XQuery 1.0 and XPath 2.0 Data Model

W3C Working Draft 7 June 2001

This version:
http://www.w3.org/TR/2001/WD-query-datamodel-20010607/
(available in: HTML, XML)
Latest version:
http://www.w3.org/TR/query-datamodel/
Previous version:
http://www.w3.org/TR/2001/WD-query-datamodel-20010215/
Editors:
Mary Fernández, AT&T Labs <mff@research.att.com >
Jonathan Marsh, Microsoft <jmarsh@microsoft.com>

Abstract

This document defines the W3C XQuery 1.0 and XPath 2.0 Data Model, which is the data model of at least [XSL Transformations], 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].

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.

This document has been produced as part of the [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 www-xml-query-comments@w3.org. (archived at http://lists.w3.org/Archives/Public/www-xml-query-comments/).

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

Table of Contents

1 Introduction

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

The XQuery 1.0 and XPath 2.0 Data Model (henceworth "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 Working Group is defining 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 Simple and Complex Values. ([XML Query Requirements])
- Representation of References. ([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.

Values in the data model fall into five categories: *nodes*, *simple values*, *sequences*, *error*, and *schema components*. A node is defined in **4 Nodes** and is one of eight node kinds. Simple values are the union of all the value spaces of XML Schema simple types and are defined in **5 Simple Values**. A sequence is an ordered collection of nodes, simple values, or any mixture of nodes and simple values. A sequence cannot be a member of a sequence. Sequences are defined in **6 Sequences**. The *error* value is defined in **7 Error**. A *schema component* represents the type of element nodes, attribute nodes, and simple values, and are defined in **8 Schema Components**.

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:

```
f : (x) -> y
```

In the psuedo-code syntax, the term *Node* denotes the category of node values, *SimpleValue* denotes the category of simple values, and *Sequence<V>* denotes the category of sequence values whose members are in category *V*. A *UnitValue* refers to a node or a simple value. In a sequence, *V* may be any *Node* or *SimpleValue*, or the union (choice) of several unit values. For example, the following denotes a sequence containing comment or processing instruction nodes:

```
Sequence<CommentNode | ProcessingInstructionNode>
```

There are some functions in the data model that are *partial functions*, for example, a node may have one parent node or no parent. We use *bounded* sequences, *Sequence(m,n)<V>*, to denote a sequence of at least *m* and at most *nV* values. The unbounded sequence *Sequence<V>* is equivalent to *Sequence(0,*)<V>*, where *** denotes unbounded. For example, the *parent* accessor returns a singleton sequence, if its node argument has a parent, or the empty sequence, if its argument has no parent. The signature of *parent* specifies that it returns an empty sequence or a sequence containing one element or document node:

```
parent : Node -> Sequence(0,1)<ElementNode | DocumentNode>
```

A *SchemaComponent* denotes the category of schema-component values.

The pseudo-code syntax defines functions to construct values, called *constructors*; and functions to access parts of values, called *accessors*.

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

The term *signature* of a function specifies the value category of its zero or more inputs and the value category 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.

f : (V_1, ..., V_m) -> 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*.

In the pseudo-code syntax describing the mapping from the Infoset to the data model, we name accessors of the Infoset using the convention *infoset-<item-name>-<property>*. For example, *infoset-elem-attributes* is the accessor that returns an element item's attributes property:

```
infoset-elem-attributes : ElementItem -> Sequence<AttributeItem>
```

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. Node identity simplifies the representation of XML reference values, e.g., IDREF, XPointer, and URI values.

Two nodes have the same identity if only if they were created by the same application of a node constructor (see **4 Nodes**).

3.2 Document Order

A *document order* is defined on all the nodes in a document. It corresponds to the order in which the first character of the XML representation of each node occurs in the XML representation of the document after expansion of general entities. Thus, the document node is the first node. Element nodes occur before their children. Thus, document order means that element nodes occur in the order of their start-tag in the XML (after expansion of entities). The namespace nodes of an element occur before its attribute nodes, and the element's attribute nodes occur before its children. The relative order of namespace nodes and the relative order of attribute nodes are 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, if *n1* and *n2* are in one document, and *m* is in a different document, then either '*n1* before *m*' is true or '*m* before *n2*' is true, but both may not be true.

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. We use the naming convention convention *psvi-<item-name>-<property>* to identify the accessor functions that return the PSV Infoset additions.

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 non-well-formed XML documents, nor documents that otherwise don’t have an XML Information Set; for example, 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. **8 Schema Components** specifies how such documents are represented in the data model.

> **Editorial Note:** JM: 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.

3.4 Schema Components and Values

The [XML Schema: Formal Description] (henceforth "XSFD") is a formal, declarative system for describing and naming XML Schema information, specifying XML instance type information, and validating instances against schemas. XSFD includes a component model that defines four *schema components* (*simple type* , *complex type*, *element*, and *attribute*), and it defines the mapping from the XML Schema component model to the XSFD model. In addition, it specifies "normalized, universal" names for all components of an XML Schema, so that they can be uniquely identified by URIs.

The data model provides a representation for schema components, which are used to represent the types of values. All simple values, element nodes, and attribute nodes have an associated schema component. We use the term *SchemaComponent* to collectively refer to the data model’s schema component values *simple-type-definition*, *complex-type-definition*, *element-declaration*, and *attribute-declaration*. The accessors for schema components are defined in **8 Schema Components**. The schema component of element and attribute nodes is derived from the PSV Infoset additions of the nodes’ corresponding element and attribute information items.

A schema *simple type* consists of a lexical space, a value space, and a set of facets [XMLSchema Part 2]. A simple type is either *primitive* (e.g., *xs:string, xs:boolean, xs:float, xs:double, xs:ID, xs:IDREF*) or *derived* (e.g., *xs:language, xs:NMTOKEN, xs:long*, etc., or user defined). We say a simple value is an *instance* of a schema simple type if the simple value is in the value space of the simple type. Because the value spaces of schema simple types may overlap, a simple value may be an instance of more than one schema simple type, e.g., an instance of *xs:integer* is also an instance of a *xs:long*.

> **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.

A schema *complex type* defines the permissible structure and content of an element [XMLSchema Part 1].

A schema *attribute declaration* specifies an attribute's name and the simple type of its value.

A schema *element declaration* specifies an element's name and the simple or complex type of its content.

3.5 Text Nodes and Simple-Typed Values

The data model supports two representations of the character data in an XML document: text nodes and simple-typed values. An element node, for example, has child nodes that may include text nodes, comment nodes, processing instruction nodes, and other element nodes. A text node contains a string of consecutive character data items and is never followed or preceded by another text node. In addition, the text content of an element may be interpreted as a simple-typed value, such as an integer, a date, or a sequence of prices. To illustrate, consider an element node whose complex 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 simple-typed value is a sequence containing the double-precision numbers 12.0 and 13.0.

We note that the data model logically supports both text nodes and simple-typed values, but it does not specify they should be implemented. An implementation might choose to only store simple-typed values and reconstruct text nodes on demand, vice versa. We note, however, that a simple-typed value does not always have a unique lexical representation (**[Issue-0027: Lexical representation of simple-typed values]**).

3.6 Ignoring Comments, Processing Instructions, and Whitespace

Although the data model can preserve all comments, processing instructions, and whitespace characters in the Infoset, preservation of these values may be unnecessary and onerous for some applications.

The data model is parameterized by three flags, *ignore-comments*, *ignore-processing-instructions*, and *ignore-whitespace*, which affect the construction of the data model from the Infoset. 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 white space is not preserved.

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

Insignificant whitespace is defined as a text node that:

1. contains no characters other than white space 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*.

Editorial Note: JM: The data model itself might benefit from whitespace information available in the schema. See **[Issue-0028: Whitespace handling]**.

4 Nodes

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

Note: The XPath 1.0 data model does not have reference nodes. Document nodes and XPath 1.0 root nodes serve many of the same purposes, but are not identical. In XPath 1.0, root nodes served as containers of document fragments. XPath 2.0 supports sequences as first-class objects in the data model.

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.

Document nodes and element nodes have a sequence of *child* nodes. 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.

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. A tree contains a root plus all nodes that are reachable directly or indirectly from the root via the *children*, *attributes*, and *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*.

Note: In XPath 1.0, Namespace nodes have parents.

There is also a way of determining the *string-value* of each kind of node. For some kinds of node, the *string-value* is part of the node; for other kinds of node, the *string-value* is computed from the *string-value* of its descendant nodes.

Element and attribute nodes contain a typed value, i.e., values that have associated schema simple type. An attribute contains a sequence of simple-typed values. An element contains either a complex-typed value, i.e., a sequence of nodes, a simple-typed value, or a sequence of simple-typed values. **4.2 Elements** and **4.3 Attributes** specify how the simple-typed value (*typed-value*) of an element or attribute is accessed.

The following accessors are defined on all eight kinds of Nodes. The *node-kind* accessor returns a string value representing the node's kind: either *"document"*, *"element"*, *"attribute"*, *"text"*, *"namespace"*, *"processing-instruction"*, *"comment"*, or *"reference"*. The *name* accessor returns the empty sequence, if the node has no name, otherwise, it returns a sequence containing one *expanded QName*. An expanded QName is in the value space of *xs:QName*, and contains a namespace URI and a local name. The *parent* accessor returns an empty sequence, if the node has no parent, otherwise, it returns a sequence containing one node. The *string-value* accessor returns the node's string representation.

```
    node-kind    : Node -> xs:string
    name         : Node -> Sequence(0,1)<xs:QName>
    parent       : Node -> Sequence(0,1)<ElementNode | DocumentNode>
    string-value : Node -> xs:string
```

The basic concept in the Infoset is an *InfoItem*. 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. Constructors are provided by an Infoset processor.

The *infoitem-kind* accessor returns a string value representing the information item's kind:

```
    infoitem-kind  : InfoItem -> xs:string
```

4.1 Documents

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

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

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

In a well-formed document, the children of the document node consist exclusively of element, processing-instruction, or 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.

The *base URI* of the document corresponds to the **[base URI]** property.

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

A document node has the constructor *document-node*, which takes a base URI value and a non-empty sequence of its children nodes:

```
document-node : (xs:anyURI,
                 Sequence(1,*)<ElementNode | TextNode | ProcessingInst
                              | CommentNode>)
            -> DocumentNode
```

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

```
base-uri : DocumentNode -> xs:anyURI
children : DocumentNode
         -> Sequence(1,*)<ElementNode | TextNode
                       | ProcessingInstructionNode | CommentNode>
```

The node accessors *node-kind*, *parent*, and *string-value* also apply to document nodes. A document node does not have an expanded-QName; the following signature specifies that *name* applied to a document node returns the empty sequence:

```
name(DocumentNode) : Sequence(0,0)<xs:QName>
```

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

```
/* Accessors for document information items: */
infoset-doc-children : DocumentItem
                     -> Sequence<ElementItem | ProcessingInstructionIt
                                | CommentItem | DocTypeItem>
infoset-doc-base-uri : DocumentItem -> xs:anyURI
```

```
dm-document-node : DocumentItem -> DocumentNode
function dm-document-node(d) {
  kids = dm-collapse-text-nodes(sequence-map(dm-node,
                                 infoset-doc-children(d)))
  return document-node(infoset-doc-base-uri(d), kids)
}
```

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. Below, *dm-node* is applied to each child of the Document Information Item value *d* and a new sequence of children nodes is constructed, each of which is a *Node*. The constructor *document-node* constructs the document node in the data model.

```
  sequence-map : ((UnitValue1 -> UnitValue2), Sequence<UnitValue1)
               -> Sequence<UnitValue2>
```

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

```
  dm-node : InfoItem -> Sequence<(0,1)Node>
  function dm-node(i) {
    return
      if (infoitem-kind(i) = "element") then
        dm-element-node(i)
      else if (infoitem-kind(i) = "character") then
        dm-char-to-text(i)
      else if (infoitem-kind(i) = "processing-instruction") then {
        if (not(ignore-processing-instructions)) then
```

```
                dm-pi-node(i)
              else empty-sequence()
            }
            else if (infoitem-kind(i) = "comment") {
              if (not(ignore-comments)) then
                dm-comment-node(i)
              else empty-sequence()
            }
            else if (infoitem-kind(i) = "doctype") then
              empty-sequence()
            else if (infoitem-kind(i) = "notation-item") then
              empty-sequence()
            else if (infoitem-kind(i) = "unparsed-entity-item") then
              empty-sequence()
        }
```

4.2 Elements

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 name of the *expanded-QName* of the element node corresponds to the **[namespace name]** property, if it has a value.

The *children* nodes 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 *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 *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.

The *declaration* of an element is a schema component and corresponds to the **[element declaration]** PSVI property. The *type* of an element is a schema component and corresponds to the **[type definition]** PSVI property. The representation of schema component information is defined in **8 Schema Components**.

An element node has an associated simple *typed-value*, e.g., an integer, date, or user-defined simple value. For a document with a schema, the element's typed-value corresponds to the **[schema normalized value]** PSVI property. If the element has a complex type, the typed-value is the empty sequence. For an element in a well-formed document with no associated schema, the element's typed-value is the empty sequence.

The *unique ID* of the element node corresponds to the **[normalized value]** property of the **attribute information item** in the **[attributes]** property that has a type *ID*.

> **Editorial Note:** JM: Need to augment attribute typing to accommodate DTD types and Schema types in a unified manner. For instance, what is the namespace of the ID type from a DTD?

> **Editorial Note:** JM: Not incorporated 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."

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

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

An element node has a constructor *element-node*, which takes an expanded-QName, a sequence of namespace nodes, a sequence of attribute nodes, a sequence of child nodes, and the node’s element declaration, which is a schema component. Like all other node constructors, the element-node constructor has the effect of creating a new node with a unique identity, distinct from all other nodes.

```
        element-node : (expanded-QName,
                        Sequence<NamespaceNode>,
                        Sequence<AttributeNode>,
                        Sequence<ElementNode | TextNode | ProcessingInstruct
                                 | CommentNode | ReferenceNode>,
                        SchemaComponent)
                     -> ElementNode
```

> **Editorial Note:** MF: The constructor only takes the element declaration, because it’s possible to derive the type of an element or attribute from its corresponding declaration. But would it be cleaner to include the type in the constructor as well?

To guarantee that the parent-child relationship is invertible, the element constructor logically creates a copy of all of its namespace, attribute, and children arguments and sets 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.

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

The accessors *name*, *namespaces*, *attributes*, *children*, and *declaration* return an element node’s constituent parts. The *type* accessor returns the schema component corresponding to the type of the element’s content: either a complex-type-definition or simple-type-definition. It is possible to derive the element’s type from its declaration.

```
name       : ElementNode -> expanded-QName
namespaces : ElementNode -> Sequence<NamespaceNode>
attributes : ElementNode -> Sequence<AttributeNode>
children   : ElementNode
           -> Sequence<ElementNode | TextNode | ProcessingInstructionN
                      | CommentNode | ReferenceNode>
declaration : ElementNode -> SchemaComponent
type        : ElementNode -> SchemaComponent
```

The accessor function *typed-value* returns a sequence of the simple-typed values of an element, if the element has a simple type, otherwise if the element has a complex type, it returns the empty sequence.

```
typed-value : ElementNode -> Sequence<SimpleValue>
```

The accessor function *unique-id* returns a sequence containing the unique ID of a node, if one exists, otherwise, it returns the empty sequence.

```
unique-id : ElementNode -> Sequence(0,1)<xs:ID>
```

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

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

```
/* Accessors for element information items: */
infoset-elem-namespace-name : ElementItem -> Sequence(0,1)<xs:anyURI>
infoset-elem-local-name     : ElementItem -> xs:string
infoset-elem-children       : ElementItem -> Sequence<InfoItem>
infoset-elem-attributes     : ElementItem -> Sequence<AttributeItem>
infoset-elem-in-scope-namespaces  : ElementItem -> Sequence<NamespaceI
infoset-elem-base-URI       : ElementItem -> xs:anyURI  /* unused ? */

psvi-elem-validity            : ElementItem -> xs:string
psvi-elem-element-declaration : ElementItem -> ElementItem
psvi-elem-type-definition     : ElementItem -> ElementItem
psvi-elem-schema-normalized-value : ElementItem -> xs:string
```

```
dm-element-node : ElementItem -> ElementNode
function dm-element-node(e) {
  name       = xfo:expanded-QName(infoset-elem-namespace-name(e),
                                  infoset-elem-local-name(e))
  nsnodes    = sequence-map(dm-namespace-node,
                            infoset-elem-in-scope-namespaces(e))
  attrnodes  = sequence-map(dm-attribute-node, infoset-elem-attributes
  kids       = dm-collapse-text-nodes(sequence-map(dm-node, infoset-el

  declaration = dm-schema-component(psvi-elem-validity(e),
                                    psvi-elem-element-declaration(e))
  type        = dm-schema-component(psvi-elem-validity(e),
                                    psvi-elem-type-definition(e))
  return element-node(name, nsnodes, attrnodes, kids, declaration)
}
```

Editorial Note: MF: Even though its possible to derive the type of an element from its corresponding element declaration, it seems cleaner to compute explicitly the schema components for both the element declaration and its simple or complex type.

Editorial Note: JM: Why is [base URI] discarded? **[Issue-0030: Base URI is a property of element nodes]**.

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

4.3 Attributes

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 *string-value*, which corresponds to the **[normalized value]** property.

The *declaration* of an attribute is a schema component and corresponds to the **[attribute declaration]** PSVI property. The *type* of an attribute is a schema component and corresponds to the **[type definition]** PSVI property. The representation of schema component information is defined in **8 Schema Components**.

An attribute node also has a *typed-value*. For a document with a schema, the attribute's typed-value corresponds to the **[schema normalized value]** PSVI property. For an attribute in a well-formed document with no associated schema, the attribute's typed-value is the empty sequence.

Editorial Note: JM: What about in the presence of a DTD? We need to recognize ID attribute types at a minimum.

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 *parent* of the attribute node corresponds to the **[owner element]** property.

An attribute node has the constructor *attribute-node*, which takes the attribute's name, a string value, and the node's attribute declaration, which is a schema component. Like all other node constructors, the attribute-node constructor has the effect of creating a new node with a unique identity, distinct from all other nodes.

```
    attribute-node : (xs:QName, xs:string, SchemaComponent) -> AttributeNo
```

The accessors *name* and *declaration* return an attribute's constituent parts. The accesor *string-value* returns its value. The *type* accessor returns the schema component corresponding to the simple type of the attribute's value. It is possible to derive the attribute's type from its declaration.

```
    name         : AttributeNode -> xs:QName
```

```
declaration : AttributeNode -> SchemaComponent
type        : AttributeNode -> SchemaComponent
```

The accessor function *typed-value* returns a sequence of the simple-typed values of an attribute.

```
typed-value : AttributeNode -> Sequence<SimpleValue>
```

The node accessors *node-kind*, *parent*, and *string-value* also apply to attribute nodes.

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

```
/* Accessors for attribute information items: */
infoset-attr-namespace-name   : AttributeItem -> Sequence(0,1)<xs:anyU
infoset-attr-local-name       : AttributeItem -> xs:string
infoset-attr-normalized-value : AttributeItem -> xs:string
infoset-attr-owner-element    : AttributeItem -> ElementItem

psvi-attr-validity            : AttributeItem -> xs:string
psvi-attr-attribute-declaration : AttributeItem -> ElementItem
psvi-attr-type-definition     : AttributeItem -> ElementItem
psvi-attr-schema-normalized-value : AttributeItem -> xs:string
```

```
dm-attribute-node : AttributeItem -> AttributeNode
function dm-attribute-node(a) {
   name = xfo:expanded-QName(infoset-attr-namespace-name(a),
                             infoset-attr-local-name(a))
   declaration = dm-schema-component(psvi-attr-validity(a),
                                     psvi-attr-attribute-declaration(a
   type = dm-schema-component(psvi-attr-validity(a),
                              psvi-attr-type-definition(a))
   return attribute-node(name, infoset-attr-normalized-value(a), decla
}
```

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

4.4 Namespaces

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 name of the *QName* is the empty sequence.

The *string-value* of the namespace node corresponds to the **[namespace URI]** property.

A namespace node has no *parent*.

Editorial Note: From XPath 1.0 : "The *parent* of the namespace node is the element node in whose namespaces collection this node appears." This is still the subject of debate.

A namespace node has the constructor *namespace-node*, which takes a namespace prefix and the absolute URI of the namespace being declared, either of which may be the empty sequence. If the URI is empty, the prefix must be empty too. 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.

```
namespace-node : (Sequence(0,1)<xs:string>, Sequence(0,1)<xs:anyURI>)
              -> NamespaceNode
```

The accessors *prefix* and *namespace-uri* return a namespace node's constituent parts:

```
prefix : NamespaceNode -> Sequence(0,1)<xs:string>
uri    : NamespaceNode -> Sequence(0,1)<xs:anyURI>
```

The accessors *name*, *node-kind* and *string-value* also apply to comment nodes. The *parent* accessor applied to a namespace-node returns the empty sequence:

```
parent(NamespaceNode) : Sequence(0,0)<ElementNode | DocumentNode>
```

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

```
infoset-ns-prefix           : NamespaceItem -> Sequence(0,1)<xs:string
infoset-ns-namespace-name   : NamespaceItem -> Sequence(0,1)<xs:anyURI
```

```
dm-namespace-node : NamespaceItem -> NamespaceNode
function dm-namespace-node(i) {
   return namespace-node(infoset-ns-prefix(i), infoset-ns-namespace-na
}
```

4.5 Processing Instructions

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

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

A processing-instruction node has the constructor *processing-instruction-node*, which takes a string 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.

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

```
processing-instruction-node : (xs:NCName, xs:string)
                            -> ProcessingInstructionNode
```

The node accessors *name*, *node-kind*, *parent*, and *string-value* also apply to processing-instruction nodes.

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

```
/* Accessors for processing instruction information items */
infoset-pi-target  : ProcessingInstructionItem -> xs:string
infoset-pi-content : ProcessingInstructionItem -> xs:string
```

```
dm-pi-node : ProcessingInstructionItem -> ProcessingInstructionNode
function dm-pi-node(i) {
  return processing-instruction-node(xfo:NCName(infoset-pi-target(i)),
                                     infoset-pi-content(i))
}
```

4.6 Comments

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

The *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 *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.

```
comment-node : xs:string -> CommentNode
```

The node accessors *node-kind*, *parent*, and *string-value* also apply to comment nodes. A comment node does not have an expanded-QName; the following signature specifies that *name* applied to a comment node returns the empty sequence:

```
    name(CommentNode) : Sequence(0,0)<xs:QName>
```

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

```
    /* Accessors for comment information items */
    infoset-comment-value    : CommentItem -> xs:string
```

```
    dm-comment-node : CommentItem -> CommentNode
    function dm-comment-node(i) {
      return  comment-node(infoset-comment-value i)
    }
```

4.7 References

The data model provides reference nodes as a general mechanism for referring to arbitrary nodes and preserving their identity.

We assume that the representation of a reference node is defined by the query system that implements the data model. The mechanism for implementing a reference node is implementation dependent, for example, a reference node might be represented by a ID or key value, an object identifier, an XPointer value [XML Pointer Language (XPointer)], etc. Node references are not serialized, i.e., they exist only for use by the query system. To serialize a node reference, an implementation may require that the reference be transformed explicitly to a valid XML value, such as an IDREF or URI reference, or the implementation may transform a reference node automatically.

Reference nodes are not guaranteed to be globally unique or persistent, although some implementation of the data model may choose to support persistent node references. Multiple techniques may exist for implementing reference nodes, and there may be multiple techniques for implementing node identity.

A reference node does not correspond to any information item.

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

The *string-value* of a reference node is implementation-defined.

A reference node has the constructor *reference-node*, which takes a document, element, attribute, text, processing-instruction, or comment node. Like all other node constructors, the reference node constructor has the effect of creating a new node with a unique identity, distinct from all other nodes.

```
      reference-node : (DocumentNode | ElementNode | AttributeNode | TextN
                        | ProcessingInstructionNode | CommentNode)
                       -> ReferenceNode
```

The accessor *dereference* returns the node referred to by a reference node.

```
      dereference    : ReferenceNode
                       -> (DocumentNode | ElementNode | AttributeNode | TextN
                         | ProcessingInstructionNode | CommentNode)
```

The node accessors *node-kind*, *parent*, and *string-value* also apply to reference nodes. The following signature specifies that *name* applied to a reference node node returns the empty sequence:

```
name(ReferenceNode) : Sequence(0,0)<xs:QName>
```

4.8 Text

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 *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**.

The *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 `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.

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

```
text-node : xs:string -> TextNode
```

The node accessors *node-kind*, *parent*, and *string-value* also apply to text nodes.

The mapping from character information items to text nodes occurs in the dm-element-node function. The *infoset-char-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.

```
infoset-char-code       : CharacterItem -> Code
```

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

```
dm-char-to-text : CharacterItem -> TextNode
function dm-char-to-text(c) {
  /* convert character code to string of length 1 */
  text-node(code2string(infoset-char-code c))
}
```

The *dm-collapse-text-node* function synthesizes a single text node from multiple text nodes. It calls *dm-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.

```
dm-collapse-text-nodes : Sequence<Node> -> Sequence<Node>
dm-text-node           : Sequence<Node> -> Sequence<Node>

function dm-collapse-text-node(nodes) {
  let newnodes := dm-text-nodes(nodes)
  return
    if (ignore-whitespace) then
      sequence-map(delete-whitespace-node, newnodes)
    else newnodes
}

function dm-text-nodes(nodes) {
  if (empty(nodes)) then empty-sequence()
  else
    let h := head(nodes),
        t := tail(nodes)
    return
      if (node-kind(h) = "text") then {
        /* Collapse two consecutive text nodes and apply
           dm-text-nodes recursively */
        if (empty(t)) then h
        else if (node-kind(head(t)) = "text") then
           dm-text-nodes(
             append(
               text-node(xfo:concat(string-value(h), string-value(he
               tail(t)))
      }
      else append(h, dm-text-nodes(t))
}
```

5 Simple Values

This section specifies how to construct and access simple values.

5.1 Primitive Values

A *primitive value* is a value contained in the union of the value spaces of the nineteen primitive XML Schema data types [XMLSchema Part 2]. They 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*.

Constructors for primitive values are specified in a forthcoming document that defines the functions and operators for XQuery and XPath 2.0. Each constructor takes a string in the lexical space of the given primitive type and returns a value in the value space of the type. For example, the constructor `xfo:integer (xfo:string)` constructs an `xs:integer` value from a string. Analogous constructors exist for the other primitive types above.

Two primitive values *xs:IDREF* and *xs:anyURI* have special accessors. The function *id* returns the element node denoted by an *xs:IDREF* value:

```
id  : xs:IDREF -> ElementNode
```

Similarly, the function *referent* returns a sequence of element nodes denoted by a *xs:anyURI*:

```
    referent  : xs:anyURI -> Sequence(0,1)<ElementNode>
```

If a referent does not correspond to an element node in the data-model, *referent* returns the empty sequence, otherwise it returns a singleton sequence containing the referenced element node. In the data-model, every IDREF and keyref value is guaranteed to refer to a `ElementNode`. This may not be the case for a URI reference value, which could refer to an element in an arbitrary document that is not contained in the data-model. In this case, *referent* may return the empty sequence.

5.2 Derived Simple Values

A derived simple value must be in the value space of its corresponding derived simple type. A derived simple type has a primitive base type and a set of constraining facets. For example, a "Sku" type is derived from string and has a *pattern* facet: :

```
    <simpleType name="Sku" base="string">
       <pattern value="\d{3}-[A-Z]{2}"/>
    </simpleType>
```

A simple value has a corresponding *type*, which is a schema component

A simple value has no identity. Simple values only may be compared for equality by value.

A simple value has the constructor *simple-value*, which takes a primitive value and the simple value’s type.

```
    simple-value  : (PrimitiveValue, SchemaComponent) -> SimpleValue
```

The accessor *value* returns a simple value’s primitive value and *type* returns its type:

```
    value  :  SimpleValue -> PrimitiveValue
    type   :  SimpleValue -> SchemaComponent
```

The accessor *string-value* returns the string representation of a simple value. This function can be used to recover a lexical representation of a string value. We note, however, that not all simple-typed values have a unique lexical representation (**[Issue-0027: Lexical representation of simple-typed values]**).

```
    string-value  :  SimpleValue -> xs:string
```

Since the data model supports sequences, a value of a simple type derived by list can be represented as a sequence of the base type of the list simple type.

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 (see **[Issue-0032: Keys and key references not represented]**).

6 Sequences

The data model supports a *sequence* collection. Unlike conventional lists, sequences are "flat", i.e., sequences may not contain other sequences. Sequences may contain duplicate nodes and simple values.

A sequence has no identity. Sequences only may be compared for equality by value.

The *string-value* of a sequence is the concatenation of the *string-value*s of each member of the sequence.

> **Note:** Sequences replace the node-sets in 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 will be provided by functions on node sequences.

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

The constructor *empty-sequence* constructs the empty sequence. The n-ary *append* constructor creates a new sequence containing the values in the its first argument followed by the appended values of its second through final arguments. Since a unit value is equivalent to a singleton sequence containing the unit value, *append* may be applied to unit values.

```
        empty-sequence : Sequence<UnitValue>
        append         : (Sequence<UnitValue>, ..., Sequence<UnitValue>) ->
```

A sequence has three accessors. The *empty* accessor returns true if its argument is the empty sequence and false otherwise. The *head* accessor returns the first value in a non-empty sequence. The *tail* accessor returns all items in a non-empty sequence excluding its first member.

```
        empty : Sequence<UnitValue> -> xs:boolean
        head  : Sequence<UnitValue> -> UnitValue
        tail  : Sequence<UnitValue> -> Sequence<UnitValue>
```

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. 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.

8 Schema Components

This section requires some familiarity with the [XML Schema: Formal Description]. In the data model, the [XML Schema: Formal Description] (XFSD) components are represented by four kinds of schema-component values: *element-declaration*, *attribute-declaration*, *simple-type-definition*, and *complex-type-definition*. A *schema component* collectively refers to these four kinds of values.

We use the notation **[f]** to refer to the XSFD component field named f. The **[name]** of an XSFD component has the form $i\#sn_1/.../sn_k$, where $sn_k = ss::j$. The symbol *i* is a namespace, *ss* is one of six symbol spaces (element, attribute, type, attribute group, model group, or notation), and *j* is a local name. Given an XSFD component with a **[name]** as above, the corresponding schema-component value has the following properties.

A schema component has an *expanded-QName* The namespace name of its expanded-QName is equal to *i*, and local part is equal to the empty string if *j* = *, otherwise it is equal to *j*.

The *parent* property is equal to the empty sequence if *k* = 1, otherwise it is the schema component with corresponding **[name]** equal to $i\#sn_1/.../sn_{(k-1)}$.

The *base* property is the component whose name is **[base]**.

The *derived-by-extension* property is true if **[derivation]** is *"extension"*, otherwise false.

The *derived-by-refinement* property is true if **[derivation]** is *"restriction"*, otherwise false.

Editorial Note: MF cite James: (An alternative to (d), (e) and (f) 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.) The [abstract], [refinement] and [content] fields of a XSFD component are not represented in this proposal (at least for the first two it would be easy to extend the proposal to cover them).

A *SchemaComponent* has the following accessors. The *component-kind* accessor returns the string *"element-declaration"*, *"attribute-declaration"*, *"simple-type-definition"*, or *"complex-type-definition"*. The other accessors are defined above.

```
component-kind         : SchemaComponent -> xs:string
name                   : SchemaComponent -> xs:QName
parent                 : SchemaComponent -> Sequence(0,1)<SchemaCompone
base                   : SchemaComponent -> SchemaComponent
derived-by-extension   : SchemaComponent -> xs:boolean
derived-by-refinement  : SchemaComponent -> xs:boolean
```

8.1 Mapping PSV Infoset additions to Schema Components

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

We note that each kind of XSFD component has a corresponding "top-most" or "root" component, named *xs:AnyElement*, *xs:AnyAttribute*, *xs:AnySimpleType*, *xs:AnyComplexType*. These root components represent the most general element, attribute, simple type, and complex type. We assume these components are pre-defined and rely on them in the definitions below.

A PSV element (attribute) information item has a **[validity]**, an **[element-declaration]** (**[attribute-declaration]**), and a **[type-definition]** property. The **[validity]** property may be *"valid"*, *"invalid"*, or *"notKnown"*. The **[element-declaration]** and **[attribute-declaration]** properties contains an element information item that is isomorphic to the XML Schema element or attribute declaration of the element or attribute information item. Similarly, 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. These properties are used to construct the schema component of an element or attribute in the data model.

The *dm-schema-component* function takes a string-valued validity property and an element information item that corresponds to an element or attribute declaration or a type definition. It constructs a schema-component value that corresponds to the schema component represented by its arguments.

```
        dm-schema-component : (xs:string, ElementItem) -> SchemaComponent
```

If its **[validity]** property is *"valid"*, *dm-schema-component* constructs a schema component that corresponds to the element or attribute declaration or type definition represented by the information item in its second argument.

If its **[validity]** property is *"invalid"* or *"notKnown"*, *dm-schema-component* returns the "root" schema component that corresponds to the element or attribute declaration or type definition represented by the information item in its second argument. For example, if the information item is an element declaration, *dm-schema-component* returns *xs:AnyElement*, and similarly, for the other three components. 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 the most general type information with the element or attribute node.

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...

Editorial Note: MF cite James' notes: Given an XSFD normalized attribute or element x[t types d], the declaration accessor would return deref(x) and the type accessor would return deref(t). Note that for any element or attribute node nd, if declaration(x) is not null, then local-name(x) = local-name (declaration(x)), and namespace-uri(x) = namespace-uri(declaration(x))

9 Equality

The functions and operators on data-model values are included in a forthcoming document that defines the functions and operators for XQuery 1.0 and XPath 2.0. Included in that document are the functions that define equality between values and equality between nodes. For completeness, we repeat the definitions of those functions here.

The data model includes two equality functions: *xfo:value-equal* and *xfo:node-equal*. The *xfo:value-equal* function denotes equality of values, and the *xfo:node-equal* function denotes equality of node identities.

```
xfo:value-equal : (Sequence<UnitValue>, Sequence<UnitValue>) -> xs:boo
xfo:node-equal  : (Node, Node) -> xs:boolean
```

We define the value-equality function, *xfo:value-equal*, as follows. We assume value equality over simple values is defined. Equality over all other data model values is defined recursively:

- Given attributes *a1* and *a2*, `xfo:value-equal`(*a1*,*a2*), if and only if `xfo:value-equal`(`name`(*a1*), `name`(*a2*)) and `xfo:value-equal`(`value`(*a1*), `value` (*a2*)).
- Given elements *e1* and *e2*, `xfo:value-equal`(*e1*, *e2*), if and only if `xfo:value-equal`(`name`(*e1*), `name`(*e2*)) and `xfo:value-equal`(`attributes`(*e1*), `attributes`(*e2*)) and `xfo:value-equal`(`children`(*e1*), `children`(*e2*)).
- Given two sequences (u_1, .., u_j) and (v_1, ..., v_k), `xfo:value-equal`((u_1, .., u_j), (v_1, ..., v_k)) holds if and only if $j = k$ and `xfo:value-equal`(u_i, v_i) holds for all 1 <= *i* <= *n*.

The function `xfo:node-equal` is only defined on nodes. For two nodes *n1* and *n2*, `xfo:node-equal`(*n1*, *n2*) holds if and only if *n1* and *n2* were created by the same application of a node constructor (See **4 Nodes**).

Editorial Note: MF: Should value equality over elements include all comment and PI children? See **[Issue-0015: Semantics of value equality operator ’=’]**.

10 Example

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.mywebsite.com/PartSchema"
      xs:schemaLocation = "http://www.mywebsite.com/PartSchema
                           http://www.mywebsite.com/PartSchema"
      name="nutbolt">
  <mfg>Acme</mfg>
  <price>10.50</price>
</p:part>
```

The document is valid with respect to the following XML schema:

```
    <xs:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema"
      targetNamespace="http://www.mywebsite.com/PartSchema">
      <xs:element name="part" type="part-type">
        <xs:complexType name="part-type">
          <xs:element name = "mfg" type="xs:string"/>
          <xs:element name = "price" type="xs:decimal"/>
          <xs:attribute name = "name" type="xs:string"/>
        </xs:complexType>
      </xs:element>
    </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 **8 Schema Components**.

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; the values *SC1, ...* represent schema component values;

```
     // Document node D1
     children(D1)     = E1
     parent(D1)       = empty-sequence()

     // Element node E1
     name(E1)         = xfo:expanded-QName("http://www.mywebsite.com/PartSc
     children(E1)     = append(E2, E3)
     attributes(E1)   = A1
     namespaces(E1)   = N1
     parent(E1)       = D1
     declaration(E1)  = SC1

     typed-valued(E1) = empty-sequence()
     type(E1)         = SC2

     // Attribte node A1
     name(A1)         = xfo:QNAME(empty-sequence(), "name")
     string-value(A1) = "nutbolt"
     parent(A1)       = E1
     declaration(A1)  = SC3

     typed-value(A1)  = "nutbolt"
     type(A1)         = SC4

     // Namespace node N1
     name(N1)       = xfo:expanded-QName(empty-sequence(), "p")
     uri(N1)        = xfo:anyURI("http://www.mywebsite.com/PartSchema")
     parent(N1)     = E1

     // Element node E2
     name(E2)        = xfo:QNAME(empty-sequence(), "mfg")
     children(E2)    = T1
     attributes(E2)  = empty-sequence()
```

```
namespaces(E2)  = N1
parent(E2)      = E1
declaration(E2) = SC5

typed-value(E2) = simple-value("Acme", SC4)
type(E2)        = SC4

// Element node E3
name(E3)       = xfo:QNAME(empty-sequence(), "price")
children(E3)   = T2
attributes(E3) = empty-sequence()
namespaces(E3) = empty-sequence()
parent(E3)     = E1
declaration(E3) = SC6

typed-value(E3) = simple-value(10.50, SC7)
type(E3)        = SC7

// Text node T1
value(T1)      = "Acme"
parent(T1)     = E2

// Text node T2
value(T2)      = "10.50"
parent(T2)     = E3

// Schema component SC1
component-kind(SC1)        = "element-declaration"
name(SC1)                  =
  xfo:expanded-QName("http://www.mywebsite.com/PartSchema", "part")
parent(SC1)                = empty-sequence()
base(SC1)                  = xs:AnyElement
derived-by-extension(SC1)  = false
derived-by-refinement(SC1) = true

// Schema component SC2
component-kind(SC2)        = "type-definition"
name(SC2)                  =
  xfo:expanded-QName("http://www.mywebsite.com/PartSchema", "part-type
parent(SC2)                = SC1
base(SC2)                  = xs:AnyComplexType
derived-by-extension(SC2)  = false
derived-by-refinement(SC2) = true

// Schema component SC3
component-kind(SC3)        = "attribute-declaration"
name(SC3)                  =
  xfo:expanded-QName("http://www.mywebsite.com/PartSchema", "name")
parent(SC3)                = SC1
base(SC3)                  = xs:AnyAttribute
derived-by-extension(SC3)  = false
derived-by-refinement(SC3) = true


// Schema component SC4
component-kind(SC4)        = "simple-type-definition"
name(SC4)                  =
  xfo:expanded-QName("http://www.w3.org/1999/XMLSchema", "string")
parent(SC4)                = empty-sequence()
base(SC4)                  = xs:AnySimpleType
derived-by-extension(SC4)  = false
derived-by-refinement(SC4) = true

// Schema component SC5
component-kind(SC5)        = "element-declaration"
name(SC5)                  =
  xfo:expanded-QName("http://www.mywebsite.com/PartSchema", "mfg")
parent(SC5)                = SC2
```

```
base(SC5)                   = xs:AnyElement
derived-by-extension(SC5)  = false
derived-by-refinement(SC5) = true

// Schema component SC6
component-kind(SC6)        = "element-declaration"
name(SC6)                  =
  xfo:expanded-QName("http://www.mywebsite.com/PartSchema", "price")
parent(SC6)                = SC2
base(SC6)                  = xs:AnyElement
derived-by-extension(SC6)  = false
derived-by-refinement(SC6) = true

// Schema component SC7
component-kind(SC7)        = "simple-type-definition"
name(SC7)                  =
  xfo:expanded-QName("http://www.w3.org/1999/XMLSchema", "decimal")
parent(SC7)                = empty-sequence()
base(SC7)                  = xs:AnySimpleType
derived-by-extension(SC7)  = false
derived-by-refinement(SC7) = true
```

Editorial Note: MF: New graphic is needed here.

11 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 URI]**, **[parent]** properties.
- **Attribute Information Items** with **[namespace URI]**, **[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 URI]** 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]**, **[element declaration]**, **[type definition]**, and **[schema normalized value]** properties on **Element Information Items**.
- **[attribute declaration]**, **[type definition]**, and **[schema normalized value]** properties on **Attribute Information Items**.

12 References

Document Object Model
World Wide Web Consortium, *Document Object Model*. See http://www.w3.org/TR/DOM-Level-2-Core/.

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

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

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/2000/WD-xmlquery-req-20000131.

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

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.

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

XSL Working Group

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

A Issues

The issues in **A Issues** 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 "W3C-members only". Some of the descriptions of the resolved issues are obsolete w.r.t. to the current version of the document.

A.1 Issues

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?

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 (W3C-members only) and http://lists.w3.org/Archives/Member/w3c-xml-query-wg/2000Sep/0079.html (W3C-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.

Issue-0004: Schema/DTD

Date: Oct-2000
Raised by: Datamodel Editors

Description: A document may refer to a DTD and have an associated schema.

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.(W3C-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.

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: TextNodes

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 (W3C-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).

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.

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.

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 in **9 Equality**.

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 '=' is undefined

Issue-0016: PSV Infoset Mapping - undefined terms

Date: 21-Feb-2001
Raised by: Michael Rys

Description: `Code` is undefined.

Resolution: Defined in **4.8 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.

Issue-0019: Element constructor that performs schema processing

Date: 15-Mar-2001
Raised by: James Clark

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  : (expanded-QName,
                 Sequence<NamespaceNode>,
                 Sequence<AttributeNode>,
                 Sequence<ElementNode | ProcessingInstructionNod
                         | TextNode | CommentNode | ReferenceNod
              -> 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?

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.

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.

Issue-0023: Support for document repositories

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

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 - 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 - 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 - 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.

Issue-0025: Types of Sequences

Date: 27-Apr-2001
Raised by: Mike Kay

Description: Should sequence values carry their type as do simple 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?

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.

Issue-0028: Whitespace handling

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

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?

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

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

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.

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]).

Issue-0032: Keys and key references not represented

Date: 17-May-2001
Raised by: Query

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 (see **[Issue-0032: Keys and key references not represented]**).

B Open Issues (Non-Normative)

- Issue-0001: PSV Infoset identity constraints
- Issue-0003: Example parent
- Issue-0004: Schema/DTD
- Issue-0005: Lists of Simple Values
- Issue-0007: TextNodes
- Issue-0009: Schema info
- Issue-0011: Access to facets
- Issue-0012: Representation of reference values
- Issue-0015: Semantics of value equality operator '='
- Issue-0018: Representation of lists of IDREFS and NMTOKENS
- Issue-0019: Element constructor that performs schema processing
- Issue-0020: Semantics of copy
- Issue-0021: Declared vs. In-scope namespaces
- Issue-0022: Abstraction of Run-time type information
- Issue-0023: Support for document repositories
- Issue-0024: Support for Schema-invalid documents
- Issue-0025: Types of Sequences
- Issue-0026: Schema Component Values vs. Nodes
- Issue-0027: Lexical representation of simple-typed values
- Issue-0028: Whitespace handling
- Issue-0029: Use of Reference Nodes
- Issue-0030: Base URI is a property of element nodes
- Issue-0031: Schema component does not reveal [content] property
- Issue-0032: Keys and key references not represented

C Resolved Issues (Non-Normative)

- Issue-0002: Representation of atomic values
- Issue-0006: Collections
- Issue-0008: Node vs edge centric data model
- Issue-0010: Node identity
- Issue-0013: Equality operators on collections
- Issue-0014: Elements with unordered children
- Issue-0016: PSV Infoset Mapping - undefined terms
- Issue-0017: Relationship between Ordered and Unordered collections