XQuery 1.0 and XPath 2.0 Data Model

W3C Working Draft 30 April 2002

This version:

http://www.w3.org/TR/2002/WD-query-datamodel-20020430/
(available in: HTML, XML)

Latest version:

http://www.w3.org/TR/query-datamodel/

Previous versions:

http://www.w3.org/TR/2001/WD-query-datamodel-20011220/
http://www.w3.org/TR/2001/WD-query-datamodel-20010607/
http://www.w3.org/TR/2001/WD-query-datamodel-20010215/

Editors:

Mary Fernández (XML Query WG), AT&T Labs <mff@research.att.com>

Jonathan Marsh (XSL WG), Microsoft <jmarsh@microsoft.com>

Marton Nagy (XML Query WG), Science Applications International Corporation (SAIC) <marton.nagy@saic.com>

Abstract

This document defines the W3C XQuery 1.0 and XPath 2.0 Data Model, which is the data model of at least [XSLT 2.0], and [XQuery 1.0: A Query Language for XML], and any other specifications that reference it. This data model is based on the data models of [XPath] and [XML Query Data Model] and replaces [XML Query Data Model]. This document is the result of joint work by the [XSL Working Group] and the [XML Query Working Group].

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 Public Working Draft for review by W3C Members and other interested parties. It is a draft document and may be updated, replaced or made obsolete 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 the W3C membership.

The most significant changes since the previous version are the following. The document has been updated to reflect decisions regarding the named typing: Schema Components have been removed and were replaced with Types. Accordingly the constructors and accessors referencing types (element node, attribute node and atomic value constructors and the type() accessor) changed, the example in the appendix updated and related open issues closed. Two functions (node-equal and node-before) have been added to sections 3.1 and 3.2. The terminology and notations (such as the term "atomic value" and the pseudo-code syntax) have been aligned with the other working drafts.

This document has been produced as part of the [W3C Style Activity] and the [W3C XML Activity], following the procedures set out for the W3C Process. The document has been written by the [XSL Working Group] and [XML Query Working Group].

Comments on this document should be sent to the W3C mailing list public-qt-comments@w3.org. (archived at http://lists.w3.org/Archives/Public/public-qt-comments/).

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

Table of Contents

Appendices

---

1 Introduction

This document defines the XQuery 1.0 and XPath 2.0 Data Model, which is the data model of [XSLT 2.0] and [XQuery 1.0: A Query Language for XML] 1.0.

The XQuery 1.0 and XPath 2.0 Data Model (henceforth "data model") serves two purposes. First, it defines precisely the information contained in the input to an XSLT or XQuery processor. Second, it defines all permissible values of expressions in the XSLT, XQuery, and XPath languages. A language is *closed* with respect to a data model if the value of every expression in a language is guaranteed to be in the data model. XSLT 2.0, XQuery 1.0, and XPath 2.0 are all closed with respect to the data

model.

The data model is based on the [XML Information Set] (henceforth "Infoset"), but it requires the following new features to meet the [XPath Requirements Version 2.0] and [XML Query Requirements]:

- Support for XML Schema types. The XML Schema recommendations define features, such as structures ([XMLSchema Part 1]) and simple data types ([XMLSchema Part 2]), that extend the XML Information Set with precise type information.
- Representation of collections of documents and of complex values. ([XML Query Requirements])

As with the Infoset, the XQuery 1.0 and XPath 2.0 Data Model specifies what information in the documents is accessible, but it does not specify the programming-language interfaces or bindings used to represent or access the data.

Every value handled by the data model is either a *sequence* of zero or more *item*s, or an error. An *item* is either a *node* or an *atomic value*. A node is defined in **4 Nodes** and is one of seven node kinds. An atomic value encapsulates an XML Schema atomic type and a corresponding value of that type. They are defined in **5 Atomic Values**. A sequence is an ordered collection of nodes, atomic values, or any mixture of nodes and atomic values. A sequence cannot be a member of a sequence. A single item appearing on its own is modeled as a sequence containing one item. Sequences are defined in **6 Sequences**. The *error* value is defined in **7 Error**.

Note: In XPath 1.0, the data model only defines nodes. The primitive data types (number, boolean, string, node-set) are part of the expression language, not the data model.

The data model can represent various values including not only the input and the output of a query, but all values of expressions used during the intermediate calculations. Examples include the input document or document repository (represented as a document node or a sequence of document nodes), the result of a path expression (represented as a sequence of nodes), the result of an arithmetic or a logical expression (represented as an atomic value), a sequence expression resulting a sequence of integers, dates, QNames or other XML Schema atomic values (represented as a sequence of atomic values), etc. Examples of values that cannot be expressed directly by the data model include schema components, sets containing both atomic values and the error value, atomic values whose type is not an XML Schema atomic type, etc.

In this document, we provide a precise definition of how values in the XQuery 1.0 and XPath 2.0 Data Model are constructed and accessed, and how they relate to values in the Infoset. We note wherever the XQuery 1.0 and XPath 2.0 Data Model differs from that of XPath 1.0.

2 Notation and Pseudo-code Syntax

In addition to using prose, we define the data model using a functional notation. We chose this notation because it is simple and permits a precise definition of the data model, suitable for use by the formal semantics of XQuery. Although the notation has a functional style, we emphasize that the data model can be realized in a variety of programming languages and styles, for example, as object classes and methods in an object-oriented language.

Pseudo-code syntax is highlighted as follows:

```
function dm:typed-value(Node $n) returns AtomicValue*
```

In the psuedo-code syntax, the term *Node* denotes the category of node values, *AtomicValue* denotes the category of atomic values, and *Item* refers to the category of either node values or atomic values. *V** denotes the category of sequence values all of whose members are in category *V*. *V?* and *V+* denote subcategories of *V**: The former denotes the category of sequences containing zero or one items, the latter refers to sequences containing one or more items. In a sequence, *V* may be a *Node* or *AtomicValue*, or the union (choice) of several categories of *Items*. For example, the following denotes a category of sequence containing any combination of comment and processing instruction nodes:

```
(CommentNode | ProcessingInstructionNode)*
```

There are some functions in the data model that are *partial functions*. We use the occurrence indicators *?* or * when specifying the return type of such functions. For example, a node may have one parent node or no parent. If the node argument has a parent, the *dm:parent* accessor returns a singleton sequence. If the node argument does not have a parent, it returns the empty sequence. The signature of *dm:parent* specifies that it returns an empty sequence or a sequence containing one element or

document node:

```
function dm:parent(Node $n) returns (ElementNode | DocumentNode)?
```

The pseudo-code syntax defines functions to construct values, called *constructors*; and functions to access parts of values, called *accessors* (see **[Issue-0033: Unclear relationship between values passed to the constructor, and those returned by the accessor]**). The constructors and accessors defined by the data model are prefixed with *dm*

Note: The XPath 1.0 data model defines accessors, but does not define constructors.

The term *signature* of a function specifies the type of its zero or more inputs and the type of its one output. The following signature denotes a function *f* that takes values in the categories V_1, ..., V_m and returns an output value in the category V_n.

function f(V_1 v_1, ..., V_m v_m) returns V_n

A member of a particular category is a permissible argument to any function that accepts the category, for example, a *ProcessingInstructionNode* is a permissible argument to a function expecting a *Node*.

This document relies on the [XML Information Set]. Information items and properties are indicated by the styles **information item** and **[property]**, respectively.

This document also provides pseudo-code syntax describing the mapping from the Infoset to the data model. To facilitate this we introduce accessors to the required infoset properties through *InfoItem* objects. These infoset accessors are for rhetorical purposes only and are not intended to be exposed outside this specification. We name the accessors of the Infoset using the convention infoset-<*item-name*>-<*property-name*>. Similarly, accessor functions that return PSV Infoset properties use the naming convention psvi-<*item-name*>-<*property-name*>. For example, `infoset-element-attributes` is the accessor that returns an element information item's **[attributes]** property:

```
function infoset-element-attributes(ElementItem $ii) returns AttributeItem*
```

An *InfoItem* is one of eleven kinds of item: document item, element item, attribute item, processing instruction item, unexpanded entity item, character item, comment item, doctype item, unparsed entity item, notation item, and namespace item.

The *infoitem-kind* accessor returns a string value representing the information item's kind, either `"document"`, `"element"`, `"attribute"`, `"character"`, `"namespace"`, `"processing-instruction"`, `"comment"`, `"doctype"`, `"notation"`, or `"unparsed-entity"`.

```
function infoitem-kind(InfoItem $ii) returns xs:string
```

Throughout this document, the namespace prefix `xs` indicates the [XMLSchema Part 1] namespace name `http://www.w3.org/2001/XMLSchema`. The namespace prefixes `xf` and `op` indicate the namespace of the functions and operators defined in [XQuery 1.0 and XPath 2.0 Functions and Operators].They are associated with the namespace names `http://www.w3.org/2001/12/xquery-functions` and `http://www.w3.org/2001/12/xquery-operators` respectively.

The following functions and operators are defined in [XQuery 1.0 and XPath 2.0 Functions and Operators]:

- op:concatenate
- op:item-at
- op:value-equal
- xf:anyURI
- xf:concat
- xf:count
- xf:decimal
- xf:empty
- xf:get-local-name

- xf:get-namespace-uri
- xf:NCName
- xf:QName
- xf:QName-from-uri
- xf:QName-from-string
- xf:string
- xf:sublist

3 Concepts

3.1 Node Identity

Because XML documents are tree-structured, we define the data model using conventional terminology for trees. The data model is a node-labeled, tree-constructor representation, but also includes a concept of node identity. The identity of a node is established when a node-constructor is applied to create the node: each application of a node constructor creates a new node that is identical to itself, and not identical to any other node (see **4 Nodes**).

The *dm:node-equal* function takes two nodes as arguments. It returns true if the arguments are identical and false otherwise.

```
function dm:node-equal(Node $n1, Node $n2) returns xs:boolean
```

This concept should not be confused with the concept of *unique ID*, which is a unique name assigned to an element by the author to represent references using ID/IDREF correlation.

3.2 Document Order

A *document order* is defined on all the nodes in a document. The document node is the first node. Element nodes, comment nodes, and processing instruction nodes occur in the order of their representation in the XML (after expansion of entities). Element nodes occur before their children. The namespace nodes of an element immediately follow the element node (see **[Issue-0051: Document order of shared namespace nodes]**). The relative order of namespace nodes is implementation-dependent. The attribute nodes of an element immediately follow the namespace nodes of the element. The relative order of attribute nodes is implementation-dependent. Reverse document order is the reverse of document order.

The relative order of nodes in distinct documents is implementation-dependent but stable. In other words, given two distinct documents A and B, if a node in document A is before a node in document B, then every node in document A is before every node in document B.

Note: The relative order of free-floating nodes (those not in a document) is not defined. See **[Issue-0050: Relative order of free-floating nodes]**.

The *dm:node-before* function takes two nodes as arguments. It returns true if the first argument is before (but not identical to) the second one in document order and false otherwise

```
function dm:node-before(Node $n1, Node $n2) returns xs:boolean
```

3.3 XML Schemas and the XML Information Set

The data model is defined in terms of the [XML Information Set] after XML Schema validity assessment. XML Schema validity assessment is the process of assessing an XML element information item with respect to an XML Schema and augmenting it and some or all of its descendants with properties that provide information about validity and type assignment. The result of schema validity assessment is an augmented Infoset, known as the Post Schema-Validation Infoset, or PSVI.

The data model supports the following classes of XML documents:

- Schema-validated documents, i.e., those validated with respect to a schema,
- DTD-valid documents, i.e., those documents validated with respect to a DTD, and
- Well-formed documents with no corresponding DTD or schema.

The data model does not support XML documents that are not supported by the XML Information Set, for example, non-well-formed documents and documents that don't conform to XML Namespaces.

Schema-validated documents include documents in which some elements or attributes have been validated by "lax" or "skip" validation ([XMLSchema Part 2]).

An "incompletely validated document" is an XML document that has a corresponding schema but whose schema-validity assessment has resulted in one or more element or attribute information items being assigned values other than 'valid' for the **[validity]** property in the PSVI.

The data model supports incompletely validated documents. See **[Issue-0024: Support for Schema-invalid documents]**.

Note: This implies accommodation for the case where both a DTD and a schema are applied. This will probably require some reconciliation of the [attribute type] property with type information from the PSVI. See issue **[Issue-0004: Schema/DTD]**.

3.4 Types

The data model supports a representation of named types as expanded-QNames. Named types include both the built-in types defined by XML Schema Part 2 and types declared in a schema and imported by a query. Since named types in XML Schema are global, an expanded-QName uniquely identifies such a type, be it simple or complex: The namespace of the expanded-QName is the target namespace of the schema and its local name is the name of the type. The data model does not uniquely identify anonymous types and represents them by xs:anyType or xs:anySimpleType.

Editorial Note: The usage of xs:unknownType or xs:anyType is currently under discussion. An alternative is to represent named types by an optional expanded-QName and use the empty sequence instead of xs:unknownType or xs:anyType.

The data model associates type information with element nodes attribute nodes and atomic values. If this type information is something other than xs:anyType or xs:anySimpleType, the item is guaranteed to be a valid instance of that type as defined by XML Schema. The data model defines an accessor *dm:type()* that returns an expanded-QName corresponding to the type of the element node, attribute node or atomic value, provided that the type is globally declared. It returns xs:anyType or xs:anySimpleType if it is locally declared or if no type information exists. When no type information exists for an element or an attribute node we frequently use the terminology "element with unknown type" or "attribute with unknown simple type".

The data model does not represent element or attribute declaration schema components, but it supports various type-related operations. The semantics of such operations, e.g. checking if a particular instance of an element node has a given type is defined in [XQuery 1.0 Formal Semantics].

3.5 Mapping PSV Infoset additions to Types

This section specifies how the type of an element or attribute node is computed from the PSV Infoset additions that specify validity and type assessment for the node's corresponding information item.

A PSV element (attribute) information item has a **[validity]** and a **[type definition]** property. The **[validity]** property may be *"valid"*, *"invalid"*, or *"notKnown"* and reflects the outcome of the element's (attribute's) schema-validity assessment. The **[type definition]** property contains an element information item that is isomorphic to the XML Schema type definition of the element or attribute information item. Roughly speaking these two properties are used to compute the type of an element or attribute in the data model, which corresponds to the {target namespace} and the {name} of the **[type definition]** property if **[validity]** is *"valid"*, or to xs:anyType (xs:anySimpleType) otherwise. Notice that the only information that can be inferred from an invalid or not known validity value is that the information item is well-formed, therefore, we must associate some general type information with the element or attribute node.

The precise definition of the type of an element or attribute information item is slightly more complex due to the following

factors. First, XML Schema only guarantees the existence of either the **[type definition]** property, or the **[type definition namespace]**, **[type definition name]** and **[type definition anonymous]** properties. Second, if the type definition refers to a union type, there are further properties defined, that refer to the type definition which actually validated the element item's normalized value. These properties are either the **[member type definition]**, or the **[member type definition namespace]**, **[member type definition name]** and **[member type definition anonymous]** properties. If these are available the type of an element or attribute will refer to the member type that actually validated the schema normalized value. Having identified all the relevant properties we can now define how to compute a type.

The *type* of an element information item is represented by an xs:QName whose namespace and local name corresponds to the first applicable item in the following list:

- xs:anyType if the **[validity]** property is *"invalid"* or *"notKnown"*, or
- the {target namespace} and {name} properties of the **[member type definition]** schema component if that exists, or
- the {target namespace} and {name} properties of the **[type definition]** schema component if that exists, or
- the **[member type definition namespace]** and the **[member type definition name]** if **[member type definition anonymous]** exists and is false, or
- the **[type definition namespace]** and the **[type definition name]** if **[type definition anonymous]** exists and is false, or
- it corresponds to xs:anyType.

The *type* of an attribute information item is represented by an xs:QName. Its definition is the same as for element information items except for using xs:anySimpleType instead of xs:anyType above.

We will find it convenient to define the following function that will be used later in various sections. The *infoitem-to-type* function takes an element or an attribute information item and returns an xs:QName that corresponds to the type of its argument.

```
function infoitem-to-type((ElementItem|AttributeItem) $ii) returns xs:QName

define function infoitem-to-type(ElementItem $ii)
{
   if (psvi-element-validity($ii) ne "valid") then
      return xf:QName-from-string("xs:anyType")
   else if (psvi-element-member-type-definition($ii) ne ()) then
      let $typedef:= psvi-element-member-type-definition($ii)
      let $name:= psvi-typedefinition-name($typedef)
      let $namespace:=  psvi-typedefinition-namespace($typedef)
      return xf:QName-from-uri($namespace,$name)
   else if (psvi-element-type-definition($ii) ne ()) then
      let $typedef:= psvi-element-type-definition($ii)
      let $name:= psvi-typedefinition-name($typedef)
      let $namespace:=  psvi-typedefinition-namespace($typedef)
      return xf:QName-from-uri($namespace,$name)
   else if (psvi-element-member-type-definition-anonymous($ii) eq false) then
      let $name:= psvi-element-member-type-definition-name($ii)
      let $namespace:= psvi-element-member-type-definition-namespace($ii)
      return xf:QName-from-uri($namespace,$name)
   else if (psvi-element-type-definition-anonymous($ii) eq false) then
      let $name:= psvi-element-type-definition-name($ii)
      let $namespace:= psvi-element-type-definition-namespace($ii)
      return xf:QName-from-uri($namespace,$name)
   else
      return xf:QName-from-string("xs:anyType")
}
```

```
function infoitem-to-type(AttributeItem $ii)
{
   if (psvi-attribute-validity($ii) ne "valid") then
      return xf:QName-from-string("xs:anySimpleType")
   else if (psvi-attribute-member-type-definition($ii) ne ()) then
      let $typedef:= psvi-attribute-member-type-definition($ii)
      let $name:= psvi-typedefinition-name($typedef)
      let $namespace:=  psvi-typedefinition-namespace($typedef)
      return xf:QName-from-uri($namespace,$name)
   else if (psvi-attribute-type-definition(a) ne ()) then
      let $typedef:= psvi-attribute-type-definition($ii)
      let $name:= psvi-typedefinition-name($typedef)
      let $namespace:=  psvi-typedefinition-namespace($typedef)
      return xf:QName-from-uri($namespace,$name)
   else if (psvi-attribute-member-type-definition-anonymous($ii) eq false) then
      let $name:= psvi-attribute-member-type-definition-name($ii)
      let $namespace:= psvi-attribute-member-type-definition-namespace($ii)
      return xf:QName-from-uri($namespace,$name)
   else if (psvi-attribute-type-definition-anonymous($ii) eq false) then
      let $name:= psvi-attribute-type-definition-name($ii)
      let $namespace:= psvi-attribute-type-definition-namespace($ii)
      return xf:QName-from-uri($namespace,$name)
   else
      return xf:QName-from-string("xs:anySimpleType")
}
```

Editorial Note: MF: Following two cases need to be completed:

Given information items that validate with respect to a DTD, ...

Given information items from a document a well-formed document, with no corresponding DTD or Schema...

3.6 Text Nodes and Sequences of Atomic Values

The data model supports two representations of the character data that appears as element content: text nodes or a sequence of atomic values. Similarly, character data of attributes can also be represented in two ways: as a string or a sequence of atomic values. A text node represents a string of consecutive character information items and never has another text node as its immediately following sibling. An element node, for example, has child nodes that may include text nodes, comment nodes, processing instruction nodes, and other element nodes. In addition, the text content of an element may be interpreted as a sequence of atomic values, such as an integer, a date, or a sequence of prices. To illustrate, consider an element node whose type is a sequence of double-precision numbers. The element's children are three nodes: a text node with string contents " 12.00 ", followed by a comment node, followed by a text node with contents " 13.0", whereas its atomic value is a sequence containing the double-precision numbers 12.0 and 13.0.

It is noted that the two representations of character data are not equivalent. A sequence of atomic values contains more type information then a text-node or string representation when the atomic values are of different types (e.g. a sequence containing an integer, a date and a string). Since the text node child does not contain enough type information to reconstruct this typed value, an implementation must store the element's typed value separately from the text node. This precludes a "text node only" implementation.

3.7 Ignoring Comments, Processing Instructions, and Whitespace

Although the data model is able to represent comments, processing instructions, and insignificant whitespace, preservation of this information may be unnecessary and onerous for some applications.

Construction of a document from an XML information set is parameterized by three flags, *ignore-comments*,

ignore-processing-instructions, and *ignore-whitespace*. If the *ignore-comments* flag is true, comment nodes are not preserved in the data model. If the *ignore-processing-instructions* flag is true, processing-instruction nodes are not preserved in the data model. If the *ignore-whitespace* flag is true, insignificant whitespace is not preserved.

```
ignore-comments                : xs:boolean
ignore-processing-instructions : xs:boolean
ignore-whitespace              : xs:boolean
```

Note: By whom these flags are set is not defined. See **[Issue-0040: Setting and examining construction flags]**.

Expressions which rely upon the presence or absence of comments, processing instructions, or insignificant whitespace may produce different results for two data models created from the same infoset (XML document), when each data model is constructed with different settings of these flags.

Insignificant whitespace is defined as a text node that:

1. contains no characters other than whitespace characters (as defined in XML 1.0), and
2. has a parent element with a **[validity]** property with the value "valid", and a **[type definition]** property yielding a complex type with *content-type* of *element-only*.

Note: See **[Issue-0034: Interaction of insignificant whitespace with comments]**. Removal of insignificant whitespace might be performed automatically when consistent with the schema. See **[Issue-0028: Whitespace handling]**. XSLT's whitespace handling mechanism needs to be supported; see **[Issue-0057: Support for XSLT whitespace stripping]**.

4 Nodes

The category of *Node* values contains seven distinct kinds of nodes: document, element, attribute, text, namespace, processing instruction, and comment. The seven kinds of nodes are defined in the following subsections.

Each kind of node has its own constructor. The effect of a node constructor is to create a new node with a unique identity, distinct from all other nodes.

A set of accessors is defined on all seven kinds of Nodes. Some accessors return a constant empty sequence on certain node kinds.

- The *dm:node-kind* accessor returns a string value representing the node's kind: either `"document"`, `"element"`, `"attribute"`, `"text"`, `"namespace"`, `"processing-instruction"`, or `"comment"`.
- The *dm:name* accessor returns a sequence containing one *expanded QName* for node kinds that can have names. For other node kinds, it always returns an empty sequence. An expanded QName is in the value space of *xs:QName*, and consists of a namespace URI and a local name. See **[Issue-0063: Is prefix preserved?]**, **[Issue-0070: Should the name accessor return "" or ()?]**.
- The *dm:base-uri* accessor returns a sequence containing zero or one uri references for node kinds that can have a base-uri (document nodes and element nodes). For other node kinds, it always returns the empty sequence.
- The *dm:string-value* accessor returns the node's string representation. For some kinds of nodes, this is part of the node; for other kinds of nodes, it is computed from the *dm:string-value* of its descendant nodes.
- The *dm:typed-value* accessor returns a sequence of atomic values corresponding to the node. This may be a non-empty sequence for element and attribute nodes, but it is always the empty sequence for other node kinds.
- The *dm:parent* accessor returns a sequence containing zero or one nodes for node kinds that can have parents. For other node kinds, it always returns the empty sequence. Every node has at most one parent, which is either an element node or the document node. A node that has no parent is regarded as the root of a tree. The one exception is a namespace node, which never has a parent.

  Note: In XPath 1.0, Namespace nodes have parents.

- The *dm:children* accessor returns a sequence containing zero or more nodes for node kinds that can have children (document nodes and element nodes). For other node kinds, it always returns the empty sequence. A document node or an element node is the parent of each of its child nodes. Nodes never share children: if two nodes have distinct identities, then no child of one node will be a child of the other node.
- The *dm:attributes* accessor returns a sequence containing zero or more attribute nodes for element nodes. For other node kinds, it always returns the empty sequence.
- The *dm:namespaces* accessor returns a sequence containing zero or more namespace nodes corresponding to the in-scope namespaces for element nodes. For other node kinds, it always returns the empty sequence.
- The *dm:type* accessor returns a sequence containing one expanded-QName corresponding to the type of the element node or attribute node, provided that the type is globally declared. It returns xs:anyType or xs:anySimpleType if it is locally declared or if no type information exists. For other node kinds, it always returns the empty sequence.
- The *dm:unique-ID* accessor returns a sequence containing zero or one ID for element nodes. For other node kinds, it always returns the empty sequence.

The return types of the Node accessors are given below. Some kinds of nodes further restrict the return types; notably, many node kinds return a constant empty sequence for some of the accessors.

```
function dm:node-kind(Node $n)    returns xs:string
function dm:name(Node $n)         returns xs:QName?
function dm:base-uri(Node $n)     returns xs:anyURI?
function dm:string-value(Node $n) returns xs:string
function dm:typed-value(Node $n)  returns AtomicValue*
function dm:parent(Node $n)       returns (ElementNode | DocumentNode)?
function dm:children(Node $n)     returns (ElementNode | TextNode
                                           | ProcessingInstructionNode
                                           | CommentNode)*
function dm:attributes(Node $n)   returns AttributeNode*
function dm:namespaces(Node $n)   returns NamespaceNode*
function dm:type(Node $n)         returns xs:QName?
function dm:unique-ID(Node $n)    returns xs:ID?
```

A tree contains a root plus all nodes that are reachable directly or indirectly from the root via the *dm:children*, *dm:attributes*, and *dm:namespace* accessors. Every node belongs to exactly one tree, and every tree has exactly one root node. A tree whose root node is a document node is referred to as a *document*. A tree whose root node is some other kind of node is referred to as a *fragment*.

4.1 Documents

Document Node accessors	possible values
dm:node-kind	"document"
dm:name	empty sequence
dm:base-uri	xs:anyURI
dm:string-value	xs:string
dm:typed-value	empty sequence
dm:parent	empty sequence
dm:children	arbitrary sequence of one or more element nodes, zero or more processing instruction nodes, and zero or more comment nodes
dm:attributes	empty sequence
dm:namespaces	empty sequence
dm:type	empty sequence
dm:unique-ID	empty sequence

A document is represented by a document node, which corresponds to a **document information item**.

Note: Document nodes and XPath 1.0 root nodes are essentially identical.

A document node does not have an *expanded-QName*.

The *dm:base-uri* of the document corresponds to the **[base URI]** property.

The *dm:string-value* of the document node is the concatenation of the string-values of all text-node descendants of the document node in document order.

The *dm:parent* of the document node is always the empty sequence. A document node always represents the root of a tree.

The *dm:children* of the document node are nodes corresponding to the information items found in the **[children]** property, omitting any **document type declaration information items**.

Note: There is no way to determine what DTD might apply to the data model. See **[Issue-0042: System Id and Public Id are not exposed]**.

In a well-formed document, the children of the document node consist exclusively of element nodes, processing-instruction nodes, and comment nodes, and exactly one of these children is an element node. A document node in the data model is more permissive: it permits more than one element node as a child and also permits text nodes as children.

Note: See **[Issue-0041: Document node permissiveness unnecessary]** and **[Issue-0074: Do we need Document fragments]**.

A document node has the constructor *dm:document-node*, which takes a base URI value and a non-empty sequence of its children nodes. Like all other node constructors, the document-node constructor has the effect of creating a new node with a unique identity, distinct from all other nodes.

```
function dm:document-node(
          xs:anyURI $baseuri,
          (ElementNode | TextNode | ProcessingInstructionNode
           | CommentNode)+ $children)
        returns DocumentNode
```

The accessors *dm:base-uri* and *dm:children* return a document node's constituent parts:

```
function dm:base-uri(DocumentNode $n) returns xs:anyURI
function dm:children(DocumentNode $n)
        returns (ElementNode | TextNode
                 | ProcessingInstructionNode | CommentNode)+
```

The accessors *dm:node-kind* and *dm:string-value* also apply to document nodes and return results other than the empty sequence. The accessors *dm:name*, *dm:typed-value*, *dm:parent*, *dm:attributes*, *dm:namespaces*, *dm:type* and *dm:unique-ID* applied to a document node always return the empty sequence.

A document node is constructed from a Document Information Item by the *infoitem-to-document-node* function:

```
/* Accessors for document information items: */
function infoset-document-children(DocumentItem $ii)
        returns (ElementItem | ProcessingInstructionItem
                 | CommentItem | DocTypeItem)*
function infoset-document-base-uri(DocumentItem $ii) returns xs:anyURI

define function infoitem-to-document-node(DocumentItem $ii) returns DocumentNode
{
  let $kids:= collapse-text-nodes(sequence-map(infoitem-to-node,
                                  infoset-document-children($ii)))
```

```
  return dm:document-node(infoset-document-base-uri($ii), $kids)
}
```

The `collapse-text-nodes` function synthesizes a single text node from multiple text nodes. The `sequence-map` function applies its first function argument to each member of its second sequence argument and returns a new sequence containing the result of applying the function to each member of the sequence. In the pseudo-code above, `infoitem-to-node` is applied to each child of the **document information item** *$ii* and a new sequence of children nodes is constructed, each of which is a *Node*. The constructor `dm:document-node` constructs the document node in the data model.

```
function sequence-map(function $map, Item* $seq) returns Item*
```

The *infoitem-to-node* function maps an information item to a sequence of zero or one node.

```
define function infoitem-to-node(InfoItem $ii) returns Node?
{
  return
    if (infoitem-kind($ii) = "element") then
      infoitem-to-element-node($ii)
    else if (infoitem-kind($ii) = "character") then
      infoitem-to-single-character-text-node($ii)
    else if (infoitem-kind($ii) = "processing-instruction") then
      if (not(ignore-processing-instructions)) then
        infoitem-to-processing-instruction-node($ii)
      else empty-sequence()
    else if (infoitem-kind($ii) = "comment") then
      if (not(ignore-comments)) then
        infoitem-to-comment-node($ii)
      else empty-sequence()
    else
      {-- infoitem-kind($ii) = "doctype" | "notation" | "unparsed-entity" --}
      empty-sequence()
}
```

4.2 Elements

Element Node accessors	possible values
dm:node-kind	"element"
dm:name	xs:QName
dm:base-uri	xs:anyURI
dm:string-value	xs:string
dm:typed-value	sequence of zero or more atomic values
dm:parent	sequence of zero or one element or document node
dm:children	sequence of zero or more element nodes, zero or more processing instruction nodes, zero or more comment nodes, and zero or more text nodes
dm:attributes	sequence of zero or more attribute nodes
dm:namespaces	sequence of zero or more namespace nodes
dm:type	xs:QName
dm:unique-ID	zero or one xs:ID

Each element node corresponds to an **element information item**.

An element node has an *expanded-QName*. The local part of the *expanded-QName* corresponds to the **[local name]** property. The namespace URI of the *expanded-QName* of the element node corresponds to the **[namespace name]** property.

The *dm:parent* of the element node corresponds to the node corresponding to the **[parent]** property.

An element node has an associated *dm:typed-value*, which is a sequence of zero or more atomic values. Examples of such values would be a sequence containing an integer, or an atomic value of some user-defined type or several dates, etc. For a document with a schema, the element's typed-value corresponds to the **[schema normalized value]** PSVI property if that exists, or the empty sequence otherwise. But if the element node has a complex type with complex content then *dm:typed-value* returns the error value. Furthermore, if the type of the element node is xs:anyType or xs:anySimpleType then *dm:typed-value* returns its string value with unknown simple type. In particular this implies the following: If an element was created with an xsi:nil attribute set to true, then its *dm:typed-value* is the empty sequence. For an element in a well-formed document with no associated schema, the element's *dm:typed-value* is its string value with unknown simple type. See **[Issue-0071: Magic Attributes]**.

The *dm:children* of the element node correspond to the element, comment, processing instruction, and character information items appearing in the **[children]** property. This correspondence is not one-to-one, as consecutive **character information item** children are coalesced into a single text node. Because the data model requires that all general entities be expanded, there will never be **unexpanded entity reference information item** children.

The *dm:attributes* of the element node are nodes corresponding to **attribute information items** appearing in the **[attributes]** property. The attributes of an element always have distinct names.

The *dm:namespaces* of the element node are nodes corresponding to **namespace information items** appearing in the **[in-scope namespaces]** property. The namespaces of an element always have distinct prefixes. See **[Issue-0062: Namespace fixups required]**.

The *dm:type* of an element is an xs:QName and its namespace and local name correspond to the following:

- xs:anyType if the **[validity]** property is *"invalid"* or *"notKnown"*, or
- the {target namespace} and {name} properties of the **[member type definition]** schema component if that exists, or
- the {target namespace} and {name} properties of the **[type definition]** schema component if that exists, or
- the **[member type definition namespace]** and the **[member type definition name]** if **[member type definition anonymous]** exists and is false, or
- the **[type definition namespace]** and the **[type definition name]** if **[type definition anonymous]** exists and is false, or
- it corresponds to xs:anyType.

It is noted that if the type referenced would be a union type then type refers to the member type that actually validated the schema normalized value. See **[Issue-0064: Including type definition in element constructors]**.

The *unique ID* of the element node is an identifier optionally assigned by the user. It corresponds to the **[normalized value]** property of the **attribute information item** in the **[attributes]** property that has a type *ID*, if one exists.

Note: Using this definition, only IDs declared in a DTD are effective. See **[Issue-0004: Schema/DTD]**. Even so, this definition is not backward compatible with XPath 1.0. See **[Issue-0038: XPath 1.0 treatment of non-unique IDs]**. Furthermore, it doesn't even work as spec'd, see **[Issue-0044: Unable to construct an element with unique ID]**.

An element node can be constructed in one of two ways: *dm:element-node* is useful for constructing an element from the PSVI, *dm:element-node-atomic* is useful when constructing an element via embedded expressions. The difference in the constructors is whether the children of the node are specified as a sequence of nodes or as a sequence of atomic values.

The constructor *dm:element-node* takes an expanded-QName, a sequence of namespace nodes, a sequence of attribute nodes, a sequence of child nodes, and the node's type.

```
function dm:element-node(
           xs:QName $qname,
           NamespaceNode* $nsnodes,
           AttributeNode* $attrnodes,
           (ElementNode | TextNode | ProcessingInstructionNode
            | CommentNode)* $children,
```

```
          xs:QName $type)
        returns ElementNode
```

The constructor *dm:element-node-atomic* takes an expanded-QName, a sequence of namespace nodes, a sequence of attribute nodes, a sequence of atomic values, and the node's type.

```
function dm:element-node-atomic(
          xs:QName $qname,
          NamespaceNode* $nsnodes,
          AttributeNode* $attrnodes,
          AtomicValue* $values,
          xs:QName $type)
        returns ElementNode
```

Like all other node constructors, the element node constructors has the effect of creating a new node with a unique identity, distinct from all other nodes.

To guarantee that the parent-child relationship is invertible, the element constructors logically create a copy of all of their namespace, attribute, and children arguments and set the parent property of these nodes to the newly created element node. As long as the parent-child constraint is satisfied, an implementation of the data model may choose to use specialized techniques to avoid creating physical copies of the arguments to an element constructor. See **[Issue-0052: Element constructor copies nodes?]**.

Note: An alternative interface is suggested by James Clark: See **[Issue-0019: Element constructor that performs schema processing]**.

Neither constructor allows specifying the children of the node as a heterogeneous sequence, i.e. a sequence mixing nodes and atomic values. It is still possible to construct an element (with mixed content) from such a sequence in two steps: First the heterogeneous sequence will need to be turned into a homogeneous one by converting the atomic values to text nodes, then the now homogeneous sequence can be passed to the element-node constructor. Note that the precise type information of the atomic values is lost in the process, which is inline with XML Schema, that cannot represent the type of such mixed content.

The accessors *dm:name*, *dm:namespaces*, *dm:attributes* and *dm:type* return an element node's constituent parts. The *dm:children* accessor returns the sequence of children nodes for an element node if it was created with *dm:element-node* and it returns a singleton sequence containing a text node corresponding to the sequence of atomic values if it was created with *dm:element-node-atomic*. The string value of the text node in the latter case is the string value of the sequence of atomic values, which is obtained by inserting a #x20 (space) separator between the dm:string-values of the individual atomic values and concatenating the result (as defined in **6 Sequences**). See **[Issue-0068: Retaining the type of a sequence of heterogeneous simple typed values.]**.

```
function dm:name(ElementNode $n)       returns xs:QName
function dm:namespaces(ElementNode $n) returns NamespaceNode*
function dm:attributes(ElementNode $n) returns AttributeNode*
function dm:children(ElementNode $n)   returns (ElementNode | TextNode
                                                | ProcessingInstructionNode
                                                | CommentNode)*
function dm:type(ElementNode $n)       returns xs:QName
```

The accessor function *dm:typed-value* returns a sequence of zero or more atomic values. If the element was created with *dm:element-node*, *dm:typed-value* corresponds to the **[schema normalized value]** PSVI property if that exists, or the empty sequence otherwise. But if the element node has a complex type with complex content then *dm:typed-value* returns the error value. Furthermore, if the type of the element node is xs:anyType or xs:anySimpleType then *dm:typed-value* returns its string value with unknown simple type. If the element was created with *dm:element-node-atomic*, *dm:typed-value* returns the sequence of atomic values that were used to construct the element.

```
function dm:typed-value(ElementNode $n) returns AtomicValue*
```

The *dm:string-value* of an element node returns the concatenation of the string-values of all text-node descendants of the element node in document order if it was created with *dm:element-node* and it returns the string representation of the sequence of atomic values if it was created with *dm:element-node-atomic*. See **[Issue-0073: Whitespace normalization of the string-value of elements with simple content]**.

```
function dm:string-value(ElementNode $n) returns xs:string
```

If an element has a unique ID, the accessor function *dm:unique-ID* returns a sequence containing the unique ID of the node; otherwise, it returns the empty sequence.

```
function dm:unique-ID(ElementNode $n) returns xs:ID?
```

The node accessors *dm:base-uri*, *dm:node-kind*, *dm:parent*, and *dm:string-value* also apply to element nodes.

An element node is constructed from an Element Information Item by the *infoitem-to-element-node* function:

```
/* Accessors for element information items: */
function infoset-element-namespace-name(ElementItem $ii)       returns xs:anyURI?
function infoset-element-local-name(ElementItem $ii)           returns xs:string
function infoset-element-children(ElementItem $ii)             returns InfoItem*
function infoset-element-attributes(ElementItem $ii)           returns AttributeItem*
function infoset-element-in-scope-namespaces(ElementItem $ii)  returns NamespaceItem*
function infoset-element-base-uri(ElementItem $ii)             returns xs:anyURI

function psvi-element-validity(ElementItem $ii)                returns xs:string
function psvi-element-type-definition(ElementItem $ii)         returns ElementItem
function psvi-element-schema-normalized-value(ElementItem $ii) returns xs:string

define function infoitem-to-element-node(ElementItem $ii) returns ElementNode
{
  let $qname:= xf:QName-from-uri(infoset-element-namespace-name($ii),
                                 infoset-element-local-name($ii)),
      $nsnodes:= sequence-map(infoitem-to-namespace-node,
                              infoset-element-in-scope-namespaces($ii)),
      $attrnodes:= sequence-map(infoitem-to-attribute-node,
                                infoset-element-attributes($ii)),
      $kids:= collapse-text-nodes(sequence-map(
                infoitem-to-node, infoset-element-children($ii))),
      $type:= infoitem-to-type($ii)
  return dm:element-node($qname, $nsnodes, $attrnodes, $kids, $type)
}
```

Note: [base URI] is discarded. See **[Issue-0030: Base URI is a property of element nodes]**.

4.3 Attributes

Attribute Node accessors	possible values
dm:node-kind	"attribute"
dm:name	xs:QName
dm:base-uri	empty sequence
dm:string-value	xs:string
dm:typed-value	sequence of zero or more atomic values
dm:parent	sequence of zero or one element nodes

dm:children	empty sequence
dm:attributes	empty sequence
dm:namespaces	empty sequence
dm:type	xs:QName
dm:unique-ID	empty sequence

Each element node has an associated set of attribute nodes, each corresponding to an **attribute information item**.

An attribute node has an *expanded-QName*. The local part of the *expanded-QName* corresponds to the **[local name]** property. The namespace name of the *expanded-QName* corresponds to the **[namespace name]** property.

An attribute node has an associated *dm:string-value*, which corresponds to the **[normalized value]** property.

An attribute node also has a *dm:typed-value*. For a document with a schema, the attribute's typed-value corresponds to the **[schema normalized value]** PSVI property. But if the type of the attribute node is xs:anySimpleType then *dm:typed-value* returns its string value with unknown simple type.

For convenience, the element node is called the "parent" of each of these attribute nodes even though an attribute node is not a "child" of its parent element. The *dm:parent* of the attribute node corresponds to the **[owner element]** property.

The *dm:type* of an attribute is an xs:QName and its namespace and local name correspond to the following:

- xs:anySimpleType if the **[validity]** property is *"invalid"* or *"notKnown"*, or
- the {target namespace} and {name} properties of the **[member type definition]** schema component if that exists, or
- the {target namespace} and {name} properties of the **[type definition]** schema component if that exists, or
- the **[member type definition namespace]** and the **[member type definition name]** if **[member type definition anonymous]** exists and is false, or
- the **[type definition namespace]** and the **[type definition name]** if **[type definition anonymous]** exists and is false, or
- it corresponds to xs:anySimpleType.

It is noted that if the type referenced would be a union type then type refers to the member type that actually validated the schema normalized value.

An attribute node can be constructed in one of two ways: *dm:attribute-node* is useful for constructing an attribute from the PSVI, *dm:attribute-node-atomic* is useful when constructing an attribute with embedded expressions.

The constructor *dm:attribute-node* takes the attribute's name, a string value and the attribute's type.

```
function dm:attribute-node(xs:QName $qname, xs:string $value, xs:QName $type)
         returns AttributeNode
```

The constructor *dm:attribute-node-atomic* takes the attribute's name, a list of atomic values and the attribute's type.

```
function dm:attribute-node-atomic(xs:QName $qname, AtomicValue* $value, xs:QName
$type)
         returns AttributeNode
```

Like all other node constructors, the attribute node constructors have the effect of creating a new node with a unique identity, distinct from all other nodes.

The accessors *dm:name* and *dm:type* return an attribute's constituent parts. The accessor *dm:string-value* returns an attribute's constituent part if it was created with *dm:attribute-node* and it returns the string representation of the sequence of atomic values if it was created with *dm:attribute-node-atomic*. The accessor function *dm:typed-value* returns a sequence of the atomic values of an attribute. In particular if the type of the attribute node is xs:anySimpleType then *dm:typed-value* returns its string value with unknown simple type.

```
function dm:name(AttributeNode $n)          returns xs:QName
function dm:string-value(AttributeNode $n) returns xs:string
function dm:type(AttributeNode $n)          returns xs:QName
function dm:typed-value(AttributeNode $n)  returns AtomicValue*
```

The node accessors *dm:node-kind* and *dm:parent* also apply to attribute nodes and may return results other than the empty sequence. The accessors *dm:base-uri*, *dm:children*, *dm:attributes*, *dm:namespaces* and *dm:unique-ID* applied to an attribute node always return the empty sequence.

An attribute node is constructed from an Attribute Information Item by the *infoitem-to-attribute-node* function:

```
/* Accessors for attribute information items: */
function infoset-attribute-namespace-name(AttributeItem $ii)   returns xs:anyURI?
function infoset-attribute-local-name(AttributeItem $ii)       returns xs:string
function infoset-attribute-normalized-value(AttributeItem $ii) returns xs:string
function infoset-attribute-owner-element(AttributeItem $ii)    returns ElementItem

function psvi-attribute-validity(AttributeItem $ii)                 returns xs:string
function psvi-attribute-type-definition(AttributeItem $ii)          returns
ElementItem
function psvi-attribute-schema-normalized-value(AttributeItem $ii) returns xs:string

define function infoitem-to-attribute-node(AttributeItem $ii) returns AttributeNode
{
  let $qname:= xf:QName-from-uri(infoset-attribute-namespace-name($ii),
                                 infoset-attribute-local-name($ii)),
      $type:= infoitem-to-type($ii)
  return dm:attribute-node($qname, infoset-attribute-normalized-value($ii), $type)
}
```

Editorial Note: JM: Update the above to accommodate the possibility of schema-less and DTD validation.

4.4 Namespaces

Namespace Node accessors	possible values
dm:node-kind	"namespace"
dm:name	sequence of zero or one xs:QName
dm:base-uri	empty sequence
dm:string-value	xs:string
dm:typed-value	empty sequence
dm:parent	empty sequence
dm:children	empty sequence
dm:attributes	empty sequence
dm:namespaces	empty sequence
dm:type	empty sequence
dm:unique-ID	empty sequence

Each element node has an associated set of namespace nodes, each corresponding to a **namespace information item**.

A namespace node has an *expanded-QName*. The local part of the *QName* corresponds to the **[prefix]** property. The namespace URI of the *QName* is the empty sequence.

The *dm:string-value* of the namespace node corresponds to the **[namespace name]** property.

A namespace node has no *dm:parent*.

Note: From XPath 1.0 : "The *parent* of the namespace node is the element node in whose namespaces collection this node appears." See **[Issue-0039: Parent of namespace nodes]** and **[Issue-0060: Sharing namespace nodes]**, and **[Issue-0061: No access to prefix on free-floating attributes]**.

A namespace node has the constructor *dm:namespace-node*, which takes a namespace prefix and the absolute URI of the namespace being declared. The namespace prefix may be the empty sequence. If the URI is the zero-length string, the prefix must be the empty sequence. Like all other node constructors, the namespace node constructor has the effect of creating a new node with a unique identity, distinct from all other nodes.

```
function dm:namespace-node(xs:string? $prefix, xs:string $uri)
         returns NamespaceNode
```

A namespace node's constituent parts may be obtained by applying the accessors *dm:name* (with the function xf:get-local-name) and *dm:string-value*.

```
function dm:name(NamespaceNode $n)         returns xs:QName?
function dm:string-value(NamespaceNode $n) returns xs:string
```

The node accessor *dm:node-kind* also applies to namespace nodes and returns a result other than the empty sequence. The accessors *dm:base-uri*, *dm:typed-value*, *dm:parent*, *dm:children*, *dm:attributes*, *dm:namespaces*, *dm:type* and *dm:unique-ID* applied to a namespace node always return the empty sequence.

A namespace node is constructed from a Namespace Information Item by the *infoitem-to-namespace-node* function:

```
function infoset-namespace-prefix(NamespaceItem $ii) returns xs:string?
function infoset-namespace-namespace-name(NamespaceItem $ii) returns xs:string

define function infoitem-to-namespace-node(NamespaceItem $ii) returns NamespaceNode
{
   return dm:namespace-node(infoset-namespace-prefix($ii),
                            infoset-namespace-namespace-name($ii))
}
```

4.5 Processing Instructions

Processing Instruction Node accessors	possible values
dm:node-kind	"processing-instruction"
dm:name	xs:QName
dm:base-uri	empty-sequence
dm:string-value	xs:string
dm:typed-value	empty sequence
dm:parent	sequence of zero or one element or document nodes
dm:children	empty sequence
dm:attributes	empty sequence
dm:namespaces	empty sequence
dm:type	empty sequence
dm:unique-ID	empty sequence

A processing instruction node corresponds to a **processing instruction information item**. There are no processing instruction nodes for processing instructions that are children of a **document type declaration information item**.

A processing instruction node has an *expanded-QName*. The local part of the *expanded-QName* corresponds to the **[target]** property. The namespace URI of the *expanded-QName* is the empty sequence. The local part is a string value that must be an

NCName.

The string '?>' may not occur within a processing instruction's target value ([XML Recommendation]).

The *dm:string-value* of the processing instruction node corresponds to the **[content]** property.

The *dm:parent* of the processing instruction node corresponds to the **[parent]** property.

A processing-instruction node has the constructor *dm:processing-instruction-node*, which takes an NCName representing the target and a string representing the content. Like all other node constructors, the processing node constructor has the effect of creating a new node with a unique identity, distinct from all other nodes.

```
function dm:processing-instruction-node(xs:NCName $target, xs:string $content)
         returns ProcessingInstructionNode
```

A processing instruction's constituent parts may be obtained by applying the accessors *dm:name* (with the function xf:get-local-name) and *dm:string-value*.

```
function dm:name(ProcessingInstructionNode $n)         returns xs:QName
function dm:string-value(ProcessingInstructionNode $n) returns xs:string
```

The node accessors *dm:node-kind* and *dm:parent* also apply to processing-instruction nodes and may return results other than the empty sequence. The accessors *dm:base-uri*, *dm:typed-value*, *dm:children*, *dm:attributes*, *dm:namespaces*, *dm:type* and *dm:unique-ID* applied to a processing-instruction node always return the empty sequence.

A processing-instruction node is constructed from an Processing Instruction Information Item by the *infoitem-to-processing-instruction-node* function:

```
/* Accessors for processing instruction information items */
function infoset-processing-instruction-target(ProcessingInstructionItem $ii)
         returns xs:string
function infoset-processing-instruction-content(ProcessingInstructionItem $ii)
         returns xs:string

define function infoitem-to-processing-instruction-node(ProcessingInstructionItem
$ii)
       returns ProcessingInstructionNode
{
  return dm:processing-instruction-node(
           xf:NCName(infoset-processing-instruction-target($ii)),
           infoset-processing-instruction-content($ii))
}
```

4.6 Comments

Comment Node accessors	possible values
dm:node-kind	"comment"
dm:name	empty sequence
dm:base-uri	empty-sequence
dm:string-value	xs:string
dm:typed-value	empty sequence
dm:parent	sequence of zero or one element or document nodes
dm:children	empty sequence
dm:attributes	empty sequence
dm:namespaces	empty sequence

dm:type	empty sequence
dm:unique-ID	empty sequence

A comment node corresponds to a **comment information item**. There are no comment nodes for comments that are children of a **document type declaration information item**.

A comment node does not have an *expanded-QName*.

The *dm:string-value* of the comment node corresponds to the **[content]** property.

The *dm:parent* of the comment node corresponds to the **[parent]** property.

The string "--" (double-hyphen) must not occur within a comment's string value ([XML Recommendation]).

A comment node has the constructor *dm:comment-node*, which takes a string value. Like all other node constructors, the comment node constructor has the effect of creating a new node with a unique identity, distinct from all other nodes.

```
function dm:comment-node(xs:string $content) returns CommentNode
```

A comment node's constituent parts may be obtained by applying the accessor *dm:string-value*.

```
function dm:string-value(CommentNode $n) returns xs:string
```

The node accessors *dm:node-kind* and *dm:parent* also apply to comment nodes and may return results other than the empty sequence. The accessors *dm:name*, *dm:base-uri*, *dm:typed-value*, *dm:children*, *dm:attributes*, *dm:namespaces*, *dm:type* and *dm:unique-ID* applied to a comment node always return the empty sequence.

A comment node is constructed from a Comment Information Item by the *infoitem-to-comment-node* function:

```
/* Accessors for comment information items */
function infoset-comment-value(CommentItem $ii) returns xs:string

define function infoitem-to-comment-node(CommentItem $ii) returns CommentNode
{
  return  dm:comment-node(infoset-comment-value($ii))
}
```

4.7 Text

Text Node accessors	possible values
dm:node-kind	"text"
dm:name	empty sequence
dm:base-uri	empty-sequence
dm:string-value	xs:string
dm:typed-value	sequence containing one atomic value
dm:parent	sequence of zero or one element or document nodes
dm:children	empty sequence
dm:attributes	empty sequence
dm:namespaces	empty sequence
dm:type	empty sequence
dm:unique-ID	empty sequence

A text node corresponds to a sequence of one or more consecutive **character information items**. As much character data as possible is grouped into each text node: a text node never has an immediately following or preceding sibling that is a text node.

A text node does not have an *expanded-QName*.

The *dm:string-value* of a text node is the character data, which corresponds to the concatenated **[character code]** properties of

each of the **character information items**.

Note: The string-value is not W3C normalized as described in the Character Model for the World Wide Web version 1.0 draft. See **[Issue-0045: Text nodes are not W3C-normalized text]**.

The *dm:parent* of the text node corresponds to the **[parent]** property of any one of the consecutive **character information items** (consecutive characters always have the same parent).

A text node has the constructor `dm:text-node` and takes a string value. Like all other node constructors, the text constructor has the effect of creating a new node with a unique identity, distinct from all other nodes.

```
function dm:text-node(xs:string $content) returns TextNode
```

The *dm:string-value* of a text node is simply its content.

```
function dm:string-value(TextNode $n) returns xs:string
```

The *dm:typed-value* accessor returns the string content of the text node with unknown simple type.

```
define function dm:typed-value(TextNode $n) returns AtomicValue
{
  return dm:atomic-value(dm:string-value($n),xs:anySimpleType)
}
```

The node accessors *dm:node-kind* and *dm:parent* also apply to text nodes and may return results other than the empty sequence. The accessors *dm:name*, *dm:base-uri*, *dm:children*, *dm:attributes*, *dm:namespaces*, *dm:type* and *dm:unique-ID* applied to a text node always return the empty sequence.

The mapping from character information items to text nodes occurs in the infoitem-to-element-node function. The *infoset-character-code* accessor maps a character information item to the ISO 10646 character code (in the range 0 to #x10FFFF, though not every value in this range is a legal XML character code) of the character.

```
function infoset-character-code(CharacterItem $ii) returns Code
```

The function *infoitem-to-single-character-text-node* takes one character information item and maps it to a text node with a string value of length one.

```
define function infoitem-to-single-character-text-node(CharacterItem $ii)
       returns TextNode
{
  {-- convert character code to string of length 1 --}
  return dm:text-node(code2string(infoset-character-code($ii)))
}
```

Editorial Note: JM: need a definition or description of code2string.

The *collapse-text-nodes* function synthesizes a single text node from multiple text nodes. It calls *infoitem-to-text-nodes* to collapse recursively one or more consecutive text nodes in its argument sequence. If insignificant whitespace is ignored, any text node containing only whitespace is eliminated. All other nodes are returned unchanged.

```
define function collapse-text-nodes(Node* $nodes) returns Node*
{
  let $newnodes:= infoitem-to-text-nodes($nodes)
  return
    if (ignore-whitespace) then
      sequence-map(delete-whitespace-node, $newnodes)
    else $newnodes
```

```
}

define function infoitem-to-text-nodes(Node* $nodes) returns Node*
{
  if (xf:empty($nodes)) then return empty-sequence()
  else
    let $head:= op:item-at($nodes, 1),
        $tail:= xf:sublist($nodes, 2)
    return
      if (dm:node-kind($head) = "text") then
        /* Collapse two consecutive text nodes and apply
           infoitem-to-text-nodes recursively */
        if (xf:empty($tail)) then $head
        else if (dm:node-kind(op:item-at($tail,1)) = "text") then
          infoitem-to-text-nodes(
            op:concatenate(
              dm:text-node(xf:concat(dm:string-value($head),
                                     dm:string-value(op:item-at($tail,1)))),
              xf:sublist($tail, 2)
            )
          )
        else op:concatenate($head,
               op:concatenate(op:item-at($tail,1),
                              infoitem-to-text-nodes($tail)))
      else op:concatenate($head, infoitem-to-text-nodes($tail))
  }
```

Editorial Note: JM: need a definition or description of delete-whitespace-node.

5 Atomic Values

Editorial Note: MN: Atomic values were previously called simple or simple-typed values. The current terminology clarifies that the type of these values is an XML Schema atomic type.

[Definition: A **atomic value** encapsulates an *XML Schema atomic type* and a corresponding value of that type.]

An XMLSchema atomic type [XMLSchema Part 2] may be *primitive* or *derived*. The primitive atomic types are named: *xs:string*, *xs:boolean*, *xs:decimal*, *xs:float*, *xs:double*, *xs:duration*, *xs:dateTime*, *xs:time*, *xs:date*, *xs:gYearMonth*, *xs:gYear*, *xs:gMonthDay*, *xs:gDay*, *xs:gMonth*, *xs:hexBinary*, *xs:base64Binary*, *xs:anyURI*, *xs:QName*, *xs:NOTATION*. A derived atomic type is derived by restriction and has a primitive base type and a set of constraining facets.

It is noted that the value space of the atomic values is the union of the value spaces of the nineteen primitive XML Schema types. This value space clearly includes those atomic values whose type is primitive, but it also includes those whose type is derived, as derivation by restriction always limits the value space.

An XMLSchema simple type [XMLSchema Part 2] may be also *primitive* or *derived* with the method of derivation being *restriction*, *list* or *union*. The atomic types are exactly those types where the derivation method is restriction only and not list or union. Values corresponding to such types are represented by atomic values. XML Schema simple types can be derived by list. Values corresponding to such types are represented by a sequence of atomic values whose type is the base type. XML Schema simple types can be derived by union. Values corresponding to such types lose the union type information and store one of the individual types.

An atomic value can be constructed from its lexical representation. The constructor *dm:atomic-value* takes a string and a corresponding atomic type.

```
function dm:atomic-value(xs:string $value, xs:QName $type) returns AtomicValue
```

This constructor corresponds very closely to the atomic value constructors specified in [XQuery 1.0 and XPath 2.0 Functions and Operators]. Each of those constructors takes a string in the lexical space of the given primitive type and returns the corresponding atomic value. For example, the constructor `xf:decimal(xf:string)` constructs an atomic value of type xs:decimal from a string. Analogous constructors exist for the other XML Schema primitive types.

The accessor *dm:type* returns an atomic value's type. The accessor *dm:string-value* can be used to recover a lexical representation of the atomic value: If the atomic value's type is primitive, it returns the atomic value's canonical lexical representation for that primitive type as specified in XML Schema Part 2 : Datatypes [XMLSchema Part 2]. If the atomic value's type is derived, it returns the atomic value's canonical lexical representation for the base type (which is a primitive type). See **[Issue-0072: Lexical representation of Schema primitive types]**.

```
function dm:type(AtomicValue $v)         returns xs:QName
function dm:string-value(AtomicValue $v) returns xs:string
```

Note: Using the canonical lexical representation for atomic values as described above may not always be compatible with XPath 1.0.

Note that the data model does not currently represent key values and key reference values as described in XML Schema Part 1 : Structures [XMLSchema Part 1]. In a future draft of this document, keys and key references may be represented in the data model (see **[Issue-0032: Keys and key references not represented]**).

6 Sequences

A *sequence* is an ordered collection of zero or more items. An *item* may be a node or an atomic value, i.e. a sequence may contain nodes, atomic values, or any mixture of nodes and atomic values (see **[Issue-0035: Eliminate heterogeneous sequences]**). When a node is added to a sequence its identity remains the same. Consequently one node may be contained in more than one sequence and a sequence may contain duplicate items. Unlike conventional lists, sequences are "flat", i.e., sequences may not contain other sequences.

An important characteristic of the data model is that there is no distinction between an item (i.e., a node or an atomic value) and a singleton sequence containing that item, i.e., an item is equivalent to a singleton sequence containing that item and vice versa.

Note: Sequences replace node-sets from XPath 1.0. In XPath 1.0, node-sets do not contain duplicates. In generalizing node-sets to sequences in XPath 2.0, duplicate removal is provided by functions on node sequences.

Note: See **[Issue-0025: Types of Sequences]**.

A collection of documents is represented in the data model as a sequence of document nodes (see **[Issue-0023: Support for document repositories]**).

A sequence has no identity. Equality comparison of sequences is performed only by comparing the items of the sequences.

The *dm:string-value* of a sequence containing atomic values only is obtained by inserting a `#x20` (space) separator between the *dm:string-value*s of the individual members of the sequence and concatenating the result.

The *dm:string-value* of a sequence containing any item other than an atomic value is the concatenation of the *dm:string-value*s of the individual members of the sequence. In particular the string value of an empty sequence is an empty string.

The constructor *empty-sequence* constructs the empty sequence. The n-ary *op:concatenate* constructor creates a new sequence containing the values in its first argument followed by the concatenated values of its second through final arguments. In the function signatures below *Item* refers to a nodes or an atomic value. Since an item is equivalent to a singleton sequence containing the item, *op:concatenate* may be applied to items.

```
function empty-sequence()                  returns Item*
function op:concatenate(Item*, ..., Item*) returns Item*
```

A sequence has several accessors. They are defined in [XQuery 1.0 and XPath 2.0 Functions and Operators]. In this document

the following four accessors are used. The *empty* accessor returns a boolean indicating whether or not the specified sequence is empty. The *item-at* accessor returns the item in the sequence that is located at the specified index. The *sublist* accessor returns the contiguous sequence of items beginning at the specified position and continuing to the end of the sequence. The *count* function returns the number of items in the sequence.

```
function empty(Item*)              returns xs:boolean
function item-at(Item*,xs:decimal) returns Item
function sublist(Item*,xs:decimal) returns Item*
function count(Item*)              returns xs:unsignedInt
```

7 Error

The data model includes a distinguished error value, called *error*. Note that *error* cannot occur in the content of any node in the data model, nor may it occur in any sequence (see **[Issue-0047: Should errors be allowed in sequences?]**). The error object is defined so that functions or operators have a mechanism for identifying an error condition. How the error value is handled in a query processor is implementation-defined (see **[Issue-0048: Imprecise behavior of errors]**).

A XML Information Set Conformance

This specification conforms to the XML Information Set [XML Information Set]. The following information items must be exposed by the infoset producer to construct an instance of the data model:

- The **Document Information Item** with **[base URI]** and **[children]** properties.
- **Element Information Items** with **[children]**, **[attributes]**, **[in-scope namespaces]**, **[local name]**, **[namespace name]**, **[parent]** properties.
- **Attribute Information Items** with **[namespace name]**, **[local name]**, **[normalized value]**, **[owner element]** properties.
- **Character Information Items** with **[character code]** and **[parent]** properties.
- **Processing Instruction Information Items** with **[target]**, **[content]** and **[parent]** properties.
- **Comment Information Items** with **[content]** and **[parent]** properties.
- **Namespace Information Items** with **[prefix]** and **[namespace name]** properties.

Other information items and properties made available by the Infoset processor are ignored. In addition to the properties above, the following properties from the PSV Infoset are required:

- **[validity]**, **[type definition]**, **[type definition namespace]**, **[type definition name]**, **[type definition anonymous]**, **[member type definition]**, **[member type definition namespace]**, **[member type definition name]**, **[member type definition anonymous]** and **[schema normalized value]** properties on **Element Information Items**.
- **[validity]**, **[type definition]**, **[type definition namespace]**, **[type definition name]**, **[type definition anonymous]**, **[member type definition]**, **[member type definition namespace]**, **[member type definition name]**, **[member type definition anonymous]** and **[schema normalized value]** properties on **Attribute Information Items**.

B References

XML Information Set

World Wide Web Consortium, *XML Information Set (Infoset)*. See http://www.w3.org/TR/xml-infoset/.

XML Recommendation

World Wide Web Consortium, *Extensible Markup Language (XML) 1.0 (Second Edition)* See http://www.w3.org/TR/REC-xml.

XML Schema: Formal Description

World-Wide Web Consortium *XML Schema: Formal Description*, Working Draft, March 2001. See http://www.w3.org/TR/xmlschema-formal/.

XMLSchema Part 1

World Wide Web Consortium, *XML Schema Part 1: Structures*. See http://www.w3.org/TR/xmlschema-1.

XMLSchema Part 2

World Wide Web Consortium, *XML Schema Part 2: Datatypes*. See http://www.w3.org/TR/xmlschema-2.

XQuery 1.0 and XPath 2.0 Functions and Operators

World Wide Web Consortium, *XQuery 1.0 and XPath 2.0 Functions and Operators*. See http://www.w3.org/TR/xquery-operators/.

C References (Non-Normative)

W3C Style Activity

World Wide Web Consortium, *Style Activity*. See http://www.w3.org/Style/Activity.

W3C XML Activity

World Wide Web Consortium, *XML Activity*. See http://www.w3.org/XML/Activity.

XML Pointer Language (XPointer)

World Wide Web Consortium, *XML Pointer Language (XPointer)*. See http://www.w3.org/TR/xptr/.

XML Query Data Model

World-Wide Web Consortium *XML Query Data Model*, Working Draft, Feb 2001. See http://www.w3.org/TR/2001/WD-query-datamodel-20010215/.

XML Query Requirements

World Wide Web Consortium, *XML Query Requirements*. See http://www.w3.org/TR/2001/WD-xmlquery-req-20010215.

XML Query Working Group

World Wide Web Consortium, *XML Query Working Group*. Home page: http://www.w3.org/XML/Activity#query-wg.

XPath

World-Wide Web Consortium *XML Path Language (XPath)*: Version 1.0. November, 1999. See http://www.w3.org/TR/xpath.html.

XPath Requirements Version 2.0

World Wide Web Consortium, *XPath Requirements Version 2.0*. See http://www.w3.org/TR/xpath20req.

XQuery 1.0 Formal Semantics

World Wide Web Consortium, *XQuery 1.0 Formal Semantics*. See http://www.w3.org/TR/query-semantics/

XQuery 1.0: A Query Language for XML

World Wide Web Consortium, *XQuery 1.0: A Query Language for XML*. See http://www.w3.org/TR/xquery/.

XSL Working Group

World Wide Web Consortium, *XSL Working Group*. Home page: http://www.w3.org/Style/XSL/.

XSLT 1.0

World Wide Web Consortium, *XSL Transformations Language (XSLT)*: Version 1.0. See http://www.w3.org/TR/xslt.

XSLT 2.0

World Wide Web Consortium, *XSL Transformations Language (XSLT)*: Version 2.0. See http://www.w3.org/TR/xslt20/.

D Example (Non-Normative)

We use the following XML document to illustrate the information contained in an instance of the data model:

```
<?xml version="1.0"?>
<p:part xmlns:p="http://www.example.com/PartSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation = "http://www.example.com/PartSchema
                              http://www.example.com/PartSchema"
        name="NB-401-nutbolt">
  <p:mfg>Acme</p:mfg>
  <p:price>10.50</p:price>
</p:part>
```

The document is associated with the URI "http://www.example.com/partlist/part0001.xml", and is valid with respect to the following XML schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.example.com/PartSchema"
           xmlns="http://www.example.com/PartSchema">
  <xs:element name="part" type="part-type"/>
  <xs:complexType name="part-type">
    <xs:sequence>
      <xs:element name="mfg" type="xs:string"/>
      <xs:element name="price" type="xs:decimal"/>
      <xs:attribute name="name" type="part-name"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="part-name">
    <xs:restriction base="xsd:string">
      <xs:pattern value="[A-Z]{2}-\d{3}-[A-Z]*"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

For this example, we chose an XML document and an XML Schema that illustrates the relationship between document content and its associated schema type information. In general, an XML Schema is not required, that is, the data model can represent a schemaless, well-formed XML document with the rules described in **3.4 Types**.

The XML document is represented by the data-model constructors below. The value *D1* represents a document node; the values *E1*, *E2*, *etc.* represent element nodes; the values *A1*, ... represent attribute nodes; the values *N1*, ... represent namespace nodes; the values *T1*, ... represent text nodes. Accessors that return a constant value for that node type are omitted.

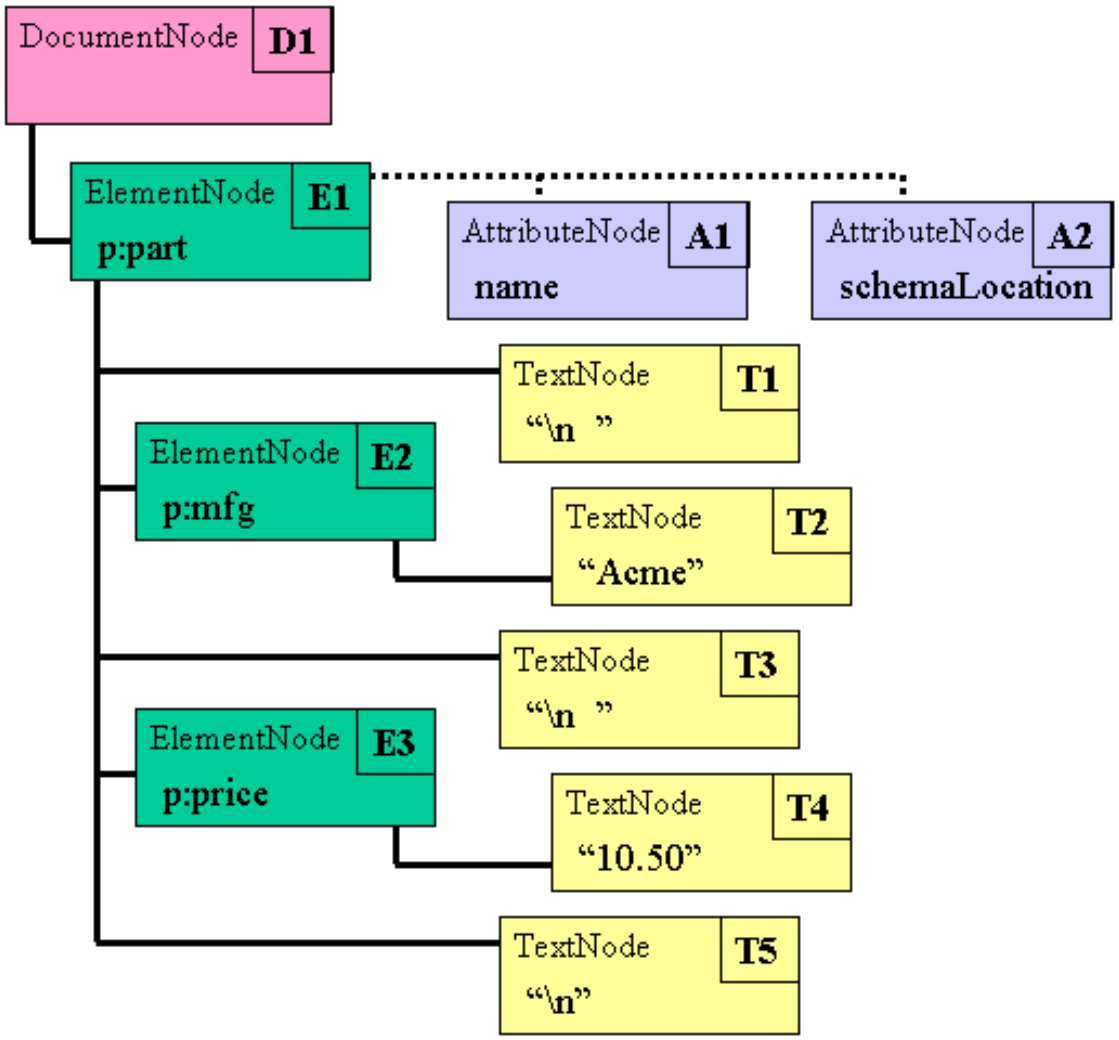
```
// Document node D1
dm:base-uri(D1)     = xf:anyURI("http://www.example.com/partlist/part0001.xml")
dm:string-value(D1) = xf:string("\n  Acme\n  10.50\n")
dm:typed-value(D1) = error
dm:children(D1)     = E1

// Element node E1
dm:name(E1)         = xf:QName-from-uri("http://www.example.com/PartSchema", "part")
dm:base-uri(E1)     = xf:anyURI("http://www.example.com/partlist/part0001.xml")
dm:string-value(E1) = xf:string("\n  Acme\n  10.50\n")
dm:typed-value(E1) = error
dm:parent(E1)       = D1
```

```
dm:children(E1)     = (T1, E2, T3, E3, T5)
dm:attributes(E1)   = A1
dm:namespaces(E1)   = (N1, N2)
dm:type(E1)         = xf:QName-from-uri("http://www.example.com/PartSchema",
"part-type")
dm:unique-ID(E1)    = ()

// Attribute node A1
dm:name(A1)         = xf:QName("name")
dm:string-value(A1) = xf:string("NB-401-nutbolt")
dm:typed-value(A1)  = dm:atomic-value("NB-401-nutbolt",
                      xf:QName-from-uri("http://www.example.com/PartSchema",
"part-name"))
dm:parent(A1)       = E1
dm:type(A1)         = xf:QName-from-uri("http://www.w3.org/2001/XMLSchema", "string")

// Attribute node A2
dm:name(A2)         = xf:QName-from-uri("http://www.w3.org/2001/XMLSchema-instance",
"schemaLocation")
dm:string-value(A2) = xf:string("http://www.example.com/PartSchema
http://www.example.com/PartSchema")
dm:typed-value(A2)  = ()
dm:parent(A2)       = E1
dm:type(A2)         = xf:QName-from-uri("http://www.w3.org/2001/XMLSchema",
"anySimpleType")

// Namespace node N1
dm:name(N1)         = xf:QName("xsi")
dm:string-value(N1) = xf:string("http://www.w3.org/2001/XMLSchema-instance")

// Namespace node N2
dm:name(N2)         = xf:QName("p")
dm:string-value(N2) = xf:string("http://www.example.com/PartSchema")

// Element node E2
dm:name(E2)         = xf:QName-from-uri("http://www.example.com/PartSchema", "mfg")
dm:base-uri(E2)     = xf:anyURI("http://www.example.com/partlist/part0001.xml")
dm:string-value(E2) = xf:string("Acme")
dm:typed-value(E2)  = dm:atomic-value("Acme",
                      xf:QName-from-uri("http://www.w3.org/2001/XMLSchema", "string"))
                  = xf:string("Acme")
dm:parent(E2)       = E1
dm:children(E2)     = T2
dm:attributes(E2)   = ()
dm:namespaces(E2)   = (N1, N2)
dm:type(E2)         = xf:QName-from-uri("http://www.w3.org/2001/XMLSchema", "string")
dm:unique-ID(E2)    = ()

// Element node E3
dm:name(E3)         = xf:QName-from-uri("http://www.example.com/PartSchema", "price")
dm:base-uri(E3)     = xf:anyURI("http://www.example.com/partlist/part0001.xml")
dm:string-value(E3) = xf:string("10.50")
dm:typed-value(E3)  = dm:atomic-value("10.50",
```

```
                    xf:QName-from-uri("http://www.w3.org/2001/XMLSchema",
"decimal"))
                 = xf:decimal("10.50")
dm:parent(E3)       = E1
dm:children(E3)     = T4
dm:attributes(E3)   = ()
dm:namespaces(E3)   = (N1, N2)
dm:type(E3)         = xf:QName-from-uri("http://www.w3.org/2001/XMLSchema",
"decimal")
dm:unique-ID(E3)    = ()

// Text node T1
dm:string-value(T1) = xf:string("\n  ")
dm:parent(T1)       = E1

// Text node T2
dm:string-value(T2) = xf:string("Acme")
dm:parent(T2)       = E2

// Text node T3
dm:string-value(T3) = xf:string("\n  ")
dm:parent(T3)       = E1

// Text node T4
dm:string-value(T4) = xf:string("10.50")
dm:parent(T4)       = E3

// Text node T5
dm:string-value(T5) = xf:string("\n")
dm:parent(T5)       = E1
```

A graphical depiction of the data model instance, containing the information described in the text above (with the exception of the namespace nodes), is also included.

Note: MN: The issue of handling whitespace-only nodes is currently under discussion. See **[Issue-0028: Whitespace handling]**. Depending on the decisions, the picture below may change. In particular the whitespace-only nodes T1, T3, T5 may be omitted from a data model representation.



E Issues (Non-Normative)

The issues in this section serve as a design history for this document. The ordering of issues is irrelevant. Each issue has a unique id of the form Issue-<dddd> (where d is a digit). This can be used for referring to the issue by <url-of-this-document>#Issue-<dddd>. Furthermore, each issue has a mnemonic header, a date, an optional description, and an optional resolution. For convenience, resolved issues are displayed in green. Some of the issues contain references to W3C internal archives. These are marked with "members only". Some of the descriptions of the resolved issues are obsolete w.r.t. to the current version of the document.

Issue-0001: PSV Infoset identity constraints

Date: Oct-2000
Raised by: Datamodel Editors

Description: What should be data-model representation, if any, of PSV Infoset identity-constraint tables?

Resolution: JM: Duplicate of **[Issue-0032: Keys and key references not represented]**.

Issue-0002: Representation of atomic values

Date: Oct-2000
Raised by: Datamodel Editors

Description: This function assumes that the character information items for an atomic value (e.g., string, integer, floating-point

number) are not interleaved with other information items (e.g., PIs or comments). The treatment of such interleaved values is not handled in this definition. This issue is addressed in threads beginning at: http://lists.w3.org/Archives/Member/w3c-archive/2000Jun/0090.html (members only) and http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0079.html (members only).

Resolution: MF: The data model does not preserve information items interleaved with the character info items of an atomic value.

Issue-0003: Example parent

Date: Oct-2000
Raised by: Datamodel Editors

Description: *Remark Michael:* An IDREF cannot point to an empty string.

Resolution: JM: Issue is obsolete, probably had something to do with Reference Nodes, which have been removed.

Issue-0004: Schema/DTD

Date: Oct-2000
Raised by: Datamodel Editors
Affects: **3.3 XML Schemas and the XML Information Set** **4.2 Elements**

Description: A document may refer to a DTD and have an associated schema. Currently, content model from the DTD is ignored, as are unique IDs from the schema. A coherent priority or merging strategy is needed.

Issue-0005: Lists of Simple Values

Date: Oct-2000
Raised by: Datamodel Editors

Description: The current data model draft takes only into account singleton value-nodes. It must represent lists of simple-type values as well. See http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000May/0060.html (members only).

Peter suggests having a special-purpose kind of a TextNode that represents lists of simple types. An advantage of this approach is that the constraint that lists of simple types be homogeneous/monomorphic can be enforced. However, lists/forests already can be modeled in current data model, without adding more complexity. For example, an attribute's value could be modeled as a list of TextNodes:

```
value    : (AttributeNode)  -> Sequence<TextNode>
```

A disadvantage of this approach is that the monomorphism constraint on lists derived from simple types is not enforced. However, given a type system for Query, such a constraint could be enforced. So Mary is in favor of not having a special-purpose kind of TextNode to represent lists, but instead model them by forests directly in the data model.

JM: note that the pseudo-code doesn't appear to handle schema list values.

Resolution: decision record (members only).

Issue-0006: Collections

Date: Oct-2000
Raised by: Datamodel Editors

Description: We need a more thorough definition of collections, perhaps in a separate section, which includes bags and defines collections formally.

In particular, the algebra (probably) will not support arbitrarily nested collections (i.e., lists of lists, sets of sets, etc.). We need to specify how collections are constructed. For example, in the data model, the basic collection is a forest, i.e., a list of Nodes. The forest constructor creates a singleton forest from one Node; or it creates a forest from two forests by concatenating the two forests:

```
Forest = Node | Sequence<Node>

forest : (Node) -> Forest
function forest(Node n) =  Sequence<n>

union : (Forest, Forest) -> Forest
function union(f1, f2) = list-append f1 f2

bagunion : (NodeBag, NodeBag) -> NodeBag
setunion : (NodeSet, NodeSet) -> NodeSet
```

Similar constructors would exist for bags with and without duplicates.

```
unordered : (Forest) -> NodeBag
unique    : (NodeBag) -> NodeSet
set = unique o unordered : (Forest) -> NodeSet
```

Resolution: Added section on Collections **6 Sequences**.

Issue-0007: Text Nodes

Date: Oct-2000
Raised by: Datamodel Editors

Description: An alternative representation is to have a single TextNode whose base type is string:

```
text-node  : (xs:string, S, Sequence<Node>) -> TextNode
```

This representation is more closely aligned with other node types in the data model, but it makes the simple type of leaf-node values opaque.

Peter Fankhauser compares and constrasts these options in :
http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Apr/0174.html (members only).

Resolution: decision record (members only).

Issue-0008: Node vs edge centric data model

Date: Oct-2000
Raised by: Michael Rys

Description: *Cite:*

Let me summarize my issues with a node-centric datamodel right at the beginning. The first two are mentioned in the doc later on:

As long as (1) the data represents a tree, (2) easy bi-directional is not required, (3) projection/extension operations with object-preserving semantics are not required, a node-centric datamodel is isomorphic to an edge-centric datamodel and is easier to represent and understand.

As soon as anyone of the above requirements change, an edge model has several advantages:

1. data represents a graph: naming the edges (relationships) becomes a must, since the names are now on the relationships and not on the objects. Uniform treatment of all edges (even the so far anonymous containment edges) makes defining operations easier since they are more orthogonal. With the possibility of distinguishing "type" from "name", even subelement names now semantically represent relationship names. For example, ShipAddr and BillAddr in

   <Order> <ShipAddr dt:dt="Address">...</ShipAddr> <BillAddr dt:dt="Address">...</BillAddr> </Order>

are denoting relationships (ownership to be exact) from the order element to the Address elements.

2. As soon as backwards pointers are introduced into a node-centric model, the representation becomes more complex and less elegant. Transforming data becomes more complex since the backwards pointer becomes part of the object state. Thus, if I define views where an element changes the parent, in the edge-centric case, this just adds a new relationship, the object state is unchanged, in the node-centric approach, I need to express now two parents in the object state.
3. Projection/extension operations. Assume that I pose a query that projects name and address but hides the age of a person element. In the edge-centric approach, this means that the query logically transforms the graph context on which the query operates by removing the age edge from the context without touching the object state (the objects keeps its basetype), in the node-centric approach, the object state needs to change since the context transformation will remove the attribute property age. While both operations transform the context, I find the former to be more elegant than the later.

Resolution: MF: To align with XPath 1.0 and the Algebra, the data model is node centric.

Issue-0009: Schema info

Date: Oct-2000
Raised by: Michael Rys

Description: *Cite:*Sometimes one wants to use different schemata over the same basic XML fragment. So I would rather start with that in principle, the data model is schemaless and can provide the data model of any XML fragment given a schema. Thus, the schema postprocessing becomes a datamodel transformation that we make explicit (and that could be optimized with other operations that transform the datamodel graph).

Resolution: MN: This was an alternate design to the named typing proposal. With the acceptance of the named typing this isssue is closed. When using a different schemata over the same XML fragment (by invoking validate, see [XQuery 1.0 Formal Semantics]) a new instance is constructed and there is no need for postprocessing.

Issue-0010: Node identity

Date: Oct-2000
Raised by: Datamodel Editors

Description: Should the data model require that an implementation guarantee that the identity of a node is always preserved?

Resolution: MF: The data model always preserves node identity; the only operator that does not preserve node identity is `copy`.

Issue-0011: Access to facets

Date: Oct-2000
Raised by: Datamodel Editors

Description: In XML Schema, facets such as "nullable" is associated with an *element declaration*, which is a element name, complex type pair. If the query language needs access to such facets, we may need to replace *ReferenceNode* by a reference to the element declaration.

Resolution: MN: The data model represents named types as expanded-QNames. This way it supports type-related operations, but the semantics of such operations and the information that must be available to support them (e.g. facets) is described in the [XQuery 1.0 Formal Semantics].

Issue-0012: Representation of reference values

Date: Oct-2000
Raised by: Michael Rys

Description: *Cite:* The current representation of reference values is too much IDREF(S) centric. I would prefer a more general representation for XLink and the schema (and potentially graph operation) introduced reference mechanisms.

Resolution: JM: Removed reference nodes.

Issue-0013: Equality operators on collections

Date: 17-Jan-2001

Raised by: Mary Fernandez

Description: Equality operators '=' on collections are not defined.

Resolution: MF: Added Functions (subsequently removed to Functions and Operators spec).

Issue-0014: Elements with unordered children
Date: 17-Jan-2001
Raised by: Mary Fernandez

Description: Should the element constructor element-node also permit bags of children?

Resolution: MF: decision to use sequences everywhere in data model.

Issue-0015: Semantics of value equality operator '='
Date: 02-Feb-2001
Raised by: Mary Fernandez

Description: The semantics of the value equality operator '=' are undefined.

Resolution: JM: Defined in the Functions and Operators document.

Issue-0016: PSV Infoset Mapping - undefined terms
Date: 21-Feb-2001
Raised by: Michael Rys

Description: `Code` is undefined.

Resolution: Defined in **4.7 Text**.

Issue-0017: Relationship between Ordered and Unordered collections
Date: 03-Mar-2001
Raised by: Mary Fernandez

Description: The relationship between ordered and unordered collections is not specified. Any ordered collection can be treated as an unordered collection.

Resolution: Unordered collections removed.

Issue-0018: Representation of lists of IDREFS and NMTOKENS
Date: 12-Mar-2001
Raised by: Michael Rys

Description: How are IDREF lists and NMTOKEN lists represented in data model.

Resolution: JM: Duplicate of **[Issue-0005: Lists of Simple Values]**.

Issue-0019: Element constructor that performs schema processing
Date: 15-Mar-2001
Raised by: James Clark
Affects: **4.2 Elements**

Description: An alternate is to separate element construction from schema validity assessment. The element constructor would construct an element corresponding to the an element information item in the Infoset before schema validity assessment. To produce elements with types, the `schema-process` function would schema process an element with respect to a schema type to yield a new element with the full PSV infoset. The `schema-process` function would ignore any type information on attributes and elements and would assess the untyped value with respect to the given type.

```
element-node  : (xs:QName,
                 Sequence<NamespaceNode>,
                 Sequence<AttributeNode>,
```

```
                  Sequence<ElementNode | ProcessingInstructionNode
                          | TextNode | CommentNode>)
               -> ElementNode
schema-process : (ElementNode | AttributeNode, SchemaComponent)
               -> ElementNode | AttributeNode
```

Issue-0020: Semantics of copy

Date: 27-Mar-2001
Raised by: Michael Kay

Description: Deep copy on a node is defined only informally. For example, does deep copy preserve base URI?

Resolution: JM: Obsolete - copy function removed (not used).

Issue-0021: Declared vs. In-scope namespaces

Date: 27-Mar-2001
Raised by: XPath 2.0 Task Force

Description: Currently, an element node preserved its declared namespace nodes, not its in-scope namespaces. Members of the XSLT WG point out this may make impossible to determine the meaning of data-model values that refer to the default namespace. This is a big, nasty problem.

Resolution: JM: Element constructor uses in-scope namespaces.

Issue-0022: Abstraction of Run-time type information

Date: 27-Mar-2001
Raised by: XPath 2.0 Task Force (Steve Zilles)

Description: The representation of run-time type information is very concrete -- it's the data model representation of a Schema type. The XPath task force would like a more abstract representation of runtime type that is not bound so tightly to XML Schema. This is an open design problem.

Resolution: MN: The data model currently represents named types in a more abstract manner, as expanded-QNames.

Issue-0023: Support for document repositories

Date: 27-Mar-2001
Raised by: XPath 2.0 Task Force
Affects: **6 Sequences**

Description: Many people would like to see support for document repositories in XPath 2.0 with a corresponding notion in the data model. A document repository is easy to model as a sequence or bag of document nodes. It may have some additional properties, like for an ordered repository, order among all the nodes in the repository.

Issue-0024: Support for Schema-invalid documents

Date: 27-Mar-2001
Raised by: Michael Sperberg-McQueen

Description: In its current state, the data model clearly does not cover schema-invalid documents: section 3.3 says "We assume that the element is an instance of the type represented by Def-Type, i.e., the document 'type checks' or is valid with respect to the given schema."

I believe we may wish to extend / modify the data model to specify that:

1. if the element is marked valid (i.e. if the [validity] property for the element information item has the value "valid"), then we assume that the element is an instance of the type represented by Def-Type
2. otherwise, if the element is marked invalid (i.e. the [validity] property has the value "invalid" or "notKnown"), and if the element has neither attributes nor child elements, then we assume [observe] that the element is an instance of the type anySimpleType

3. otherwise, we assume that the element is an instance of the type anyType

This would allow / require schema systems to be robust in the face of invalid documents. At first glance, that seems like a win.

Resolution: The data model answers the above issue in the following way: Section 3.3 specifically states that the data model supports incompletely validated documents. Section 3.4 (formerly 8.1) describes a function used in the pseudo-code segments that maps the elements with the different [validity] properties to the appropriate types. The mapping is inline with the solution suggested by the issue above. For further details see the discussion thread and decision record (members only).

Issue-0025: Types of Sequences

Date: 27-Apr-2001
Raised by: Mike Kay
Affects: **6 Sequences**

Description: Should sequence values carry their type as do simple typed values and element and attribute nodes?

Issue-0026: Schema Component Values vs. Nodes

Date: 27-Apr-2001
Raised by: Mary Fernandez

Description: If schema component values becomes nodes, then does that mean they can occur any where in a document tree? I.e., can they be children of other nodes? What does this mean when a data model is serialized as a document?

Resolution: MN: The data model represents named types as expanded-QNames. They can be associated with element nodes, attribute nodes and atomic values. Consequently the question raised by the issue no longer applies.

Issue-0027: Lexical representation of simple-typed values

Date: 01-May-2001
Raised by: Mary Fernandez

Description: Given a simple-typed value, it may be necessary to recover its lexical representation, for example, when creating a text node that contains the value. It is not always possible to compute a unique lexical representation of a simple typed value.

Resolution: Changed the definition of the string-value accessor to return the canonical lexical representation of the simple typed value (as defined by XML Schema Part 2: Datatypes). decision record (members only)

Issue-0028: Whitespace handling

Date: 04-May-2001
Raised by: Jonathan Marsh
Affects: **3.7 Ignoring Comments, Processing Instructions, and Whitespace** **D Example**

Description: Whitespace handling needs to be more explicit. In the presence of a schema we have full knowledge of which whitespace is significant and which isn't, and can either mark whitespace as insignificant (and thus exclude it from text() and string-range() for instance), or automatically suppress whitespace in the data model. The former is appropriate given the dual representation of text nodes and values, the latter is appropriate if we only expose values.

Issue-0029: Use of Reference Nodes

Date: 04-May-2001
Raised by: Jonathan Marsh

Description: Reference nodes may be part of the data model, but will never appear from a mapping from the infoset. In addition they cannot be serialized. Without these two features there doesn't seem to be much point in having them. Should we leverage an existing syntax (e.g. IDREFS) or design a new syntax to represent them?

Resolution: Removed Reference Nodes.

Issue-0030: Base URI is a property of element nodes

Date: 04-May-2001
Raised by: Jonathan Marsh
Affects: **4.2 Elements**

Description: With external entities, and now with XML Base, the base URI can be scoped to various parts of the document. A base URI property should be added to Element Nodes, and the constructor and infoset mapping updated. Otherwise relative URIs in content cannot be correctly resolved.

Processing instructions also require an infoset-derived base URI. The base URI of attributes, for instance, should probably not be the empty sequence, if that does not adequately imply that the base URI of the element should be used instead.

Issue-0031: Schema component does not reveal [content] property

Date: 17-May-2001
Raised by: Ashok Malhotra

Description: Schema component does not reveal [content] property of [XML Schema: Formal Description] schema component. MF: Problem with revealing [content] property is that we/Schema/Query have to agree on syntax for component content (Sec 2.2.1 in [XML Schema: Formal Description]).

Resolution: MN: The data model represents named types as expanded-QNames. This way it supports type-related operations, but the semantics of such operations and the information that must be available to support them (e.g. [content] property) is described in the [XQuery 1.0 Formal Semantics].

Issue-0032: Keys and key references not represented

Date: 17-May-2001
Raised by: Query
Affects: **5 Atomic Values** **Issue-0001: PSV Infoset identity constraints**

Description: Note that the data model does not currently represent key values and key reference values as described in XML Schema Part 1 : Structures [XMLSchema Part 1]. In a future draft of this document, keys and key references will be represented in the data model.

Issue-0033: Unclear relationship between values passed to the constructor, and those returned by the accessor

Date: 28-April-2001
Raised by: James Clark
Affects: **2 Notation and Pseudo-code Syntax**

Description: http://lists.w3.org/Archives/Member/w3c-xsl-query/2001Apr/0312.html (members only). Asks for inference rules, especially for the constuctor, describing when values returned by an accessor are the same as those set by the corresponding constructor. Especially unclear are when adjacent text nodes are collapsed, base URI and namespace declarations.

Issue-0034: Interaction of insignificant whitespace with comments

Date: 8-May-2001
Raised by: Michael Kay
Affects: **3.7 Ignoring Comments, Processing Instructions, and Whitespace**

Description: http://lists.w3.org/Archives/Member/w3c-xsl-query/2001May/0053.html (members only). Clarify whether whitespace is classified as insignificant before or after PI and comment removal.

Issue-0035: Eliminate heterogeneous sequences

Date: 8-May-2001
Raised by: XSL WG (Michael Kay)
Affects: **6 Sequences**

Description: http://lists.w3.org/Archives/Member/w3c-xsl-query/2001May/0054.html (members only), http://lists.w3.org/Archives/Member/w3c-xsl-query/2001May/0048.html (members only). Simplify operations such as distinct() by disallowing sequences mixing nodes and simple typed values. Suggests converting nodes in such a heterogeneous sequence to their typed values.

Issue-0036: Support for abstract types

Date: 19-July-2001
Raised by: XSL WG

Affects:

Description: xsd:anySimpleType (and other abstract types) are not supported in the data model. The string-value of an xsd:anySimpleTyped is apparently the empty sequence - indicating that you can't really do useful operations on anySimpleType'd values. However, we apply a default schema which assigns xsd:anySimpleType, resulting in a proliferation of these types. How can we operate on these documents? Should we at least return a non-empty string-value() for an anySimpleType? Should we define additional operations such as +, *, for compatibility with XPath 1.0?

Issue-0037: Axis functions

Date: 19-July-2001
Raised by: XSL WG
Affects:

Description: Define (somewhere other than the data model document?) axis functions for non-primitive axes like descendants-or-self.

Issue-0038: XPath 1.0 treatment of non-unique IDs

Date: 13-August-2001
Raised by: Datamodel Editors
Affects: **4.2 Elements**

Description: From XPath 1.0: "If an XML processor reports two elements in a document as having the same unique ID (which is possible only if the document is invalid) then the second element in document order must be treated as not having a unique ID." This has not been incorporated into this document.

Issue-0039: Parent of namespace nodes

Date: 13-August-2001
Raised by: Datamodel Editors
Affects: **4.4 Namespaces**

Description: In XPath 1.0 namespace nodes have a parent. Should we adopt the XPath 1.0 behavior, the current behavior (no parent), or some other parent (e.g. the document)?

Issue-0040: Setting and examining construction flags

Date: 15-August-2001
Raised by: Jim Melton
Affects: **3.7 Ignoring Comments, Processing Instructions, and Whitespace**

Description: [Jim] found [him]self wondering how those flags (parameters) get set/passed. More importantly, can a process ask whether "this instance" of the data model has those flags set or not? If so, how? If not, why not?

Issue-0041: Document node permissiveness unnecessary

Date: 15-August-2001
Raised by: Jonathan Marsh
Affects: **4.1 Documents**

Description: What are the use cases for being permissive about children of the document node? According to the note above, sequences of elements do not need a container node. Suggest removing this permissiveness.

Issue-0042: System Id and Public Id are not exposed

Date: 15-August-2001
Raised by: Jim Melton
Affects: **4.1 Documents**

Description: In our model, there is no way for a query to determine what DTD is relevant for the data model instance. That seems like a piece of information that might be wanted occasionally (though probably not often).

Issue-0043: Treatment of common accessors inconsistent

Date: 15-August-2001

Raised by: Jonathan Marsh

Description: In some cases, when an accessor is inappropriate for a node, we omit that accessor. In other cases, we define it to always return the empty sequence. We need to rationalize these two design patterns.

Resolution: Use empty sequence throughout. decision record (members only).

Issue-0044: Unable to construct an element with unique ID

Date: 15-August-2001
Raised by: Jonathan Marsh
Affects: **4.2 Elements**

Description: The unique ID property is defined on an element node, but is a function of an attribute information item. When an element node is constructed it is given a attribute node - not an info item. An attribute node is insufficient to remember the appropriate properties from the infoset in order for the element constructor to detect when an attribute is an ID declared in the DTD.

Issue-0045: Text nodes are not W3C-normalized text

Date: 17-August-2001
Raised by: Jim Melton
Affects: **4.7 Text**

Description: The Character Model for the World Wide Web version 1.0 working draft defines W3C-normalized text. The algorithm for constructing text nodes from character information items does not perform normalization to this form. Should it?

Issue-0046: Value of derived types?

Date: 17-August-2001
Raised by: Jim Melton

Description: In XML Schema, the type unsignedInt is a simple type, but it's a derived type (not a primitive type); the nearest corresponding primitive type is decimal. If I were to invoke the value accessor with something analogous to "value(cast as unsignedInt('10'))", does it return a decimal value, and is the SchemaComponent returned by type a component that describes decimal? I do not believe that this a non-issue because the only types supported in the data model are XPath 1.0's types (string, number, boolean, and node-set), since this document makes a strong point about using all of Schema's simple types. Thus, does value return the primitive value, or does it return something like the schema-normalized value of the specified simple type?

Resolution: Simple typed values encapsulate both the value and the type. The type of an unsignedInt is the SchemaComponent corresponding to the unsignedInt. The string representation on the other hand is the value's canonical lexical representation for the base type (which is a primitive type) as specified by XML Schema. Decision record (members only).

Issue-0047: Should errors be allowed in sequences?

Date: 17-August-2001
Raised by: Jim Melton
Affects: **7 Error**

Description: The possibility that generating sequences in which one or more members are Error may sometimes be a useful way to produce partial results that can satisfy some sorts of queries.

Issue-0048: Imprecise behavior of errors

Date: 17-August-2001
Raised by: Jim Melton
Affects: **7 Error**

Description: I am very concerned that "How the error value is handled in a query processor is implementation-defined." I think we have to do a bit better than this, although leaving flexibility for implementations to do more for their users is a Good Thing.

Issue-0049: Alternative design of schema components

Date: 17-August-2001
Raised by: James Clark

Description: MF cite James: An alternative would be to have an extends accessor that returns deref([base]) if [derivation] is extension and empty sequence otherwise, and a restricts accessor that returns deref([base]) if [derivation] is restriction and empty sequence otherwise.

Resolution: MN: This proposal is not in the scope of the data model: The data model represents named types as expanded-QNames. The semantics of the type operations and the information that must be available to support them is described in the [XQuery 1.0 Formal Semantics].

Issue-0050: Relative order of free-floating nodes

Date: 17-August-2001
Raised by: Jim Melton
Affects: **3.2 Document Order** **Issue-0058: Node constructors formalism of questionable value**

Description: Are newly-constructed nodes in any particular order, such as some kind of document order? Does the order of these nodes have any relationship to the document order of the "input" data model instance? In fact, is the process being described properly characterized as "create a new data model instance from information derived from an existing data model instance", or something similar? Can more than one "new" document instance be created by a single query? In the fourth paragraph [Section 4], we see the phrase "the document node" --- is this the "existing document"'s document node, the "new document"'s document node, both, neither? Can more than one "existing" document instance be the source of information for a query?

Issue-0051: Document order of shared namespace nodes

Date: 28-August-2001
Raised by: Michael Kay
Affects: **3.2 Document Order**

Description: Section 3.2 states that in the document ordering, the namespace nodes of an element follow the element but precede its attributes. This is inconsistent with the idea, suggested but not spelled out in 4.4, that a namespace node can be shared by several elements. In fact, the question of namespace node identity is not really tackled. My view is that namespace node identity should be determined by the combination of (document identity, namespace prefix, namespace URI), that the parent of a namespace node should be the document node, and that namespace nodes should be ordered after every other node in the document. (This is easier for implementations than placing them at the start of the document, because the number of namespace nodes is not known until parsing is complete).

Issue-0052: Element constructor copies nodes?

Date: 28-August-2001
Raised by: Michael Kay
Affects: **4.2 Elements**

Description: In section 4.2 Elements, the notion that the constructor makes a copy of the supplied child nodes seems strange. It's hard to square this with the definition of node identity. Also, I don't see why the provision is needed here, but not for the document node constructor. Wouldn't it be cleaner to define a precondition that all the child nodes supplied to the constructor must be parentless?

Issue-0053: Semantics of head and tail on empty sequences

Date: 28-August-2001
Raised by: Michael Kay

Description: Head would appear to be a partial function, it does not apply to empty sequences. If we follow the same conventions as elsewhere, that means head returns a Sequence(0,1)<item>, which perhaps begs the question as to how you extract the first member of this sequence...

Resolution: Replaced the accessors head and tail by the some of the sequence accessors defined in the Functions and Operators document.

Issue-0054: Complex types with simple content

Date: 6-September-2001
Raised by: Evan Lenz

Description: Currently, the typed-value of a complex type is the empty sequence. Should complex types with simple content (i.e. an element that contains only text but that also has an attribute) be treated differently than complex types with complex content?

Resolution: Now typed-value corresponds to the **[schema normalized value]** PSVI property if that exists, or the empty sequence otherwise. This way if an element has a complex type with simple content its typed-value may be something other than the empty sequence. For further details see the discussion thread and decision record (members only).

Issue-0055: Effect of xsi:nil

Date: 6-September-2001
Raised by: XSL WG

Description: The XSL WG wishes xsi:nil="true" to result in a typed-value of the empty sequence. This allows the differentiation of a null string value and an empty string.

Resolution: This is a duplicate of the resolved XPath Issue 0021: Handling of xsi:nil on Input. See the review of the XPath issues (members only). Accordingly closed this issue, updated the document to reflect the decision and added the related data model Issue-0071.

Issue-0056: Lightweight Schema Components

Date: 20-September-2001
Raised by: XPath TF

Description: Summary: The only necessary operation on Schema Components is instance-of(schema-component1, schema-component2). There is no need to expose the internal structure of Schema Components to enable this functionality. Instead we could provide an abstract Type object whose internal structure can be treated as a black box. See http://lists.w3.org/Archives/Member/w3c-xsl-query/2001Jul/0042.html (members only).

Resolution: MN: The data model represents named types as expanded-QNames, which is inline with the suggestion of the issue.

Issue-0057: Support for XSLT whitespace stripping

Date: 25-September-2001
Raised by: Michael Kay
Affects: **3.7 Ignoring Comments, Processing Instructions, and Whitespace**

Description: XSLT allows a stylesheet to designate elements whose whitespace is to be stripped. We need to support this in the data model, or possibly elsewhere.

Similarly, a data model instance for a stylesheet has provisions for stripping comments and processing instructions.

Issue-0058: Node constructors formalism of questionable value

Date: 25-September-2001
Raised by: Michael Kay
Affects:

Description: Node constructors. I'm a bit concerned that this is lacking in rigor. Some of this is exposed in **[Issue-0050: Relative order of free-floating nodes]**. The idea that the constructor for a parent node (Element or Document) takes a copy of the supplied children node doesn't seem to be fully worked out. What does "taking a copy" mean, where is it defined? It has to be a deep copy to make sense; it has to preserve its name, its type, and its children, but not its base URI or its node identity, and it acquires a new position in document order. What happens about the namespace nodes when an element or attribute is copied? Altogether, I'm worried that this idea of node constructors looks formal, but is actually just as informal as the 1.0 specification. It's actually a very procedural description, and I can't really see why it's needed: if it's intended as a target vocabulary for the formal semantics of the language, then it's a pretty shaky foundation. I'd be much happier with a model that only defines the valid states in the system; if we are going to define the permitted state transitions, we need to be much more rigorous about it.

Issue-0059: Pseudo-formalism provides no value

Date: 25-September-2001
Raised by: Michael Kay

Affects:

Description: "The sequence-map function applies its first function argument to each member of its second sequence argument and returns a new sequence containing the result of applying the function to each member of the sequence." I really wonder whether it is a good idea to define the data model using a pseudo-formal language that we make up and half-explain as we go along? If we can't use an existing formal notation like Z or VDM that has a good specification we can reference, we should do the whole thing in English.

Issue-0060: Sharing namespace nodes

Date: 25-September-2001
Raised by: Michael Kay
Affects: **4.4 Namespaces**

Description: We need to say something about the identity of namespace nodes, and about the fact that two namespace nodes for the same prefix and uri may need to be combined when an element is added to a document. Also "the element constructor logically creates a copy of all of its namespace..." - namespace nodes do not need to be copied(?)

Issue-0061: No access to prefix on free-floating attributes

Date: 25-September-2001
Raised by: Michael Kay
Affects: **4.4 Namespaces**

Description: An observation: if an attribute node has no parent element (a floating attribute), then there is also no access to any namespace nodes. This makes it impossible to support the XPath 1.0 name() function, which returns a lexical QName for the node by finding a prefix that maps to the node's namespace URI.

[JM: Sounds like we need a QName object that is a triple, local-name, namesapce-uri, and prefix, to support XSLT.]

Issue-0062: Namespace fixups required

Date: 16-October-2001
Raised by: Michael Kay
Affects: **4.2 Elements**

Description: The current XSLT draft includes a substantial piece of text on namespace fixup. This was introduced in the XSLT 1.1 working draft, and is basically designed to ensure that when an element or attribute is added to a result tree, namespace nodes are also added to declare any namespace URI used by the element or attribute. (At XSLT 1.0, this was handled at serialization time, but this had to change when temporary trees became accessible to the stylesheet). The description of namespace fixup logically belongs with the description of the node construction process in the data model document. http://lists.w3.org/Archives/Member/w3c-xsl-query/2001Oct/0036.html (members only).

Issue-0063: Is prefix preserved?

Date: 16-October-2001
Raised by: Michael Kay
Affects: **4 Nodes**

Description: Although XSLT and the XPath 2.0 data model agree that element and attribute nodes do not hold a namespace prefix, XSLT has always hinted that prefixes might be preserved through a transformation where possible. http://lists.w3.org/Archives/Member/w3c-xsl-query/2001Oct/0036.html (members only).

[JM: the ability to recover the lexical value of an xs:QName simple type seems useful, perhaps even necessary. It is at least needed to support the name() function. A better description of the xs:QName accessors is required, perhaps unifying xs:QName and expanded-QName.]

Issue-0064: Including type definition in element constructors

Date: 16-October-2001
Raised by: Jeni Tennison

Description: The constructor for the element node probably should include the type definition in the constructor as well, for cases where the [type definition] of the element information item is not the same as the [type definition] of the [element

declaration], which can occur if xsi:type is used in the document. Should the same be done for attribute node constructors for consistency, (even though attributes cannot have xsi:type attributes)? http://lists.w3.org/Archives/Public/www-xml-query-comments/2001Sep/0012.html.

Resolution: MN: The definition of the element an attribute constructors have been changed to take an xs:QName that represents the type definition and not the element or attribute declaration. The type definition can be either the [type definition] of the element information item or the more specific [type definition] of the [element declaration] when xsi:type is used.

Issue-0065: Proposal for alternative constructor

Date: 19-November-2001
Raised by: Mary Fernandez

Description: The element and attribute constructors are not convenient when constructing elements or attributes with simple-type content. An alternative constructor that takes a typed value directly (instead of only text nodes or normalized string values) would simplify dependent specifications. http://lists.w3.org/Archives/Member/w3c-xsl-query/2001Nov/0068.html (members only).

Resolution: Add the two new constructors proposed by Mary with the change that the element/attribute declarations are not optional and they are named differently (so we won't overload the constructors). decision record (members only)

Issue-0066: SchemaComponent support for substitutions groups

Date: 29-November-2001
Raised by: Ashok Malhotra

Description: SchemaComponents currently don't expose any substitution group information. http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2001Nov/0358.html (members only).

Resolution: MN: The data model represents named types as expanded-QNames. This way it supports type-related operations, but the semantics of such operations and the information that must be available to support them (e.g. substitution groups) is described in the [XQuery 1.0 Formal Semantics].

Issue-0067: Align function call syntax

Date: 5-December-2001
Raised by: Marton Nagy

Description: The syntax of the function calls need to be aligned with the other working documents, in particular the Functions and Operators and the XQuery documents. Possible candidates are f:(T1,T2)->T or f(T1 $v1, T2 $v2)=>T.

Resolution: MN: The pseudo-calls and examples now use the XQuery syntax.

Issue-0068: Retaining the type of a sequence of heterogeneous simple typed values.

Date: 4-December-2001
Raised by: XPath Task Force
Affects: **4.2 Elements**

Description: When a sequence of heterogeneous simple typed values is passed as the content of an element constructor, is the type information associated with the simple typed values preserved? e.g. when using a sequence containing a decimal, a string and a boolean are these types preserved (or possibly changed to anySimpleType* ?). Note that XML Schema cannot describe this former. Related to this, should we ever allow an element constructor to create an instance that can not be serialized as XML text without loss of type information?

Issue-0069: Canonical form for derived types.

Date: 4-December-2001
Raised by: XPath Task Force
Affects:

Description: Should derived types have a canonical form? Should we ask XML Schema to fix this?

Issue-0070: Should the name accessor return "" or ()?

Date: 5-December-2001
Raised by: XPath Task Force
Affects: **4 Nodes**

Description: Should the name accessor return "" or () for nodes that have no name?

Issue-0071: Magic Attributes

Date: 7-December-2001
Raised by: Michael Rys
Affects: **4.2 Elements**

Description: Should the following attributes be represented in the data model? xmlns attributes, xml:lang, xml:space, xsi:nil (etc). If so, how?

Issue-0072: Lexical representation of Schema primitive types

Date: 13-January-2002
Raised by: Jeni Tennison
Affects: **5 Atomic Values**

Description: Unfortunately the XML Schema Datatypes Rec doesn't detail the canonical lexical representation of all of the primitive types. In particular, no canonical lexical representation is specified for:

1. xs:string, xs:base64Binary, xs:anyURI (but that's OK, I think we can guess)
2. xs:duration - presumably the lexical representation contains all components of the duration (years, months, days, hours, minutes and seconds, even those that occur 0 times? Or are these omitted? In the latter case, what's the canonical lexical representation of PT0S? Since the number of seconds can be a decimal, is this decimal represented with a decimal point (i.e. using the canonical lexical representation for xs:decimal)?
3. xs:date - what happens to the timezone component? Presumably, unlike xs:dateTime and xs:time, this isn't normalized to Z? (And similarly for xs:gYearMonth, xs:gYear, xs:gMonthDay, xs:Month, and xs:Day)
4. xs:QName and xs:NOTATION - these are the trickiest (their value spaces are the same). The XML Schema Rec states that the lexical representation of a QName depends on the in-scope namespaces. Does this mean the ones in the query/stylesheet or the ones from the source document? What if there's more than one namespace declaration for the namespace URI? What if there aren't any?

http://lists.w3.org/Archives/Public/www-xml-query-comments/2002Jan/0268.html

Issue-0073: Whitespace normalization of the string-value of elements with simple content

Date: 13-January-2002
Raised by: Jeni Tennison
Affects: **4.2 Elements**

Description: Currently the data model says that the string value of an attribute node is normalized (according to the whitespace facet of its type); on the other hand, the string value of an element node is not normalized. The string value of elements with simple content should also be normalized in the same way as the string value of attributes.
http://lists.w3.org/Archives/Public/www-xml-query-comments/2002Jan/0269.html

Issue-0074: Do we need Document fragments

Date: 12-February-2002
Raised by: Michael Rys
Affects: **4.1 Documents**

Description: Currently the Document node in the data model is permissive in the number of element nodes it can directly contain whereas the infoset only allows a single element node. Since preserving the single element node constraint is important to enforce queries to generate only well-formed documents when generating documents, the question is whether the data model needs to introduce a docfragment node that is permissive and keep the document node to be non-permissive. See http://lists.w3.org/Archives/Member/w3c-xsl-query/2002Feb/0111.html (members only).

F Open Issues (Non-Normative)

- Issue-0004: Schema/DTD
- Issue-0019: Element constructor that performs schema processing
- Issue-0023: Support for document repositories
- Issue-0024: Support for Schema-invalid documents
- Issue-0025: Types of Sequences
- Issue-0028: Whitespace handling
- Issue-0030: Base URI is a property of element nodes
- Issue-0032: Keys and key references not represented
- Issue-0033: Unclear relationship between values passed to the constructor, and those returned by the accessor
- Issue-0034: Interaction of insignificant whitespace with comments
- Issue-0035: Eliminate heterogeneous sequences
- Issue-0036: Support for abstract types
- Issue-0037: Axis functions
- Issue-0038: XPath 1.0 treatment of non-unique IDs
- Issue-0039: Parent of namespace nodes
- Issue-0040: Setting and examining construction flags
- Issue-0041: Document node permissiveness unnecessary
- Issue-0042: System Id and Public Id are not exposed
- Issue-0044: Unable to construct an element with unique ID
- Issue-0045: Text nodes are not W3C-normalized text
- Issue-0047: Should errors be allowed in sequences?
- Issue-0048: Imprecise behavior of errors
- Issue-0050: Relative order of free-floating nodes
- Issue-0051: Document order of shared namespace nodes
- Issue-0052: Element constructor copies nodes?
- Issue-0057: Support for XSLT whitespace stripping
- Issue-0058: Node constructors formalism of questionable value
- Issue-0059: Pseudo-formalism provides no value
- Issue-0060: Sharing namespace nodes
- Issue-0061: No access to prefix on free-floating attributes
- Issue-0062: Namespace fixups required
- Issue-0063: Is prefix preserved?
- Issue-0068: Retaining the type of a sequence of heterogeneous simple typed values.
- Issue-0069: Canonical form for derived types.
- Issue-0070: Should the name accessor return "" or ()?
- Issue-0071: Magic Attributes
- Issue-0072: Lexical representation of Schema primitive types
- Issue-0073: Whitespace normalization of the string-value of elements with simple content
- Issue-0074: Do we need Document fragments

G Resolved Issues (Non-Normative)

- Issue-0001: PSV Infoset identity constraints
- Issue-0002: Representation of atomic values
- Issue-0003: Example parent
- Issue-0005: Lists of Simple Values
- Issue-0006: Collections
- Issue-0007: Text Nodes
- Issue-0008: Node vs edge centric data model
- Issue-0009: Schema info
- Issue-0010: Node identity
- Issue-0011: Access to facets
- Issue-0012: Representation of reference values
- Issue-0013: Equality operators on collections
- Issue-0014: Elements with unordered children
- Issue-0015: Semantics of value equality operator '='
- Issue-0016: PSV Infoset Mapping - undefined terms
- Issue-0017: Relationship between Ordered and Unordered collections
- Issue-0018: Representation of lists of IDREFS and NMTOKENS
- Issue-0020: Semantics of copy
- Issue-0021: Declared vs. In-scope namespaces
- Issue-0022: Abstraction of Run-time type information
- Issue-0026: Schema Component Values vs. Nodes
- Issue-0027: Lexical representation of simple-typed values
- Issue-0029: Use of Reference Nodes
- Issue-0031: Schema component does not reveal [content] property
- Issue-0043: Treatment of common accessors inconsistent
- Issue-0046: Value of derived types?
- Issue-0049: Alternative design of schema components
- Issue-0053: Semantics of head and tail on empty sequences
- Issue-0054: Complex types with simple content
- Issue-0055: Effect of xsi:nil
- Issue-0056: Lightweight Schema Components
- Issue-0064: Including type definition in element constructors
- Issue-0065: Proposal for alternative constructor
- Issue-0066: SchemaComponent support for substitutions groups
- Issue-0067: Align function call syntax