Document Object Model (DOM) Level 3 Validation Specification

Version 1.0

W3C Working Draft 08 October 2002

This version:
http://www.w3.org/TR/2002/WD-DOM-Level-3-Val-20021008
Latest version:
http://www.w3.org/TR/DOM-Level-3-Val
Previous version:
http://www.w3.org/TR/2002/WD-DOM-Level-3-Val-20020725

Editors:
Ben Chang, *Oracle*
Joe Kesselman, *IBM (until September 2001)*
Rezaur Rahman, *Intel Corporation (until July 2001)*

This document is also available in these non-normative formats: XML fileplain text, PostScript file, PDF file, single HTML file, and ZIP file.

Abstract

This specification defines the Document Object Model Validation Level 3, a platform- and language-neutral interface. This module provides the guidance to programs and scripts to dynamically update the content and the structure of documents while ensuring that the document remains valid, or to ensure that the document becomes valid.

Status of this document

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

This is a Last Call Working Draft for review by W3C members and other interested parties. The Last Call review period ends on 27 November 2002. Please send reviews before the review period ends to the public mailing list www-dom@w3.org. An archive is available at http://lists.w3.org/Archives/Public/www-dom/.

Individuals or organizations are also invited to send a message to the public mailing list if they intend to produce an implementation of this module.

It is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress". This is work in progress and does not imply endorsement by, or the consensus of, either W3C or members of the DOM working group.

This document has been produced as part of the W3C DOM Activity. The authors of this document are the DOM WG members.

Patent disclosures relevant to this specification may be found on the Working Group’s patent disclosure page.

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

Table of contents

Expanded Table of Contents

Copyright Notice

This document is published under the W3C Document Copyright Notice and License [p.5] . The bindings within this document are published under the W3C Software Copyright Notice and License [p.6] . The software license requires "Notice of any changes or modifications to the W3C files, including the date changes were made." Consequently, modified versions of the DOM bindings must document that they do not conform to the W3C standard; in the case of the IDL definitions, the pragma prefix can no longer be 'w3c.org'; in the case of the Java language binding, the package names can no longer be in the 'org.w3c' package.

W3C Document Copyright Notice and License

Note: This section is a copy of the W3C Document Notice and License and could be found at http://www.w3.org/Consortium/Legal/copyright-documents-19990405.

http://www.w3.org/Consortium/Legal/

Public documents on the W3C site are provided by the copyright holders under the following license. The software or Document Type Definitions (DTDs) associated with W3C specifications are governed by the Software Notice. By using and/or copying this document, or the W3C document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, and distribute the contents of this document, or the W3C document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on *ALL* copies of the document, or portions thereof, that you use:

1. A link or URL to the original W3C document.
2. The pre-existing copyright notice of the original author, or if it doesn't exist, a notice of the form: "Copyright © [$date-of-document] World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/" (Hypertext is preferred, but a textual representation is permitted.)
3. *If it exists*, the STATUS of the W3C document.

When space permits, inclusion of the full text of this **NOTICE** should be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of W3C documents is granted pursuant to this license. However, if additional requirements (documented in the Copyright FAQ) are satisfied, the right to create modifications or derivatives is sometimes granted by the W3C to individuals complying with those requirements.

THIS DOCUMENT IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

W3C Software Copyright Notice and License

Note: This section is a copy of the W3C Software Copyright Notice and License and could be found at http://www.w3.org/Consortium/Legal/copyright-software-19980720

Copyright © 1994-2002 World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved.

http://www.w3.org/Consortium/Legal/

This W3C work (including software, documents, or other related items) is being provided by the copyright holders under the following license. By obtaining, using and/or copying this work, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, and modify this software and its documentation, with or without modification, for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the software and documentation or portions thereof, including modifications, that you make:

1. The full text of this NOTICE in a location viewable to users of the redistributed or derivative work.
2. Any pre-existing intellectual property disclaimers. If none exist, then a notice of the following form: "Copyright © [$date-of-software] World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/."

3. Notice of any changes or modifications to the W3C files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permission. Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

1. Validation

Editors:
Ben Chang, Oracle
Joe Kesselman, IBM (until September 2001)
Rezaur Rahman, Intel Corporation (until July 2001)

1.1. Overview

This chapter describes the optional DOM Level 3 Validation feature. This module provides APIs to query information about the XML document.

A DOM application can use the `hasFeature` method of the `DOMImplementation` interface to determine whether a given DOM supports these capabilities or not. This module defines 1 feature string: `"VAL-DOC"` for document-editing interfaces.

This chapter focuses on the editing aspects used in the XML document-editing world and usage of such information.

1.1.1. General Characteristics

In the October 9, 1997 DOM requirements document, the following appeared: "There will be a way to determine the presence of a DTD. There will be a way to add, remove, and change declarations in the underlying DTD (if available). There will be a way to test conformance of all or part of the given document against a DTD (if available)." In later discussions, the following was added, "There will be a way to query element/attribute (and maybe other) declarations in the underlying DTD (if available)," supplementing the primitive support for these in Level 1.

That work was deferred past Level 2, in the hope that XML Schemas would be addressed as well. Work was deferred on lowest common denominator general grammar APIs due to heightened interest in XML Schema- specific APIs; however, work on querying information on the grammar was only done for DOM Level 3.

1.1.2. Use Cases and Requirements

Here are the following use cases and requirements that prompted the functionality in this document:

Use Cases:

1. DU1. For editing documents with an associated grammar, provide the guidance necessary so that valid documents can be modified and remain valid.
2. DU2. For editing documents with an associated grammar, provide the guidance necessary to transform an invalid document into a valid one.

Requirements:

1. DR1. Be able to determine if the document is well-formed, and if not, be given enough guidance to locate the error.
2. DR2. Be able to determine if the document is namespace well-formed, and if not, be given enough guidance to locate the error.
3. DR3. Be able to determine if the document is valid with respect to its associated grammar.
4. DR4. Be able to determine if specific modifications to a document would make it become invalid.
5. DR5. Retrieve information from all grammar. One example might be getting a list of all the defined element names for document editing purposes.

1.2. Exceptions

This section describes the "VAL-DOC" exceptions.

Exception *ExceptionVAL*

These operations may throw a `ExceptionVAL` [p.10] as described in their descriptions.

IDL Definition

```
exception ExceptionVAL {
  unsigned short   code;
};
// ExceptionVALCode
const unsigned short      NO_GRAMMAR_AVAILABLE          = 71;
const unsigned short      VALIDATION_ERR                = 72;
```

Definition group *ExceptionVALCode*

An integer indicating the type of error generated.

Defined Constants

`NO_GRAMMAR_AVAILABLE`
: If the `DocumentEditVAL` [p.11] related to the node does not have any grammar and `wfValidityCheckLevel` is set to `PARTIAL` or `STRICT_VALIDITY_CHECK`.

`VALIDATION_ERR`
: Raised if document is invalid.

1.3. Document-Editing Interfaces

This section contains "Document-editing" methods (includes `Node`, `Element`, `Text` and `Document` methods).

A DOM application may use the `hasFeature(feature, version)` method of the `DOMImplementation` interface with parameter values "VAL-DOC" and "3.0" (respectively) to determine whether or not the Document-Editing interfaces are supported by the implementation.

Interface *DocumentEditVAL*

This interface extends the `NodeEditVAL` [p.11] interface with additional methods for document editing.

IDL Definition

```
interface DocumentEditVAL : NodeEditVAL {
           attribute boolean         continuousValidityChecking;
  void               validateDocument()
                                        raises(ExceptionVAL);
};
```

Attributes

`continuousValidityChecking` of type `boolean`

An attribute specifying whether continuous checking for the validity of the document is enforced or not. Setting this to `true` will result in an exception being thrown, i.e., `VALIDATION_ERR` [p.10] , for documents that are invalid at the time of the call. When set to true, the implementation if free to raise the `VALIDATION_ERR` exception on DOM operations that would make the document invalid with respect to "partial validity." If the document is invalid, then this attribute will remain `false`. This attribute is `false` by default.

Methods

`validateDocument`

Validates the document against the grammar. If the document is mutated during validation, a warning will be issued. In addition, the validation cannot modify the document, e.g., for default attributes. This method makes use of the passed-in error handler, as described in [DOM Level 3 Core] interface.

Exceptions

`ExceptionVAL` [p.10]	NO_GRAMMAR_AVAILABLE: Raised if an error occurs when the grammar is not available for the document.

No Parameters

No Return Value

Interface *NodeEditVAL*

This interface is similar to the [DOM Level 3 Core] `Node` interfaces, with methods for guided document editing.

IDL Definition

```
interface NodeEditVAL {

  // CheckTypeVAL
  const unsigned short      WF_CHECK                    = 1;
  const unsigned short      NS_WF_CHECK                 = 2;
  const unsigned short      PARTIAL_VALIDITY_CHECK      = 3;
  const unsigned short      STRICT_VALIDITY_CHECK       = 4;
```

```
  boolean            canInsertBefore(in Node newChild,
                                     in Node refChild);
  boolean            canRemoveChild(in Node oldChild);
  boolean            canReplaceChild(in Node newChild,
                                     in Node oldChild);
  boolean            canAppendChild(in Node newChild);
  boolean            isNodeValid(in boolean deep,
                                 in unsigned short wFValidityCheckLevel)
                                     raises(ExceptionVAL);
};
```

Definition group *CheckTypeVAL*

An integer indicating which type of validation this is.

Defined Constants

`NS_WF_CHECK`
: Check for namespace well-formedness includes `WF_CHECK`.

`PARTIAL_VALIDITY_CHECK`
: Checks for whether this node is *partially valid* [p.35] . It includes `NS_WF_CHECK`.

`STRICT_VALIDITY_CHECK`
: Checks for strict validity of the node with respect to active grammar which by definition includes `NS_WF_CHECK`.

`WF_CHECK`
: Check for well-formedness of this node.

Methods

`canAppendChild`

Has the same arguments as `Node.appendChild`.

Parameters

`newChild` of type `Node`
: `Node` to be appended.

Return Value

`boolean` `true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`canInsertBefore`

Determines whether the `Node.insertBefore` operation would make this document not partially valid with respect to the currently active grammar.

Parameters

`newChild` of type `Node`
: `Node` to be inserted.

`refChild` of type `Node`
: Reference `Node`.

Return Value

`boolean` `true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`canRemoveChild`

Has the same arguments as `Node.removeChild`.

Parameters

`oldChild` of type `Node`

`Node` to be removed.

Return Value

`boolean`	`true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`canReplaceChild`

Has the same arguments as `Node.replaceChild`.

Parameters

`newChild` of type `Node`

New `Node`.

`oldChild` of type `Node`

`Node` to be replaced.

Return Value

`boolean`	`true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`isNodeValid`

Determines if the Node is valid relative to currently active grammar. It doesn't normalize before checking if the document is valid. To do so, one would need to explicitly call a normalize method.

Parameters

`deep` of type `boolean`

Setting the `deep` flag on causes the `isNodeValid` method to check for the whole subtree of the current node for validity. Setting it to `false` only checks the current node and its immediate child nodes. The `validateDocument` method on the `DocumentVAL` interface, however, checks to determine whether the entire document is valid.

`wFValidityCheckLevel` of type `unsigned short`

Flag to tell at what level validity and well-formedness checking is done.

Return Value

`boolean`	`true` if the node is valid/well-formed in the current context and check level defined by `wfValidityCheckLevel`, `false` if not.

Exceptions

`ExceptionVAL` [p.10]	NO_GRAMMAR_AVAILABLE: Exception is raised if the DocumentEditVAL related to this node does not have any grammar associated with it and `wfValidityCheckLevel` is set to `PARTIAL` or `STRICT_VALIDITY_CHECK`.

Interface *ElementEditVAL*

This interface extends the `Element` interface with additional methods for guided document editing. An object implementing this interface must also implement NodeEditVAL interface.

IDL Definition

```
interface ElementEditVAL : NodeEditVAL {
  readonly attribute NodeList        definedElementTypes;
  unsigned short     contentType();
  boolean            canSetAttribute(in DOMString attrname,
                                     in DOMString attrval);
  boolean            canSetAttributeNode(in Attr attrNode);
  boolean            canSetAttributeNS(in DOMString namespaceURI,
                                       in DOMString qualifiedName,
                                       in DOMString value);
  boolean            canRemoveAttribute(in DOMString attrname);
  boolean            canRemoveAttributeNS(in DOMString namespaceURI,
                                          in DOMString localName);
  boolean            canRemoveAttributeNode(in Node attrNode);
  NodeList           getChildElements();
  NodeList           getParentElements();
  NodeList           getAttributeList();
  boolean            isElementDefined(in DOMString name);
  boolean            isElementDefinedNS(in DOMString name,
                                        in DOMString namespaceURI);
};
```

Attributes

`definedElementTypes` of type `NodeList`, readonly

The list of all element nodes defined in non-namespace aware grammar or list of all element nodes belonging to the particular namespace. These are not nodes from the instance document, but rather are new nodes that could be inserted in the document.

Methods

`canRemoveAttribute`

Verifies if an attribute by the given name can be removed.

Parameters

`attrname` of type `DOMString`

Name of attribute.

Return Value

`boolean`	`true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`canRemoveAttributeNS`

Verifies if an attribute by the given local name and namespace can be removed.

Parameters

`namespaceURI` of type `DOMString`

The namespace URI of the attribute to remove.

`localName` of type `DOMString`

Local name of the attribute to be removed.

Return Value

`boolean` `true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`canRemoveAttributeNode`

Determines if an attribute node can be removed.

Parameters

`attrNode` of type `Node`

The `Attr` node to remove from the attribute list.

Return Value

`boolean` `true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`canSetAttribute`

Determines if the value for specified attribute can be set.

Parameters

`attrname` of type `DOMString`

Name of attribute.

`attrval` of type `DOMString`

Value to be assigned to the attribute.

Return Value

`boolean` `true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`canSetAttributeNS`

Determines if the attribute with given namespace and qualified name can be created if not already present in the attribute list of the element. If the attribute with the same qualified name and namespaceURI is already present in the element's attribute list, it tests whether the value of the attribute and its prefix can be set to the new value. See DOM core `setAttributeNS`.

Parameters

`namespaceURI` of type `DOMString`

`namespaceURI` of namespace.

`qualifiedName` of type `DOMString`
: Qualified name of attribute.

`value` of type `DOMString`
: Value to be assigned to the attribute.

Return Value

`boolean`	`true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`canSetAttributeNode`

Determines if an attribute node can be added with respect to the validity check level.

Parameters

`attrNode` of type `Attr`
: `Node` in which the attribute can possibly be set.

Return Value

`boolean`	`true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`contentType`

Determines element content type.

Return Value

`unsigned short`	Constant for one of `EMPTY_CONTENTTYPE`, `ANY_CONTENTTYPE`, `MIXED_CONTENTTYPE`, `ELEMENTS_CONTENTTYPE`.

No Parameters

No Exceptions

`getAttributeList`

Returns a `NodeList` containing all the possible `Attr`s that can appear with this type of element. These are not nodes from the instance document, but rather are new nodes that could be inserted in the document.

Return Value

`NodeList`	List of possible attributes of this element.

No Parameters

No Exceptions

`getChildElements`

Returns a `NodeList` containing the possible `Element` nodes that can appear as children of this type of element, with certain conditions as specified below. These are not nodes from the instance document, but rather are new nodes that could be inserted in the document.

Return Value

`NodeList` List of possible children element types of this element. Note that if no context of this element exists, then a `NULL` is returned; an empty list is returned if the element is not in the document tree.

No Parameters

No Exceptions

`getParentElements`

Returns a `NodeList` containing the possible `Element` nodes that can appear as a parent of this type of element, with certain conditions as specified below. These are not nodes from the instance document, but rather are new nodes that could be inserted in the document.

Return Value

`NodeList` List of possible parent element types of this element. Note that if no context of this element exists, for example, the parent element of this element, then a `NULL` is returned; an empty list is returned if the element is not in the document tree.

No Parameters

No Exceptions

`isElementDefined`

Determines if `name` is defined in the currently active grammar.

Parameters

`name` of type `DOMString`

Name of element.

Return Value

`boolean` A boolean that is `true` if the element is defined, false otherwise.

No Exceptions

`isElementDefinedNS`

Determines if `name` in this namespace is defined in the currently active grammar.

Parameters

`name` of type `DOMString`

Name of element.

`namespaceURI` of type `DOMString`

`namespaceURI` of namespace.

Return Value

`boolean` A boolean that is `true` if the element is defined, false otherwise.

No Exceptions

Interface *CharacterDataEditVAL*

This interface extends the `NodeEditVAL` [p.11] interface with additional methods for document editing. An object implementing this interface must also implement NodeEditVAL interface.

IDL Definition

```
interface CharacterDataEditVAL : NodeEditVAL {
  readonly attribute boolean         isWhitespaceOnly;
  boolean            canSetData(in unsigned long offset,
                                in DOMString arg);
  boolean            canAppendData(in DOMString arg);
  boolean            canReplaceData(in unsigned long offset,
                                    in unsigned long count,
                                    in DOMString arg);
  boolean            canInsertData(in unsigned long offset,
                                   in DOMString arg);
  boolean            canDeleteData(in unsigned long offset,
                                   in unsigned long count);
};
```

Attributes

`isWhitespaceOnly` of type `boolean`, readonly

`true` if content only whitespace; `false` for non-whitespace.

Methods

`canAppendData`

Determines if data can be appended.

Parameters

`arg` of type `DOMString`

Data to be appended.

Return Value

`boolean` `true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`canDeleteData`

Determines if data can be deleted.

Parameters

`offset` of type `unsigned long`

Offset.

`count` of type `unsigned long`

Number of 16-bit units to delete.

Return Value

`boolean` `true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`canInsertData`

Determines if data can be inserted.

Parameters

`offset` of type `unsigned long`

Offset.

`arg` of type `DOMString`

Argument to be set.

Return Value

`boolean` `true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`canReplaceData`

Determines if data can be replaced.

Parameters

`offset` of type `unsigned long`

Offset.

`count` of type `unsigned long`

Replacement.

`arg` of type `DOMString`

Argument to be set.

Return Value

`boolean` `true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

`canSetData`

Determines if data can be set.

Parameters

`offset` of type `unsigned long`

Offset.

`arg` of type `DOMString`

Argument to be set.

Return Value

`boolean` `true` if no reason it can't be done; `false` if it can't be done.

No Exceptions

1.4. Document Manipulation

Applications would like to be able to use functionality to guide construction and editing of documents, which falls into the document-editing world. Examples of this sort of guided editing already exist, and are becoming more common. The necessary queries can be phrased in several ways, the most useful of which may be a combination of "what does the DTD allow me to insert here" and "if I insert this here, will the document still be valid". The former is better suited to presentation to humans via a user interface, and when taken together with sub-tree validation may subsume the latter.

It has been proposed that in addition to asking questions about specific parts of the grammar, there should be a reasonable way to obtain a list of all the defined symbols of a given type (element, attribute, entity) independent of whether they’re valid in a given location; that might be useful in building a list in a user-interface, which could then be updated to reflect which of these are relevant for the program’s current state.

Remember that namespaces also weigh in on this issue, in the case of attributes, a "can-this-go-there" may prompt a namespace-well-formedness check and warn you if you’re about to conflict with or overwrite another attribute with the same namespaceURI/localName but different prefix, or same nodeName but different namespaceURI.

We have to deal with the fact that "the shortest distance between two valid documents may be through an invalid one". Users may want to know several levels of detail (all the possible children, those which would be valid given what precedes this point, those which would be valid given both preceding and following siblings). Also, once XML Schemas introduce context sensitive validity, we may have to consider the effect of children as well as the individual node being inserted.

1.5. Validating a Document

The most obvious use for a DTD or XML Schema or any grammar is to use it to validate a given XML document. This again falls into the document-editing world. The XML spec only discusses performing this test at the time the document is loaded into the "processor", which most of us have taken to mean that this check should be performed at parse time. But it is obviously desirable to be able to validate again a document -- or selected subtrees -- at other times. One such case would be validating an edited or newly constructed document before serializing it or otherwise passing it to other users. This issue also arises if the "internal subset" is altered -- or if the grammar changes.

In the past, the DOM has allowed users to create invalid documents, and assumed the serializer would accept the task of detecting problems and announcing/repairing them when the document was written out in XML syntax... or that they would be checked for validity when read back in. We considered adding validity checks to the DOM’s existing editing operations to prevent creation of invalid documents, but are currently inclined against this for several reasons. First, it would impose a significant amount of computational overhead to the DOM, which might be unnecessary in many situations, e.g., if the change is occurring in a context where we know the result will be valid. Second, "the shortest distance between two good documents may be through a bad document". Preventing a document from becoming temporarily invalid may impose a considerable amount of additional work on higher-level code and users Hence our current plan is to continue to permit editing to produce invalid DOMs, but provide operations which

permit a user to check the validity of a node on demand. If needed one can use `continuousValidityChecking` flag to ensure that the DOM remains valid during the editing process.

Note that validation includes checking that ID attributes are unique, and that IDREFs point to IDs which actually exist.

1.6. Well-formedness Testing

XML defined the "well-formed" (*WF*) state for documents which are parsed without reference to their DTDs. Knowing that a document is well-formed may be useful by itself even when a DTD is available. For example, users may wish to deliberately save an invalid document, perhaps as a checkpoint before further editing. Hence, the "Validation" features will permit both full validity checking (see previous section) and "lightweight" WF checking, as requested by the caller, as well as processing entity declarations in the AS even if validation is not turned on.

While the DOM inherently enforces some of XML’s well-formedness conditions (proper nesting of elements, constraints on which children may be placed within each node), there are some checks that are not yet performed. These include:

- Character restrictions for text content and attribute values. Some characters aren’t permitted even when expressed as numeric character entities
- The three-character sequence "]]>" in CDATASections.
- The two-character sequence "--" in comments. (Which, be it noted, some XML validators don’t currently remember to test...)

In addition, Namespaces introduce their own concepts of well-formedness. Specifically:

- No two attributes on a single Element may have the same combination of namespaceURI and localName, even if their prefixes are different and hence they don’t conflict under XML 1.0 rules.
- NamespaceURIs must be legal URI syntax. (Note that once we have this code, it may be reusable for the URI "datatype" in document content; see discussion of datatypes.)
- The mapping of namespace prefixes to their URIs must be declared and consistent. That isn’t required during normal DOM operation, since we perform "early binding" and thereafter refer to nodes primarily via their namespaceURIs and localName. But it does become an issue when we want to serialize the DOM to XML syntax, and may be an issue if an application is assuming that all the declarations are present and correct. This may imply that we should provide a `namespaceNormalize` operation, which would create the implied declarations and reconcile conflicts in some reasonably standardized manner. This may be a major undertaking, since some DOMs may be using the namespace to direct subclassing of the nodes or similar special treatment; as with the existing `normalize` method, you may be left with a different-but-equivalent set of node objects.

In the past, the DOM has allowed users to create documents which violate these rules, and assumed the serializer would accept the task of detecting problems and announcing/repairing them when the document was written out in XML syntax. We considered adding WF checks to the DOM’s existing editing operations to prevent WF violations from arising, but are currently inclined against this for two reasons.

First, it would impose a significant amount of computational overhead to the DOM, which might be unnecessary in many situations (for example, if the change is occurring in a context where we know the illegal characters have already been prevented from arising). Second, "the shortest distance between two good documents may be through a bad document" -- preventing a document from becoming temporarily ill-formed may impose a considerable amount of additional work on higher-level code and users. (Note possible issue for Serialization: In some applications, being able to save and reload marginally poorly-formed DOMs might be useful -- editor checkpoint files, for example.) Hence our current plan is to continue to permit editing to produce ill-formed DOMs, but provide operations which permit a user to check the well-formedness of a node on demand, and possibly provide some of the primitive (e.g., string-checking) functions directly.

Appendix A: IDL Definitions

This appendix contains the complete OMG IDL [OMG IDL] for the Level 3 Document Object Model Validation definitions.

The IDL files are also available as: http://www.w3.org/TR/2002/WD-DOM-Level-3-Val-20021008/idl.zip

validation.idl:

```
// File: validation.idl

#ifndef _VALIDATION_IDL_
#define _VALIDATION_IDL_

#include "dom.idl"

#pragma prefix "dom.w3c.org"
module validation
{

  typedef dom::Node Node;
  typedef dom::NodeList NodeList;
  typedef dom::DOMString DOMString;
  typedef dom::Attr Attr;

  exception ExceptionVAL {
    unsigned short   code;
  };
  // ExceptionVALCode
  const unsigned short      NO_GRAMMAR_AVAILABLE          = 71;
  const unsigned short      VALIDATION_ERR                = 72;


  interface NodeEditVAL {

    // CheckTypeVAL
    const unsigned short      WF_CHECK                      = 1;
    const unsigned short      NS_WF_CHECK                   = 2;
    const unsigned short      PARTIAL_VALIDITY_CHECK        = 3;
    const unsigned short      STRICT_VALIDITY_CHECK         = 4;

    boolean            canInsertBefore(in Node newChild,
                                       in Node refChild);
    boolean            canRemoveChild(in Node oldChild);
    boolean            canReplaceChild(in Node newChild,
                                       in Node oldChild);
    boolean            canAppendChild(in Node newChild);
    boolean            isNodeValid(in boolean deep,
                                   in unsigned short wFValidityCheckLevel)
                                        raises(ExceptionVAL);
  };

  interface ElementEditVAL : NodeEditVAL {
    readonly attribute NodeList        definedElementTypes;
```

```
    unsigned short     contentType();
    boolean            canSetAttribute(in DOMString attrname,
                                       in DOMString attrval);
    boolean            canSetAttributeNode(in Attr attrNode);
    boolean            canSetAttributeNS(in DOMString namespaceURI,
                                         in DOMString qualifiedName,
                                         in DOMString value);
    boolean            canRemoveAttribute(in DOMString attrname);
    boolean            canRemoveAttributeNS(in DOMString namespaceURI,
                                            in DOMString localName);
    boolean            canRemoveAttributeNode(in Node attrNode);
    NodeList           getChildElements();
    NodeList           getParentElements();
    NodeList           getAttributeList();
    boolean            isElementDefined(in DOMString name);
    boolean            isElementDefinedNS(in DOMString name,
                                          in DOMString namespaceURI);
  };

  interface CharacterDataEditVAL : NodeEditVAL {
    readonly attribute boolean         isWhitespaceOnly;
    boolean            canSetData(in unsigned long offset,
                                  in DOMString arg);
    boolean            canAppendData(in DOMString arg);
    boolean            canReplaceData(in unsigned long offset,
                                      in unsigned long count,
                                      in DOMString arg);
    boolean            canInsertData(in unsigned long offset,
                                     in DOMString arg);
    boolean            canDeleteData(in unsigned long offset,
                                     in unsigned long count);
  };

  interface DocumentEditVAL : NodeEditVAL {
             attribute boolean         continuousValidityChecking;
    void               validateDocument()
                                       raises(ExceptionVAL);
  };
};

#endif // _VALIDATION_IDL_
```

Appendix B: Java Language Binding

This appendix contains the complete Java [Java] bindings for the Level 3 Document Object Model Validation.

The Java files are also available as
http://www.w3.org/TR/2002/WD-DOM-Level-3-Val-20021008/java-binding.zip

org/w3c/dom/validation/ExceptionVAL.java:

```
package org.w3c.dom.validation;

public class ExceptionVAL extends RuntimeException {
    public ExceptionVAL(short code, String message) {
       super(message);
       this.code = code;
    }
    public short   code;
    // ExceptionVALCode
    public static final short NO_GRAMMAR_AVAILABLE      = 71;
    public static final short VALIDATION_ERR            = 72;

}
```

org/w3c/dom/validation/DocumentEditVAL.java:

```
package org.w3c.dom.validation;

public interface DocumentEditVAL extends NodeEditVAL {
    public boolean getContinuousValidityChecking();
    public void setContinuousValidityChecking(boolean continuousValidityChecking);

    public void validateDocument()
                                throws ExceptionVAL;

}
```

org/w3c/dom/validation/NodeEditVAL.java:

```
package org.w3c.dom.validation;

import org.w3c.dom.Node;

public interface NodeEditVAL {
    // CheckTypeVAL
    public static final short WF_CHECK                  = 1;
    public static final short NS_WF_CHECK               = 2;
    public static final short PARTIAL_VALIDITY_CHECK    = 3;
    public static final short STRICT_VALIDITY_CHECK     = 4;

    public boolean canInsertBefore(Node newChild,
                                   Node refChild);
```

```
    public boolean canRemoveChild(Node oldChild);

    public boolean canReplaceChild(Node newChild,
                                   Node oldChild);

    public boolean canAppendChild(Node newChild);

    public boolean isNodeValid(boolean deep,
                               short wFValidityCheckLevel)
                               throws ExceptionVAL;

}
```

org/w3c/dom/validation/ElementEditVAL.java:

```
package org.w3c.dom.validation;

import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.w3c.dom.Attr;

public interface ElementEditVAL extends NodeEditVAL {
    public NodeList getDefinedElementTypes();

    public short contentType();

    public boolean canSetAttribute(String attrname,
                                   String attrval);

    public boolean canSetAttributeNode(Attr attrNode);

    public boolean canSetAttributeNS(String namespaceURI,
                                     String qualifiedName,
                                     String value);

    public boolean canRemoveAttribute(String attrname);

    public boolean canRemoveAttributeNS(String namespaceURI,
                                        String localName);

    public boolean canRemoveAttributeNode(Node attrNode);

    public NodeList getChildElements();

    public NodeList getParentElements();

    public NodeList getAttributeList();

    public boolean isElementDefined(String name);

    public boolean isElementDefinedNS(String name,
                                      String namespaceURI);

}
```

org/w3c/dom/validation/CharacterDataEditVAL.java:

```
package org.w3c.dom.validation;

public interface CharacterDataEditVAL extends NodeEditVAL {
    public boolean getIsWhitespaceOnly();

    public boolean canSetData(int offset,
                              String arg);

    public boolean canAppendData(String arg);

    public boolean canReplaceData(int offset,
                                  int count,
                                  String arg);

    public boolean canInsertData(int offset,
                                 String arg);

    public boolean canDeleteData(int offset,
                                 int count);

}
```

org/w3c/dom/validation/CharacterDataEditVAL.java:

Appendix C: ECMAScript Language Binding

This appendix contains the complete ECMAScript [ECMAScript] binding for the Level 3 Document Object Model Validation definitions.

Properties of the **ExceptionVAL** Constructor function:

ExceptionVAL.NO_GRAMMAR_AVAILABLE

The value of the constant **ExceptionVAL.NO_GRAMMAR_AVAILABLE** is **71**.

ExceptionVAL.VALIDATION_ERR

The value of the constant **ExceptionVAL.VALIDATION_ERR** is **72**.

Objects that implement the **ExceptionVAL** interface:

Properties of objects that implement the **ExceptionVAL** interface:

code

This property is a **Number**.

Objects that implement the **DocumentEditVAL** interface:

Objects that implement the **DocumentEditVAL** interface have all properties and functions of the **NodeEditVAL** interface as well as the properties and functions defined below.

Properties of objects that implement the **DocumentEditVAL** interface:

continuousValidityChecking

This property is a **Boolean**.

Functions of objects that implement the **DocumentEditVAL** interface:

validateDocument()

This function has no return value.

This function can raise an object that implements the **ExceptionVAL** interface.

Properties of the **NodeEditVAL** Constructor function:

NodeEditVAL.WF_CHECK

The value of the constant **NodeEditVAL.WF_CHECK** is **1**.

NodeEditVAL.NS_WF_CHECK

The value of the constant **NodeEditVAL.NS_WF_CHECK** is **2**.

NodeEditVAL.PARTIAL_VALIDITY_CHECK

The value of the constant **NodeEditVAL.PARTIAL_VALIDITY_CHECK** is **3**.

NodeEditVAL.STRICT_VALIDITY_CHECK

The value of the constant **NodeEditVAL.STRICT_VALIDITY_CHECK** is **4**.

Objects that implement the **NodeEditVAL** interface:

Functions of objects that implement the **NodeEditVAL** interface:

canInsertBefore(newChild, refChild)

This function returns a **Boolean**.

The **newChild** parameter is an object that implements the **Node** interface.

The **refChild** parameter is an object that implements the **Node** interface.

canRemoveChild(oldChild)

This function returns a **Boolean**.

The **oldChild** parameter is an object that implements the **Node** interface.

canReplaceChild(newChild, oldChild)

This function returns a **Boolean**.

The **newChild** parameter is an object that implements the **Node** interface.

The **oldChild** parameter is an object that implements the **Node** interface.

canAppendChild(newChild)

This function returns a **Boolean**.

The **newChild** parameter is an object that implements the **Node** interface.

isNodeValid(deep, wFValidityCheckLevel)

This function returns a **Boolean**.

The **deep** parameter is a **Boolean**.

The **wFValidityCheckLevel** parameter is a **Number**.

This function can raise an object that implements the **ExceptionVAL** interface.

Objects that implement the **ElementEditVAL** interface:

Objects that implement the **ElementEditVAL** interface have all properties and functions of the **NodeEditVAL** interface as well as the properties and functions defined below.

Properties of objects that implement the **ElementEditVAL** interface:

definedElementTypes

This read-only property is an object that implements the **NodeList** interface.

Functions of objects that implement the **ElementEditVAL** interface:

contentType()

This function returns a **Number**.

canSetAttribute(attrname, attrval)

This function returns a **Boolean**.

The **attrname** parameter is a **String**.

The **attrval** parameter is a **String**.

canSetAttributeNode(attrNode)

This function returns a **Boolean**.

The **attrNode** parameter is an object that implements the **Attr** interface.

canSetAttributeNS(namespaceURI, qualifiedName, value)

This function returns a **Boolean**.

The **namespaceURI** parameter is a **String**.

The **qualifiedName** parameter is a **String**.

The **value** parameter is a **String**.

canRemoveAttribute(attrname)

This function returns a **Boolean**.

The **attrname** parameter is a **String**.

canRemoveAttributeNS(namespaceURI, localName)

This function returns a **Boolean**.

The **namespaceURI** parameter is a **String**.

The **localName** parameter is a **String**.

canRemoveAttributeNode(attrNode)

This function returns a **Boolean**.

The **attrNode** parameter is an object that implements the **Node** interface.

getChildElements()

This function returns an object that implements the **NodeList** interface.

getParentElements()

This function returns an object that implements the **NodeList** interface.

getAttributeList()

This function returns an object that implements the **NodeList** interface.

isElementDefined(name)

This function returns a **Boolean**.

The **name** parameter is a **String**.

isElementDefinedNS(name, namespaceURI)

This function returns a **Boolean**.

The **name** parameter is a **String**.

The **namespaceURI** parameter is a **String**.

Objects that implement the **CharacterDataEditVAL** interface:

Objects that implement the **CharacterDataEditVAL** interface have all properties and functions of the **NodeEditVAL** interface as well as the properties and functions defined below.

Properties of objects that implement the **CharacterDataEditVAL** interface:

isWhitespaceOnly

This read-only property is a **Boolean**.

Functions of objects that implement the **CharacterDataEditVAL** interface:

canSetData(offset, arg)

This function returns a **Boolean**.

The **offset** parameter is a **Number**.

The **arg** parameter is a **String**.

canAppendData(arg)

This function returns a **Boolean**.

The **arg** parameter is a **String**.

canReplaceData(offset, count, arg)

This function returns a **Boolean**.

The **offset** parameter is a **Number**.

The **count** parameter is a **Number**.

The **arg** parameter is a **String**.

canInsertData(offset, arg)

This function returns a **Boolean**.

The **offset** parameter is a **Number**.

The **arg** parameter is a **String**.

canDeleteData(offset, count)

This function returns a **Boolean**.

The **offset** parameter is a **Number**.

The **count** parameter is a **Number**.

Appendix D: Acknowledgements

Many people contributed to the DOM specifications (Level 1, 2 or 3), including members of the DOM Working Group and the DOM Interest Group. We especially thank the following:

Andrew Watson (Object Management Group), Andy Heninger (IBM), Angel Diaz (IBM), Arnaud Le Hors (W3C and IBM), Ashok Malhotra (IBM and Microsoft), Ben Chang (Oracle), Bill Smith (Sun), Bill Shea (Merrill Lynch), Bob Sutor (IBM), Chris Lovett (Microsoft), Chris Wilson (Microsoft), David Brownell (Sun), David Ezell (Hewlett Packard Company), David Singer (IBM), Dimitris Dimitriadis (Improve AB), Don Park (invited), Elena Litani (IBM), Eric Vasilik (Microsoft), Gavin Nicol (INSO), Ian Jacobs (W3C), James Clark (invited), James Davidson (Sun), Jared Sorensen (Novell), Jeroen van Rotterdam (X-Hive Corporation), Joe Kesselman (IBM), Joe Lapp (webMethods), Joe Marini (Macromedia), Johnny Stenback (Netscape/AOL), Jon Ferraiolo (Adobe), Jonathan Marsh (Microsoft), Jonathan Robie (Texcel Research and Software AG), Kim Adamson-Sharpe (SoftQuad Software Inc.), Lauren Wood (SoftQuad Software Inc., *former Chair*), Laurence Cable (Sun), Mark Davis (IBM), Mark Scardina (Oracle), Martin Dürst (W3C), Mary Brady (NIST), Mick Goulish (Software AG), Mike Champion (Arbortext and Software AG), Miles Sabin (Cromwell Media), Patti Lutsky (Arbortext), Paul Grosso (Arbortext), Peter Sharpe (SoftQuad Software Inc.), Phil Karlton (Netscape), Philippe Le Hégaret (W3C, *W3C team contact and former Chair*), Ramesh Lekshmynarayanan (Merrill Lynch), Ray Whitmer (iMall, Excite@Home, and Netscape/AOL, *Chair*), Rezaur Rahman (Intel), Rich Rollman (Microsoft), Rick Gessner (Netscape), Rick Jelliffe (invited), Rob Relyea (Microsoft), Scott Isaacs (Microsoft), Sharon Adler (INSO), Steve Byrne (JavaSoft), Tim Bray (invited), Tim Yu (Oracle), Tom Pixley (Netscape/AOL), Vidur Apparao (Netscape), Vinod Anupam (Lucent).

Thanks to all those who have helped to improve this specification by sending suggestions and corrections (Please, keep bugging us with your issues!).

D.1: Production Systems

This specification was written in XML. The HTML, OMG IDL, Java and ECMAScript bindings were all produced automatically.

Thanks to Joe English, author of cost, which was used as the basis for producing DOM Level 1. Thanks also to Gavin Nicol, who wrote the scripts which run on top of cost. Arnaud Le Hors and Philippe Le Hégaret maintained the scripts.

After DOM Level 1, we used Xerces as the basis DOM implementation and wish to thank the authors. Philippe Le Hégaret and Arnaud Le Hors wrote the Java programs which are the DOM application.

Thanks also to Jan Kärrman, author of html2ps, which we use in creating the PostScript version of the specification.

Glossary

Editors:
Arnaud Le Hors, W3C
Robert S. Sutor, IBM Research (for DOM Level 1)

Several of the following term definitions have been borrowed or modified from similar definitions in other W3C or standards documents. See the links within the definitions for more information.

document element
: There is only one document element in a `Document`. This element node is a child of the `Document` node. See *Well-Formed XML Documents* in XML [XML 1.0].

document order
: There is an ordering, *document order*, defined on all the nodes in the document corresponding to the order in which the first character of the XML representation of each node occurs in the XML representation of the document after expansion of general entities. Thus, the *document element* [p.35] node will be the first node. Element nodes occur before their children. Thus, document order orders element nodes in order of the occurrence of their start-tag in the XML (after expansion of entities). The attribute nodes of an element occur after the element and before its children. The relative order of attribute nodes is implementation-dependent.

event
: An event is the representation of some asynchronous occurrence (such as a mouse click on the presentation of the element, or the removal of child node from an element, or any of unthinkably many other possibilities) that gets associated with an *event target* [p.35] .

event target
: The object to which an *event* [p.35] is targeted.

partially valid
: A node in a DOM tree is *partially valid* if it is *well formed* [p.35] (this part is for comments and processing instructions) and its immediate children are those expected by the content model. The node may be missing trailing required children yet still be considered *partially valid*.

target node
: The target node is the node representing the *event target* [p.35] to which an *event* [p.35] is targeted using the DOM event flow defined in [DOM Level 3 Events].

tokenized
: The description given to various information items (for example, attribute values of various types, but not including the StringType CDATA) after having been processed by the XML processor. The process includes stripping leading and trailing white space, and replacing multiple space characters by one. See the definition of tokenized type.

well-formed document
: A document is *well-formed* if it is tag valid and entities are limited to single elements (i.e., single sub-trees).

References

For the latest version of any W3C specification please consult the list of W3C Technical Reports available at http://www.w3.org/TR.

F.1: Normative references

[DOM Level 3 Core]
Document Object Model Level 3 Core Specification, A. Le Hors, et al., Editors. World Wide Web Consortium, January 2002. This version of the Document Object Model Level 3 Core Specification is http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020114. The latest version of DOM Level 3 Core is available at http://www.w3.org/TR/DOM-Level-3-Core.

[DOM Level 3 Events]
Document Object Model Level 3 Events Specification, T. Pixley, Editor. World Wide Web Consortium, February 2002. This version of the Document Object Model Level 3 Events Specification is http://www.w3.org/TR/DOM-Level-3-Events. The latest version of Document Object Model Level 3 Events is available at http://www.w3.org/TR/DOM-Level-3-Events.

[ECMAScript]
ECMAScript Language Specification, Third Edition. European Computer Manufacturers Association, December 1999. This version of the ECMAScript Language is available at http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM.

[Java]
The Java Language Specification, J. Gosling, B. Joy, and G. Steele, Authors. Addison-Wesley, September 1996. Available at http://java.sun.com/docs/books/jls

[OMG IDL]
"OMG IDL Syntax and Semantics" defined in *The Common Object Request Broker: Architecture and Specification, version 2*, Object Management Group. The latest version of CORBA version 2.0 is available at http://www.omg.org/technology/documents/formal/corba_2.htm.

F.2: Informative references

[XML 1.0]
Extensible Markup Language (XML) 1.0 (Second Edition), T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, Editors. World Wide Web Consortium, 10 February 1998, revised 6 October 2000. This version of the XML 1.0 Recommendation is http://www.w3.org/TR/2000/REC-xml-20001006. The latest version of XML 1.0 is available at http://www.w3.org/TR/REC-xml.

F.2: Informative references

Index