Standardization

General Terms
Design, Human Factors, Standardization

Keywords
Accessibility API, W3C DOM

1. INTRODUCTION
When supporting accessibility, cross-platform application developers have to support the platform-specific accessibility API for each operating environment under which their application will run. For instance, Adobe Reader 7.0 supports MSAA under Windows, the Mac Accessibility API under Mac OS X, and ATK under Linux [8].

W4A at WWW2005, 10th May 2005, Chiba, Japan

In addition to supporting multiple APIs, application developers that want to support accessibility often face the problem that the existing platform-specific accessibility APIs are not rich enough to convey the full functionality of their applications UI and the content that they render. This is most evident when supporting accessibility in applications that render documents that can contain complex text layouts, tables, interactive form fields, different types of annotations and multimedia content.

This paper propose an approach based on the definition of an Accessible Document Object Model (Accessible DOM), that addresses both the problem of supporting accessibility in a cross-platform fashion and that will overcome the limitations of existing accessibility APIs. The Accessible DOM is based on the W3C DOM specifications. It builds upon the DOM Level 3 Core specification [11] to define a new module, the accessibility module that contains interfaces specific to accessibility.

The Accessible DOM will provide a superset of the functionality provided by existing platform-specific APIs. Therefore, the Accessible DOM will allow support of any platform-specific API through the implementation of thin-layer adapters that adapt the Accessible DOM interfaces to those of a particular API. The architecture resulting of this approach is illustrated in figure 1.1.

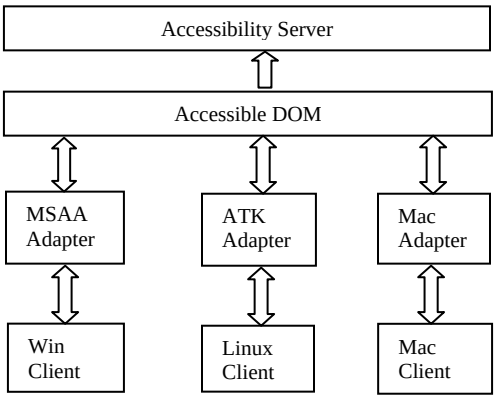

Figure 1.1. Cross-platform accessibility server implements Accessible DOM. Three adapter layers adapt the Accessible DOM to MSAA, Mac Accessibility and ATK. Platform-specific Accessibility clients hook to the respective adapters.

The Accessible DOM will also allow the development of cross-platform accessibility clients, as illustrated in figure 1.2.

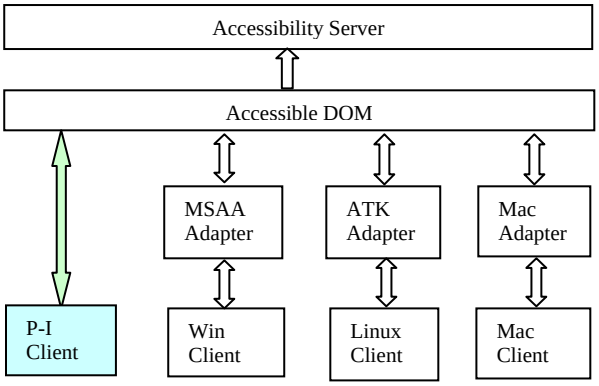

Figure 1.2. Cross-platform accessibility server implements the Accessible DOM. Platform-independent accessibility client hooks directly into the Accessible DOM. Three adapter layers adapt the Accessible DOM to MSAA, Mac Accessibility and ATK. Platform-specific Accessibility clients hook to the respective adapters.

Section 2. describes the common features of existing accessibility APIs and the additional features and benefits that the Accessible DOM provides. Section 3. defines the Accessible DOM, by defining the interfaces that constitute the accessibility module, and the DOM modules required in order to support accessibility. Section 4. illustrates how the Accessible DOM can be adapted to a platform-specific accessibility API.

2. ACCESSIBILITY APIS

2.1 What Is an Accessibility API?

An accessibility API is an application programming interface (API) by which an application (the server) exposes its graphical user interface (UI) and content to another application (the client). Through the accessibility API, the client discovers, represents, and modifies the server's UI and content.

Example of accessibility servers can be any user agent like a web browser or document viewer. Accessibility clients can be assistive technologies, like screen readers or magnifiers, UI automation and testing scripts, or dynamic content scripts like JavaScript scripts.

There are several accessibility APIs for different operating environments:

- Microsoft Active Accessibility (MSAA) [6] for the Windows operating system.
- Java Accessibility API [3], for Java applications and applets running inside a Java VM.
- Linux Accessibility Toolkit (ATK) [4], for the Linux operating system.
- Mac Accessibility API [5], for the Mac OS X operating system.

Accessibility APIs share the following core features:

- The definition of an accessible object.
- A hierarchical tree structure in which accessible objects live.
- A number of events to notify clients of UI or content changes.
- A server-client inter-process communication mechanism.

Other important features supported by some accessibility APIs include:

- Definition of relationships or associations between accessible objects [3, 4].
- A mechanism to simulate keyboard and mouse input.

This section analyzes the common aspects of existing accessibility APIs. It also enumerates the desired features of a new accessibility API, the Accessible DOM.

2.2 Accessible Objects

Accessible objects can represent a UI component such as a button or a menu item. They can also represent a content fragment, like some text in a document, more complex objects like a table or an interactive form field, an entire document or the application itself. They possess properties that describe their nature, state and functionality.

Clients use the accessible object interfaces as the basic means to retrieve information about the server's UI and content. An accessible object that represents an interactive UI component supports interfaces to simulate user's input. It is the server's responsibility to implement and expose the accessible object interfaces for every UI component and content fragment.

2.3 Accessible Object Properties

Some properties are common to all accessible objects. For instance, all accessible objects possess a name or textual description, and a role or type. Other attributes, like value and state, vary in applicability to different types of accessible objects.

The role or type of an accessible object is the attribute that best summarizes the nature and functionality of the object. Examples of types are "button", "menu item", "list" or "document". Knowing the type of an object, a client may determine what other properties are relevant in order to represent the object, and what general behavior the object may exhibit. For example, a "button" is described by its name property and pressed state, and it can be used to invoke an action. On the other hand, a "document" is a container object that may include text, tables, forms, multimedia content, therefore, a much more complex representation is necessary, and a richer behavior is expected. Most accessibility APIs define an "unknown" role for those UI components that don't fit any of the behaviors predefined by the platform. In those cases, clients can only assume that the "unknown" accessible object possesses the generic properties and behaviors common to all accessible objects.

In some accessibility APIs, in addition to a single type property, the nature and functionality of an accessible object is specified by a combination of predefined behaviors or patterns. This approach may allow a more flexible, accurate description of the object. A

"list" object will support the "multi-selectable" behavior, if multiple selected list items can exist at the same time, and the "scrollable" behavior, it not all the list items are visible on the screen.

The value property is relevant for those objects that can have a textual or numeric value, such as an edit field or a slider control. The state property is most pertinent for accessible objects that preserve a state, such as check boxes, radio buttons, or buttons. There are however states that are transient in nature, like "busy" or "downloading", that can be relevant for accessible objects of type, say, document. There are also states applicable to most object types, such as "enabled", "focused", and "visible."

Screen location is another attribute relevant for all accessible objects representing visible UI components or content.

2.4 Accessibility Tree

Accessible objects are contained in a hierarchical tree structure. Each accessible object constitutes a node in the tree. The root node of the tree typically represents the application object itself. An accessible object that is a container for other objects will be the parent node of its contained accessible objects. For instance, the accessible object representing a menu will be the parent node of the menu items. Some accessible objects like a check box or a button have no children.

Child/parent and siblings relationships are used by clients to walk and discover the entire application accessibility tree. It is the responsibility of the server to implement complete navigation throughout the tree.

2.5 Events

Most accessibility API events fall into one of the following three categories:

- Changes to the structure of the accessibility tree. Accessible objects are created, destroyed, or repositioned in the tree.
- Changes to one or more property of an accessible object
- Changes to a global property, like the focused object.

2.6 Relationships

Clients are interested not only in individual objects, but in the relationships between them. The accessibility tree imposes structural relationships. This hierarchy, however, does not express many important relationships. Examples of these relationships are:

- Label for: An object is the label (or prompt) for another object.
- Header for: An object of role "table cell" is the header for another object of role "table row" or "table column".
- Annotation for: An object is a textual comment for another object of role "document fragment".

2.7 Input

Accessibility APIs have different levels of support for simulated mouse and keyboard input. MSAA [6] for example, only allows the client to invoke a default action in a particular accessible object, which in most cases it is equivalent to a left mouse click.

In some cases, it is desirable to allow client input that is specific to the role or type of the target object. For instance for an object of type "list", it would make sense to allow the client to select a specific list item.

2.8 Inter-Process Communication

Accessibility APIs utilize an inter-process communication mechanism to allow clients and servers running in different processes to interact. MSAA [6], for instance, uses a COM based mechanism, while ATK [4] uses CORBA.

It is beyond the scope of this paper to discuss choosing an inter-process communication mechanism that would be most appropriate for a cross-platform accessibility API.

2.9 Features of the Accessible DOM

As seen throughout this section, existing platform-specific accessibility APIs share a common set of features. In the following section, a new accessibility API, the Accessible DOM, is defined. The Accessible DOM will share the common features of existing accessibility APIs and also meet the following requirements:

- It is defined in a platform- and language-neutral way.
- It is extensible. Servers and clients can add accessible object properties, relationships and events as needed in a declarative way.
- It supports complex, dynamic content documents.

3. ACCESSIBLE DOCUMENT OBJECT MODEL

The relationship between developing accessible applications and conformance with the W3C DOM recommendations has been stated in the Guideline 6, Checkpoints 6.2 and 6.9, of the User Agent Accessibility Guidelines 1.0 recommendations (UAAG 1.0) [14], part of the W3C Web Accessibility Initiative [15].

Some of the documentation related to the W3C DOM specification also allude in general terms to the idea that conformance to the DOM will facilitate accessibility. For example, the DOM FAQ [13] states:

"The DOM will make it much easier for accessibility tools to access documents, since they will be able to take a document and feed it into an accessibility-enabled application, such as a screen reader."

The DOM for HTML 4.01 and XHTML 1.0 has been successfully used to support MSAA and ATK in the Mozilla web browser [1]. The accessibility of complex, interactive web applications using JavaScript has been also addressed by leveraging platform-specific accessibility APIs with the functionality provided by the Mozilla's implementation of the DOM for XHTML [2].

This paper formalizes the connection between supporting accessibility and conformance with the DOM specifications. It establishes the relationship between the DOM implementation and an accessibility API by proposing the Accessible DOM as the appropriate framework to define a platform-independent accessibility API.

The Accessible DOM consists of the accessibility module defined in sections 3.1 through 3.3, and a subset of the modules defined

by the W3C DOM specifications. Section 3.6 lists the specific modules that need to be supported in order to provide an Accessible DOM implementation.

The AccessibleNode, AccessibleDocument, Relationship, and Collection interfaces defined in the accessibility module guarantee that the Accessible DOM meets the requirements for an accessibility API put forth in section 2.9. These features, in addition to the features inherited from the rest of the DOM modules comprised by the Accessible DOM, make the Accessible DOM a superset of the existing platform-specific accessibility APIs.

Note: Following the style of the W3C DOM specifications, the accessibility module interfaces will be defined using OMG IDL [7]. See the Appendix for a complete IDL definition of the accessibility module of the Accessible DOM.

3.1 AccessibleNode Interface

The AccessibleNode interface is the fundamental interface that characterizes the Accessible DOM. It contains the basic accessible properties that are common to existing accessibility APIs and that have proven to be useful to accessibility servers and clients.

The Accessible DOM extends the Node interface defined in the DOM level 3 Core specification [11] as follows:

```
interface AccessibleNode : Node {
        attribute DOMString accessibleName;
        readonly attribute DOMString accessibleType;
        attribute DOMString accessibleValue;
        attribute DOMString accessibleState;
        attribute Rect boundingRect;
        Collection getCollection(in DOMString accessibleType);
        attribute RelationshipList relationships;
};
```

The accessibleName attribute contains a string that best describes the node. For instance, for a button element in HTML 4.01, accessibleName is equivalent to HTMLButtonElement.name, while for an image element, it would be more appropriate to use HTMLImageElement.alt [9].

The accessibleType attribute represents the role or behavior of the Node as discussed in section 2.3. For an HTML 4.01 select element, accessibleType would correspond to HTMLSelectElement.type, which can be “select-multiple” or “select-one”, if the element allows multiple or single selection respectively.

The accessibleValue and accessibleState attributes may be null if they don’t apply to a particular AccessibleNode. For example, the accessibleValue of an HTMLButtonElement would be null, but its accessibleState may be “enabled, pressed”.

The boundingRect of type Rect represents the screen coordinates of a bounding rectangle for the visual rendering of an AccessibleNode (see the Appendix for the IDL definition of Rect). If a certain Node is never rendered on the screen or it is invisible at the time, this attribute returns null. In most cases, the user agent would need to compute the screen coordinates of the bounding rectangle based on the DOM implementation’s client area coordinates.

A Collection is a set of descendents of the AccessibleNode with the same accessibleType attribute (see the Appendix for its IDL definition). There is no restriction on the relative locations on the DOM tree between the items in the Collection, i.e., different items can be at different levels of depth in the subtree rooted at the AccessibleNode. For instance, the DOM Level 2 HTML specification defines several collections as attributes of the HTMLDocument object, namely, images, applets, links, forms and anchors. Other HTML Elements also possess Collections, i.e., HTMLFormElement, HTMLMapElement, HTMLTableElement, HTMLTableSectionElement, and HTMLTableRowElement.

The getCollection method returns a Collection of AccessibleNodes with the specified accessibleType or null, if not such a Collection exists.

The relations attribute provides the mechanism to express arbitrary associations between AccessibleNodes. The next section analyzes in detail the definition of the RelationshipList and Relationship interfaces.

In addition to encapsulating the most commonly used accessible properties, the NamedNodeMap attribute inherited from the Node interface makes the AccessibleNode an easily extensible interface. Accessibility servers and clients will be able to exchange a rich set of accessibility properties, not limited by a fixed number of API methods.

The definition of a complete set of accessible attributes, as well as a complete set of values for those attributes, is beyond the scope of this paper. Future work in the definition of the Accessible DOM will define a complete set of values for the AccessibleNode’s accessibleType and accessibleState attributes, as well as a complete set of relevant relationships between AccessibleNodes.

3.2 Relationship and RelationshipList Interfaces

As mentioned in section 2.6, an important aspect that accessibility APIs need to convey is the relevant relationships between objects, beyond the parent-child relationships determined by the structure tree. To accomplish this, the AccessibleNode interface includes the attribute relationships of type RelationshipList, which interface and related types are defined as follows:

```
interface Relationship {
        readonly attribute DOMString name;
        readonly attribute Node owner;
        readonly attribute Node relatedNode;
        readonly attribute Document ownerDocument;
        void        initRelation(in DOMString relationName,
                    in Node ownerNode,
                    in Node relatedNode);
        boolean     isSameRelationship(in Relationship arg);
};
```

where name is the name of the relationship, ownerNode is the owner or subject of the relationship, and relatedNode is the node that relates to ownerNode as described by name.

For instance, let’s assume that n1 is a textual annotation for the document fragment n2 in a PDF document. This may be expressed with the Relationship rel as follows:

```
rel.name = “annotation for”
rel.ownerNode = n2
rel.relatedNode = n1
```

The ownerDocument attribute is the AccessibleDocument node that created the relationship, see the createRelationship method of the AccessibleDocument interface.

The initRelationship method allows the initialization of the Relationship object. Both the ownerNode and relatedNode parameters have to be nodes belonging to the same Document.

The isSameRelationship method determines whether the specified Relationship arg is the same as this Relationship by comparing the name attributes, as specified in the DOM Level 3 Core specification [11] for comparing DOMStrings, and evaluating the Node method isSameNode for both the ownerNode and relatedNode. That is, rel1.isSameRelationship(rel2) returns true if and only if the following expression is true:

```
( rel1.name == rel2.name &&
        rel1.ownerNode.isSameNode(rel2.ownerNode) &&
        rel1.relatedNode.isSameNode(rel2.relatedNode) )
```

```
interface RelationshipList {
        readonly attribute unsigned long   length;
        Relationship item(in unsigned long index);
        RelationshipList namedItems(in DOMString name);
        RelationshipList relatedNodeItems(in Node relatedNode);
        Relationship add(in Relationship arg);
        Relationship remove(in Relationship arg);
};
```

The length attribute and item method provide a way to retrieve a particular Relationship from the set of relationships owned by an AccessibleNode. The namedItems and relatedNodeItems methods are convenient to extract a subset of the relationships owned by a Node that have the same name or the same relatedNode, respectively. The add and remove methods simply allow the addition or removal of a Relationship to the set. Add and remove will raise an exception if the Relationship being added or removed is not owned by the same Node as the existing items in the set.

3.3 AccessibleDocument Interface

The AccessibleDocument interface is implemented by the same object that implements the Document interface, i.e., the root node of the DOM tree. It extends the Document interface by adding the capability of creating relationships and inherits the ability of retrieving relevant Collections from the AccessibleNode interface. It also provides a mechanism for focus and selection tracking.

```
interface AccessibleDocument : AccessibleNode, Document {
        Relationship createRelationship();
        attribute AccessibleNode currentFocus;
        attribute Range currentSelection;
};
```

The createRelationship method provides the mechanism by which both the accessibility server and the client can create a Relationship object. It returns a Relationship object whose ownerDocument attribute is the Document implementing the interface. The name, ownerNode, and relatedNode attributes are null. Therefore, the user of the created Relationship needs to call initRelation before it can be added to any RelationshipList of an AccessibleNode.

The getCollection method inherited from AccessibleNode covers particular significance for the AccessibleDocument object because it would provide a mechanism for easy navigation and summarization of a document. As stated in UAAG 1.0 [14], Guideline 9. “Provide navigation mechanisms”, direct and structured navigation are crucial for accessibility. For example, by retrieving a Collection of all the headings in a document, the user may jump to the desired section without having to go sequentially through the entire content.

The currentFocus and currentSelection attributes correspond to the current user interface focus and current selection concepts defined in UAAG 1.0. Notice that different views of the same document can have different focused objects and selections, but only the current focus and current selection respond to user’s input. Defining the currentFocus and currentSelection as attributes of the AccessibleDocument object, and not of an AccessibleNode in general, reflects their global, unique nature as the target of user input.

3.4 Ranges and Traversal

An accessible DOM implementation needs to support both the ranges and traversal DOM modules defined in the DOM Level 2 specification [10].

Ranges are appropriate objects to keep track and manipulate content selection and content focus, required by Guideline 9 of UAAG 1.0 [14]. The AccessibleDocument interface defined in section 3.3 uses a Range to keep track of the document current selection. User agents that render textual content may utilize Ranges to navigate text by units like paragraphs, lines or words.

Accessibility clients need to traverse a filtered view of the DOM tree. There are at least two filtering criteria that are important for an accessibility client:

1. Traverse only nodes that support the AccessibleNode interface.
2. Traverse only those AccessibleNodes that are visible on the screen in a visual rendering.

Traversing AccessibleNode objects only may be implemented by defining an additional whatToShow constant as follows:

```
const unsigned long      SHOW_ACCESSIBLE = 0x000001000;
```

The DOM implementation would need to honor this flag when supporting the navigation methods of the NodeIterator and TreeWalker interfaces.

Visible AccessibleNode objects may be filtered by providing an acceptNode method as follows:

```
short          acceptVisibleNode(in Node n)
{
        if ( (AccessibleNode)n.boundingRect )
                return NodeFilter::FILTER_ACCEPT;
        return NodeFilter::FILTER_SKIP;
};
```

Then a TreeWalker that traverse only visible AccessibleNodes objects may be created by doing the following:

```
TreeWalker tw = (DocumentTraversal)doc.createTreeWalker(root,
        show | SHOW_ACCESSIBLE,
        acceptVisibleNode, false);
```

where show is a combination of the DOM Level 2 NodeFilter whatToShow constants excluding SHOW_ALL.

3.5 Events

The DOM Level 3 Events specification [12] provides an appropriate framework for accessibility-related events. An Accessible DOM implementation should support the following event modules:

- User Interface event module. Events related to activation of UI elements and focus tracking.

- Text event module. Related to input of text.
- Mutation and mutation name event modules. Notifications of any changes to the structure of the document tree and its nodes' attributes, text, or name.
- Basic event module. Basic event types associated with document manipulation such as load, unload, select, resize, and scroll.
- Mouse and keyboard event modules. Device-dependent events such as mouse click and key ups and downs.

Clients may register event listeners to be notified of any of the above-mentioned types of events in any particular subtree of the DOM tree. By using the dispatchEvent method of the EventTarget interface, clients may also trigger events, which, for instance, provides a mechanism to simulate mouse and keyboard input.

As stated in the DOM Level 3 Events specification, the DOM does not attempt to define all possible events. Further types and categories of events may be necessary to define for accessibility in the future. As a complete set of accessible events is defined, specialized event interfaces will be derived from the existing DOM event types in order to express the appropriate event context information.

Section 4.2 illustrates how to map the DOM Level 3 event types to a platform-specific accessibility API events.

3.6 Accessible DOM Conformance Profile

Summarizing, an Accessible DOM implementation will have to conform to the following DOM modules:

- DOM Level 3 Core.
- DOM Level 3 Events.
- DOM Level 3 User interface Events.
- DOM Level 3 Mouse Events.
- DOM Level 3 Text Events.
- DOM Level 3 Keyboard Events.
- DOM Level 3 Mutation Events.
- DOM Level 3 Mutation name Events.
- DOM Level 2 Range.
- DOM Level 2 Traversal.

It will also have to support the accessibility interfaces and associated semantics discussed in this section. See the Appendix for a complete IDL definition of the accessibility module.

4. PLATFORM-SPECIFIC ADAPTERS

As shown in Figure 1.1 in the introduction, a platform-independent accessibility API would need to be adapted to platform-specific APIs in order to support accessibility clients that rely upon them. This section illustrates how the Accessible DOM can be adapted and used to support a platform-specific accessibility API.

A platform-specific adapter is nothing more than an Accessible DOM client, whose purpose is to adapt the Accessible DOM interfaces to a particular API. An adapter will implement the interfaces of the accessibility API based on the Accessible DOM implementation. It will also forward any events generated by the Accessible DOM implementation to the accessibility API event triggering mechanism. Adapters for accessibility APIs that support input will also need to feed any input from the accessibility API back into the Accessible DOM. To accomplish this proxy role, the adapter will need to translate between the DOM accessible attributes and events and the properties and events of the accessibility API.

Section 4.1 explains how the methods of the IAccessible MSAA interface can be implemented given an implementation of an Accessible DOM. Section 4.2 shows how DOM events can be mapped into MSAA events.

4.1 Supporting MSAA IAccessible

The IAccessible get and put accName and accValue correspond to the AccessibleNode's accessibleName and accessibleValue attributes respectively. Likewise, get_accLocation, get_accRole and get_accState can be implemented straightforwardly from the AccessibleNode attributes boundingRect, accessibleType and accessibleState, respectively. Other descriptive properties of IAccessible such as accDescription or accHelp can be obtained by using the AccessibleNode NamedNodeMap attributes attribute or the Element interface to retrieve attribute values directly.

Hierarchical navigation methods like get_accParent, get_accChild and accNavigate can be naturally implemented using the DOM tree navigation mechanisms. The accNavigate method also provides spatial navigation to the objects to the left, right, above and below in a bi-dimensional visual rendering. This may be implemented by defining and supporting the AccessibleNode Relationships "to the left", "to the right", "above" and "below".

The get_accSelection and accSelect methods can be implemented by iterating over the NodeList childNodes attribute and adding or removing the "selected" value from the accessibleState attribute of each accessible child. A more specialized extension of the AccessibleNode interface may be defined in the future to better support complex selection operations.

The get_accFocus method can be supported by using the currentFocus attribute of the ownerDocument of the AccessibleNode, that is:

```
AccessibleNode focus = ( (AccessibleNode)(((Node) accnode).
                        ownerDocument) ).currentFocus;
```

Then return (IAccessible)focus, if focus is a descendent of accnode or null otherwise.

The accHitTest method can be implemented by traversing the subtree rooted at the corresponding AccessibleNode and checking the given coordinates against the boundingRect. The deepest AccessibleNode in the tree whose boundingRect contains the given point is returned.

Finally, the accDoDefaultAction method can be implemented by dispatching the DOMActivate event to the corresponding Element. In most cases this is equivalent to dispatching a mouse click event.

4.2 Mapping The DOM Event Types To MSAA Events

To forward Accessible DOM events to the accessibility API, an adapter can add listeners for different types of events to all

accessible EventTargets in the DOM tree. The handleEvent method of the EventListener interface will map each event and its context information into the corresponding accessibility API event.

The following table summarizes the correspondence between the DOM Level 3 event types and MSAA events. When more than one MSAA event should be fired as the result of a DOM event, ampersand ('&') is used to indicate so. When different events should be fired for different object types, a vertical bar ('|') is used.

Table 4.1. The first column contains the DOM event types and the second column, the corresponding MSAA events. Third column contains comments.

DOM event type	MSAA event	Comments
DOMActivate	EVENT_SYSTEM_FOREGROUND | EVENT_OBJECT_FOCUS	A link gained focus, or a new viewport became the active view.
DOMFocusIn	EVENT_OBJECT_FOCUS	
DOMFocusOut	n/a	No equivalent in MSAA
textInput	EVENT_OBJECT_VALUECHANGE	The accessibleValue of the Text object changed.
click	n/a	No mouse input support in MSAA.
mousedown	n/a	No mouse input support in MSAA.
mouseup	n/a	No mouse input support in MSAA.
mouseover	n/a	No mouse input support in MSAA.
mousemove	n/a	No mouse input support in MSAA.
mouseout	n/a	No mouse input support in MSAA.
keydown	n/a	No keyboard input support in MSAA.
keyup	n/a	No keyboard input support in MSAA.
DOMSubtreeModified	EVENT_OBJECT_REORDER	
DOMNodeInserted	EVENT_OBJECT_REORDER & EVENT_OBJECT_CREATE	
DOMNodeRemoved	EVENT_OBJECT_REORDER & EVENT_OBJECT_DESTROY	
DOMNodeRemovedFromDocument	EVENT_OBJECT_DESTROY	
DOMNodeInsertedIntoDocument	EVENT_OBJECT_CREATED	
DOMAttrModified	EVENT_OBJECT_STATECHANGE | EVENT_OBJECT_LOCATIONCHANGE | EVENT_OBJECT_NAMECHANGE | EVENT_OBJECT_VALUECHANGE	Or any other EVENT_OBJECT_ constant, corresponding to an MSAA descriptive property.
DOMCharacterDataModified	EVENT_OBJECT_VALUECHANGE	The accessibleValue of the Text object changed.
DOMElementNameChanged	EVENT_OBJECT_NAMECHANGE	
DOMAttributeNameChanged	n/a	No support for attribute name changes in MSAA.

load	EVENT_OBJECT_VALUECHANGE	No document support in MSAA. But this event may be used to indicate that something changed in the document object.
unload	EVENT_OBJECT_VALUECHANGE	No document support in MSAA. But this event may be used to indicate that something changed in the document object.
abort	n/a	No event to indicate document loading errors in MSAA.
error	n/a	No event to indicate document loading errors in MSAA.
select	EVENT_OBJECT_SELECTION	
change	EVENT_OBJECT_VALUECHANGE	
submit	n/a	No forms support in MSAA
reset	n/a	No forms support in MSAA.
resize	EVENT_OBJECT_LOCATIONCHANGE | EVENT_SYSTEM_MOVESIZESTART | EVENT_SYSTEM_MOVESIZEEND	
scroll	EVENT_SYSTEM_SCROLLINGSTART | EVENT_SYSTEM_SCROLLINGEND	

5. CONCLUSIONS

The Accessible DOM provides a suitable framework to define a platform-independent accessibility API. The AccessibleNode interface encapsulates the basic properties of existing accessibility APIs. At the same time, it remains easily extensible since new accessible attributes, relationships and events can be added in a declarative fashion, without having to change its core interfaces.

The AccessibleDocument, Relationship and Collection interfaces create the foundation for supporting accessibility of documents with dynamic, complex content and layout.

The Accessible DOM can be used to support existing and future platform-specific APIs through the implementation of thin-layer adapters. These adapters will allow the decoupling of defining and supporting accessibility requirements from the details and mechanisms that are specific to a particular operating environment.

As more work is done in the definition of a complete set of relevant accessibility attribute values, relationships and event types, the interfaces defined in the accessibility module of the Accessible DOM will be refined, enriched, and more specialized accessibility interfaces will be added.

6. APENDIX

// IDL definition of the accessibility module.

```
#ifndef _ACCESSIBLE_IDL_
#define _ACCESSIBLE_IDL_

#include "dom.idl"
#include "ranges.idl"

module accessibility
{
        typedef dom::DOMString DOMString;
        typedef dom::Node Node;

        interface Rect;
        interface RelationshipList;
        interface Collection;

        interface AccessibleNode : Node {
                attribute DOMString accessibleName;
                readonly attribute DOMString accessibleType;
                attribute DOMString accessibleValue;
                attribute DOMString accessibleState;
                attribute Rect boundingRect;
                Collection getCollection(in DOMString accessibleType);
                attribute RelationshipList relationships;
        };

        interface Rect {
                attribute unsigned long top;
                attribute unsigned long left;
                attribute unsigned long bottom;
                attribute unsigned long right;
        };

        interface Collection {
                readonly attribute unsigned long   length;
                AccessibleNode item(in unsigned long index);
        };

        interface Relationship {
                readonly attribute DOMString name;
                readonly attribute Node owner;
                readonly attribute Node relatedNode;
                readonly attribute Document ownerDocument;
                void            initRelation(in DOMString relationName,
                                in Node ownerNode,
                                in Node relatedNode);
                boolean         isSameRelationship(in Relationship arg);
        };
```

```
        interface RelationshipList {
                readonly attribute unsigned long   length;
                Relationship item(in unsigned long index);
                RelationshipList namedItems(in DOMString name);
                RelationshipList relatedNodeItems(in Node relatedNode);
                Relationship add(in Relationship arg);
                Relationship remove(in Relationship arg);
        };

        interface AccessibleDocument : AccessibleNode, Document {
                Relationship createRelationship();
                attribute AccessibleNode focusedNode;
                attribute Range selection;
        };
};

#endif // _ACCESSIBLE_IDL_
```

7. ACKNOWLEDGMENTS

Special thanks to the members of the Acrobat Accessibility team Chika Kono, Richard Potter, Ray Fischer and Ashutosh Sharma. Many of the ideas presented on this paper are the result of long hours of discussion, research and development carried out by this team. Big thanks to Abhishek Shrivastava for helping us out with the diagrams of the introduction. Our deepest appreciation to Chika Kono for also taking care of the formatting of the document.

8. REFERENCES

[1] Leventhal, A. Mozilla Accessibility Architecture. http://www.mozilla.org/access/architecture

[2] Schwerdtfeger, R. and Gibson, B. DHTML Accessibility: Fixing the JavaScript Accessibility Problem. http://www.csun.edu/cod/conf/2005/proceedings/2524.htm

[3] Java Accessibility API. http://java.sun.com/products/jfc/accessibility/reference/index.html

[4] Linux ATK Accessibility Toolkit. http://developer.gnome.org/doc/API/2.0/atk/

[5] Mac Accessibility API. http://developer.apple.com/accessibility/

[6] Microsoft Active Accessibility Version 2.0. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msaa/msaastart_9w2t.asp

[7] OMG IDL Syntax and Semantics. Common Object Request Broker: Architecture and Specification (CORBA), Version 2, Object Management Group. http://www.omg.org/technology/documents/formal/corba_2.htm

[8] Reading PDF Through Accessibility Interfaces. http://partners.adobe.com/public/developer/en/pdf/access.pdf

[9] W3C Document Object Model (DOM) Level 2 HTML Specification. http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109

[10] W3C Document Object Model (DOM) Level 2 Traversal and Range Specification. http://www.w3.org/TR/2000/REC-DOM-Level-2-Traversal-Range-20001113/

[11] W3C Document Object Model (DOM) Level 3 Core Specification. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407

[12] W3C Document Object Model (DOM) Level 3 Events Specification. http://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107/

[13] W3C Document Object Model FAQ. http://www.w3.org/DOM/faq.html#accessibility

[14] W3C User Agent Accessibility Guidelines 1.0. http://www.w3.org/TR/2002/REC-UAAG10-20021217/

[15] W3C Web Accessibility Initiative (WAI). http://www.w3.org/WAI/