



XForms 1.0

W3C Working Draft *16 February 2001*

This version:

<http://www.w3.org/TR/2001/WD-xforms-20010216>

(Available as: [PDF](#), [zip archive of HTML](#))

Latest version:

<http://www.w3.org/TR/xforms/>

Previous version:

<http://www.w3.org/TR/2000/WD-xforms-20001219>

Editors:

Micah Dubinko (Cardiff) <mdubinko@Cardiff.com>

Josef Dietl (Mozquito Technologies) <josef@mozquito.com>

Roland Merrick (IBM) <Roland_Merrick@uk.ibm.com>

Dave Raggett (W3C/OpenWave) <dsr@w3.org>

T. V. Raman (IBM) <tvraman@almaden.ibm.com>

Linda Bucsay Welsh (Intel) <linda@intel.com>

Authors:

See [author list](#)

[Copyright](#) ©1998, 1999, 2000, 2001 [W3C](#)® ([MIT](#), [INRIA](#), [Keio](#)), All Rights Reserved. W3C [liability](#), [trademark](#), [document use](#) and [software licensing](#) rules apply.

Abstract

This document presents a description of the architecture, concepts, processing model, and terminology underlying XForms, the next generation Web forms. Except as noted, it represents the current consensus of the Working Group.

"XForms" is W3C's name for a specification of Web forms that can be used with a wide variety of platforms of varying capabilities, for instance, desktop computers, television sets, personal digital assistants, cell phones, computer peripherals and even paper.

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 Working Draft that incorporates new material regarding the XForms Processing Model. Additionally, this draft includes minor editorial clarifications and changes to highlight areas where the Working Group desires additional input from outside sources. Interested parties are encouraged to provide feedback and comments.

This document is a W3C Working Draft for review by W3C members and other interested parties. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress". A list of current public W3C Working Drafts can be found at <http://www.w3.org/TR>.

This document has been produced as part of the [W3C HTML Activity](#).

Please send detailed comments on this document to www-forms@w3.org, the public forum for discussion of W3C's work on Web forms. To subscribe, send an email to www-forms-request@w3.org with the word `subscribe` in the subject line (include the word `unsubscribe` if you want to unsubscribe). The [archive](#) for the list is accessible online.

Table of Contents

- [Copyright notice](#)
- [1 About the XForms 1.0 Specification](#)
- [2 Concepts](#)
- [3 Terminology](#)
- [4 Datatypes](#)
- [5 XForms Model](#)
- [6 Dynamic Constraint Language](#)
- [7 XForms User Interface](#)
- [8 Binding](#)
- [9 Using XForms with Other Document Types](#)

- [10 Processing Model and Conformance](#)
 - [Appendix A: Schema for XForms Model](#)
 - [Appendix B: XSLT from Simple to Schema Syntax](#)
 - [Appendix C: Sample Forms](#)
 - [Appendix D: Optional Function Libraries](#)
 - [Appendix E: References](#)
 - [Appendix F: Change History](#)
-

Authors:

Steven Pemberton, CWI (co-chair)

Sebastian Schnitzenbaumer, Mozquito Technologies (co-chair)

Micah Dubinko, Cardiff

Peter Stark, Ericsson

Roland Merrick, IBM

T. V. Raman, IBM

Linda Bucsay Welsh, Intel

Gavin McKenzie, JetForm

Rob McDougall, JetForm

John McCarthy, Lawrence Berkeley National Laboratory (Until November 2000)

Frank Olken, Lawrence Berkeley National Laboratory (Until November 2000)

Ray Waldin, Lexica

Tantek Çelik, Microsoft

Reveliotis Panagiotis, Philips (Until December 2000)

David Cleary, Progress Software

Mike Mansell, PureEdge

Josef Dietl, Mozquito Technologies

Michael Fergusson, SoftQuad Software

Dave Raggett, W3C/Openwave

Leigh Klotz, Xerox

Copyright Notice

Copyright © 2001 [World Wide Web Consortium](#), ([Massachusetts Institute of Technology](#), [Institut National de Recherche en Informatique et en Automatique](#), [Keio University](#)). All Rights Reserved.

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

W3C Document Copyright Notice and License

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

Copyright © 1994-2001 [World Wide Web Consortium](#), ([Massachusetts Institute of Technology](#), [Institut National de Recherche en Informatique et en Automatique](#), [Keio University](#)). All Rights Reserved.

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

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

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

1. A link or URL to the original W3C document.
2. The pre-existing copyright notice of the original author, or if it doesn't exist, a notice of the form: "Copyright © [*\$date-of-document*] [World Wide Web Consortium](#), ([Massachusetts Institute of Technology](#), [Institut National de Recherche en Informatique et en Automatique](#), [Keio University](#)). All Rights Reserved.

<http://www.w3.org/Consortium/Legal/>" (Hypertext is preferred, but a textual representation is permitted.)

3. *If it exists*, the STATUS of the W3C document.

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

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

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

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

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

W3C Software Copyright Notice and License

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

Copyright © 1994-2001 [World Wide Web Consortium](#), ([Massachusetts Institute of Technology](#), [Institut National de Recherche en Informatique et en Automatique](#), [Keio University](#)). All Rights Reserved.

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

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

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

1. The full text of this NOTICE in a location viewable to users of the redistributed or derivative work.
2. Any pre-existing intellectual property disclaimers. If none exist, then a notice of the following form: "Copyright © [\$date-of-software] [World Wide Web Consortium](#), ([Massachusetts Institute of Technology](#), [Institut National de Recherche en Informatique et en Automatique](#), [Keio University](#)). All Rights Reserved.
<http://www.w3.org/Consortium/Legal/>."
3. Notice of any changes or modifications to the W3C files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

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

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

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

[previous](#) [next](#) [contents](#)

1 About the XForms 1.0 Specification

Contents

- [1.1 Background](#)
- [1.2 Reading the Specification](#)
- [1.3 How the Specification is Organized](#)
- [1.4 Document Conventions](#)

This chapter is informative.

1.1 Background

Forms are an important part of the Web, and they continue to be the primary means of interactivity used by many Web sites. Web applications and eCommerce solutions have sparked the demand for better Web forms with richer interactions. XForms are the response to this demand--extended analysis, followed by the creation of a new platform-independent markup language for online interaction between a [XForms Processor](#) and a remote entity. XForms are the successor to XHTML forms, and benefit from the lessons learned in the years of HTML forms implementation experience.

Further background information on XForms can be found at <http://www.w3.org/MarkUp/Forms>.

1.2 Reading the Specification

This specification has been written with various types of readers in mind--In particular XForms authors and XForms implementors. We hope the specification will provide authors with the tools they need to write efficient, attractive, and accessible documents, without overexposing them to the XForms implementation details. Implementors, however, should find all they need to build conforming [XForms Processors](#). The specification begins with a general presentation of XForms and becomes more and more technical and specific towards the end. For quick access to information, a general table of contents, specific tables of contents at the beginning of each section, and an index provide easy navigation, in both the electronic and printed versions.

The specification has been written with two modes of presentation in mind: electronic and printed. In case of a discrepancy, the electronic version is considered the authoritative version

of the document.

1.3 How the Specification is Organized

The specification is organized into the following chapters:

Chapters 1 and 2: An introduction to XForms

The introduction includes a brief tutorial on XForms and a discussion of design principles behind XForms.

Chapters 3 and up: XForms reference manual.

The bulk of the reference manual consists of the specification of XForms. This reference defines what may go into XForms and how [XForms Processor](#) must interpret the various components in order to claim conformance.

Appendixes:

Appendixes contain a normative description of XForms described in XML Schema, information on optional function libraries, references, a change history, and other useful information.

1.4 Documentation Conventions

The following highlighting and typography is used to present technical material in this document and other documents from the XForms Working Group:

Special terms are defined in their own chapter; hyperlinks connect uses of the term to the definition.

Throughout this document, the namespace prefixes "xform:" and "xsd:" are used to denote the XForms and XML Schema namespaces respectively. This is by convention only; any namespace prefix may be used in practice.

BNF grammar productions are presented as follows:

```
[1] name1 ::= BNF Grammar 1 /* Comments */  
[2] name2 ::= BNF Grammar 2  
[3] name3 ::= BNF Grammar 3  
[4] name4 ::= BNF Grammar 4  
[5] name5 ::= BNF Grammar 5
```

Non-normative short examples are set off typographically:

- Example item

While lengthier non-normative examples are set off typographically and may include a short explanation:

Good Example, using Tables

```
<foo href="http://www.example.com/XForms" />
Multiple lines in length
```

And an explanation of the example of proper syntax or usage

Bad Example, using Tables

```
<foo href=http://www.example.com/XForms>
Multiple lines in length
```

And an explanation of the example of incorrect syntax or usage.

References to external documents are in [[Square Brackets](#)] with links to the references section of this document.

The XML representations of various elements within XForms are presented as follows: Listed are the element name, names of all attributes, allowed values of attributes appearing after a "=" character, default values of attributes appearing after a ":" character, and allowed content. One or more headings below the table provide additional explanatory information.

XML Representation : <code><example></code>
<pre><example count = integer size = (small medium large) : medium > <!-- Content: (allowed-content) --> </example></pre>

Attribute Definitions:

count = integer
description of this attribute

Attributes Defined Elsewhere:

size

The following highlighting is used for non-normative commentary:

Editor's Note: Inline informational note to provide annotations or point out design decisions.

[Editor's Feedback Request **1.4.sample**: A uniquely identified request on behalf of the Working Group for specific feedback on XForms.]

[Ed. General comments intended for removal before final publication.]

2 Concepts

Contents

- [2.1 What are XForms?](#)
- [2.2 What is the XForms Model?](#)
- [2.3 What is the XForms User Interface?](#)
- [2.4 What is the XForms Submit Protocol?](#)

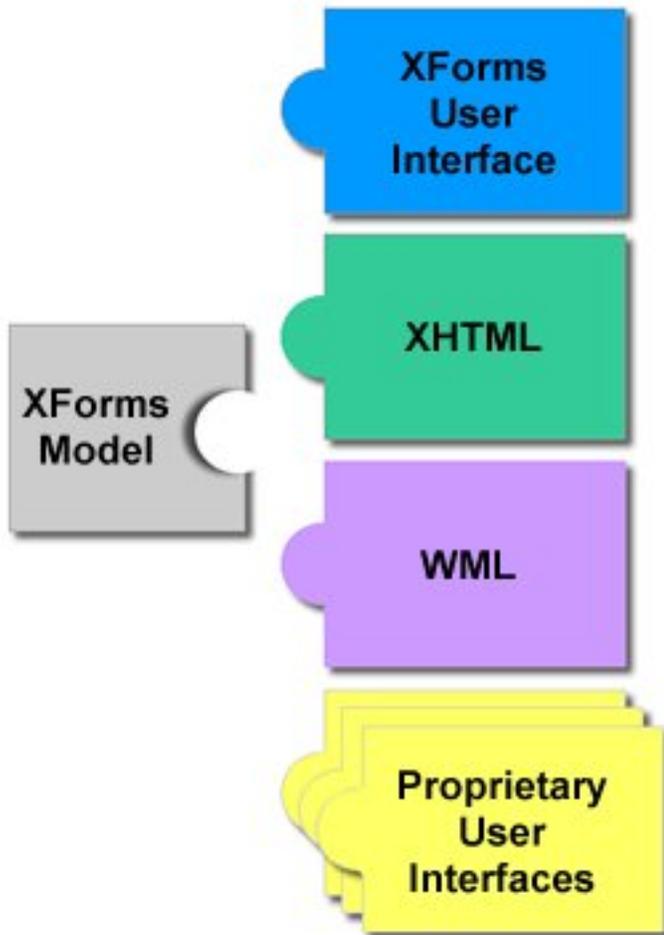
This chapter is informative.

2.1 What are XForms?

The design of existing Web forms didn't separate the *purpose* from the *presentation* of a form. XForms, in contrast, are comprised of separate sections that describe what the form does, and how the form looks. This allows for flexible presentation options, including classic XHTML forms, to be attached to an XML form definition.

The following illustrates how a single device-independent XML form definition, called the [XForms Model](#), has the capability to work with a variety of standard or proprietary user interfaces:

Presentation Options

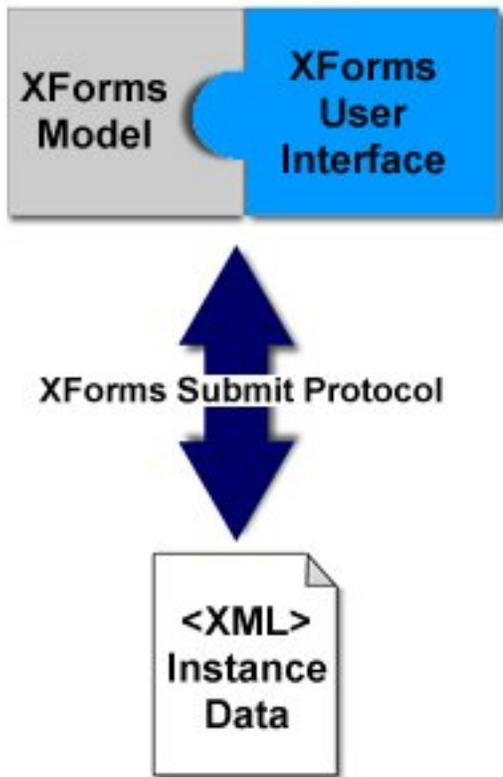


The [XForms User Interface](#) provides a standard set of visual controls that are targeted toward replacing today's XHTML [form controls](#). These [form controls](#) are directly usable inside XHTML and other XML documents, like SVG. Other groups, such as the Voice Browser Working Group, are independently developing user interface components for XForms.

An important concept in XForms is that forms collect data, which is expressed as XML [instance data](#). Among other duties, the [XForms Model](#) describes the structure of the [instance data](#). This is important, since like XML, forms represent a structured interchange of data. Workflow, auto-fill, and pre-fill form applications are supported through the use of [instance data](#).

Finally, there needs to be a channel for [instance data](#) to flow to and from the [XForms Processor](#). For this, the [XForms Submit Protocol](#) defines how XForms send and receive data, including the ability to suspend and resume the completion of a form.

The following illustration summarizes the main aspects of XForms:



The following sections will explain these in greater detail.

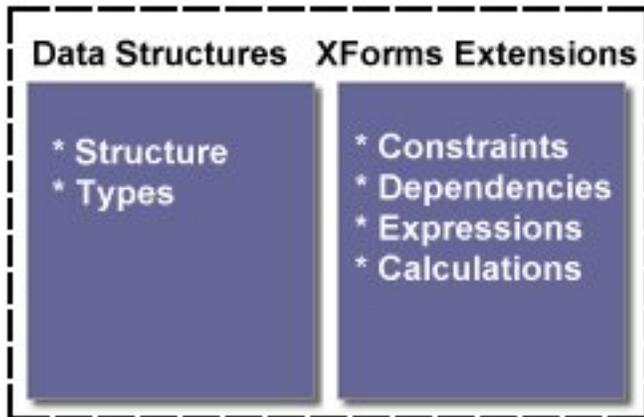
2.2 What is the XForms Model?

The [XForms Model](#) is the non-visual definition of an XML form. It serves several purposes:

- It defines the [model items](#) that make up the [XForms Model](#).
- It describes the structure of the associated XML [instance data](#).
- It defines types that apply to [model items](#) and can be reused in the same or different [XForms Model](#).
- It defines limits and restrictions that apply to [model items](#) when the form is being filled.
- It defines relationships and dependencies between [model items](#).

The [XForms Model](#) is subdivided into two components: Data Structures and XForms Extensions.

The XForms Model



The Data Structures Component provides a schema that describes the [instance data](#) and provides reusable data types, a role that fits well with XML Schemas. Due to this, the XForms Working Group is focusing on an XML Schema compatible definition for the Data Structures Component. In addition, we are investigating the option of a simpler syntax, more in tune with current XHTML authors.

Additionally, the [XForms Model](#) includes aspects that are not typically expressed in schemas, like additional constraints, dependencies, an Dynamic Constraints Language, and calculations. For instance, stating that a [model item](#) is "read only" is outside the scope of XML Schema, yet clearly an important capability for an XML form definition. XForms Extensions Components such as these will be handled through attributes that will extend the XML Schema syntax.

The [XForms Model](#) is capable of representing potentially complex interrelationships between [model items](#). One example of this might be a "spouse information" section that is only applicable when the user has previously indicated that he or she has a spouse. Relationships such as these, along with calculations, will be represented in a lightweight, XML-friendly Dynamic Constraints Language.

The [XForms Model](#) will also keep track of numeric values using decimal arithmetic. Unlike "floating point" math which is unable to exactly represent certain commonly-used values, decimal arithmetic works with numbers in the way most users expect, and avoids many types of rounding errors.

Lastly, the [XForms Model](#) is closely related to the XML [instance data](#). The definition of the [XForms Model](#) provides an area with an open content model where arbitrary well-formed [instance data](#) in any namespace can reside. XForms authors are typically not required to include [instance data](#), although they are free to do so. Some useful applications of [instance data](#) include:

- A [XForms Processor](#) can submit the completed form as XML [instance data](#).
- A server might pre-populate certain [model items](#) by sending XML [instance data](#) to the [XForms Processor](#).

- A user might suspend form-filling, storing the partially-completed XML [instance data](#) on the server.
- A workflow application might route XML [instance data](#) to various users in a list.
- A data storage application might store only one copy of the [XForms Model](#) and [XForms User Interface](#), but store many XML [instance data](#) entities.

2.3 What is the XForms User Interface?

The user interface is the part of electronic forms most immediately noticeable to users. By separating the user interface from the rest of the form, XForms provides flexibility in presentation options. In order to provide increased functionality, a separate [XForms User Interface](#) will be specified. This will include all the functionality of current XHTML forms, and more.

Typically, XHTML [form controls](#) are expressed in fairly generic terms, for instance `<select>` to represent menu controls. Web application designers have expressed a desire for more control over presentation. XForms enable this additional level of control, since presentations specific to desktop browsers, handheld devices, and even paper can be attached to a single [XForms Model](#). This modular construction also leads to alternate user interfaces. In particular, the Voice Browser Working Group is interested in developing a specification for non-visual [form controls](#) for use with XForms.

[XForms User Interface](#) consists of a set of XHTML modularization elements used to define user interface widgets ([form controls](#)) and a [binding](#) mechanism for connecting these [form controls](#) to the underlying [XForms Model](#). Even though the XForms [form controls](#) are designed for use with XHTML, we do not mandate a complete XHTML rendering engine for displaying [XForms User Interfaces](#). Specifically, the intent is for [XForms User Interface](#) to be rendered without the need to implement a full table rendering model as required today by HTML 4.0.

The XForms [form controls](#) are designed for use with XHTML and are initially inspired by the set of [form controls](#) available in HTML 4.0 as well as common constructs in use today that require the use of scripting --rather than declarative markup. The intent is for these new set of XForms [form controls](#) to be used within XHTML documents to provide a rich end-user experience while using declarative markup.

The final necessary component of XForms is "[binding](#)", which in this case refers to making the connection between the [XForms Model](#) and [XForms User Interface](#). The XForms Working Group will define a [binding](#) for XHTML [form controls](#) and [XForms User Interface form controls](#). The [binding](#) mechanism is designed to be generic and device-independent i.e., it is intended that this mechanism be usable for [binding](#) a multiplicity of user interfaces suitable for use on different devices to the same underlying [XForms Model](#). Other groups that define XForms presentations may define corresponding [bindings](#).

2.4 What is the XForms Submit Protocol?

Conventional XHTML forms send [instance data](#) over HTTP, in flat name/value pairs, with unsophisticated URL-encoding. Later, the ability to include multipart mime-encoded data was added to support file upload. XForms are designed from the ground up to allow rich, internationalized, hierarchical XML [instance data](#), transferred over the [XForms Submit Protocol](#).

Besides just submitting data, XForms have the additional requirement of suspending a form-filling session, later resuming it. This also calls for another use for the [XForms Submit Protocol](#)--a way to remotely serialize the [instance data](#).

As of this writing, an XML Protocol Working Group has been formed within the W3C, and is currently accepting requirements. The XForms Working Group will work closely with this effort to ensure that the end results will be compatible with the [XForms Submit Protocol](#).

[previous](#) [next](#) [contents](#)

3 Terminology

This chapter is normative.

binding

The connection between a model item in the XForms Model and a form control in the XForms User Interface, or other presentation. Can also refer to the connection between a model item and its representation as a instance data item.

binding expression

An addressing expression used by the binding to connect form controls to other parts of XForms.

facet

A single defining aspect of a value space. Generally speaking, each facet characterizes a value space along independent axes or dimensions.

form control

A user interface control or "widget" that serves as a point of user interaction. The XForms User Interface defines several form controls.

instance data

Representation of the values of all the model items. When represented as XML, instance data can be accessed through the XML DOM.

instance data item

Representation of a single piece of data, constrained by the definition of a model item, and typically presented to the user through one or more form controls.

lexical space

A lexical space is the set of valid literals for a datatype.

model item

An abstract unit of data-collection within the XForms Model, which defines a type and possibly other constraints on a single piece of collected data.

value space

A set of values for a given datatype. Each value in the value space of a datatype is denoted by one or more literals in its lexical space.

XForms Model

The non-visible definition of an XML form as specified by XForms. The XForms Model defines the model items that exist, the structure of the instance data to represent the data, and constraints and other run-time aspects of XForms.

XForms Processor

A software application or program that implements the XForms specification.

XForms Submit Protocol

The means by which instance data is transported from one place to another, such as part of a submit, suspend, or resume operation.

XForms User Interface

A set of graphical form controls designed to provide a higher-quality user experience when compared to conventional XHTML forms.

[previous](#) [next](#) [contents](#)

4 Datatypes

Contents

- [4.1 Introduction](#)
- [4.2 String](#)
- [4.3 Boolean](#)
- [4.4 Number](#)
- [4.5 Currency](#)
- [4.6 Monetary](#)
- [4.7 Date](#)
- [4.8 Time](#)
- [4.9 Duration](#)
- [4.10 URI](#)
- [4.11 Binary](#)
- [4.12 XForms Facets](#)

This chapter is normative.

4.1 Introduction

This chapter sets out a core set of datatypes that all [XForms Processors](#) support. The [XForms Model chapter](#) describes the different ways that an [XForms Model](#) can be represented. For the purposes of this chapter, the most important distinction is "simple syntax" (defined by XForms) versus "Schema Syntax" (defined in [\[XSchema-1\]](#) and [\[XSchema-2\]](#)).

All XForms datatypes reuse and extend XML Schema [\[XSchema-2\]](#) datatypes, including all the constraining [facets](#). These are called static [facets](#), since they are predefined and unchangeable. In many cases, XForms defines like-named [facets](#) in the XForms namespace, which are semantically equivalent, however an XForms Dynamic Constraint can be used to define the [facet](#). This allows the XForms [facets](#) to be dynamic, and freely change their evaluated constraint at run-time.

The [XForms Model](#) uses these datatypes to define how the submitted values bind to the [instance](#)

[data](#). For instance, here is how you could define a "leavingDate", bound to an element in the [instance data](#), which must have a value later in time than a "startingDate":

```
<date name="leavingDate" min="startingDate" />
<date name="startingDate" max="leavingDate" />
```

Here the constraints are totally dynamic, and in fact mutually dependent.

It is important to note that XForms dynamic constraints cannot change any underlying static constraints specified in an XML Schema.

XForms Validation Rule: Whenever XForms [facets](#) and Schema [facets](#) have potentially conflicting values, the one that restricts the [value space](#) to the smallest subset is used.

For instance, here's an example of an XForms [facet](#) that conflict with underlying schema static constraints:

Example of conflicting facets:

```
<xsd:complexType name="myNumber" >
  <xsd:restriction base="xform:number" >
    <xsd:maxInclusive value="20" />
    <xform:maxInclusive value="100" />
  </xsd:restriction>
</xsd:complexType>
```

This example shows a Schema maxInclusive [facet](#) alongside a conflicting XForms maxInclusive [facet](#). In this case, the XForms [facet](#) would have no effect, since the more restrictive constraint, value=" 20 ", is used.

Like XML Schema, XForms datatypes are based on [value spaces](#) which have [lexical space](#) representations. For each datatype defined, the following will be specified:

- [Lexical space](#) - how are values represented when appearing as attributes or element content.
- Canonical Representation - When multiple lexical representations are possible, which is preferred? For instance, both "100" and "1e2" are valid as a xform:number, but only the first is the canonical representation.
- Example - One or more examples of canonical lexical values.
- XForms [Facets](#) Used - Like Schema, XForms supports the concept of [facets](#). In addition to constraining [facets](#), XForms supports additional property [facets](#) that provide useful functionality for XForms processing. All XForms [facets](#) are listed later in this chapter.

4.2 String

The XForms datatype `string` is derived from the XML Schema [\[XSchema-2\]](#) datatype `string`.

Lexical Representation: as in Schema.

Canonical Representation: as in Schema.

Example: A value of "hello" would be represented as:

- hello

XForms Facets defined:

- [enumeration](#)
- [mask](#)
- [maxLength](#)
- [minLength](#)

The XForms `string` datatype inherits the static [facets](#) of its XML schema base datatype, which are:

```
length, minLength, maxLength, pattern, enumeration,  
whiteSpace
```

4.3 Boolean

The XForms datatype `boolean` is derived from the XML Schema [\[XSchema-2\]](#) datatype `Boolean`.

Lexical Representation: as in Schema.

Canonical Representation: as in Schema.

Example: A true value would be represented as:

- true

XForms Facets defined: No additional [facets](#).

The XForms `Boolean` datatype inherits the static [facets](#) of its XML schema base datatype, which are:

```
pattern, whiteSpace
```

4.4 Number

The XForms datatype `number` is derived from the XML Schema [\[XSchema-2\]](#) datatype `decimal`.

Lexical Representation: as in Schema.

Canonical Representation: as in Schema.

Example: A value of -42 would be represented as:

- -42

XForms Facets defined:

- [enumeration](#)
- [max](#)
- [maxExclusive](#)
- [maxInclusive](#)
- [min](#)
- [minExclusive](#)
- [minInclusive](#)
- [precision](#)
- [scale](#)

The XForms `number` datatype inherits the static [facets](#) of its XML schema base datatype, which are:

```
precision, scale, pattern, whiteSpace, enumeration,  
maxInclusive, maxExclusive, minInclusive, minExclusive
```

Numeric calculations should be performed on the internal [value space](#) values (not the [lexical space](#) values) using decimal arithmetic, except where the resource constraints preclude this.

4.5 Currency

The XForms datatype `currency` is derived from the XForms datatype `string`.

Lexical Representation: a list of 3 character currency codes, as defined in defined in [\[ISO 4217\]](#). This is considered an open list--additional currency codes not defined there are allowable.

Canonical Representation: No leading or trailing whitespace is allowed.

Example: A value 'US Dollars' would be represented as:

- USD

XForms Facets defined: The XForms currency datatype inherits all [facets](#) of its XForms base datatype. These are:

- [enumeration](#)
- [mask](#)
- [maxLength](#)
- [minLength](#)

4.6 Monetary Values

[Editor's Feedback Request **4.6.money**: The monetary datatype is essentially `xform:number` plus an `xform:currency` designator. The XForms Working Group does not have consensus on whether a momentary datatype should be an atomic datatype, or a new compound type mechanism is needed for this special case. Both alternatives are specified here. We would appreciate feedback on which alternative is better.]

Alternative A - atomic datatype

The XForms datatype `money` is derived from the XForms datatype `number`.

Lexical Representation: `currency` is represented by concatenating the `xform:currency` string to the end of the `xform:number` string, with no whitespace in between.

Canonical Representation: both the `xform:number` and `xform:currency` portions must be their individual canonical representation.

Example: A value of 4.37 Euro would be represented as:

- 4.37EUR

XForms Facets defined:

- [allowCurrency](#)
- [enumeration](#)
- [max](#)
- [maxExclusive](#)
- [maxInclusive](#)
- [min](#)
- [minExclusive](#)
- [minInclusive](#)
- [precision](#)

- [scale](#)

Alternative B - compound datatype

The XForms datatype `money` consists of two parts, the currency identifier and the value. The currency datatype is derived from the XForms datatype `currency`. The value subtype is derived from the XForms datatype `number`.

Lexical Representation: as a compound datatype, there is no single [value space](#), and therefore no single [lexical space](#). See example below.

Canonical Representation: See example below.

Example: A value of 4.37 Euro would be represented as two distinct lexical values:

- 4.37
- EUR

Editor's Note: This is essentially a shortcut for separately defining instances of the `xform:number` and `xform:currency` datatypes, but without independently settable [facets](#). When mapped to [instance data](#), if the datatype is bound to an element, the `xform:number` and `xform:currency` portions can be bound to child elements or attributes, possibly in the XForms namespace. If the datatype is bound to an attribute, there is no way to present both portions, since attributes have no children.

XForms Facets defined: same as alternative A.

4.7 Date

The XForms datatype `date` is derived from the XML Schema [\[XSchema-2\]](#) datatype `date`.

Lexical Representation: as in Schema.

Canonical Representation: as in Schema.

Example: A value of '31st January 2000' would be represented as:

- 2000-01-31

A value of '4 years from now' would be represented as:

- +P4Y

A value of 'right now' would be represented as a Dynamic Constraint:

- `now()`

XForms Facets defined:

- [enumeration](#)

- [max](#)
- [maxExclusive](#)
- [maxInclusive](#)
- [min](#)
- [minExclusive](#)
- [minInclusive](#)
- [precision](#)

The XForms date datatype inherits the static [facets](#) of its XML schema base datatype, which are:

```
duration, period, pattern, enumeration, whiteSpace,
maxInclusive, maxExclusive, minInclusive, minExclusive
```

Note that a special Dynamic Constraint function, `now()` is defined. This can be used to specify any min or max [facet](#). Additionally, time values relative to the submission date can be specified using positive or negative durations. The syntax for dates and durations are as per the subset of [\[ISO 8601\]](#) specified for XML Schemas for time instants and durations.

It is recommended that [XForms Processors](#) offer date and time pickers which offer date validation and choices from the distant past to the distant future. Small portable devices will likely validate and pick only dates in the range likely for business appointments near the current time; whereas, a full-featured desktop browser, which supports use cases such as historical records search and long-term financial obligations, should offer an extended range of dates. As always, the server must assume that the client has not performed the validation specified in the [XForms Model](#) and perform its own validation on the entered date.

4.8 Time

The XForms datatype `time` is derived from the XML Schema [\[XSchema-2\]](#) datatype `time`.

Lexical Representation: as in Schema.

Canonical Representation: as in Schema. For [XForms Processors](#) that have access to the correct local time zone, this must be included in the canonical representation.

Example: A value of '1:20 PM Eastern Standard Time' (5 hours behind of UTC) would be represented as:

- 13:20:00-5

XForms Facets defined:

- [enumeration](#)
- [max](#)

- [maxExclusive](#)
- [maxInclusive](#)
- [min](#)
- [minExclusive](#)
- [minInclusive](#)
- [precision](#)

The XForms `time` datatype inherits the static [facets](#) of its XML schema base datatype, which are:

`duration, period, pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive`

4.9 Duration

The XForms datatype `duration` is derived from the XML Schema [\[XSchema-2\]](#) primitive datatype `timeDuration`.

Lexical Representation: as in Schema.

Canonical Representation: as in Schema.

Example: A value of '1 year, 2 months, 3 days, 10 hours, and 30 minutes' would be represented as:

- P1Y2M3DT10H30M

A value of 'negative 120 days' would be represented as:

- -P120D

XForms Facets defined:

- [enumeration](#)
- [max](#)
- [maxExclusive](#)
- [maxInclusive](#)
- [min](#)
- [minExclusive](#)
- [minInclusive](#)
- [precision](#)

The XForms `duration` datatype inherits the static [facets](#) of its XML schema base datatype, which are:

pattern, enumeration, whiteSpace, maxInclusive, maxExclusive, minInclusive, minExclusive

[Editor's Feedback Request **4.9.precision**: Months only provide an approximate means to specify duration since individual months vary in length. The same holds true for years. Is this acceptable?]

4.10 URI

The XForms datatype `uri` is derived from the XML Schema [\[XSchema-2\]](#) datatype `uriReference`.

Lexical Representation: `uri` represents a Uniform Resource Identifier (URI) Reference as defined in Section 4 of [\[RFC 2396\]](#). A `uri` may be absolute or relative, and may have an optional fragment identifier. This datatype is used for values representing an absolute Uniform Resource Identifier (URI) as defined in [\[RFC 2396\]](#).

Canonical Representation: as in Schema.

Editor's Note: currently an open issue in Schema is how to handle non-ASCII values in either the URI [value space](#) or [lexical space](#). We plan to adopt whatever solution the XML Schema Working Group chooses.

Example: A value of 'http://www.w3.org/' would be represented as:

- <http://www.w3.org/>

XForms Facets defined:

- [enumeration](#)
- [length](#)
- [maxLength](#)
- [minLength](#)
- [uriScheme](#)

The XForms `uri` datatype inherits the static [facets](#) of its XML schema base datatype, which are:

length, minLength, maxLength, pattern, enumeration, whiteSpace

The `uriScheme` [facet](#) allows you to restrict URIs to a limited set of URI schemes. For instance, `http` or `mailto`. [XForms Processors](#) are encouraged to provide a means to pick or browse addresses, for instance an email address picker. The [XForms User Interface](#) may allow users to enter relative URIs, but the internal values will always be absolute URIs.

4.11 Binary

The XForms datatype `binary` is derived from the XML Schema [\[XSchema-2\]](#) datatype `binary`.

Lexical Representation: as in Schema.

Canonical Representation: as in Schema.

XForms Facets defined:

- [enumeration](#)
- [length](#)
- [maxLength](#)
- [mediaType](#)
- [minLength](#)

The XForms `binary` datatype inherits the static [facets](#) of its XML schema base datatype, which are:

```
encoding, length, minLength, maxLength, pattern,  
enumeration, whiteSpace
```

This is a datatype for use with data appropriate to specific Internet media types. The [XForms Processor](#) could use the media type to determine how to prompt the user. For example, an image could be acquired from a digital camera, an image scanner, or a disk file.

In simple syntax, here is a possible representation for a [model item](#) that accepts JPEG or PNG images:

```
<binary name="photo">  
  <mediaType>image/jpeg</mediaType>  
  <mediaType>image/png</mediaType>  
</binary>
```

Binary data could be packaged either in-place as part of XML [instance data](#) or held separately and referenced from XML. Further work is needed to cover the details.

[Editor's Feedback Request **4.11.binary-constraints**: Is there a need for [facets](#) to further constrain the data, for instance, to place limits on the size of the data? Other kinds of constraints may be appropriate, for example, for audio clips, you might want to set constraints on the bit rate and duration.]

4.12 XForms Facets

For each XForms [facet](#), the following is specified:

- Availability - does this [facet](#) apply to the simple syntax, Schema syntax, or both?
- Description - a description of the function of the [facet](#).
- Legal Values - a description of permissible values for the [facet](#).
- Default Values - a description of behavior when the [facet](#) is not specified.
- Applies to - a listing of all XForms datatypes that support this [facet](#).

4.12.1 XForms Facet: allowCurrency

Availability: Simple, Schema.

Description: indicates allowable currency datatypes.

Legal Values: a list of zero or more 3 letter currency codes (of subtype `currency`), e.g. USD or GBP.

Default Value: unrestricted.

Applies to:

- [money](#)

4.12.2 XForms Facet: enumeration

Availability: Simple, Schema.

Description: restricts the [value space](#) of the datatype to a specified list of possible values.

Legal Values: a list of datatype values compatible with the parent datatype.

Default Value: unrestricted.

Applies to:

- [string](#)
- [number](#)
- [currency](#)
- [money](#)
- [date](#)
- [time](#)
- [duration](#)

- [URI](#)
- [binary](#)

Editor's Note: the value of an XForms enumeration [facet](#) will almost always be a Dynamic Constraint, since the Schema enumeration [facet](#) handles static enumerations.

4.12.3 XForms Facet: length

Availability: Simple, Schema.

Description: restricts the [value space](#) of the datatype to only values which have the specified length. For string-based and binary-based datatypes, length is measured as in [\[XSchema-2\]](#).

Legal Values: any expression that evaluates to a non-negative `xform:number`.

Default Value: unrestricted.

Applies to:

- [string](#)
- [currency](#)
- [URI](#)

4.12.4 XForms Facet: mask

Availability: only Simple.

Description: restricts the [value space](#) of the datatype according to a specified set of rules.

Legal Values: a list of 0 or more values representing legal mask syntax.

Default Value: unrestricted.

Applies to:

- [string](#)
- [currency](#)

XML Schema has defined a Regular Expression language which is "similar to the regular expression language used in the Perl Programming language", and can be applied to most built-in datatypes. Regular expression syntax, however, is considered complex by some. Therefore, XForms defines the concept of a mask [facet](#). All mask [facets](#) are convertible into regular expressions.

The mask [facet](#) is available only using simple syntax. XML schema allows multiple [pattern facets](#) to be specified. Similarly, multiple mask or [pattern facets](#), but not a mixture, are permitted in simple syntax.

XForms mask uses the syntax and processing from [\[WML1.3\]](#) format. Some examples:

- A matches "A", "X", "\$", "%", or "."
- a matches "a", "x", "\$", "%", or "."
- X matches "A", "X", "\$", "%", ".", or "4"
- x matches "a", "x", "\$", "%", ".", or "4"
- N matches "0", "4", or "7"
- 3N matches "0", "63", or "999" but not "1234" (Note: only allowed at end of mask)
- *X matches "\$", "3.0", or "ABCDEFG" (Note: only allowed at end of mask)
- \ causes the next literal character to be inserted into the mask
- NNN\-NNNN matches "123-4567" but not "1234567"

As with WML format processing, an [XForms Processor](#) must ignore invalid masks.

4.12.5 XForms Facet: max

Availability: only Simple.

Description: for numeric datatypes, serves as a shortcut to the maxExclusive [facet](#).

Legal Values: see maxExclusive.

Default Value: see maxExclusive.

Applies to:

- [number](#)
- [money](#)
- [date](#)
- [time](#)
- [duration](#)

4.12.6 XForms Facet: maxExclusive

Availability: only Schema.

Description: restricts the [value space](#) to values below the specified exclusive upper bound.

Legal Values: any expression that evaluates to a value compatible with the parent datatype.

Default Value: unrestricted.

Applies to:

- [number](#)
- [money](#)

- [date](#)
- [time](#)
- [duration](#)

4.12.7 XForms Facet: maxInclusive

Availability: only Schema.

Description: restricts the [value space](#) to values below the specified inclusive upper bound.

Legal Values: any expression that evaluates to a value compatible with the parent datatype.

Default Value: unrestricted.

Applies to:

- [number](#)
- [money](#)
- [date](#)
- [time](#)
- [duration](#)

4.12.8 XForms Facet: maxLength

Availability: only Schema.

Description: restricts the [value space](#) of the datatype to a maximum length. For string-based, binary-based, and list-based datatypes, length is measured as in [\[XSchema-2\]](#).

Legal Values: any expression that evaluates to a non-negative `xform:number`.

Default Value: unrestricted.

Applies to:

- [string](#)
- [currency](#)
- [URI](#)
- [binary](#)

4.12.9 XForms Facet: mediaType

Availability: Simple, Schema.

Description: restricts the [value space](#) of the datatype to specific mime-types.

Legal Values: a list of zero or more mime-types.

Default Value: unrestricted.

Applies to:

- [binary](#)

4.12.10 XForms Facet: min

Availability: only Simple.

Description: for numeric datatypes, serves as a shortcut to the minExclusive [facet](#).

Legal Values: see minExclusive.

Default Value: see minExclusive.

Applies to:

- [number](#)
- [money](#)
- [date](#)
- [time](#)
- [duration](#)

4.12.11 XForms Facet: minExclusive

Availability: only Schema.

Description: restricts the [value space](#) to values above the specified exclusive lower bound.

Legal Values: any expression that evaluates to a value compatible with the parent datatype.

Default Value: unrestricted.

Applies to:

- [number](#)
- [money](#)
- [date](#)
- [time](#)
- [duration](#)

4.12.12 XForms Facet: minInclusive

Availability: only Schema.

Description: restricts the [value space](#) to values above the specified inclusive lower bound.

Legal Values: any expression that evaluates to a value compatible with the parent datatype.

Default Value: unrestricted.

Applies to:

- [number](#)
- [money](#)
- [date](#)
- [time](#)
- [duration](#)

4.12.13 XForms Facet: minLength

Availability: only Schema.

Description: restricts the [value space](#) of the datatype to a minimum length. For string-based and binary-based datatypes, length is measured as in [\[XSchema-2\]](#).

Legal Values: any expression that evaluates to a non-negative `xform:number`.

Default Value: unrestricted.

Applies to:

- [string](#)
- [currency](#)
- [URI](#)
- [binary](#)

4.12.14 XForms Facet: precision (applied to numeric datatypes)

Availability: Simple, Schema.

Description: for numeric datatypes, restricts the [value space](#) to values with the specified number of digits total are significant. Numeric datatypes always use precision along with scale, and precision must be greater than or equal to scale.

Legal Values: for numeric datatypes, any expression that evaluates to a non-negative

`xform:number`.

Default Value: unrestricted.

Applies to:

- [number](#)
- [money](#)

4.12.14.1 XForms Facet: precision (applied to date and time datatypes)

Availability: only Simple.

Description: for date and time datatypes, restricts the [value space](#) to values that include significant data in the specified precision.

Legal Values: for date and time datatypes, any expression that evaluates to "years", "months", "days", "hours", "minutes", or "seconds". Note that large precisions, such as "years" may be meaningless for time datatypes, and small precisions, such as "seconds" may be meaningless for date datatypes.

Default Value: unrestricted.

Applies to:

- [date](#)
- [time](#)
- [duration](#)

4.12.15 XForms Facet: scale

Availability: Simple, Schema.

Description: for numeric datatypes, restricts the [value space](#) to values with the specified number of digits after the decimal. E.g. `scale="0"` restricts the [value space](#) to integers. Numeric datatypes always use precision along with scale, and precision must be greater than or equal to scale.

Legal Values: any expression that evaluates to a non-negative `xform:number`.

Default Value: unrestricted.

Applies to:

- [number](#)
- [money](#)

4.12.16 XForms Facet: uriScheme

Availability: Simple, Schema.

Description: restricts the [value space](#) of a datatype to the set of values that conform to one or more URI schemes, such as `http`, `ftp`, `mailto`, `news`, or `ldap`.

Legal Values: a list of zero or more strings that represent URI schemes.

Default Value: unrestricted.

Applies to:

- [URI](#)

[previous](#) [next](#) [contents](#)

5 XForms Model

Contents

- [5.1 Introduction](#)
- [5.2 Model Item Properties](#)
- [5.3 Using Datatypes](#)
- [5.4 Combining XForms and Schema](#)

This chapter is normative.

5.1 Introduction

The [XForms Model](#) supports XML Schema built-in datatypes such as `string` and custom datatypes defined using XML Schema syntax.

This chapter goes into further detail on the XForms simple syntax. XForms simple syntax is designed for easy hand-authoring and is targeted at HTML authors. Simple syntax provides an easy to learn interface for HTML authors; it achieves its simplicity by providing a smaller range of functionality than that afforded by XML Schemas. Specifically, XForms simple syntax allows the author to:

- Declare elements that conform to any of the elementary datatypes e.g., `string`;
- Declare elements that conform to complex types defined elsewhere using XML Schema e.g., `USPostalAddress`;
- Declare additional run-time (dynamic) XForms constraints e.g., date of birth precedes date of graduation;

In order to stay simple, simple syntax **does not** permit the definition of reusable datatypes. Instead, reusable datatypes can be defined in XML Schema and instantiated using XForms simple syntax as described here.

Later on, this chapter discusses ways to combine XForms simple syntax with XML Schema.

5.2 Model Item Properties

The following [model item](#) properties are similar in syntax to constraining datatype [facets](#). Instead of constraining the [value space](#) of a datatype, however, they add properties such as `readOnly` or `required` to individual [model items](#). The following properties are available for all datatypes, and their syntax is explained later in this document.

Many [model item](#) properties can be represented by the Dynamic Constraints Language, and may evaluate to values that can change at any time. [XForms Processors](#) that implement [form controls](#) need to dynamically update as the evaluated properties change.

5.2.1 Model Item Property: name

Description: provides a specific name for the declaring datatype.

Legal Values: only values of type `xsd:NCNAME`

Default Value: none.

Authors can associate a human-readable name with a declaring datatype through the use of the name property. Each name should be unique within the scope of the [XForms Model](#) where it is declared.

When using simple syntax to define elements that appear in the [instance data](#), the name property provides the corresponding element name, as shown in the [atomic datatype example](#) below.

5.2.2 Model Item Property: readOnly

Description: describes whether the value is restricted from changing. The ability of [form controls](#) to have focus and appear in the tab order is unaffected by this property.

Legal Values: any expression that evaluates to `xform:boolean`.

Default Value: `false`.

In addition to restricting value changes, the `readOnly` property provides a hint to the [XForms User Interface](#). [Form controls](#) bound to a [model item](#) with the `readOnly` property should indicate that entering or changing the value is not allowed. The hint provided has no effect on visibility, focus, or tab order.

[Editor's Feedback Request **5.2.2.constant**: Should it be possible to make the `readOnly` [facet](#) immutable? i.e. if an element is marked as read-only and immutable, then the [XForms Processor](#) could rely on the [facet](#) not changing and employ a different rendering. Do we need a "constant"?]

5.2.3 Model Item Property: required

Description: describes whether a value is required before the [instance data](#) is submitted.

Legal Values: any expression that evaluates to `xform:Boolean`

Default Value: `false`.

Often forms require certain values to be entered. Within XForms, this may be a static requirement and defined in an XML Schema (e.g. `xsd:minOccurs="1"`). Alternatively a value may only be required if some condition is satisfied.

Except as noted below, the `required` property does not provide a hint to the [XForms User Interface](#) regarding visibility, focus, or tab order. XForms authors are strongly encouraged to make sure that [form controls](#) that accept `required` data are visible. An [XForms Processor](#) may provide a unique indication that a [form control](#) is required.

Note: Suspend and resume operations are not restricted by the `required` property.

Note: As with the constraining [facets](#), conflicting properties are resolved by choosing the most restrictive. For instance, if `xform:required="false"` was specified but `xsd:minOccurs="1"` was also defined by the Schema for the element, the element would still be required.

[Editor's Feedback Request **5.2.3.default-default**: it might be useful to set the default for the `required` attribute for an entire [XForms Model](#). What should the default default be? How could we assign a default for a single [XForms Model](#)? This could apply to other attributes as well, e.g. `readOnly`, etc..]

[Editor's Feedback Request **5.2.3.null**: Null values; XML Schema `nullable="true"`. We have not yet addressed the subject of *null*. It is of particular relevance for required items.]

5.2.4 Model Item Property: relevant

Description: indicates whether the [model item](#) is currently relevant to the rest of the [XForms Model](#). [XForms Processors](#) would typically not render an associated [form control](#), including children, when the value is `false`.

Legal Values: any expression that evaluates to `xform:Boolean`

Default Value: `true`.

Many forms have fields dependent on other conditions. For example, a form might ask whether the respondent owns a car. It is only appropriate to ask for information about their car if they have indicated that they own one. In XForms, this occurs through the `relevant` property.

The `relevant` property provides hints to the [XForms User Interface](#) regarding visibility, focus, and tab order. In general, when `true`, associated [form controls](#) should be made visible. When `false`, associated [form controls](#) should be hidden, though an [XForms Processor](#) may only disable the [form controls](#). In either case, the [form controls](#) should be removed from the tab order and not allowed focus.

The following table shows the interaction between `required` and `relevant`.

	<code>required="true"</code>	<code>required="false"</code>
<code>relevant="true"</code>	The form control (and any children) should be visible or available to the user. The XForms User Interface may indicate that a value is required.	The form control (and any children) should be visible or available to the user. The XForms User Interface may indicate that a value is optional.

<code>relevant="false"</code>	The form control (and any children) should be hidden or unavailable to the user. Entering a value or obtaining focus should not be allowed. The XForms User Interface may indicate that should the form control become relevant, a value would be required.	The form control (and any children) should be hidden or unavailable to the user. Entering a value or obtaining focus should not be allowed.
-------------------------------	---	---

5.2.5 Model Item Property: calculate

Description: indicates that the value of the declaring datatype is to be dynamically calculated.

Legal Values: any expression that evaluates to a datatype compatible with the declaring datatype.

Default Value: none.

An [XForms Model](#) may include [model items](#) that are computed from the other values elsewhere. For example, the sum over line items for quantity times unit price, or the amount of tax to be paid on an order. The computed value can be represented as an Dynamic Constraint using the values of other [model items](#).

5.2.6 Model Item Property: priority

Description: indicates the relative priority for calculations of the [model item](#).

Legal Values: any expression that evaluates to an integer in the range 0-32767.

Default Value: 0.

For [model items](#) that are calculated, this optional property specifies a calculation order.

5.2.7 Model Item Property: validate

Description: indicates that the value of the declaring datatype is to be dynamically validated.

Legal Values: any expression that evaluates to `xform:Boolean`

Default Value: `true`.

An [XForms Model](#) may include [model items](#) that need to be validated. The specified Dynamic Constraint is invoked every time the value of the declaring datatype changes. The expression must evaluate to `true` for the [model item](#) to be considered valid. Future versions of this specification will describe details such as immediate validation vs. onsubmit validation.

Dynamic Constraints used here are not restricted to examining the [instance data item](#) they are invoked on. The Dynamic Constraints Language provides the means to traverse the [instance data](#), as well as call-outs to external script.

The [XForms User Interface](#) may indicate whether a [form control](#) is currently valid or invalid.

[Editor's Feedback Request **5.2.7.cascade**: Will the `validate` property be evaluated on all the parent or child [model items](#) whenever a value changes? We need to make sure that inter-[model item](#) constraints will get evaluated.]

5.3 Using Datatypes

The basic datatypes defined by XForms can be used individually or aggregated to build appropriate structures for use in XForms applications. XForms provides a number of structures that are mapped to corresponding constructs in XML Schema.

[XForms Processors](#) that are Schema compliant can use arbitrarily complex Schema constructs in the [XForms Model](#). The full Schema syntax can be found in [\[XSchema-1\]](#) and [\[XSchema-2\]](#).

5.3.1 Atomic Datatype

Description: inserts an atomic datatype into the [XForms Model](#).

Simple Syntax: for each atomic datatype in the [Datatypes chapter](#), the simple syntax specifies an element with a matching name in the XForms Namespace. Optional allowed attributes in the XForms Namespace are the XForms constraining [facets](#) and [model item](#) properties, except as noted below. The element content is empty, except as noted below:

XML Representation: atomic datatypes

```

<datatype-name
  id = xsd:ID
  datatype-facet-name = value
  <!-- ... additional datatype facets ... -->
  model-item-property-name = value
  <!-- ... additional model item properties ... -->
  enum = ("open" | "closed") : "open"
  choices = dynamic-constraint

  <!-- exceptions noted below -->
>
  <!-- Content: described in this and the following section -->
</datatype-name>

```

Attribute Definitions:

id = xsd:ID

Optional unique identifier.

enum = ("open" | "closed") : "open"

Optional enumeration specifier, with details given below.

choices = dynamic-constraint

Optional dynamic constraint that produces an enumeration list dynamically.

Syntax Exceptions:

- <string> has 0 or more <mask> children representing the list members of the mask [facet](#). mask is not allowed as an attribute.
- <money> has 0 or more <allowedCurrency> children representing the list members of the allowedCurrency [facet](#). allowedCurrency is not allowed as an attribute.
- <uri> has 0 or more <scheme> children representing the list members of the scheme [facet](#). scheme is not allowed as an attribute.
- <binary> has 0 or more <mediaType> children representing the members of the mediaType [facet](#). mediaType is not allowed as an attribute.
- The enumeration [facet](#) is described in the following section.
- The name [model item](#) property is required.

Editor's Note: <money> may not be considered an atomic datatype (see the Editor's Feedback Request in the [Datatypes chapter](#)), and thus may be considered a structure rather than a datatype.

A method is needed to include datatypes defined elsewhere (such as in Schema format) into the [XForms Model](#). One syntax proposal is:

- <xform:element type="Schema datatype"/>

Example Simple Syntax:

```
<xform:string name="foo" minLength="1" />
```

Example Equivalent Schema Syntax:

```

<xsd:element name="foo">
  <xsd:complexType>
    <xsd:restriction base="xform:string">
      <xform:minLength value="1"/>
    </xsd:restriction>
  </xsd:complexType>
</xsd:element>

```

Second Example Simple Syntax:

```

<xform:binary name="foo">
  <xform:mediaType>image/jpeg</xform:mediaType>
  <xform:mediaType>image/png</xform:mediaType>
</xform:binary>

```

Second Example Equivalent Schema Syntax:

```
<xsd:element name="foo">
  <xsd:complexType>
    <xsd:restriction base="xform:binary">
      <xsd:mediaType value="image/jpg"/>
      <xsd:mediaType value="image/png"/>
    </xsd:restriction>
  </xsd:complexType>
</xsd:element>
```

5.3.2 Enumerated Datatype

Description: inserts an atomic datatype, with a restricting enumeration, into the [XForms Model](#). Enumerations are mentioned separately here only because of their special syntax.

Simple Syntax: an enumerated datatype is declared as an atomic datatype above, with the additional syntax as follows:

- An additional attribute `enum` can be either `open` (the default) or `closed`. When `closed`, the enumeration is strictly limited to the defined values. When `open`, other values are allowed, as long as they satisfy all other constraining [facets](#).
- Any datatype can be restricted to an enumerated [value space](#) by providing 1 or more `<value>` children with string content representing the individual enumerated value.
- Any datatype can be restricted to an enumerated [value space](#) by providing an attribute `choices` which contains an XForms Dynamic Constraint that returns a list of values at runtime. This list is used in addition to any `<value>` children.

XML Representation : `<value>`

```
<value
  id = xsd:ID
>
  <!-- Content: #PCDATA -->
</value>
```

Attribute Definitions:

`id = xsd:ID`

Optional unique identifier.

Note that an open enumeration is useful for scenarios like the following, where a multiple choice question has an "other" option:

- Visa
- MasterCard
- Diner's Club
- American Express
- Other...

Editor's Note: functionality similar to open enumerations is available in XML Schema through the combination of union and enumeration features.

Simple Syntax:

```
<xform:string name="foo" enum="closed">
  <xform:value>Visa</xform:value>
  <xform:value>MasterCard</xform:value>
  <xform:value>Diner's Club</xform:value>
  <xform:value>American Express</xform:value>
</xform:string>
```

Equivalent Schema Syntax:

```

<xsd:element name="foo" enum="closed">
  <xsd:complexType>
    <xsd:restriction base="xform:string">
      <xsd:enumeration value="Mastercard"/>
      <xsd:enumeration value="Diner's Club"/>
      <xsd:enumeration value="American Express"/>
    </xsd:restriction>
  </xsd:complexType>
</xsd:element>

```

Simple Syntax with Dynamic Choices:

```

<xform:string name="foo" enum="closed" choices="getCreditCardList()"/>

```

5.3.3 Group

Description: allows aggregate hierarchical arrangement of ordered datatypes.

Simple Syntax: the simple syntax specifies the element <group>, with the possible attributes representing the [model item](#) properties.

XML Representation : <group>

```

<group
  id = xsd:ID
  name = xsd:NCNAME
>
  <!-- Content: (group | union | array | switch | atomic-datatypes)* -->
</group>

```

Attribute Definitions:

id = xsd:ID

Optional unique identifier.

name = xsd:NCNAME

Required name for the group.

Example Simple Syntax:

```

<xform:group name="person">
  <xform:string name="personName"/>
  <xform:string name="personTitle"/>
</xform:group>

```

Example Equivalent Schema Syntax:

```

<xsd:group name="person">
  <xsd:sequence>
    <xsd:element name="personName" type="xform:string"/>
    <xsd:element name="personTitle" type="xform:string"/>
  </xsd:sequence>
</xsd:group>

```

Note that here element `person` does not define a new datatype; when mapped to XML Schema syntax; i.e. `person` is an anonymous type that cannot be reused.

[Editor's Feedback Request **5.3.3.unordered-group**: The Working Group is looking for feedback on whether the option for an unordered group is necessary in addition to the ordered group as described above.]

5.3.4 Union

Description: allows differing datatypes to be bound to a single [model item](#).

Simple Syntax: the simple syntax specifies the element <union>, with the possible attributes representing the [model item](#) properties. Child datatypes do not require the name attribute.

[Editor's Feedback Request **5.3.4.names**: Should the name attribute be required for each of the elements within a union?

XML Representation : <union>

```
<union
  id = xsd:ID
  name = xsd:NCNAME
>
  <!-- Content: (group | union | array | switch | atomic-datatypes)* -->
</union>
```

Attribute Definitions:

id = xsd:ID

Optional unique identifier.

Name = xsd:NCNAME

Required name for the union.

Example Simple Syntax:

```
<xform:union name="weekday">
  <xform:string enum="closed">
    <xform:value>Monday</xform:value>
    <xform:value>Tuesday</xform:value>
    <xform:value>Wednesday</xform:value>
    <xform:value>Thursday</xform:value>
    <xform:value>Friday</xform:value>
    <xform:value>Saturday</xform:value>
    <xform:value>Sunday</xform:value>
  </xform:string>
  <xform:number min="1" max="7" scale="0"/>
</xform:union>
```

Example Equivalent Schema Syntax:

```
<xsd:element name="weekday">
  <xsd:simpleType>
    <xsd:union>
      <xsd:simpleType>
        <xsd:restriction base="xform:string">
          <xsd:enumeration value="Monday"/>
          <xsd:enumeration value="Tuesday"/>
          <xsd:enumeration value="Wednesday"/>
          <xsd:enumeration value="Thursday"/>
          <xsd:enumeration value="Friday"/>
          <xsd:enumeration value="Saturday"/>
          <xsd:enumeration value="Sunday"/>
        </xsd:restriction>
      </xsd:simpleType>
      <xsd:simpleType>
        <xsd:restriction base="xform:decimal">
          <xsd:maxInclusive value="7"/>
          <xsd:minInclusive value="1"/>
          <xsd:scale value="0"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:union>
  </xsd:simpleType>
</xsd:element>
```

5.3.5 Array

Description: allows homogeneous collections (i.e., all members of an array are the same structure).

Simple Syntax: the simple syntax specifies the element <array>, with the possible attributes representing the [model item](#) properties.

XML Representation : <array>

```
<array
  id = xsd:ID
  name = xsd:NCNAME
  minOccurs = xform:number : 1
  maxOccurs = (xform:number | "unbounded") : 1
>
  <!-- Content: (group | union | array | switch | atomic-datatypes)* -->
</array>
```

Attribute Definitions:

id = xsd:ID

Optional unique identifier.

Name = xsd:NCNAME

Optional name for the array. Note: The author has the ability to specify whether a containing element for the repeating elements is produced. If a name is specified for the array then a containing element will be created where the containing element name will be derived from the name of the array. If no name is provided for the array then no containing element will be created.

minOccurs = xform:number

Optional restriction for the minimum number of elements in the array.

maxOccurs = xform:number | "unbounded"

Optional restriction for the maximum number of elements in the array, or "unbounded" for no restriction.

Example Simple Syntax:

```
<xform:array name="children" minOccurs="0" maxOccurs="unbounded">
  <xform:string name="child" />
</xform:array>
```

Example Equivalent Schema Syntax:

```
<xsd:element name="children">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="child" type="xform:string" minOccurs="0"
maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Second Example Simple Syntax:

```
<xform:array minOccurs="0" maxOccurs="unbounded">
  <xform:string name="child" />
</xform:array>
```

Second Example Equivalent Schema Syntax:

```
<xsd:element name="child" type="xform:string" minOccurs="0" maxOccurs="unbounded" />
<!-- note this syntax is only valid inside a group -->
```

5.3.6 Switch

Description: allows alternative structures to be defined within simple syntax and for a Dynamic Constraint to determine which should be created in the [instance data](#).

Simple Syntax: the simple syntax specifies the element <switch>, with the possible attributes representing the [model item](#) properties. The content model consists of <case> elements, which in turn contain datatype declarations. The <case> elements have two attributes: name from the [model item](#) properties, and condition, which consists of a script to validate.

XML Representation : <switch>

```

<switch
  id = xsd:ID
  name = xsd:NCNAME
>
  <!-- Content: (case)* -->
</switch>

```

Attribute Definitions:

- id** = xsd:ID
Optional unique identifier.
- Name** = xsd:NCNAME
Required name for the switch.

XML Representation : <case>

```

<case
  id = xsd:ID
  name = xsd:NCNAME
  condition = dynamic-constraint
>
  <!-- Content: (group | union | array | switch | atomic-datatypes)* -->
</case>

```

Attribute Definitions:

- id** = xsd:ID
Optional unique identifier.
- Name** = xsd:NCNAME
Required name for a specific case
- condition** = *dynamic-constraint*
Optional dynamic constraint; the condition used to select this case statement. A maximum of one case may be without a condition, this will be considered the default case.

Example Simple Syntax:

```

<xform:switch name="address">
  <xform:case name="us" condition="property(locale) is 'US'" >
    <xform:string name="street"/>
    <xform:string name="city"/>
    <xform:string name="state"/>
    <xform:string name="zip"/>
  </xform:case>
  <xform:case name="uk" condition="property(locale) is 'UK'" >
    <xform:string name="street"/>
    <xform:string name="town"/>
    <xform:string name="county"/>
    <xform:string name="postcode"/>
  </xform:case>
  <xform:case name="default">
    <xform:string name="street"/>
    <xform:string name="town"/>
    <xform:string name="county"/>
    <xform:string name="postcode"/>
  </xform:case>
</xform:switch>

```

Example Equivalent Schema Syntax:

```

<xsd:choice>
  <xsd:element name="address">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="street" type="xform:string"/>
        <xsd:element name="city" type="xform:string"/>
        <xsd:element name="state" type="xform:string"/>
        <xsd:element name="zip" type="xform:string"/>
      </xsd:sequence>
      <xsd:attribute name="xform:caseName" type="xsd:string"
        use="fixed" value="us"/>
      <xsd:attribute name="xform:caseCondition" type="xform:xmExpr"
        use="fixed" value=" property(locale) is 'US'"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="address" >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="street" type="xform:string"/>
        <xsd:element name="town" type="xform:string"/>
        <xsd:element name="county" type="xform:string"/>
        <xsd:element name="postcode" type="xform:string"/>
      </xsd:sequence>
      <xsd:attribute name="xform:caseName" type="xsd:string"
        use="fixed" value="UK"/>
      <xsd:attribute name="xform:caseCondition" type="xform:xmExpr"
        use="fixed" value=" property::locale is 'UK'"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="address" >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="street" type="xform:string"/>
        <xsd:element name="town" type="xform:string"/>
        <xsd:element name="county" type="xform:string"/>
        <xsd:element name="postcode" type="xform:string"/>
      </xsd:sequence>
      <xsd:attribute name="xform:caseName" type="xsd:string"
        use="fixed" value="default"/>
    </xsd:complexType>
  </xsd:element>
</xsd:choice>

```

Note that this capability is an extension of the XML Schema choice.

Editor's Note: This impacts [instance data](#), [XForms User Interface](#), the [XForms Model](#), and the [binding](#) between them. An [XForms Model](#) can exploit the switch construct to define a number of variants and to determine at runtime which single variant should be created in the instance.

5.4 Combining XForms and XML Schema

XML Schema allows annotations as foreign namespaced attributes. Adding elements is more problematic. Future versions of this specification will specify in greater detail how XForms datatypes, properties, and structures can be used within XML Schema.

- XForms namespaced datatypes can be used within XML Schema
- [Model item](#) properties can be used as namespaced attributes within XML Schema
- Using compound structures (group, array, union, switch, etc.) require further discussion.

5.4.1 Annotating an External Schema

XForms authors may wish to reuse datatypes and structures already defined in an external XML Schema. For use in an XForms application, such schemas may need to be annotated using XForms dynamic properties for expressing run-time constraints.

Future versions of this specification will describe how to add XForms annotations to an external Schema by reference without modifying the original Schema.

One syntax proposal is:

- `<xform:annotateElement elementID="age" min="17" max="63" />`
-

[previous](#) [next](#) [contents](#)

6 Dynamic Constraint Language

Contents

- [6.1 Introduction](#)
- [6.2 Datatypes](#)
- [6.3 Binding Expressions](#)
- [6.4 Operators](#)
- [6.5 XForms Core Function Library](#)
- [6.6 Lexical Structure](#)
- [6.7 Extensibility](#)

This chapter is normative.

6.1 Introduction

Many forms define integrity constraints that act over multiple fields. For example, the total value of an order can be defined in terms of a computation over other values such as unit prices, quantities, discounts, and tax and shipping costs. Such computations can be conveniently represented using the syntax outlined in here. This chapter describes an XForms Dynamic Constraints Language (DCL) based on XPath that enables these types of expressions without the use of a separate scripting language.

Dynamic Constraints are also useful for declaratively stating when a [form control](#) or subform needs to be filled out, according to some other value. A further use is to functionally define the acceptable choices for some [form control](#), when this depends on other values.

Editor's Note: For simplicity, this chapter currently defines DCL as being based on XPath without subsetting. Readers should note, however, there is not yet consensus within the XForms Working Group on this matter. Specific points under consideration are noted throughout this chapter.

In the following grammar, the non-terminal [NCName](#) is defined in [\[XML Names\]](#), and [S](#) is defined in [\[XML 1.0\]](#).

6.2 Datatypes

XForms Dynamic Constraints are built out of the XForms data types:

- **string** with single or double quote marks for delimiters
- **boolean** with the values *true* and *false*.
- **number** with subtypes for integers etc.
- **date**
- **time**
- **monetary value**
- **currency code**
- **binary data**
- **Internet media type** (also known as MIME types)
- **enumeration**

The **string**, **Boolean**, and **number** types correspond to those defined in XPath. Dates, time durations and monetary values etc. are subtypes of string. Additionally, the XPath datatypes **node-set** and **null** are allowed in the XForms Dynamic Constraint Language.

Resource-limited [XForms Processors](#) may define implementation limits on the maximum size of a node-set.

[Editor's Feedback Request **6.2.conversions**: XPath defines specific type conversions. The XForms Working Group is considering whether to include or exclude these as part of XForms Dynamic Constraints. Either way, there will be well-defined semantics of operations involving differing types. Comments from the community are welcome.]

If an operation cannot be performed an exception will be thrown. Exceptions are treated as events and can be caught using event handlers, declared in XML or in scripts.

Standalone XForms Datatypes are considered valid XForms Dynamic Constraints.

Editor's Note: The productions here currently do not properly reference and extend those found in XPath.

- [1] Digit ::= [0-9]
- [2] HexDigit ::= Digit
| ['a' - 'f']
| ['A' - 'F']
- [3] NullExp ::= 'null'
- [4] BoolExp ::= 'true' | 'false'
- [5] NumberExp ::= 0x[HexDigit]+
| [['-']Digit+['.' Digit*] [('e' | 'E')['+' | '-']Digit+]
- [6] StringExp ::= "" NCName? "" | "" NCName? ""
- [7] ArrayExp ::= '[' | [Expr [, Expr]*]

6.3 Binding Expressions

Like XPath, the XForms DCL models an XML document as a tree of nodes. There are different types of nodes, including element nodes, attribute nodes and text nodes. XPath uses '/' as a location-step separator. XML doesn't permit the '/' character within element or attribute names, so this is unambiguous.

XPath additionally allows an array index notation to address the n-th element in a sequence, for example, the line items in a purchase order.

```
<purchaseOrder orderDate="1999-10-20">
  <item partNum="872-AA">
    ...
  </item>
  <item partNum="926-AA">
    ...
  </item>
</purchaseOrder>
```

The second item in the purchase order could be addressed as follows:

- `purchaseOrder/item[2]`

As in XPath, array indexes start at 1, not 0. Authors should be aware of this, especially when writing applications that combine scripting and XForms.

XPath also allows you to address attributes. For instance, the `orderDate` attribute in the `purchaseOrder` element could be addressed as follows:

- `purchaseOrder/@orderDate`

To address the `partNum` attribute in the second item you could write:

- `purchaseOrder/item[2]/@partNum`

As with XPath, all addressing is based on the concept of a context node. In many situations, using a context node can lead to shorter identifiers. As an example, if the second `item` element above was selected as the context node, the `partNum` attribute could be addressed as follows:

- `@partNum`

Identifiers are evaluated from left to right. The value of an identifier must resolve to one of the above types. The identifier syntax is based on XPath and follows the same semantics. If an identifier starts with an element name, then the name must be in the current context (scope) or an ancestor context. If an identifier can't be resolved, an invalid identifier exception is thrown.

[Editor's Feedback Request **6.3.ns**: To what extent do [binding expressions](#) need to concern themselves with namespaces? (Note that for authoring simplicity, this proposal so far largely treats [binding expressions](#) in a non-namespace-aware fashion)]

- [8] Identifier ::= (('/'
| './'
| element-name) [' Expr '])+
- [9] PathExp ::= identifier ['@' attribute-name]
| '@' attribute-name

6.3.1 Root and Context Nodes

[Binding expressions](#) are often used to point to a specific [instance data item](#). When used in this role, the following rules determine how data is selected:

1. The root node for [binding expressions](#) refers to the virtual [instance data](#) <instance> element. For example:
 - / (selects <instance> and therefore all child [instance data](#))
 - /foo (selects the <foo> element, which must be a child of <instance>)
2. For purposes of addressing, all [instance data](#) is treated as if it were a local resource.
3. If a [binding expression](#) selects a node-set in a context where only a single node is required, by default only the first node from the node-set is used.
4. The context node for outermost [binding](#) elements (such as XForms UI elements) is the virtual [instance data](#) root, defined above. A "[binding](#) element" is any element that is explicitly allowed to have an `xform:ref` attribute. An XForms element is "outermost" when the XPath expression `ancestor::*` includes no [binding](#) element nodes.
5. The context node for non-outermost [binding](#) elements (such as XForms UI elements) is determined by evaluating the [binding expression](#) of the immediately enclosing element. An element is "immediately enclosing" when it is the first [binding](#) element node in the XPath expression `ancestor::*`. This is also called "scoped resolution".

Example:

Scoped Binding Expressions

```
<someGroupingWidget ref="element1/foo/bar" >
  <anotherWidget ref="element2"/>
  <anotherWidget ref="@attr"/>
</someGroupingWidget>
```

In this example, the `someGroupingWidget` has a [binding expression](#) of `element1/foo/bar`. According to the rules above, this outermost element would have a context node of `/`, which is the `<instance>` element. Both of the `anotherWidgets` then inherit a context node from their parent, the context node being `/element1/foo/bar`. Based on this, the `anotherWidget` [binding expressions](#) evaluate respectively to `/element1/foo/bar/element2` and `/element1/foo/bar/@attr`. Matching [instance data](#) follows:

Sample Instance Data

```
<instance xmlns="http://www.w3.org/2001/02/xforms">
  <element1 xmlns="...">
    <foo>
      <bar attr="xyz">
        <element2>xyz</element2>
      </bar>
    </foo>
  </element1>
</instance>
```

Here is a sample [instance data](#) that fits the above UI markup. "xyz" indicates sample data.

In some cases, [binding expressions](#) address the [XForms Model](#), for instance to select dynamic constraints or calculations. This operates similar to the above.

Editor's Note: A future revision will explain this in greater detail, including an example.

6.3.2 Canonical Binding Expressions

As with XPath, it is possible to construct many different [binding expressions](#) that end up pointing to the same location. That said, it is often useful to express a [binding expression](#) in a standard, compact representation, called a canonical [binding expression](#).

Canonical [binding expressions](#) are represented as a `AbsoluteLocationPath` as defined in [\[XPath\]](#). Additionally, canonical [binding expressions](#) use only default axis-specifiers (for elements) or the '@' abbreviation (for attributes). Examples:

- (canonical) `/a/b/c`
- (canonical) `/a/b/@c`
- (non-canonical) `a/b/c`
- (non-canonical) `child::a/child::b/child::c`
- (non-canonical) `/a/b/c/d/ancestor::c`

6.4 Operators

Editor's Note: Some work is still needed to specify operator precedence, associativity and the event names thrown upon exceptions.

XPath reserves '/' as a location-step separator, making it impractical to also use this symbol for division. The Dynamic Constraints Language makes consistent use of English names for operators is intended to minimize the potential for authoring errors, and to avoid the need for using character entity references for symbols.

The **not** operator can only be used with an operand that evaluates to a Boolean value. If the operand is `true`, the `not` operator evaluates to `false`. If the operand is `false`, the `not` operator evaluates to `true`.

The **if cond then expr1 else expr2** construct requires `cond` to evaluate to `true` or `false`. If `cond` is `true`, the value of the construct is the result of evaluating `expr1`, otherwise it is obtained from evaluating `expr2`.

The **is** operator compares two values, and produces a Boolean result.

The **expr is within(expr1, expr2)** construct evaluates to `true` if the result from evaluating `expr` falls within the inclusive range defined by `expr1` and `expr2`. If it falls outside the range the construct evaluates to `false`. The operands must be of the same type, and are restricted to numbers, strings, dates, times or monetary values with the same currency code. String comparison is defined as per the Unicode standard.

The **expr is not within(expr1, expr2)** construct evaluates to `false` if the result from evaluating `expr` falls within the inclusive range defined by `expr1` and `expr2`. If it falls outside the range the construct evaluates to `true`. The operands must be of the same type, and are restricted to numbers, strings, dates, times or monetary values with the same currency code. String comparison is defined as per the Unicode standard. Here are some examples of `true` statements:

- `3 is within(1,5)`
- `3 is not within(1,2)`
- `"aab" is within("aaa", "aac")`

The **is before** and related operators provide comparison operations similar to **is within**. The operands must be of the same type, and are restricted to numbers, strings, dates, times or monetary values with the same currency code. String comparison is defined as per the Unicode standard. Before and below denote earlier in the scalar range, while after and above denote later in the scalar range. For instance, here are some examples of `true` statements:

- `age is 60`
- `26 is not 27`
- `3 is below 4`
- `"Mary" is after "Mandy"`

The **and**, **or** and **xor** require Boolean operands and perform the corresponding Boolean operations. For instance, the following examples are all true:

- false is true and false
- true is true or false
- true is true xor false
- false is true xor true

The **plus**, **minus**, **times** and **over** operators require numeric operands (see below for exceptions) and perform the corresponding arithmetic operations. The **over** operator performs division and throws an overflow exception if the denominator is zero. The **plus** operator can also be applied to string operands, to perform string concatenation. The following examples are all true:

- 5 is 1 plus 4
- 3 is 6 over 2
- 3 is 5 minus 2
- "happy days" is "happy" plus " " plus "days"

The **%** operator is a postfix operator that divides its operand by 100.

- 9 is 15% times 60

The **=** operator performs assignment. The mechanism for [binding form controls](#) generally assumes that each [form control](#) is bound to a single [model item](#). Some [XForms User Interface](#) controls such as buttons and image-maps may need to set the values of several [model items](#) in the same action. It is proposed that this is handled using one or more assignment statements separated by semicolons:

Here is a simple example which sets both the city and state:

- city="London"; state="Ontario"

```
[10] InfixOperator ::= 'and'
                        | 'or'
                        | 'xor'
                        | 'plus'
                        | 'minus'
                        | 'times'
                        | 'over'
                        | 'is' [[PrefixOperator] ('above' | 'below' | 'before' | 'after')]
[11] InfixExp      ::= Expr InfixOperator Expr
[12] PrefixOperator ::= 'not'
[13] PrefixExp     ::= PrefixOperator Expr
[14] PostfixOperator ::= '%'
[15] PostfixExp    ::= Expr PostfixOperator
[16] SpecialOperator ::= 'is' ['not'] 'within'
```

```

[17] SpecialExp      ::= Expr SpecialOperator '(' Expr ',' Expr ')'
[18] IfThenElseExp  ::= 'if' Expr 'then' Expr ['else' Expr]
[19] Assignment     ::= [Lexpr '=' ]+ Expr [ ';' Assignment ]* [ ';' ]
[20] Lexpr          ::= Identifier |
                    Function |
                    (Lexpr)

```

6.5 XForms Core Function Library

This section defines a set of required functions useful within XForms. Function syntax is based on XPath:

```

[21] Arg            ::= Expr
[22] FunctionExp   ::= function-name '([arg [' , ' arg]*] )'

```

6.5.1 XPath Core Function Library

The XForms Core Function Library includes the entire [XPath](#) Core Function Library, including operations on node-sets, strings, numbers, and booleans.

Editor's Note: Further input is required on the ability for resource-constrained devices to implement the complete XPath Core Function Library.

6.5.2 Number Methods

Note: the following are defined within [XPath](#) - `number()`, `sum()`, `floor()`, `ceiling()`, and `round()`

Function: *number average(node-set)*

The [average](#) function returns the arithmetic average value, for each node in the argument node-set, of the result of converting the string-values of the node to a number. Numbers are added with **plus**, and then taken **over** the **count()** of the specified node-set.

Function: *number min(node-set)*

The [min](#) function returns the minimum value, for each node in the argument node-set, of the result of converting the string-values of the node to a number. Numbers are compared with **is below**.

Function: *number max(node-set)*

The [max](#) function returns the arithmetic average value, for each node in the argument node-set, of the result of converting the string-values of the node to a number. Numbers are compared with **is below**.

6.5.3 String Methods

Note: the following are defined within [XPath](#) - `string()`, `concat()`, `starts-with()`, `contains()`, `substring-before()`, `substring-after()`, `substring()`, `string-length()`, `normalize-space()`, and `translate()`.

6.5.4 Date/Time Methods

Function: *string* `now()`

The [now](#) function returns the current system time as a string value, in the canonical format defined within the XForms specification. If local time zone information is available, it is included in the string.

6.5.5 Miscellaneous Methods

Function: *null* `submit()`

The [submit](#) function immediately submits the [instance data](#) bound to the node that contains the expression.

Function: *null* `reset()`

The [reset](#) function immediately resets the [instance data](#) bound to the node that contains the expression.

6.6 Lexical Structure

When tokenizing, the longest possible token is always returned.

Whitespace is permitted between tokens with the following exceptions:

- before or after the / or . . / within compound identifiers.
- within number tokens
- within name tokens

Whitespace is required between adjacent alphanumeric tokens, e.g. white space is required between the operator "not" and the name of a function. Names follow the lexical rules for XML NAME tokens. Function names, however, are not permitted to include - or . for compatibility with externally defined functions.

Parentheses can be used for grouping, but otherwise have no effect on the semantics of Dynamic Constraints. The syntax caters for literals for null, booleans, numbers, and strings.

```
[23] Expr ::= NullExp
          | BoolExp
          | NumberExp
          | StringExp
          | ArrayExp
          | PathExp
          | InfixExp
          | PrefixExp
          | PostfixExp
          | SpecialExp
          | IfThenElseExp
```

6.7 Extensibility

This section will be expanded in future revisions, to cover extension functions and methods for calling out to script.

[previous](#) [next](#) [contents](#)

7 XForms User Interface

Contents

- [7.1 Introduction](#)
- [7.2 Abstract Form Controls](#)
- [7.3 Core Form Controls](#)
- [7.4 Custom Form Controls](#)
- [7.5 Multiple Pages](#)
- [7.6 Layout](#)

This chapter is normative.

7.1 Introduction

This document describes:

- Markup for the [XForms User Interface](#)
- New style properties for use in combination with Cascading Style Sheets (CSS) style properties for laying out [form controls](#).
- A mechanism for [binding](#) the [XForms User Interface](#) to the [XForms Model](#) and [instance data](#).

[XForms User Interface](#) controls are declared using markup elements, and their behavior refined via markup attributes. This markup may be further decorated with style properties that can be set using CSS stylesheets to deliver a customized look and feel. [XForms User Interface](#) controls defined here are bound to the underlying data instance by using the [binding](#) attributes as defined in the [Binding chapter](#).

This chapter addresses accessibility by taking a uniform approach to such features as captions, help text, tabbing and short cuts.

The group plans to address internationalization and conformance profiles in a later Working Draft.

7.1.1 Design Input

The following cases have been considered in this design:

- *Group boxes used to group [form controls](#)*
- *Explanatory text and graphics*

- Output [form controls](#) used for computed values
- Single-line and multi-line text entry [form controls](#)
- Check boxes for yes/no questions
- radio buttons and drop down menus for multiple choice questions
- Lists allowing multiple selections
- Buttons for navigation or [instance data](#) submission
- Image maps functioning as one or more buttons
- Tree controls with the ability to open and close nodes
- Sliders or rotary controls for picking from a range
- Spin controls for incrementing or decrementing a value
- Custom pickers e.g. for dates or colors
- Additional (pop-up) help
- Mechanisms for navigation through [form controls](#)
- Keyboard shortcuts for moving to particular [form controls](#)
- The ability to disable particular [form controls](#), thereby removing them from the navigation order, and from the submitted data
- The ability to prevent users from changing the values of particular [form controls](#)
- The ability for [form controls](#) to indicate an error in some manner
- The ability to selectively hide or reveal groups of [form controls](#)
- The ability to present XForms as a sequence of cards

Eventually, this chapter will describe functionality for most or all of the above cases.

7.2 Abstract Form Controls

Because many types of [form controls](#) share similar properties, we here specify a few abstract [form controls](#) which serves no purpose but to contain common attributes, properties, and elements. It is an error to include an abstract [form control](#) directly. Other [form controls](#) are later defined in terms of these abstract [form controls](#).

Note: This approach does not constrain implementations to utilize abstract [form controls](#).

The following pieces of markup (core elements and attributes) are common to all [XForms User Interface](#) controls defined in this document.

7.2.1 anyControl

An abstract [form control](#) anyControl is the basis for all XForms [form controls](#). It is defined as follows:

XML Representation : abstract form control

```

<anyControl
  xmlns = namespace-identifier
  xml:lang = xml-language-identifier
  id = xsd:ID
  class = TBD
  style = TBD
  ref = binding-expression
  xform = context-node-selector
>
  <!-- Content: caption, help?, hint?, onevent? -->
</anyControl>

```

Attribute Definitions:

xmlns = *namespace-identifier*

Optional standard XML attribute for identifying an XML namespace. It is frequently useful to include this standard attribute at this point.

xml:Lang = *xml-language-identifier*

Optional standard XML attribute to specify a human language for this element.

id = *xsd:ID*

Optional unique identifier used to refer to this particular `<xform>` element.

class = *TBD*

Optional selector for a style rule.

style = *TBD*

Optional inline style specification.

Attributes Defined Elsewhere:

ref = [binding-expression](#)

Optional [binding expression](#). Details in the [Binding chapter](#).

xform = *context-node-selector*

Optional context node selector. Details in the [Binding chapter](#).

Common Style Properties

Style properties specify custom look and feel, such as color and border styles. The specification uses the box model and style properties defined in the Cascading Style Sheets level 2 specification [\[CSS2\]](#) and also introduces a few new properties for an extended layout model for XForms. Note that the XForms Working Group is aware that the definition of new CSS properties is beyond the scope of our work. The additional style properties mentioned in this chapter are meant as an initial basis for discussions with the CSS Working Group. Style properties can be specified via style sheets, and changed via scripts through the DOM.

Some common properties include:

- `display`: used to hide or display the [form control](#)

- `width`: the width of the [form control](#)
- `height`: the height of the [form control](#)
- `border`: (and related longhand properties) the style of the [form control](#)'s border
- `caption-style`: Style for the [form control](#) label.

Note that interpretation of attributes like `height` are dependent on the layout model we use.

The new `caption-style` style property may be used to specify the position of the caption text relative to the [form control](#). The value is `top`, `left`, `right`, `bottom` or `hidden`. It may be used together with the CSS *text-align* property to set left, center or right alignment.

Editor's Note: The above list is illustrative; for the exhaustive list of CSS style properties, refer to the CSS specification.

Common Child Elements

The `caption` element

This element labels the [form control](#) with a descriptive summary. The caption makes it possible for someone who can't see the [form controls](#) to obtain a short description as they navigate between [form controls](#). This element is required. It contains inline content as defined by XHTML. Attributes `xml:Lang`, `class` and `style` can be used to further qualify the `caption` element. Attribute `style` on element `caption` styles the caption text; attribute `captionStyle` on the [form control](#) containing the caption determines the position of the caption text relative to the associated [form control](#).

The `help` element

This element provides a longer description that will help users to understand how to fill out this [form control](#). The `help` text will normally be shown only on request. This optional element contains inline content as defined by XHTML. Attributes `xml:Lang`, `class` and `style` can be used to further qualify the `help` element. Attribute `style` on element `help` styles the caption text; attribute `helpStyle` on the containing [form control](#) determines the position of the help text relative to the associated [form control](#).

The `hint` element

This optional element provides a short hint for the user, typically represented as a tooltip by graphical [XForms Processors](#). The tooltip text will normally be shown when the user remains on the [form control](#) for more than a certain length of time. It contains inline content as defined by XHTML. Attributes `xml:Lang`, `class` and `style` can be used to further qualify the `hint` element. Attribute `style` on element `hint` styles the hint text; attribute `hintStyle` on the containing [form control](#) determines the position of the hint text relative to the associated [form control](#).

The `onevent` element

This element can be used to bind event handlers to [form controls](#). It is defined in [\[XHTML Events\]](#). Details on XForms events can be found in the [Reference Processing Model chapter](#).

7.2.2 anyNavControl

The abstract [form control](#) `anyNavControl` represents any navigable [form control](#). It is based on `anyControl`, sharing all its attributes and stylable properties. The following attributes are additionally defined:

XML Representation: abstract navigable [form control](#)

```
<anyNavControl
  <!-- all attributes from <anyControl> -->
  accesskey = xsd:string
  navindex = xsd:integer
>
  <!-- Content: same as <anyControl> -->
</anyNavControl>
```

Attribute Definitions:

accesskey = xsd:string

This attribute defines a shortcut for moving the input focus directly to a particular [form control](#). The value of this is typically a single character which when pressed together with a platform specific modifier key (e.g. the *alt* key) results in the focus being set to this [form control](#).

navindex = xsd:integer

This attribute is a non-negative integer in the range of 0-32767 used to define the navigation sequence. This gives the author control over the sequence in which [form controls](#) are traversed. The default navigation order is specified in the [Reference Processing Model chapter](#).

7.3 Core Form Controls

The following [form controls](#) are defined in terms of the abstract [form control](#) on which they are based in terms of syntax. Unless noted otherwise, all [form controls](#) here are treated as inline text for purposes of XHTML processing.

7.3.1 Hidden

[Editor's Feedback Request **7.3.1.hidden**: Given a separate [XForms Model](#) that can store data values not visible to the end user, is there a need for a "hidden" [form control](#), perhaps for compatibility with XHTML forms?]

7.3.2 Output

Description: The output [form control](#) allows the display of a view of a data value, typically as part of other content. The resulting [form control](#) cannot be modified by the user.

Based On: `anyControl`

Syntax Definition: The element name is `output`.

Example:

```
The total comes to <output ref="order/totalPrice"/>
```

The output [form control](#) may be used in a `caption`, for instance when authors want to say: "I charged you 100.0 - and here is why".

7.3.3 Text Entry

Description: allows for the single or multiple line entry of text values.

Based On: `anyNavControl`

Syntax Definition: The element name is `textbox`.

Two additional attributes, `rows` and `cols` specify a number to be used as the [form control](#)'s height and width in characters, respectively. `rows` defaults to 1. Note that these do not constrain the amount of text that can be entered.

[Editor's Feedback Request **7.3.3.cols-default:** What should be the default for `cols`?]

Example:

```
<textbox ref="order/shipTo/street">  
  <caption>Street</caption>  
  <help>Please enter the number and street name</help>  
</textbox>
```

7.3.4 Checkbox

Description: allows for binary (yes/no) input.

Based On: `anyNavControl`

Syntax Definition: The element name is `checkbox`.

Example:

```
<checkbox ref="questionnaire/married">
  <caption>Are you married?</caption>
  <help>We need this to determine your tax allowance</help>
</checkbox>
```

The checkbox element is used for yes/no questions.

7.3.5 Single Select: radio buttons, drop-down menus and list boxes

Description: allows for various representations of [form controls](#) that allow the user to choose one option out of many.

Based On: `anyNavControl`

Syntax Definition: The element name is `exclusiveSelect`. One or more child `item` elements define the caption text for an individual radio button or menu item, with an optional attribute of `value`, which specifies the associated value to be used in the [instance data](#).

Example:

```
<exclusiveSelect ref="icecream/flavor">
  <caption>Flavor</caption>
  <item value="a">Vanilla</item>
  <item value="b">Strawberry</item>
  <item value="c">Chocolate</item>
</exclusiveSelect>
```

These constructs are used to encapsulate various forms of selection. We support two broad classes of selection characterized by the type of data being obtained from the user. When the [XForms Model](#) allows the user to pick one or more from a set of choices, the type of the underlying instance is a container; contrast this with the case where the [XForms Model](#) permits the user to pick **only one**. We parallel this distinction in the [XForms User Interface](#) markup by introducing elements `multipleSelect` and `exclusiveSelect`.

User interfaces typically support a wide range of selection widgets characterized by distinctive appearances and behaviors. We capture these distinctions e.g., pull down list versus a combo box, via `style` attributes on elements `exclusiveSelect` and `multipleSelect`.

The `list-ui` style property must be one of the following values:

- `radio`
- `checkbox`
- `menu`
- `listbox`

Each option may have a platform-specific behavior.

Editor's Note: We need a default representation, to be used in the absence of a style sheet.

The items are bound to the choices in the data type in the lexical order in which they appear in the markup. This avoids the need to redundantly name things in both the presentation markup and in the [XForms Model](#).

Layout can be handled through the automated layout mechanisms described later, or can use absolute positioning via style properties.

[Editor's Feedback Request **7.3.5.accessibility**: Accessibility requirement: long lists of choices are easier to use if they are grouped in some way, preferably with associated captions. Is there a need for hierarchical menus? An `itemgroup` or empty `separator` element might do the trick.]

7.3.6 Multiple Select: Lists

Description: allows for various representations of [form controls](#) that allow the user to choose several options out of many.

Based On: `anyNavControl`

Syntax Definition: The element name is `multipleSelect`. One or more child `item` elements are allowed, as for `exclusiveSelect`.

Example:

```
<multipleSelect ref="icecream/flavors">
  <caption>Flavors</caption>
  <item value="a">Vanilla</item>
  <item value="b">Strawberry</item>
  <item value="c">Chocolate</item>
</multipleSelect>
```

This construct can be used to populate structures like `array` in the [XForms Model](#).

7.3.7 Buttons

Description: similar to the XHTML [form control](#) of the same name, allows for one-time actions to occur.

Based On: `anyNavControl`

Syntax Definition: The element name is `button`. The attribute `action` may contain an XForms Dynamic Constraint to call when the [form control](#) is activated.

Example:

```
<button action="city = 'London'; state = 'Ontario'">
  <caption>Set Values</caption>
</button>
```

The `button` element is derived from XHTML. It allows you to submit the [instance data](#), or to set one or more data values at the same time. The caption is generally shown on the face of the button.

7.3.8 Submit

Description: a specific control for submitting the [instance data](#).

Based On: `button`

Syntax Definition: The element name is `submit`. The attribute `action` defaults to the [binding expression](#) `submit()`.

Example:

```
<submit xform="foo">
  <caption>Submit XML</caption>
</submit>
```

The [binding expression](#) on `submit` should select the [instance data](#) that will be submitted.

In future revisions, we will define similar controls, like `reset` or `suspend`.

7.3.9 Future Work

The XForms Working Group is continuing to investigate and design additional controls for use within the [XForms User Interface](#). These include sliders, spin controls, rotary controls, image maps, tree controls, and scrolling record controls.

7.4 Custom Form Controls

The following section outlines some of our ideas for reusing [XForms User Interface](#) markup.

7.4.1 Custom Pickers

Custom pickers allow us to create reusable user interface components. Just as we can define data types and structures that can be reused within the [XForms Model](#), reusable user interface components allow us to design complex XForms using the basic building blocks described above, and then reuse these components in multiple situations. As with any component framework, this has two basic requirements:

- Components need to declare what aspects of the component are parameterizable by the caller.
- The caller needs to be able to override the default values of the parameters declared in the component.

Here, we describe such a component framework along with sample markup. For this example, assume that `USShippingAddress` is a reusable data type that is used in multiple places in the [XForms Model](#), e.g. the user will be asked for a `billingAddress` and `shippingAddress`

--both of type `USShippingAddress`.

First, we show a simple example that is designed to bind [XForms User Interface](#) to a [model item](#) of type `USShippingAddress` with no attention to making the component reusable.

```
<groupbox>
  <textbox ref="address/street">
    <caption>Please enter your street address</caption>
  </textbox>
  <textbox ref="address/zip">
    <caption>Zip Code</caption>
  </textbox>
</groupbox>
```

Next, we prepare the above [XForms Model](#) fragment to become a reusable component that could be used for obtaining both the shipping and billing address. To do this, we need to parameterize those portions of the *component* that the caller will wish to modify.

```
<component name="AddressWidget" dataType="USShippingAddress">
  <param name="streetPrompt"/>
  <param name="zipPrompt"/>
  <param name="border" value="line"/>
  <groupbox border="$border">
    <textbox ref="address/street">
      <caption>
        <value-of name="streetPrompt"/>
      </caption>
    </textbox>
    <textbox ref="address/zip">
      <caption>
        <value-of name="zipPrompt"/>
      </caption>
    </textbox>
  </groupbox>
</component>
```

Note that the UI component as defined above does not create a user interface; user interface is created by explicitly instantiating the component via element `use-component` described next.

Finally, we use this component to instantiate the [XForms User Interface](#) for obtaining the shipping and billing address.

```
<Use-component name="address" component="AddressWidget">
  <with-param name="streetPrompt">
    Shipping Street Address
  </with-param>
  <with-param name="zipPrompt">
    Zip Code for state where we are shipping to
  </with-param>
  <with-param name="border" value="dotted"/>
</use-component>
```

The reusable component is instantiated by element `use-component`; parameter values are specified by the contained `with-param` elements.

7.5 Multiple Pages

The following section outlines some of our ideas for allowing multiple-page forms.

7.5.1 Subpages

Subpages provide a means to present XForms one bit at a time, breaking a complex task into smaller, simpler parts. Presentation of a subpage can occupy the entire "page" or just part of a page. Different presentations are possible, e.g. a stack of *pages* with visible name tags, or as a set of buttons for flipping through the stack or navigating directly to a particular subpage. One possible representation is a `formset` element enclosing one or more subpage elements, each of which starts with a `caption` element.

As the name implies subpage is not specific to XForms --our intent is to design subpage so that it can be used within XForms --and more generally within XHTML to create presentations where document views are presented to progressively reveal the document structure and content.

7.6 Layout

The following section outlines some of our ideas for making XForms independent from any particular presentation technology, for example XHTML tables.

Editor's Note: All style properties used by XForms to specify layout will be as defined by the CSS working group.

7.6.1 Grouping

The `groupbox` element is used as a container for defining a hierarchy of [form controls](#). A `groupbox` element can contain other `groupbox` elements.

The hierarchy defined by nested `groupbox` elements is used to determine the traversal order specified by attribute `navindex` on [form controls](#). Setting the input focus on a group box results in the focus being set to the lowest [form control](#) in the tabbing order within that group box.

[Editor's Feedback Request **7.6.1.binding-on-groups**: Currently, `groupbox` is being used to both group controls and provide [binding](#) context. It is an open issue whether the [binding](#) attribute `ref` is allowed here.]

7.6.2 Additional Style Properties

The following are preliminary ideas intended to start a dialog with the CSS Working Group and other interested parties.

Element `groupbox` allows the following additional layout attributes to control the layout of the [form controls](#) being packed inside the container. The `layout` property can be used to control whether the [form controls](#) within a group box are laid out from left to right or top to bottom. The value of the property is `horizontal`, `vertical` or `inherit`. It can be used together with CSS padding properties.

Editor's Note: As we develop the layout model, we may factor out some of the more common behaviors desired of element `groupbox` into elements such as `grid` to avoid overloading element `groupbox` with multiple functions.

The `field-align` property controls the amount of whitespace between the caption and the [form control](#) so as to ensure the desired alignment of all the [form controls](#) within the group box. The value of the property can be one of: `left`, `right`, `top`, `bottom`, `center`, `justify` or `inherit`. The default is `justify`.

Editor's Note: This could meet our requirement to not be dependent on XHTML layout/tables.

The width and height of a group box can be set via the corresponding CSS properties. The default is to size the box to the minimum needed for the contents of the group box. Setting the width to 100% ensures that the box is the maximum width permitted by the enclosing CSS block.

In the following example, the [form controls](#) are laid out horizontally and justified to fill the available space.

Quality Settings

Best Normal Draft Custom

Here is another example, but this time with a vertical layout:

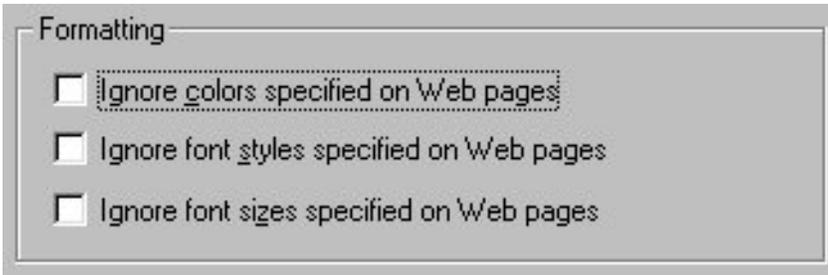
Tray Selection

Paper Source:

Media:

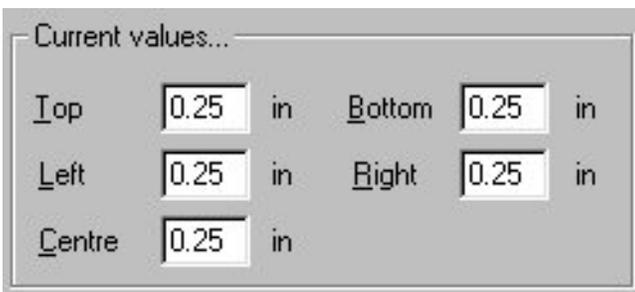
To justify the [form controls](#) for a vertical layout, the [XForms Processor](#) adjusts the spacing between

each [form control](#) and its caption. In the next example, the `field-align` property has been set to *left*. As a result, the spacing between the controls and the caption is the same for all [form controls](#).



7.6.3 Grid Layout

Sometimes a simple horizontal or vertical layout is not enough. The next example uses a grid layout:



Editor's Note: "in" in the example is an output [form control](#) that can display either "in" or "cm".

Grid layout is potentially complex, and further study is needed to find the most appropriate solution for XForms. Here are several possibilities:

- Use explicit positioning for each [form control](#). This is how most Windowing systems specify dialog boxes. It is common for the [form controls](#) to be constrained to positions on a grid with a spacing relative to the dialog font size. A subtlety is to make text input [form controls](#) use the same baseline as the caption text. This is a simple approach, and works fine provided the width and height of the group box are fixed.
- Use explicit positioning, but distinguish between hard and soft spacings. The soft spacings are adjusted automatically to match the width and height of the group box, and the font size etc. This is inspired by the approach used by Donald Knuth for T_EX.
- Use a style property on [form controls](#) to force that [form control](#) to appear on the next row, e.g. `break-before`. This could be combined with another style property to specify which [form controls](#) need to be vertically aligned with one another. One possibility for this is a named tab stop (such as "foo" in `tab-stop: foo`), where the position of the stop is determined automatically by the [XForms Processor](#). Another possibility is to define a penalty scheme where layout engine adjusts the positioning to minimize the sum over the positioning penalties. This approach would involve minimal effort on behalf of the author.
- Use explicit markup for each row, either a container element similar to XHTML's table row element `<tr>...</tr>` or an empty element such as `
`

- A generalization is to allow explicit markup to divide the remaining space up progressively horizontally or vertically, similar to the mechanism provided by XHTML's frames.

[Editor's Feedback Request **7.6.3.repeating**: What about "repeating structures" when these are specified in the [XForms Model](#)? One possibility is a record set control which allows users to add and delete records, where all of the records satisfy the same [XForms Model](#). An example is a line item in an order form, consisting of the quantity, product code and unit price.]

7.6.4 Text and graphics

It is often desirable to include graphics and additional text within group boxes, for instance, the following uses graphics to illustrate the idea of printing in black & white versus printing in color.



Group boxes can, in principle, include graphics marked up using the XHTML `img` element, or vector graphics marked up in SVG etc.

[previous](#) [next](#) [contents](#)

8 Binding

Contents

- [8.1 Introduction](#)
- [8.2 Binding Attributes](#)
- [8.3 Direct Binding](#)
- [8.4 Indirect Binding](#)
- [8.5 Multiple Forms per Page](#)

This chapter is normative.

8.1 Introduction

[Binding](#) is the glue that connects the separate pieces of XForms -- the [XForms Model](#), [instance data](#), and [form controls](#). The [binding](#) is independent of the language used in the [XForms Model](#) (Schema or simple syntax) and also independent of the user interface language used.

[Binding](#) is specified through the use of [binding expressions](#). The syntax and details of [binding expressions](#) are defined in the [Dynamic Constraint Language chapter](#). This chapter describes the wider topic of how [binding expressions](#) are used within XForms.

A future revision of this chapter will address [binding](#) across [XForms Models](#), for instance declaring an "address" in one [XForms Model](#) and referencing it from another.

[Editor's Feedback Request **8.1.binding**: Examples in this chapter heavily lean towards [binding](#) between [form controls](#) and [instance data](#). Future revisions of XForms will include greater representation of [binding](#) between [form controls](#) and the [XForms Model](#). Feedback on this chapter is especially appreciated.]

8.2 Binding Attributes

XForms defines an attribute `ref` that can be placed on any [form control](#). Note that when placed on [form controls](#) outside of XForms, it must be appropriately namespaced. The value of the attribute is a [binding expression](#), based on the XForms Dynamic Constraints Language, that links the [form control](#) to a particular location in the [instance data](#) (and therefore a particular [model item](#)). For example:

XForms User Interface Markup with Binding Attributes

```
<xform:textbox ref="binding-expression">
  <xform:caption>Your first name</xform:caption>
</xform:textbox>
```

The `ref` attribute links the [form control](#) to the [instance data](#) and [XForms Model](#) declared elsewhere in the containing document.

This can also be used on non-XForms [form controls](#), for instance XHTML:

XHTML with Binding Attributes

```
<html:input type="text" name="ncname" xform:ref="binding-expression" />
```

Here the `ref` attribute links an XHTML [form control](#) to the [instance data](#) and [XForms Model](#) contained elsewhere in the containing document. Note that the `html:` prefix is used here to represent the XHTML namespace.

Details on the operation of [binding expressions](#) are given throughout the rest of this chapter.

8.3 Direct Binding

When a containing document has at most a single [XForms Model](#) and at most a single set of [instance data](#), [binding](#) is simpler because there is no possibility of ambiguity as to which [XForms Model](#) and [instance data](#) will participate. The following syntax can be used for the [binding expression](#):

- A [binding expression](#) that evaluates to an element or attribute in the [instance data](#), or to a location in the [XForms Model](#).

For example:

Binding Expression with an XForms Dynamic Constraint

```
<xform:textbox ref="orderForm/shipTo/firstName">  
...
```

Here the `ref` attribute specifies a path through the [XForms Model](#) or [instance data](#) to the desired location.

Instance Data

```
<orderForm>  
  <shipTo>  
    <firstName>value</firstName>  
  </shipTo>  
</orderForm>
```

Here is the matching [instance data](#) for the above example.

A special case applies when [binding](#) to an element in the [instance data](#) that contains an attribute of type `xsd:ID`. In this case, an XPath function `id()`, can be used:

Binding Expression with XPath `id()` Syntax

```
<xform:textbox ref="id('myfirstname')">  
...
```

Here the `ref` attribute specifies a link to an [instance data](#) element with an `id` of `myfirstname`.

Instance Data

```
<a>  
  <b id="myfirstname">value</b>  
</a>
```

Here is the [instance data](#) for the above example.

For this syntax to be valid, the following conditions must be true:

- There must be exactly one instance
- The [instance data](#) must be included in the same document as the [XForms User Interface](#)
- Every referenced element in the [instance data](#) must be marked with a valid ID (i.e. the ID is unique throughout the whole document)
- There must be no Dynamic Constraints used to determine the data element to be referenced. (Note that this does not prevent changing the IDREF through scripting)

Note also that:

- this method is limited, because it requires exactly one instance, decorated with ID attributes.
- It is still legal to have elements without ID attributes in the [instance data](#).
- Only elements can be addressed

8.4 Indirect Binding

In situations where a form is designed by collaboration (such as between a graphic designer and a database/XML specialist) it is desirable to locate all [binding expressions](#) in a single area in the containing document. XForms allows the [binding expression](#) to appear in a separate element `<bind>`, a sibling to `<model>` and `<instance>`.

The attributes of `<bind>` are `id` of type `xsd:ID` and `ref` which takes a [binding expression](#). When a [binding expression](#) is defined this way, the [form control](#) can reference the `id` of the `<bind>` element, as seen here:

Binding Expression Using Indirection

```
<xform:textbox ref="id('myfirstname')">
...
```

Here the `ref` attribute specifies a link to a [binding expression](#) defined elsewhere.

Instance Data

```
<xform:bind id="myfirstname" ref="orderForm/shipTo/firstName"/>
<xform:instance>
  <orderForm>
    <shipTo>
      <firstName>value</firstName>
    </shipTo>
  </orderForm>
</xform:instance>
```

Here is the [instance data](#) for the above example.

This accomplishes the following:

- It is no longer necessary to add IDs to the instance.
- The [binding](#) mechanism is *independent* of the schema and user interface mechanisms.

8.5 Multiple Forms per Page

One design goal of XForms is to support multiple forms per page. This is accomplished by having multiple `<xform>` elements within a containing document. Each `<xform>` element might have a separate [XForms Model](#) or [instance data](#) defined. This makes [binding](#) slightly more complex, because the correct `<xform>` element needs to be referenced.

By default, [binding expressions](#) are based on an [XForms Model](#) or [instance data](#) that are children of the first

<xform> element in document order. To refer to the contents of subsequent <xform> elements, these must be decorated with id attributes of type xsd:ID. Attached to the [form control](#) element, an additional attribute xform serves to adjust the context node for the attached [binding expression](#) to the virtual [instance data](#) or [XForms Model](#) associated with the <xform> element with the matching id.

For example:

Binding Expression Specifying Non-default <xform>

```
<xform:textbox xform="b" ref="/orderForm/shipTo/firstName">  
...
```

Here the xform and ref attributes specify a [binding](#) to the [instance data](#) and [XForms Model](#).

Instance Data

```
<xform:xform id="b">  
  <xform:model>  
    ...  
  </xform:model>  
  <xform:instance>  
    <orderForm xmlns="...">  
      <shipTo>  
        <firstName>value</firstName>  
      </shipTo>  
    </orderForm>  
  </xform:instance>  
</xform:xform>
```

Here is the markup for the above example.

Scoped resolution of [binding expressions](#), as defined in the [Dynamic Constraint Language chapter](#) can in some cases be used to avoid repetitive uses of the xform attribute.

Acknowledgments: The editor would like to thank Kai Scheppe, Malte Wedel and Götz Bock for lots of constructive criticism on early versions of this document and their contributions to its present content.

[previous](#) [next](#) [contents](#)

9 Using XForms with Other Document Types

Contents

- [9.1 The XForms Namespace](#)
- [9.2 The <xform> Element](#)

This chapter is normative.

XForms have been designed for use within other XML vocabularies, in particular XHTML. This chapter discusses some of the high-level features of XForms that allow it to be used with other document types.

9.1 The XForms Namespace

The XForms namespace has the URI: `http://www.w3.org/2001/02/xforms`. Future revisions are expected to use a different identifier. This document uses the convention of an `xform:` prefix to represent elements and attributes that are part of the XForms Namespace.

[XForms Processors](#) must use the XML namespaces mechanism [\[XML Names\]](#) to recognize elements and attributes from this namespace.

9.2 The <xform> Element

The `<xform>` element is used as a container for other XForms elements, and can serve as the root element of a standalone document or be embedded in other document types such as XHTML. A single containing document may contain any number of `<xform>` elements.

XML Representation : `<xform>`

```
<xform
  xmlns = namespace-identifier
  id = xsd:ID
>
  <!-- Content: (submit | bind)*, model, instance -->
</xform>
```

Attribute Definitions:

xmlns = *namespace-identifier*

Optional standard XML attribute for identifying an XML namespace. It is frequently useful to include this standard attribute at this point.

id = `xsd:ID`

Optional unique identifier used to refer to this particular `<xform>` element.

For example:

```
<xform xmlns="http://www.w3.org/2001/02/xforms" id="Person">
  <model href="Schema-Questionnaire.xform" />
  <instance href="URL-to-retrieve-defaults" />
  ...
</xform>
```

9.2.1 The `<model>` Element

The `<model>` element is used to define the [XForms Model](#). The content of the [XForms Model](#) may be defined inline or obtained from an external URI.

Editor's Note: The Working Group is actively investigating the potential of using XLink simple links within XForms.

XML Representation : `<model>`

```
<model
  id = xsd:ID
  href = xsd:uriReference
>
  <!-- Content: (xsd:schema | simple) -->
</model>
```

Attribute Definitions:

id = `xsd:ID`

Optional unique identifier.

href = `xsd:uriReference`

Optional link to an externally defined [XForms Model](#).

The content of the `<model>` element is restricted to either XML Schema content, or XForms simple syntax. The content of `<simple>` is defined in the [XForms Model chapter](#).

Editor's Note: We still need to define behavior if both an inline [XForms Model](#) and an external [XForms Model](#) are used together.

9.2.2 The <instance> Element

The <instance> element is used to define initial [instance data](#). The [instance data](#) may be defined inline or obtained from an external URI.

<p>XML Representation : <instance></p> <pre><instance id = xsd:ID href = xsd:uriReference > <!-- Content: (##other) --> </instance></pre>
--

Attribute Definitions:

id = xsd:ID

Optional unique identifier.

href = xsd:uriReference

Optional link to externally defined [instance data](#).

The content of the <instance> element is arbitrary XML in any namespace other than the XForms namespace. Authors must ensure that proper namespace declarations are used for content within the <instance> element.

[Editor's Feedback Request **9.2.2.schemalocation**: Should a `schemaLocation` attribute, linking the [instance data](#) with a schema definition, be present here?]

9.2.3 SubmitInfo

The <submitInfo> element provides information on how and where to submit the [instance data](#).

<p>XML Representation : <submit></p> <pre><submitInfo id = xsd:ID target = xsd:uriReference method = xsd:string > <!-- Content: (##empty) --> </submitInfo></pre>
--

Attribute Definitions:

id = xsd:ID

Optional unique identifier.

href = xsd:uriReference

Required destination for submitted [instance data](#).

method = xsd:string

Optional indicator to provide details on the submit protocol. With HTTP, the default is "POST".

9.2.4 Bind

The <bind> element represents a connection between the different parts of XForms.

XML Representation : <bind>
<pre><bind id = xsd:ID ref = XForms binding expression > <!-- Content: (##empty) --> </bind></pre>

Attribute Definitions:

id = xsd:ID

Required unique identifier.

ref = XForms [binding expression](#)

A link to an externally defined [XForms Model](#).

Additional details are found in the [Binding chapter](#).

[previous](#) [next](#) [contents](#)

10 Processing Model and Conformance

Contents

- [10.1 Introduction](#)
- [10.2 XForms Properties](#)
- [10.3 Events](#)
- [10.4 Virtual Instance Data](#)
- [10.5 XForms Processing](#)
- [10.6 Submit, Suspend and Reset](#)
- [10.7 Conformance](#)

This chapter is normative.

10.1 Introduction

The XForms Reference Processing Model is a normative explanation of the components, predictive behavior, and mechanisms of [XForms Processors](#). It is not intended to constrain implementations. [XForms Processor](#)s may be implemented in any manner, so long as the end results are identical to that described in this chapter.

This chapter uses the terms **may**, **must**, and **should** (when rendered as in this paragraph) in accord with RFC 2119.

[Editor's Feedback Request **10.1.processing**: This chapter is still at an early phase and may contain errors or omissions. Feedback on this chapter is especially appreciated.]

Design Rationale

The Reference Processing Model set out in this chapter will:

- Be simple enough to implement across a wide range of devices, including resource-constrained handhelds and appliances.
- Define a predictive processing model with enough detail for implementors to create interoperable software.
- Define a well-ordered system for calculations and dependencies independent of processor speed or threading.
- Provide a unified addressing scheme for [binding expressions](#), independent of how the structure of the [instance data](#) is defined.

- Be simple enough for the existing base of HTML authors to quickly get up to speed.
- Be compatible (to the extent reasonably possible) with existing form processing.
- Support hand author cut/paste operations.

10.2 XForms Properties

For each `<xform>` element, the [XForms Processor](#) maintains a set of properties, as described here.

- `version`
- `conformance-level`
- `locale`
- `timezone`
- `immediate-refresh`
- `immediate-revalidate`
- `immediate-recalculate`
- `use-nulls`

`version` (readonly) is defined as the string "1.0" for XForms 1.0

`conformance-level` (readonly) strings are TBD

`locale` (readonly) strings are TBD, as is the process for determining the locale

`timezone` (readonly) strings are signed integers representing the number of minutes offset from GMT

`immediate-refresh` (read-write) controls whether changes in the [instance data](#) are immediately updated in the UI

`immediate-revalidate` (read-write) controls whether changes in the [instance data](#) immediately trigger a validation

`immediate-recalculate` (read-write) controls whether changes in the [instance data](#) immediately trigger a recalculation

`use-nulls` (read-write) controls whether XML Schema Instance nulls are placed in the [instance data](#)

The Working Group is currently discussing the syntax for accessing these properties from the Dynamic Constraints Language.

10.3 Events

XForms uses an events system as defined in [\[DOM2 Events\]](#), with a Capture phase, arrival at the Event Target, and then a Bubbling Phase.

Events fall into different groupings. One class of events indicates that some processing is about to happen. That processing may be halted by the event handler:

- `xforms-submit`

- `xforms-reset`
- `xforms-value-changing`
- `xforms-interactive-value-changing`
- `xforms-instance-changed`

Another class of events indicates that some processing has already happened or is in progress. Such processing can not be halted by the event handler:

- `xforms-construct`
- `xforms-destruct`
- `xforms-suspend`
- `xforms-resume`
- `xforms-exception`

Finally, certain events are used by the author or the [XForms Processor](#) to cause processing to happen:

- `xforms-recalculate`
- `xforms-refresh`

Unless otherwise noted, the target node for all events is the `<xform>` element. When a containing document has multiple `<xform>` elements, the [binding](#) is used to determine which `<xform>` element is used.

The Working Group is considering syntax proposals for event handlers, mainly based on [\[XHTML Events\]](#).

10.4 Virtual Instance Data

For each containing document, an [XForms Processor](#) must behave as if it internally maintains a set of [instance data](#), called virtual [instance data](#). In this document, this is represented as a virtual `<instance>` element, which contains and maintains all [instance data](#) associated with an `<xform>` element.

It is not allowed for elements in the virtual [instance data](#) to be in the XForms namespace (for example if the XML data to be submitted itself consisted of XForms elements).

Representation of virtual instance data

```
<instance>
  ... instance data goes here ...
</instance>
```

This is an example of virtual [instance data](#) as used by the XForms Reference Processing Model.

[Editor's Feedback Request **10.4.dom**: Should virtual [instance data](#) be mapped to the DOM of the containing document, or should it exist in a separate document space?]

[Editor's Feedback Request **10.4.access**: From the DOM, should virtual [instance data](#) be read-only or

read-write? What are possible security implications here?]

Additionally, for purposes of the Reference Processing Model, virtual [instance data](#) keeps track of which [instance data items](#) (whether elements, attributes, complex types, etc.) are "dirty", or in need of update.

10.5 XForms Processing

10.5.1 Initialization/Resume

The following describes the initialization process for XForms. Initialization must occur before any other processing. For each `<xform>` element in the containing document, in document order, the following processing occurs:

1. An `xforms-construct` event is fired; this is the place for authors to handle any initialization tasks.
2. If the `<model>` element contains a reference to a non-local [XForms Model](#), it is retrieved by traversing the link to it. An [XForms Model](#) that is unretrievable for any reason **must** be regarded as a fatal error, preventing the form from being filled.
3. If the `<instance>` element contains a reference to non-local [instance data](#), it is retrieved by traversing the link to it. [Instance data](#) that is unretrievable for any reason is ignored, in which case an [XForms Processor](#) **may** issue a warning.
4. If the `<xform>` element currently being processed 1) contains no `<instance>` child, and 2) contains no `<model>` child, the following processing occurs:
 1. Each [form control](#) bound to the `<xform>` element currently being processed is visited in document order. Each [form control](#)'s [binding expression](#) is evaluated.
 2. If the [instance data item](#) result of evaluating the [binding expression](#) doesn't already exist, it is created, and if the `use-nulls` property is true, populated with a null value (in whatever representation Schema finalizes upon). Note that the way Schema specifies things, only elements can hold null values. The [form control](#) receives a default blank value. The algorithm for creating [instance data items](#) is as follows: For each location step in the canonical [binding expression](#), left to right, where no matching node exists in the virtual [instance data](#), a new node is inserted.
5. An `xforms-resume` event is fired.
6. A [recalculation](#) takes place.
7. A UI [refresh](#) takes place.

10.5.2 Navigation Sequence Algorithm

Navigation is determined on a document-wide basis. The navigation sequence is determined as follows:

1. Those [form controls](#) that support `navindex` and assign a positive value to it are navigated first. Navigation proceeds from the [form control](#) with the lowest `navindex` value to the [form control](#) with the highest value. Values need not be sequential nor must they begin with any particular

value. [Form controls](#) that have identical `navindex` values should be navigated in document order.

2. Those [form controls](#) that do not supply `navindex` or supply a value of "0" are navigated next. These [form controls](#) are navigated in document order.
3. Those [form controls](#) that are disabled, hidden, or on a non relevant subtree are assigned a relative order in the overall sequence but do not participate as navigable controls.
4. The navigation sequence past the the last [form control](#) (or before the first) is undefined. [XForms Processors](#) may cycle back to the first/last control, remove focus from the form, or other possibilities.

10.5.3 Interactivity

XForms provides similar processing to the HTML `onChange` event. As users indicate completion of a [form control](#) by navigating away the following occurs:

1. If the display value has changed since the user last navigated to the [form control](#), an `xforms-value-changing` event is fired. If the display value hasn't changed, processing for this event ends.
 1. Any listener may prevent default processing (one option under consideration provides a `preventDefault()` method), which will end event processing immediately after the Capture and Bubbling phases. Alternatively, a listener may perform a custom translation from display value to canonical value. Any listener may have side-effects that modify any [instance data item](#), in which case the modified parts must be marked "dirty".
 2. Default processing is to convert the display value of the [form control](#) to the canonical value as specified in the Datatypes chapter. Default processing should automatically take into account regional settings (if any), such as decimal character symbol, date formats, etc.
2. If the `immediate-revalidate` property is true, all [validations](#) bound to the [form control](#) are run. Note that validation is performed against the canonical value, not the display value.
 1. If any validation fails, the user **must** be notified, and **may** not be allowed to navigate away from the control. The invalid entry in the [form control](#) **should** be preserved. The associated [instance data item](#) is left unchanged, thereby ending processing for this event.
3. The [instance data item](#) is updated with the new value, and marked "dirty".
4. If the `immediate-recalculate` property is true, a [recalculate](#) occurs to perform any defined calculations.
5. If the `immediate-refresh` property is true, a [refresh](#) occurs to update any [form controls](#) that might be dependent on this newly changed value.

Certain [form controls](#) allow interactive response without finalizing on a value. Examples of this include edit boxes (users can type various characters before "tabbing out") and slider controls (users can be continuously adjusting the value before releasing at a certain value). Interactive temporary values such as this are expressly allowed to be "invalid", that is outside the permissible [value space](#). This is because incomplete data may be present while the user is entering transitional values.

Example: A partially entered currency value of "U" is not valid because it doesn't (yet) have 3 characters. This is permitted temporarily, as long as the user remains on the [form control](#). [XForms](#)

[Processors](#) with sufficient processing resources would typically update/refresh on every character. Resource-constrained [XForms Processors](#) would typically only update/refresh on the final value.

1. Any time the display value of a [form control](#) changes (such as through character or cut/paste activities), even without indication that this is a final value, an `xforms-interactive-value-changing` event is fired. Resource-constrained [XForms Processor](#) implementations **may** choose to ignore all `xforms-interactive-value-changing` events.
 1. Event listeners may prevent default processing.
 2. Otherwise, default handling is as follows: The current [form control](#) is [revalidated](#). This is for internal purposes only, and happens regardless of the `immediate-revalidate` setting. If all validations on the [form control](#) are successful, the [instance data item](#) is updated, and marked "dirty". If any validations fail (indicating a transitional value) all [form controls](#) bound to the same [instance data item](#) **may** be directly updated with the display value. Otherwise, the following occurs:
 3. If the `immediate-recalculate` property is true, a [recalculation](#) occurs to perform any defined calculations.
 4. If the `immediate-refresh` property is true, a [refresh](#) occurs to update any [form controls](#) that might be dependent on this newly changed value.

Typical implementations that choose to respond `xforms-interactive-value-changing` are expected to optimize processing (for instance not flashing the entire screen for each character entered, etc.).

10.5.4 Recalculate Algorithm

[XForms Processors](#) are free (and encouraged) to skip or change any steps in this algorithm, as long as the end result is the same. Each [form control](#) may have a [model item](#) `priority` value, which is the main factor in determining calculation order.

Following is the default handling for an `xforms-recalculate` event:

1. Each [model item](#) with a bound `calculate` [model item](#) property is visited in calculation order, which is defined as follows:
 1. Those [model items](#) that are bound to a `priority` and assign a positive integer to it are computed first. Computation proceeds from the [model item](#) with the lowest bound `priority` to the [model item](#) with the highest bound `priority`. Values need not be sequential nor must they begin with any particular value. [Model items](#) with the same bound `priority` value are computed in document order.
 2. Those [model items](#) not bound to a `priority` or bound to one with the value "0" are computed next. These [model items](#) are computed in document order.
2. For each [model item](#), the expression in the `calculate` [model item](#) property is evaluated. Any [instance data item](#) changes as a result of this are marked with a "dirty" flag.
3. The [instance data item](#) bound to the [model item](#) is updated with the result of the `calculate` expression, and the "dirty" flag is set.

10.5.5 UI Refresh Algorithm

Following is the default handling for an `xforms-refresh` event:

1. For purposes of UI refresh, the virtual [instance data](#) as it exists at the beginning of processing the `xforms-refresh` event is used.
2. Each [form control](#) is visited in refresh order, which is defined as follows:
 1. Those [form controls](#) that have a given or computed navigation sequence value are visited first, in the navigation sequence.
 2. Those [form controls](#) outside the navigation sequence are visited next. These [form controls](#) are visited in document order.
3. For each [form control](#), the relevant constraint is evaluated, which might result in the [form control](#) being disabled/hidden/etc. as specified in the [XForms Model chapter](#).
4. For each [form control](#), the [binding expression](#) is evaluated. If the virtual [instance data](#) indicates that the [instance data item](#) is not "dirty", processing for that particular [form control](#) completes.
 1. Otherwise, if the [instance data item](#) is "dirty", an `xforms-instance-changed` event is fired.
 2. Listeners to the `xforms-instance-changed` event are free to compute a new display value.
 3. Listeners to the `xforms-instance-changed` event are prohibited from directly updating any [form controls](#) present.
 4. Listeners to the `xforms-instance-changed` event are prohibited from altering any portion of the virtual innate data. To attempt to do so results in an `xforms-exception` being fired.
 5. Listeners may prevent the default processing of the `xforms-instance-changed` event.
 6. Default processing is to convert the canonical value into a display value, taking into account regional settings (if any) such as decimal separator character, etc.
5. The [form control](#) is updated with the display value.
6. After all [form controls](#) have been updated, all "dirty" flags in the virtual [instance data](#) are cleared.

Editor's Note: Still to be addressed is the processing when a datatype [facet](#) or [model item](#) property are changed--what gets marked "dirty"?; what gets recalculated?; what gets revalidated?; what gets refreshed?

10.5.6 Revalidate Algorithm

Revalidation always occurs within the scope of a context [form control](#). Following is the [revalidation](#) process:

1. The bound [instance data item](#) is checked against any bound XForms Datatype constraining [facets](#). If any fail, the context [form control](#) is considered invalid.
2. The bound [instance data item](#) is checked against any bound Schema Datatype constraining

- [facets](#). If any fail, the context [form control](#) is considered invalid.
3. If a `validate` [model item](#) property is bound to the context [form control](#), the expression within is evaluated. If it evaluates to false, the context [form control](#) is considered invalid.
 4. If the context [form control](#) is invalid, the [XForms Processor](#) **must** notify the user. The [XForms Processor](#) **may** combine messages before presentation to the user.

10.6 Submit, Suspend and Reset

The form filling experience ends with submitting the form, saving it for later, or starting over. The XForms processing for these events are covered here.

The following sections describe how to [instance data](#) is prepared for submission.

10.6.1 Submit

In response to an `xforms-submit` event, the following takes place:

1. Event listeners may prevent default processing of the submit request. Otherwise, default handling as described below occurs.
2. Every [form control](#) is [revalidated](#). Any invalid values **must** be reported to the user and submit processing **must** not continue.
3. A subset or all of the virtual [instance data](#) is selected based on the [binding expression](#) used to invoke the submit request.
 1. If the [instance data](#) selection results in an empty node-set, the submit **must** be aborted and submit processing **must** not continue.
4. [Instance data](#) is packaged according to one of the processes defined below.
5. [Instance data](#) is delivered over the network using the [XForms Submit Protocol](#).

10.6.2 Suspend

In response to an `xforms-suspend` event, the following takes place:

1. Event listeners may prevent default processing of the suspend request. Otherwise, default handling as described below occurs.
2. No [form control revalidation](#) occurs.
3. A subset or all of the virtual [instance data](#) is selected based on the [binding expression](#) used to invoke the suspend request.
 1. If the [instance data](#) selection results in an empty node-set, the suspend **must** be aborted and suspend processing **must** not continue.
4. [Instance data](#) is packaged using one of the persistence formats described below, possibly including a TBD flag that says this isn't final data.
5. [Instance data](#) is persisted either locally or remotely in a manner that is TBD.

10.6.3 Reset

In response to an `xforms-reset` event, the following takes place:

1. Event listeners may prevent default processing of the reset request. Otherwise, default handling as described below occurs.
2. A subset or all of the virtual [instance data](#) is selected based on the [binding expression](#) used to invoke the suspend request.
 1. If the [instance data](#) selection results in an empty node-set, the reset has no effect.
3. New [instance data](#) for the selected [instance data](#) is prepared according to the rules for initialization above.
4. The selected [instance data](#) is replaced with the new [instance data](#).

10.6.4 application/x-www-form-urlencoded

This format is intended to facilitate the integration of XForms into HTML forms processing environments, and represents an extension of the [\[XHTML 1.0\]](#) form content type of the same name that expresses the hierarchical nature of [instance data](#).

This format is not suitable for the persistence of binary content. Therefore, it is recommended that XForms capable of containing binary content use either the [multipart/form-data](#) or [text/xml](#) formats.

[Editor's Feedback Request **10.6.4.urlencoding**: The urlencoding technique given here does not exactly match how legacy implementations produce urlencoded data. (In particular, we are adding contextual information with slashes and multiple location-steps) Will this approach interfere with legacy urlencoding parsers?]

[Editor's Feedback Request **10.6.4.utf8**: Under discussion is the intent to have the data be UTF8 encoded; however, this is dependent upon IETF developments. Would UTF8 meet the needs of the forms community?]

The steps for building this persistence format is as follows:

1. Prepare a new UTF-8 encoded string buffer to hold the persisted [instance data](#).
2. Beginning with the root element of the virtual [instance data](#), iterate over the content of the instance in document order and build an ordered set of strings by performing the following steps:
 1. For each element with an attribute, append to the set a string of the format "*path=value*" where *path* is the canonical [binding expression](#) that refers to each attribute, and *value* is the character content of each attribute (urlencoded if necessary).
 2. For each element enclosing character content, append to the set a string of the format "*path=value*" where *path* is the canonical [binding expression](#) that refers to the element, and *value* is the character content of the element (urlencoded if necessary)
 3. For each element enclosing element content, continue the iteration.
3. Append the strings from the ordered set together, delimiting the strings with an ampersand '&' character, and place the result of the append into the UTF-8 encoded string buffer.

Example:

application/x-www-form-urlencoded

```
/PersonName/@title=Mr&/PersonName/FirstName=Roland
```

This format consists of sets of a canonical [binding expression](#) paired with a value.

Corresponding Instance Data

```
<PersonName title="Mr">  
  <FirstName>Roland</FirstName>  
</PersonName>
```

Here is the [instance data](#) for the above example.

10.6.5 multipart/form-data

This format is intended to facilitate the integration of XForms into HTML forms processing environments, and represents an extension of the [\[XHTML 1.0\]](#) form content type of the same name that expresses the hierarchical nature of [instance data](#). Unlike the [application/x-www-form-urlencoded](#) format, this format is suitable for the persistence of binary content.

This format follows the rules of all multipart MIME data streams as outlined in [\[RFC 2045\]](#). Each part is expected to contain:

1. A "Content-Disposition" header whose value is "form-data".
2. A name attribute specifying the canonical [binding expression](#) of the corresponding value from the [instance data](#). Names originally encoded in non-ASCII character encoding schemes may be encoded using the method outlined in [\[RFC 2045\]](#).

Example:

multipart/form-data

```
Content-Type: multipart/form-data; boundary=AaB03x  
--AaB03x  
  Content-Disposition: form-data; name="/PersonName/@title"  
  
Mr  
--AaB03x  
  Content-Disposition: form-data; name="/PersonName/FirstName"  
  
Roland  
--AaB03x  
  
...Possibly more data...  
  
--AaB03x-
```

This format consists of sets of a canonical [binding expression](#) paired with a value.

Corresponding Instance Data

```
<PersonName title="Mr">  
  <FirstName>Roland</FirstName>  
</PersonName>
```

Here is the [instance data](#) for the above example.

Binary Content

Each part may be encoded and the "Content-Transfer-Encoding" header supplied if the value of that part does not conform to the default (7 bit) encoding.

Where a value within the [instance data](#) represents binary content, the value should be identified by the appropriate content type (e.g., "application/octet-stream"). If multiple values of binary content are to be returned as the result of a single [model item](#), they should be returned as "multipart/mixed" embedded within the "multipart/form-data".

The [XForms Processor](#) may wish to supply a file name for each value of binary content. The file name may be specified with the "filename" parameter of the 'Content-Disposition: form-data' header, or in the case of multiple values of binary content, in a 'Content-Disposition: file' header of the subpart. If the file name of the client's operating system is not in US-ASCII, the file name might be approximated or encoded using the method of [RFC 2045](#). This is convenient for those cases where, for example, the uploaded files might contain references to each other (e.g., a TeX file and its ".sty" auxiliary style description).

10.6.6 text/xml

This format permits the expression of the [instance data](#) as an XML-based format that is straightforward to process with off-the-shelf XML processing tools. In addition, this format is suitable for the persistence of binary content.

The steps for building this persistence format is as follows:

1. Prepare a new empty XML document to hold the persisted [instance data](#).
2. Serialize, into the XML document the entire content of the virtual [instance data](#) `<instance>` node.

Binary Content

Handling of binary content will likely be based on the ongoing work in the XML Protocol Working Group.

[Editor's Feedback Request **10.6.5.metadata**: Where a value within the [instance data](#) represents binary content, can we store meta-information with an `xform:mediaType` attribute reflecting the appropriate content type (e.g., "image/jpg")?]

10.7 Conformance

XForms have been designed for use among a wide variety of [XForms Processors](#), of varying size and resource constraints. Because of this, multiple conformance levels are being discussed. This chapter will be updated in the future with more details.

[previous](#) [next](#) [contents](#)

Appendix A: Schema for XForms Model

Contents

- [A.1 XForms Model Schema](#)

A.1 XForms Model Schema

```
<?xml version="1.0"?>
<xsd:schema targetNamespace="http://www.w3.org/2001/02/xforms"
xmlns:xform="http://www.w3.org/2001/02/xforms"
xmlns="http://www.w3.org/2001/02/xforms"
xmlns:xsd="http://www.w3.org/2000/10/XMLSchema" elementFormDefault="qualified">
  <!--
  Import the schema for XMLSchema, to use the
  <xsd:schema> element.
  -->
  <xsd:import namespace="http://www.w3.org/2000/10/XMLSchema"/>
  <!--

  Defines the schema for the XForm <model> element, and everything
  contained in it. This includes the XForms defined data types, and XForms
  specific properties whose values are dynamic, and may may change at
  runtime. It also contains a definition of the XForms Simple Syntax
  elements, which can be used as an alternative to XML syntax, for
  defining the XForms model's elements.

  -->
  <!--

      XForms structure element definitions.

  -->
  <!--

  Definition of the xform container element.

  -->
  <xsd:element name="xform">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element ref="model"/>
        <xsd:element ref="instance"/>
        <xsd:element ref="submit"/>
        <xsd:element ref="bind"/>
      </xsd:choice>
      <xsd:attribute name="id" type="xsd:ID" use="optional"/>
    </xsd:complexType>
  </xsd:element>
  <!--
```

Definition of top-level model element.

```
-->
<xsd:element name="model">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element ref="xsd:schema"/>
      <xsd:element ref="simple"/>
      <xsd:sequence>
        <xsd:element ref="xsd:schema"/>
        <xsd:element ref="simple"/>
      </xsd:sequence>
    </xsd:choice>
    <xsd:attribute name="id" type="xsd:ID" use="optional"/>
    <xsd:attribute name="name" type="xsd:NCName" use="optional"/>
    <xsd:attribute name="href" type="xsd:uriReference" use="optional"/>
  </xsd:complexType>
</xsd:element>
<!--
```

Definition of top-level instance element.

```
-->
<xsd:element name="instance">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:any namespace="##other" processContents="skip" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID" use="optional"/>
    <xsd:attribute name="model" type="xsd:IDREF" use="optional"/>
    <xsd:attribute name="href" type="xsd:uriReference" use="optional"/>
  </xsd:complexType>
</xsd:element>
<!--
```

Definition of submitExtension element.

Elements contained in SubmitExtension must be namespace qualified.
If the namespace is not understood and the mustUnderstand attribute is set to true, then the contents of the Submit should not be used.

```
-->
<!-- -->
<xsd:element name="submitExtension">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:any namespace="##other" processContents="skip"/>
    </xsd:sequence>
    <xsd:attribute name="mustUnderstand" type="xsd:boolean" use="optional"
value="true"/>
  </xsd:complexType>
</xsd:element>
<!--
```

Definition of top-level submit element.

```
-->
<xsd:element name="submit">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="submitExtension" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<!--
```

```

    <!-- allow zero or more submitExtension elements -->
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID" use="optional"/>
  <xsd:attribute name="target" type="xsd:uriReference" use="required"/>
  <xsd:attribute name="method" type="xsd:string" use="optional"/>
</xsd:complexType>
</xsd:element>
<!--

```

Definition of top-level bind element.

```

-->
<xsd:element name="bind">
  <xsd:complexType>
    <xsd:attribute name="id" type="xsd:ID" use="optional"/>
    <xsd:attribute name="ref" type="xsd:string" use="optional"/>
  </xsd:complexType>
</xsd:element>
<!--

```

Definition of XForms expression type.

Note: we need to replace this with a real XForm expression type, if we can manage to define one that fits our expression language, if there is a way to do that using XML Schema (may be possible with the correct combination of string & patterns).

```

-->
<xsd:simpleType name="xfmExpr">
  <xsd:restriction base="xsd:string"/>
</xsd:simpleType>
<!--

```

XForms types used in attribute group definitions.

Note: all of these are simple derivations from xfmExpr. we need to change them to enforce the desired typed result, if there is a way to do that using XML schema.

```

-->
<!--

```

Definition of stringExpr type, used for expressions that must return a String value.

```

-->
<xsd:simpleType name="stringExpr">
  <xsd:union memberTypes="xsd:string xfmExpr"/>
</xsd:simpleType>
<!--

```

Definition of boolExpr type, used for expressions that must return a Boolean value.

```

-->
<xsd:simpleType name="boolExpr">
  <xsd:union memberTypes="xsd:boolean xfmExpr"/>
</xsd:simpleType>
<!--

```

Definition of numberExpr type, used for expressions that must

return a Number value.

```
-->  
<xsd:simpleType name="numberExpr">  
  <xsd:union memberTypes="xsd:decimal xfmExpr"/>  
</xsd:simpleType>  
<!--
```

Definition of positiveIntExpr type, used for expressions that must return a positive integer value.

```
-->  
<xsd:simpleType name="positiveIntExpr">  
  <xsd:union memberTypes="xsd:positiveInteger xfmExpr"/>  
</xsd:simpleType>  
<!--
```

Definition of dateExpr type, used for expressions that must return a Date value.

```
-->  
<xsd:simpleType name="dateExpr">  
  <xsd:union memberTypes="xsd:date xfmExpr"/>  
</xsd:simpleType>  
<!--
```

Definition of timeExpr type, used for expressions that must return a Time value.

```
-->  
<xsd:simpleType name="timeExpr">  
  <xsd:union memberTypes="xsd:time xfmExpr"/>  
</xsd:simpleType>  
<!--
```

Definition of durationExpr type, used for expressions that must return a Duration value.

```
-->  
<xsd:simpleType name="durationExpr">  
  <xsd:union memberTypes="xsd:timeDuration xfmExpr"/>  
</xsd:simpleType>  
<!--
```

Definition of schemeExpr type, used for expressions that must return a list of zero or more scheme values. Scheme is used to restrict the value space of URIs to specific URI schemes.

```
-->  
<xsd:simpleType name="schemeType">  
  <xsd:restriction base="xsd:string">  
    <xsd:pattern value="[a-z]+"/>  
  </xsd:restriction>  
</xsd:simpleType>  
<xsd:simpleType name="schemeExpr">  
  <xsd:union memberTypes="schemeType xfmExpr"/>  
</xsd:simpleType>  
<!--
```

Definition of uriExpr type, used for expressions that must return a URI value.

```
-->
<xsd:simpleType name="uriExpr">
  <xsd:union memberTypes="xsd:uriReference xfmExpr"/>
</xsd:simpleType>
<!--
```

Definition of mediaTypeExpr type, used for expressions that must return a list of zero or more MediaType values. MediaType is used to restrict the value space of Binary elements to one or more MIME media Types.

```
-->
<xsd:simpleType name="mediaTypeType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[a-z]+/[a-z]+"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="mediaTypeExpr">
  <xsd:union memberTypes="mediaTypeType xfmExpr"/>
</xsd:simpleType>
<!--
```

Definition of binaryExpr type, used for expressions that must return a Binary value.

```
-->
<xsd:simpleType name="binaryExpr">
  <xsd:union memberTypes="xsd:binary xfmExpr"/>
</xsd:simpleType>
<!--
```

Definition of currencyType type, a type used for a single, 3-character currency code.

```
-->
<xsd:simpleType name="currencyType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Z]{3}"/>
  </xsd:restriction>
</xsd:simpleType>
<!--
```

Definition of currencyExpr type, used for expressions that must return a list of zero or more 3-character currency codes.

```
-->
<xsd:simpleType name="currencyExpr">
  <xsd:union memberTypes="currencyType xfmExpr"/>
</xsd:simpleType>
<!--
XForms Specific Properties attribute groups
-->
<xsd:attributeGroup name="XFSPcommonMinusEnum">
  <xsd:attribute name="id" type="xsd:ID" use="optional"/>
  <xsd:attribute name="required" type="boolExpr" use="default" value="false"/>
</xsd:attributeGroup>
```

```

    <xsd:attribute name="readOnly" type="boolExpr" use="default" value="false"/>
    <xsd:attribute name="relevant" type="boolExpr" use="default" value="true"/>
    <xsd:attribute name="validate" type="boolExpr" use="default" value="true"/>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPcommon">
  <xsd:attributeGroup ref="XFSPcommonMinusEnum"/>
  <xsd:attribute name="enum" use="optional">
    <xsd:simpleType>
      <xsd:restriction base="xsd:NMTOKEN">
        <xsd:enumeration value="open"/>
        <xsd:enumeration value="closed"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPname">
  <xsd:attribute name="name" type="xsd:NCName" use="required"/>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPcommonMinusEnumSimple">
  <xsd:attributeGroup ref="XFSPname"/>
  <xsd:attributeGroup ref="XFSPcommonMinusEnum"/>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPcommonSimple">
  <xsd:attributeGroup ref="XFSPname"/>
  <xsd:attributeGroup ref="XFSPcommon"/>
</xsd:attributeGroup>
<!--

```

Attribute group defining @calculate and @choices. choices is an expression that returns a list of enumerated, properly typed values at runtime. calculate returns a single value of the correct type, at runtime.

```

-->
<xsd:attributeGroup name="XFSPcalcAndChoices">
  <xsd:attribute name="calculate" type="xfmExpr" use="optional"/>
  <xsd:attribute name="choices" type="xfmExpr" use="optional"/>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPcalcAndChoicesString">
  <xsd:attribute name="calculate" type="stringExpr" use="optional"/>
  <xsd:attribute name="choices" type="stringExpr" use="optional"/>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPcalcAndChoicesBoolean">
  <xsd:attribute name="calculate" type="boolExpr" use="optional"/>
  <xsd:attribute name="choices" type="boolExpr" use="optional"/>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPcalcAndChoicesNumber">
  <xsd:attribute name="calculate" type="numberExpr" use="optional"/>
  <xsd:attribute name="choices" type="numberExpr" use="optional"/>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPcalcAndChoicesDate">
  <xsd:attribute name="calculate" type="dateExpr" use="optional"/>
  <xsd:attribute name="choices" type="dateExpr" use="optional"/>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPcalcAndChoicesTime">
  <xsd:attribute name="calculate" type="timeExpr" use="optional"/>
  <xsd:attribute name="choices" type="timeExpr" use="optional"/>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPcalcAndChoicesDuration">

```

```

    <xsd:attribute name="calculate" type="durationExpr" use="optional"/>
    <xsd:attribute name="choices" type="durationExpr" use="optional"/>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPcalcAndChoicesURI">
    <xsd:attribute name="calculate" type="uriExpr" use="optional"/>
    <xsd:attribute name="choices" type="uriExpr" use="optional"/>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPcalcAndChoicesBinary">
    <xsd:attribute name="calculate" type="binaryExpr" use="optional"/>
    <xsd:attribute name="choices" type="binaryExpr" use="optional"/>
</xsd:attributeGroup>
<xsd:attributeGroup name="XFSPcalcAndChoicesCurrency">
    <xsd:attribute name="calculate" type="currencyExpr" use="optional"/>
    <xsd:attribute name="choices" type="currencyExpr" use="optional"/>
</xsd:attributeGroup>
<!-- applied to Number, and Money -->
<xsd:attributeGroup name="XFSPmmNumber">
    <xsd:attribute name="minInclusive" type="numberExpr" use="optional"/>
    <xsd:attribute name="minExclusive" type="numberExpr" use="optional"/>
    <xsd:attribute name="maxInclusive" type="numberExpr" use="optional"/>
    <xsd:attribute name="maxExclusive" type="numberExpr" use="optional"/>
</xsd:attributeGroup>
<!--
-->
<xsd:attributeGroup name="XFSPmmNumberSimple">
    <xsd:attribute name="min" type="numberExpr" use="optional"/>
    <xsd:attribute name="max" type="numberExpr" use="optional"/>
</xsd:attributeGroup>
<!--
-->
<!-- applied to Date -->
<xsd:attributeGroup name="XFSPmmDate">
    <xsd:attribute name="minInclusive" type="dateExpr" use="optional"/>
    <xsd:attribute name="minExclusive" type="dateExpr" use="optional"/>
    <xsd:attribute name="maxInclusive" type="dateExpr" use="optional"/>
    <xsd:attribute name="maxExclusive" type="dateExpr" use="optional"/>
</xsd:attributeGroup>
<!--
-->
<xsd:attributeGroup name="XFSPmmDateSimple">
    <xsd:attribute name="min" type="dateExpr" use="optional"/>
    <xsd:attribute name="max" type="dateExpr" use="optional"/>
</xsd:attributeGroup>
<!--
-->
<!-- applied to Duration -->
<xsd:attributeGroup name="XFSPmmDuration">
    <xsd:attribute name="minInclusive" type="durationExpr" use="optional"/>
    <xsd:attribute name="minExclusive" type="durationExpr" use="optional"/>
    <xsd:attribute name="maxInclusive" type="durationExpr" use="optional"/>
    <xsd:attribute name="maxExclusive" type="durationExpr" use="optional"/>
</xsd:attributeGroup>
<!--
-->
<xsd:attributeGroup name="XFSPmmDurationSimple">
    <xsd:attribute name="min" type="durationExpr" use="optional"/>
    <xsd:attribute name="max" type="durationExpr" use="optional"/>
</xsd:attributeGroup>
<!--

```

```

-->
<!-- applied to Time -->
<xsd:attributeGroup name="XFSPmmTime">
  <xsd:attribute name="minInclusive" type="timeExpr" use="optional"/>
  <xsd:attribute name="minExclusive" type="timeExpr" use="optional"/>
  <xsd:attribute name="maxInclusive" type="timeExpr" use="optional"/>
  <xsd:attribute name="maxExclusive" type="timeExpr" use="optional"/>
</xsd:attributeGroup>
<!--
-->
<xsd:attributeGroup name="XFSPmmTimeSimple">
  <xsd:attribute name="min" type="timeExpr" use="optional"/>
  <xsd:attribute name="max" type="timeExpr" use="optional"/>
</xsd:attributeGroup>
<!--
-->
<!-- applied to String, Binary, URI -->
<xsd:attributeGroup name="XFSPlengthNumber">
  <xsd:attribute name="length" type="positiveIntExpr" use="optional"/>
  <xsd:attribute name="minLength" type="positiveIntExpr" use="optional"/>
  <xsd:attribute name="maxLength" type="positiveIntExpr" use="optional"/>
</xsd:attributeGroup>
<!--
-->
<!-- applied to Number, and Money -->
<xsd:attributeGroup name="XFSPspNumber">
  <xsd:attribute name="scale" type="positiveIntExpr" use="optional"/>
  <xsd:attribute name="precision" type="positiveIntExpr" use="optional"/>
</xsd:attributeGroup>
<!--

```

XForms basic Data types for Models in Schema syntax

```

-->
<!--

```

Definition of string type.

```

-->
<xsd:complexType name="string">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attributeGroup ref="XFSPcommon"/>
      <xsd:attributeGroup ref="XFSPlengthNumber"/>
      <xsd:attributeGroup ref="XFSPcalcAndChoicesString"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<!--

```

Definition of boolean type.

```

-->
<xsd:complexType name="boolean">
  <xsd:simpleContent>
    <xsd:extension base="xsd:boolean">
      <xsd:attributeGroup ref="XFSPcommonMinusEnum"/>
      <xsd:attribute name="calculate" type="boolExpr" use="optional"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

```

```
</xsd:simpleContent>
</xsd:complexType>
<!--
```

Definition of number type.

```
-->
<xsd:complexType name="number">
  <xsd:simpleContent>
    <xsd:extension base="xsd:decimal">
      <xsd:attributeGroup ref="XFSPcommon"/>
      <xsd:attributeGroup ref="XFSPmmNumber"/>
      <xsd:attributeGroup ref="XFSPspNumber"/>
      <xsd:attributeGroup ref="XFSPcalcAndChoicesNumber"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<!--
```

Definition of currency type.

```
-->
<xsd:complexType name="currency">
  <xsd:simpleContent>
    <xsd:extension base="currencyType">
      <xsd:attributeGroup ref="XFSPcommon"/>
      <xsd:attributeGroup ref="XFSPcalcAndChoicesCurrency"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<!--
```

Definition of money type.
Alternative B: compound datatype

```
-->
<xsd:complexType name="money">
  <xsd:sequence>
    <xsd:element name="value">
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:decimal">
            <xsd:attribute name="calculate" type="numberExpr"/>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="currency">
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="currencyType">
            <xsd:attribute name="calculate" type="currencyExpr"/>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="allowCurrency" type="currencyExpr" minOccurs="0"
maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<!--
```

allow zero or more allowCurrency elements to restrict the value space of money.

-->

```
</xsd:sequence>
<xsd:attributeGroup ref="XFSPcommon"/>
<xsd:attributeGroup ref="XFSPmmNumber"/>
<xsd:attributeGroup ref="XFSPspNumber"/>
</xsd:complexType>
<!--
```

Definition of date type.

-->

```
<xsd:complexType name="date">
  <xsd:simpleContent>
    <xsd:extension base="xsd:date">
      <xsd:attributeGroup ref="XFSPcommon"/>
      <xsd:attributeGroup ref="XFSPmmDate"/>
      <xsd:attributeGroup ref="XFSPcalcAndChoicesDate"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<!--
```

Definition of time type.

-->

```
<xsd:complexType name="time">
  <xsd:simpleContent>
    <xsd:extension base="xsd:time">
      <xsd:attributeGroup ref="XFSPcommon"/>
      <xsd:attributeGroup ref="XFSPmmTime"/>
      <xsd:attributeGroup ref="XFSPcalcAndChoicesTime"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<!--
```

Definition of duration type.

-->

```
<xsd:complexType name="duration">
  <xsd:simpleContent>
    <xsd:extension base="xsd:timeDuration">
      <xsd:attributeGroup ref="XFSPcommon"/>
      <xsd:attributeGroup ref="XFSPmmDuration"/>
      <xsd:attributeGroup ref="XFSPcalcAndChoicesDuration"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<!--
-->
<!--
```

Definition of uri type.

-->

```
<xsd:complexType name="uri">
  <xsd:simpleContent>
```

```

    <xsd:extension base="xsd:uriReference">
      <xsd:attributeGroup ref="XFSPcommon"/>
      <xsd:attributeGroup ref="XFSPlengthNumber"/>
      <xsd:attributeGroup ref="XFSPcalcAndChoicesURI"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<!--

```

Definition of binary type.

```

-->
<xsd:complexType name="binary">
  <xsd:simpleContent>
    <xsd:extension base="xsd:binary">
      <xsd:attributeGroup ref="XFSPcommon"/>
      <xsd:attributeGroup ref="XFSPlengthNumber"/>
      <xsd:attributeGroup ref="XFSPcalcAndChoicesBinary"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<!--

```

XForms simple syntax element definitions
Structure elements, followed by datatype elements.

```

-->
<!--

```

Definition of the simple element, used to contain the simple syntax
element declarations.

```

-->
<xsd:element name="simple">
  <xsd:complexType>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element ref="string"/>
      <xsd:element ref="boolean"/>
      <xsd:element ref="number"/>
      <xsd:element ref="currency"/>
      <xsd:element ref="money"/>
      <xsd:element ref="date"/>
      <xsd:element ref="time"/>
      <xsd:element ref="duration"/>
      <xsd:element ref="uri"/>
      <xsd:element ref="binary"/>
      <xsd:element ref="element"/>
      <xsd:element ref="array"/>
      <xsd:element ref="group"/>
      <xsd:element ref="switch"/>
      <xsd:element ref="union"/>
    </xsd:choice>
    <xsd:attribute name="id" type="xsd:ID" use="optional"/>
    <xsd:attribute name="name" type="xsd:NCName" use="optional"/>
  </xsd:complexType>
</xsd:element>
<!--

```

Definition of simple syntax group element.

```

-->
<xsd:element name="group">
  <xsd:complexType>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element ref="string"/>
      <xsd:element ref="boolean"/>
      <xsd:element ref="number"/>
      <xsd:element ref="currency"/>
      <xsd:element ref="money"/>
      <xsd:element ref="date"/>
      <xsd:element ref="time"/>
      <xsd:element ref="duration"/>
      <xsd:element ref="uri"/>
      <xsd:element ref="binary"/>
      <xsd:element ref="element"/>
      <xsd:element ref="switch"/>
      <xsd:element ref="union"/>
    </xsd:choice>
    <xsd:attributeGroup ref="XFSPname"/>
  </xsd:complexType>
</xsd:element>

```

<!--
The occurs attribute group used on array.

```

-->
<xsd:attributeGroup name="occurs">
  <xsd:attribute name="minOccurs" use="default" value="1">
    <xsd:simpleType>
      <xsd:union memberTypes="xfmExpr xsd:nonNegativeInteger"/>
    </xsd:simpleType>
  </xsd:attribute>
  <xsd:attribute name="maxOccurs" use="default" value="1">
    <xsd:simpleType>
      <xsd:union memberTypes="xfmExpr xsd:nonNegativeInteger">
        <xsd:simpleType>
          <xsd:restriction base="xsd:NMTOKEN">
            <xsd:enumeration value="unbounded"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:union>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:attributeGroup>
<!--

```

Definition of simple syntax array element.
Issue: should we allow switches or unions?

```

-->
<xsd:element name="array">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element ref="string"/>
      <xsd:element ref="boolean"/>
      <xsd:element ref="number"/>
      <xsd:element ref="currency"/>
      <xsd:element ref="money"/>
      <xsd:element ref="date"/>
      <xsd:element ref="time"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

```

```

    <xsd:element ref="duration"/>
    <xsd:element ref="uri"/>
    <xsd:element ref="binary"/>
    <xsd:element ref="element"/>
  </xsd:choice>
  <xsd:attribute name="name" type="xsd:NCName" use="optional"/>
  <xsd:attributeGroup ref="occurs"/>
</xsd:complexType>
</xsd:element>
<!--

```

Definition of simple syntax union element.

Note: the schema for <union> requires a name attribute on its child datatypes because it simply reuses their definitions, which do. The spec says name isn't required on child datatypes. This is a known inconsistency that will be fixed in a future version.

```

-->
<xsd:element name="union">
  <xsd:complexType>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element ref="string"/>
      <xsd:element ref="boolean"/>
      <xsd:element ref="number"/>
      <xsd:element ref="currency"/>
      <xsd:element ref="money"/>
      <xsd:element ref="date"/>
      <xsd:element ref="time"/>
      <xsd:element ref="duration"/>
      <xsd:element ref="uri"/>
      <xsd:element ref="binary"/>
    </xsd:choice>
    <xsd:attributeGroup ref="XFSPname"/>
  </xsd:complexType>
</xsd:element>
<!--

```

Definition of simple syntax switch element.

```

-->
<xsd:element name="switch">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="case" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:choice maxOccurs="unbounded">
            <xsd:element ref="string"/>
            <xsd:element ref="boolean"/>
            <xsd:element ref="number"/>
            <xsd:element ref="currency"/>
            <xsd:element ref="money"/>
            <xsd:element ref="date"/>
            <xsd:element ref="time"/>
            <xsd:element ref="duration"/>
            <xsd:element ref="uri"/>
            <xsd:element ref="binary"/>
            <xsd:element ref="element"/>
            <xsd:element ref="group"/>
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

```

```

        <xsd:element ref="array" />
    </xsd:choice>
    <xsd:attributeGroup ref="XFSPname" />
    <xsd:attribute name="condition" type="xfmExpr" use="optional" />
    </xsd:complexType>
</xsd:element>
</xsd:sequence>
<xsd:attributeGroup ref="XFSPname" />
</xsd:complexType>
</xsd:element>
<!--

```

Definition of simple syntax element element.

```

-->
<xsd:element name="element" id="element">
  <xsd:complexType>
    <xsd:attribute name="type" type="xsd:QName" use="required" />
    <xsd:attributeGroup ref="XFSPcommonMinusEnumSimple" />
  </xsd:complexType>
</xsd:element>
<!--

```

Definition of the mask facet, used in strings and currency,
in simple syntax only.
Modeled after WAP/WML's format attribute.

```

-->
<xsd:simpleType name="maskType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="((A|a|X|x|N|n|M|m)|(\.\.))*([0-9\*](A|a|X|x|N|n|M|m))?" />
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="maskExpr">
  <xsd:union memberTypes="maskType xfmExpr" />
</xsd:simpleType>
<!--

```

Definition of simple syntax string element.

```

-->
<xsd:element name="string" id="string">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attributeGroup ref="XFSPcommonSimple" />
        <xsd:attributeGroup ref="XFSPlengthNumber" />
        <xsd:attributeGroup ref="XFSPcalcAndChoicesString" />
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
<!--

```

Definition of simple syntax boolean element.

```

-->
<xsd:element name="boolean">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:boolean">

```

```

        <xsd:attributeGroup ref="XFSPcommonMinusEnumSimple"/>
        <xsd:attribute name="calculate" type="boolExpr" use="optional"/>
    </xsd:extension>
</xsd:simpleContent>
</xsd:complexType>
</xsd:element>
<!--

```

Definition of simple syntax number element.

```

-->
<xsd:element name="number">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
        <xsd:attributeGroup ref="XFSPcommonSimple"/>
        <xsd:attributeGroup ref="XFSPmmNumberSimple"/>
        <xsd:attributeGroup ref="XFSPspNumber"/>
        <xsd:attributeGroup ref="XFSPcalcAndChoicesNumber"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
<!--
    TBD: mask is supposedly allowed on the currency element but, should
    I allow it if the currency element is part of money?
-->
<!--

```

Definition of simple syntax currency element.

```

-->
<xsd:element name="currency">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="currencyType">
        <xsd:attributeGroup ref="XFSPcommonSimple"/>
        <xsd:attributeGroup ref="XFSPcalcAndChoices"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
<!--

```

Definition of simple syntax money element.
Alternative B: compound datatype

```

-->
<xsd:element name="money">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="value">
        <xsd:complexType>
          <xsd:simpleContent>
            <xsd:extension base="xsd:decimal">
              <xsd:attribute name="calculate" type="numberExpr"/>
            </xsd:extension>
          </xsd:simpleContent>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

```

```

</xsd:element>
<xsd:element name="currency">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="currencyType">
        <xsd:attribute name="calculate" type="currencyExpr"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
<xsd:element name="allowCurrency" type="currencyExpr" minOccurs="0"
maxOccurs="unbounded"/>
<!--
  allow zero or more allowCurrency elements to restrict
  the value space of money.
-->
</xsd:sequence>
<xsd:attributeGroup ref="XFSPcommonSimple"/>
<xsd:attributeGroup ref="XFSPmmNumberSimple"/>
<xsd:attributeGroup ref="XFSPspNumber"/>
</xsd:complexType>
</xsd:element>
<!--

```

Definition of simple syntax date element.

```

-->
<xsd:element name="date">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:date">
        <xsd:attributeGroup ref="XFSPcommonSimple"/>
        <xsd:attributeGroup ref="XFSPmmDateSimple"/>
        <xsd:attribute name="precision" use="optional">
          <xsd:simpleType>
            <xsd:restriction base="xsd:NMTOKEN">
              <xsd:enumeration value="years"/>
              <xsd:enumeration value="months"/>
              <xsd:enumeration value="days"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attributeGroup ref="XFSPcalcAndChoicesDate"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
<!--

```

Definition of simple syntax time element.

```

-->
<xsd:element name="time">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:time">
        <xsd:attributeGroup ref="XFSPcommonSimple"/>
        <xsd:attributeGroup ref="XFSPmmTimeSimple"/>
        <xsd:attribute name="precision" use="optional">

```

```

        <xsd:simpleType>
            <xsd:restriction base="xsd:NMTOKEN">
                <xsd:enumeration value="hours"/>
                <xsd:enumeration value="minutes"/>
                <xsd:enumeration value="seconds"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:attribute>
    <xsd:attributeGroup ref="XFSPcalcAndChoicesTime"/>
</xsd:extension>
</xsd:simpleContent>
</xsd:complexType>
</xsd:element>
<!--

```

Definition of simple syntax duration element.

```

-->
<xsd:element name="duration">
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:extension base="xsd:timeDuration">
                <xsd:attributeGroup ref="XFSPcommonSimple"/>
                <xsd:attributeGroup ref="XFSPmmDurationSimple"/>
                <xsd:attribute name="precision" use="optional">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:NMTOKEN">
                            <xsd:enumeration value="years"/>
                            <xsd:enumeration value="months"/>
                            <xsd:enumeration value="days"/>
                            <xsd:enumeration value="hours"/>
                            <xsd:enumeration value="minutes"/>
                            <xsd:enumeration value="seconds"/>
                        </xsd:restriction>
                    </xsd:simpleType>
                </xsd:attribute>
                <xsd:attributeGroup ref="XFSPcalcAndChoicesDuration"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
<!--

```

Definition of simple syntax uri element.

```

-->
<xsd:element name="uri">
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:extension base="xsd:uriReference">
                <xsd:attributeGroup ref="XFSPcommonSimple"/>
                <xsd:attributeGroup ref="XFSPlengthNumber"/>
                <xsd:attributeGroup ref="XFSPcalcAndChoicesURI"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
<!--

```

Definition of simple syntax binary element.

```
-->
<xsd:element name="binary">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:binary">
        <xsd:attributeGroup ref="XFSPcommonSimple"/>
        <xsd:attributeGroup ref="XFSPlengthNumber"/>
        <xsd:attributeGroup ref="XFSPcalcAndChoicesBinary"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

[previous](#) [next](#) [contents](#)

Appendix B: XSLT from Simple to Schema Syntax

Contents

- [B.1 XSLT](#)
- [B.2 Required DTD for Transformation](#)

B.1 XSLT

The following non-normative XSLT can be used to convert XForms simple syntax into XML Schema syntax.

```
<?xml version='1.0'?>

<!-- NOTE: this XSLT transforms a XForms simple syntax into an
XML Schema conforming to the October 24, 2000 candidate
recomendation -->

<xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
  xmlns="http://www.w3.org/2000/10/XMLSchema"
  xmlns:xform="http://www.w3.org/2001/02/xforms" >

<xsl:strip-space elements="xform"/>

<xsl:output method="xml" indent="yes" encoding="UTF-8"
  standalone="no" />

<!-- the special indicator used for maxOccurs to mean unbounded -->
<xsl:variable name="maxOccursSpecial">*</xsl:variable>
<!-- the special value used for maxLength to mean unbounded -->
<xsl:variable name="maxLengthSpecial">unlimited</xsl:variable>
<!-- the special value used for min to mean minus infinity -->
<xsl:variable name="minInfinity">minus infinity</xsl:variable>
<!-- the special value used for max to mean plus infinity -->
<xsl:variable name="maxInfinity">plus infinity</xsl:variable>
<!-- the special value used for scale to mean unlimited decimals -->
<xsl:variable name="unlimitedScale">unlimited</xsl:variable>
<!-- the special value used for precision to mean unlimited precision -->
<xsl:variable name="unlimitedPrecision">unlimited</xsl:variable>

<!-- the list of WML classes -->
<xsl:variable name="classesList" >AaNnXxMm</xsl:variable>
<xsl:variable name="classesTrans">01234567</xsl:variable>
```

```

<!-- special characters to escape in patterns -->
<xsl:variable name="specialCharacters">\\|.-^?*\+{}()[]</xsl:variable>

<!-- template:      match="/"
      function:     matches the root node, calls the template "xform" for
                    all <xform> elements. drops any other elements.
      parameters:   none
      output:       &lt;schema> skeleton.
-->
<xsl:template match="/">
  <xsl:element name="xsd:schema">
    <xsl:element name="xsd:annotation">
      <xsl:element name="xsd:documentation">
<xsl:text>Automatically generated from an XForms data model.</xsl:text>

<xsl:text>Using</xsl:text>
<xsl:value-of select="system-property('xsl:vendor')"/>
<xsl:text>at XSL version</xsl:text>
<xsl:value-of select="format-number(system-property('xsl:version'),
                                   '#0.0')"/>.

<xsl:if test="system-property('xsl:version') > 1.0">
Note, the stylesheet was designed for a XSLT version 1.0 processor.
</xsl:if>
      </xsl:element>
    </xsl:element>
    <!-- output the xform -->
    <xsl:apply-templates />
  </xsl:element>
</xsl:template>

<!--
xform element
-->
<!-- template:      match="xform"
      function:     matches an xform, calls the templates for all childs
      parameters:   none
      output:       none of it's own
-->
<xsl:template match="xform">
  <!-- output the elements -->
  <xsl:apply-templates />
</xsl:template>

<!--
model element
-->
<!-- template:      match="model"
      function:     matches an xform, calls the templates for all childs
      parameters:   none
      output:       none of it's own
-->

```

```

<xsl:template match="model">
  <!-- output the elements -->
  <xsl:apply-templates />
</xsl:template>

<!--
schema element
-->
<!-- template:   match="schema"
function:       matches an schema, and copies the inlined schema definition
                to the result tree
parameters:     none
output:         copy of the input
-->
<xsl:template match="xsd:schema">
  <!-- copy XML Schema definition -->
  <xsl:copy-of select="node()"/>
</xsl:template>

<!--
simple element
-->
<!-- template:   match="simple"
function:       matches an simple, and processes the child elements
parameters:     none
output:         none of its own
-->
<xsl:template match="simple">
  <!-- output the elements -->
  <xsl:apply-templates />
</xsl:template>

<!--
instance element
-->
<!-- template:   match="instance"
function:       filters the instance
parameters:     none
output:         none
-->
<xsl:template match="instance"/>

<
binding element
-->
<!-- template:   match="binding"
function:       filters the binding
parameters:     none
output:         none

```

```

-->
<xsl:template match="binding"/>

<!--
submit element
-->

<!-- template:   match="submit"
      function:   filters the submit elements
      parameters: none
      output:     none
-->
<xsl:template match="submit"/>

<!--
group element
-->
<!-- template:   matches="group"
      function:   converts the group into a complex type and processes
                  all childs
      parameters: none
      output:     a complex type representing the group
-->
<xsl:template match="group">
  <xsl:element name="xsd:element">
    <xsl:attribute name="name">
      <xsl:value-of select="@name"/>
    </xsl:attribute>
    <!-- check attributes -->
    <xsl:call-template name="checkAttributes"/>

    <xsl:element name="xsd:complexType">
      <xsl:element name="xsd:complexContent">
        <xsl:element name="xsd:extension">
          <!-- check for a base -->
          <xsl:attribute name="base">xsd:anyType</xsl:attribute>
          <!-- the rest of the elements are always a sequence -->
          <xsl:element name="xsd:sequence">
            <!-- include all child elements -->
            <xsl:apply-templates/>
          </xsl:element>
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:element>
</xsl:template>

<!--

```

```

union element
-->
<!-- template:    matches="union"
      function:    converts a union into a union of simpleTypes.
      parameters:  none
      output:      converted definition
-->
<xsl:template match="union">
  <xsl:choose>
    <xsl:when test="string-length(@name)=0">
      <xsl:message terminate="no">
        An anonymous union definition is not supported.
      </xsl:message>
      <xsl:comment> Anonymous union definition dropped </xsl:comment>
    </xsl:when>
    <xsl:otherwise>
      <xsl:element name="xsd:element">
        <xsl:attribute name="name">
          <xsl:value-of select="@name"/>
        </xsl:attribute>
        <!-- check common attributes -->
        <xsl:call-template name="checkAttributes"/>

        <xsl:element name="xsd:simpleType">
          <xsl:element name="xsd:union">
            <xsl:apply-templates />
          </xsl:element>
        </xsl:element>
      </xsl:element>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!--
switch element
-->

<!-- template:    matches="switch"
      function:    converts the switch into a choice and processes all
                   cases
      parameters:  none
      output:      a choice representing the variant
-->
<xsl:template match="switch">
  <xsl:choose>
    <xsl:when test="string-length(@name)=0">
      <xsl:message terminate="no">
        An anonymous variant definition is not supported.
      </xsl:message>
      <xsl:comment> Anonymous variant definition dropped </xsl:comment>
    </xsl:when>
    <xsl:otherwise>
      <xsl:element name="xsd:element">

```

```

    <xsl:attribute name="name">
      <xsl:value-of select="@name"/>
    </xsl:attribute>
    <!-- check common attributes -->
    <xsl:call-template name="checkAttributes"/>

    <xsl:element name="xsd:complexType">
      <xsl:element name="xsd:complexContent">
        <xsl:element name="xsd:extension">
          <xsl:attribute name="base">xsd:anyType</xsl:attribute>
          <xsl:element name="xsd:choice">
            <xsl:apply-templates select="case"/>
          </xsl:element>
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:element>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<!--
case element
-->

<!-- template:   matches="caase"
function:       converts the case into a sequence of other elements
parameters:    none
output:        a sequence representing the case
-->
<xsl:template match="case">
  <xsl:element name="xsd:sequence">
    <xsl:if test="string-length(@name) > 0">
      <xsl:attribute name="xform:name">
        <xsl:value-of select="@name"/>
      </xsl:attribute>
    </xsl:if>
    <xsl:if test="string-length(@condition) > 0">
      <xsl:attribute name="xform:condition">
        <xsl:value-of select="@condition"/>
      </xsl:attribute>
    </xsl:if>

    <!-- include elements -->
    <xsl:apply-templates />

  </xsl:element>
</xsl:template>

<!--
string element

```

```

-->
<!-- template:  matches="string"
function:      converts a string.
parameters:    none
output:        converted definition
note:          it is assumed that all value childs
                conform to all other restrictions, otherwise their
                value will be allowed despite the further
                restrictions.
-->
<xsl:template match="string">
  <!-- create definition -->
  <xsl:variable name="definition">

    <xsl:element name="xsd:simpleType">
      <xsl:choose>
        <!-- if we have a closed range, or no enumeration value, we can
              create a simple type -->
        <xsl:when test="@enum='closed' or
                      count(value)=0">
          <xsl:element name="xsd:restriction">
            <xsl:attribute name="base">xsd:string</xsl:attribute>
            <xsl:call-template name="restrictString"/>
            <xsl:apply-templates select="value"/>
          </xsl:element>
        </xsl:when>
        <xsl:otherwise>
          <xsl:element name="xsd:union">
            <xsl:element name="xsd:simpleType">
              <xsl:element name="xsd:restriction">
                <xsl:attribute name="base">xsd:string</xsl:attribute>
                <xsl:call-template name="restrictString"/>
                <xsl:apply-templates select="value"/>
              </xsl:element>
            </xsl:element>
            <xsl:element name="xsd:simpleType">
              <xsl:element name="xsd:restriction">
                <xsl:attribute name="base">xsd:string</xsl:attribute>
                <xsl:call-template name="restrictString"/>
              </xsl:element>
            </xsl:element>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:element>
    </xsl:variable>

    <!-- if we have a name, we can create an element or attribute,
          otherwise we're part of a union -->
    <xsl:choose>
      <xsl:when test="string-length(@name) > 0">
        <!-- create element or attribute -->
        <xsl:element name="xsd:element">
          <xsl:attribute name="name">

```

```

        <xsl:value-of select="@name"/>
    </xsl:attribute>
    <!-- check common attributes -->
    <xsl:call-template name="checkAttributes"/>

    <xsl:choose>
        <!-- if we have no children, and no other restrictions,
             we can make a short definition -->
        <xsl:when test='count(child::node())=0 and @minLength="0" and
                        @maxLength="unlimited" and
                        string-length(@pattern)=0 and
                        string-length(@mask)=0'>
            <xsl:attribute name="type">xsd:string</xsl:attribute>
        </xsl:when>
        <xsl:otherwise>
            <!-- dump full definition -->
            <xsl:copy-of select="$definition"/>
        </xsl:otherwise>
    </xsl:choose>
</xsl:element>
</xsl:when>
<xsl:otherwise>
    <!-- just dump definition -->
    <xsl:copy-of select="$definition"/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<!-- template:   name="restrictString"
function:       add restrictions to a string
parameters:    a string context node with the following attributes
                @maxLength
                @minLength
                @mask
                @pattern
                and the following child elements
                <pattern/>
                <mask/>
output:        elements to represent the restrictions
-->
<xsl:template name="restrictString">
    <!-- create pattern -->
    <xsl:variable name="pattern">
        <xsl:for-each select="mask|pattern">
            <xsl:text>(</xsl:text>
            <xsl:choose>
                <xsl:when test='name()="mask"'>
                    <xsl:call-template name="make-pattern">
                        <xsl:with-param name="mask">
                            <xsl:value-of select="."/>
                        </xsl:with-param>
                    </xsl:call-template>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:value-of select="."/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:for-each>
    </xsl:variable>

```

```

        </xsl:otherwise>
    </xsl:choose>
    <xsl:text>)</xsl:text>
    <xsl:if test="not(position()=last())">
        <xsl:text>|</xsl:text>
    </xsl:if>
</xsl:for-each>

<xsl:if test="@mask">
    <xsl:if test="count(mask|pattern) > 0">
        <xsl:text>|</xsl:text>
    </xsl:if>
    <xsl:text>(</xsl:text>
    <xsl:call-template name="make-pattern">
        <xsl:with-param name="mask">
            <xsl:value-of select="@mask"/>
        </xsl:with-param>
    </xsl:call-template>
    <xsl:text>)</xsl:text>
</xsl:if>

<xsl:if test="@pattern">
    <xsl:if test="string-length(@mask) > 0 and
        count(mask|pattern) > 0">
        <xsl:text>|</xsl:text>
    </xsl:if>
    <xsl:text>(</xsl:text>
    <xsl:value-of select="@pattern"/>
    <xsl:text>)</xsl:text>
</xsl:if>
</xsl:variable>

<!-- check if @length is an non negative integer -->
<xsl:variable name="lengthIsNNI">
    <xsl:call-template name="checkNonNegInt">
        <xsl:with-param name="test">
            <xsl:value-of select="@length"/>
        </xsl:with-param>
    </xsl:call-template>
</xsl:variable>

<!-- check if @max is an non negative integer -->
<xsl:variable name="maxIsNNI">
    <xsl:call-template name="checkNonNegInt">
        <xsl:with-param name="test">
            <xsl:value-of select="@max"/>
        </xsl:with-param>
    </xsl:call-template>
</xsl:variable>

<!-- check if @min as an non negative integer -->
<xsl:variable name="minIsNNI">
    <xsl:call-template name="checkNonNegInt">
        <xsl:with-param name="test">
            <xsl:value-of select="@min"/>

```

```

    </xsl:with-param>
  </xsl:call-template>
</xsl:variable>

<!-- write xform:length restriction -->
<xsl:if test="string-length(@length) > 0 and
             not(@length=$maxLengthSpecial) and
             $lengthIsNNI='false'">
  <xsl:attribute name="xform:length">
    <xsl:value-of select="@length"/>
  </xsl:attribute>
</xsl:if>

<!-- write xform:maxLength restriction -->
<xsl:if test="string-length(@max) > 0 and
             not(@max=$maxLengthSpecial) and
             $maxIsNNI='false'">
  <xsl:attribute name="xform:maxLength">
    <xsl:value-of select="@max"/>
  </xsl:attribute>
</xsl:if>

<!-- write xform:minLength restriction -->
<xsl:if test="string-length(@min) > 0 and
             not(@min='0') and
             $minIsNNI='false'">
  <xsl:attribute name="xform:minLength">
    <xsl:value-of select="@min"/>
  </xsl:attribute>
</xsl:if>

<!-- write xsd:length restriction -->
<xsl:choose>
  <xsl:when test="$lengthIsNNI='true'">
    <xsl:if test="not(@length=$maxLengthSpecial)">
      <xsl:element name="xsd:length">
        <xsl:attribute name="value">
          <xsl:value-of select="@length"/>
        </xsl:attribute>
      </xsl:element>
    </xsl:if>
  </xsl:when>
  <xsl:when test="count(length) > 0">
    <xsl:element name="xsd:length">
      <xsl:attribute name="value">
        <xsl:value-of select="length[1]"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
</xsl:choose>

<!-- write xsd:maxLength restriction -->
<xsl:choose>
  <xsl:when test="$maxIsNNI='true'">
    <xsl:if test="not(@max=$maxLengthSpecial)">

```

```

        <xsl:element name="xsd:maxLength">
            <xsl:attribute name="value">
                <xsl:value-of select="@max"/>
            </xsl:attribute>
        </xsl:element>
    </xsl:if>
</xsl:when>
<xsl:when test="count(max) > 0">
    <xsl:element name="xsd:maxLength">
        <xsl:attribute name="value">
            <xsl:value-of select="max[1]"/>
        </xsl:attribute>
    </xsl:element>
</xsl:when>
</xsl:choose>

<!-- write xsd:minLength restriction -->
<xsl:choose>
    <xsl:when test="$minIsNNI='true'">
        <xsl:if test="not(@min='0')">
            <xsl:element name="xsd:minLength">
                <xsl:attribute name="value">
                    <xsl:value-of select="@min"/>
                </xsl:attribute>
            </xsl:element>
        </xsl:if>
    </xsl:when>
    <xsl:when test="count(min) > 0">
        <xsl:element name="xsd:min">
            <xsl:attribute name="value">
                <xsl:value-of select="min[1]"/>
            </xsl:attribute>
        </xsl:element>
    </xsl:when>
</xsl:choose>

<!-- write xsd:pattern -->
<xsl:if test="string-length($pattern) > 0">
    <xsl:element name="xsd:pattern">
        <xsl:attribute name="value">
            <xsl:value-of select="$pattern"/>
        </xsl:attribute>
    </xsl:element>
</xsl:if>
</xsl:template>

<!-- template:      name="make-pattern"
function:          converts a mask to a pattern
parameters: mask   : the mask to transform
                  i   : current position inside mask
                  last : last source character
                  count : how many times did the last character
                           occur
output:            a pattern with the same meaning

```

```

-->
<xsl:template name="make-pattern">
  <xsl:param name="mask"/>
  <xsl:param name="i">0</xsl:param>
  <xsl:param name="last"/>
  <xsl:param name="count"/>

  <xsl:choose>
    <!-- if there are characters to process, do so -->
    <xsl:when test="$i < string-length($mask)">
      <!-- get current character -->
      <xsl:variable name="c">
        <xsl:value-of select="substring($mask, $i + 1, 1)"/>
      </xsl:variable>
      <!-- process it -->
      <xsl:choose>
        <!-- check for special characters first -->
        <!-- the backslash (\): the next character is ment literally -->
        <xsl:when test="$c='\'">
          <!-- output old count -->
          <xsl:if test="$count > 1">
            <xsl:text>{</xsl:text>
            <xsl:value-of select="$count"/>
            <xsl:text>}</xsl:text>
          </xsl:if>
          <!-- check if the character has to be escaped -->
          <xsl:call-template name="escape-char">
            <xsl:with-param name="char">
              <xsl:value-of select="substring($mask, $i + 2, 1)"/>
            </xsl:with-param>
          </xsl:call-template>

          <!-- call recursively -->
          <xsl:call-template name="make-pattern">
            <xsl:with-param name="mask">
              <xsl:value-of select="$mask"/>
            </xsl:with-param>
            <xsl:with-param name="i">
              <xsl:value-of select="$i + 2"/>
            </xsl:with-param>
            <xsl:with-param name="last">
              <xsl:text>\</xsl:text>
            </xsl:with-param>
            <xsl:with-param name="count">
              <xsl:value-of select="0"/>
            </xsl:with-param>
          </xsl:call-template>
        </xsl:when>
        <!-- the asterix (*): any other number of the following class -->
        <xsl:when test="$c='*'">
          <!-- get next character -->
          <xsl:variable name="next">
            <xsl:value-of select="substring($mask, $i + 2, 1)"/>
          </xsl:variable>
          <xsl:choose>

```

```

<!-- it the next is equal to the last, create special output -->
<xsl:when test="$last=$next">
  <xsl:text>{</xsl:text>
  <xsl:value-of select="$count"/>
  <xsl:text>,</xsl:text>
</xsl:when>
<xsl:otherwise>
  <!-- output old count -->
  <xsl:if test="$count > 1">
    <xsl:text>{</xsl:text>
    <xsl:value-of select="$count"/>
    <xsl:text>}</xsl:text>
  </xsl:if>
  <!-- converte wml character class into unicode class -->
  <xsl:call-template name="convert-characterClass">
    <xsl:with-param name="class">
      <xsl:value-of select="$next"/>
    </xsl:with-param>
  </xsl:call-template>

  <xsl:text>*</xsl:text>
</xsl:otherwise>
</xsl:choose>
<!-- keep in mind, that any mask ends after a *. -->
</xsl:when>
<!-- a number [1-9]: repeat next character n times -->
<xsl:when test="contains('0123456789', $c)">
  <!-- get next character -->
  <xsl:variable name="next">
    <xsl:value-of select="substring($mask, $i + 2, 1)"/>
  </xsl:variable>
  <xsl:choose>
    <!-- when the next character equals the last,
    just increase count -->
    <xsl:when test="$last=$next">
      <xsl:call-template name="make-pattern">
        <xsl:with-param name="mask">
          <xsl:value-of select="$mask"/>
        </xsl:with-param>
        <xsl:with-param name="i">
          <xsl:value-of select="$i + 2"/>
        </xsl:with-param>
        <xsl:with-param name="last">
          <xsl:value-of select="$last"/>
        </xsl:with-param>
        <xsl:with-param name="count">
          <xsl:value-of select="$count + $c"/>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <!-- output old count -->
      <xsl:if test="$count > 1">
        <xsl:text>{</xsl:text>
        <xsl:value-of select="$count"/>

```

```

        <xsl:text>}</xsl:text>
    </xsl:if>
    <!-- convert character class -->
    <xsl:variable name="out">
        <xsl:call-template name="convert-characterClass">
            <xsl:with-param name="class">
                <xsl:value-of select="$next"/>
            </xsl:with-param>
        </xsl:call-template>
    </xsl:variable>
    <!-- output unicode character classes -->
    <xsl:value-of select="$out"/>
    <!-- call recursively -->
    <xsl:call-template name="make-pattern">
        <xsl:with-param name="mask">
            <xsl:value-of select="$mask"/>
        </xsl:with-param>
        <xsl:with-param name="i">
            <xsl:value-of select="$i + 2"/>
        </xsl:with-param>
        <xsl:with-param name="last">
            <xsl:value-of select="$next"/>
        </xsl:with-param>
        <xsl:with-param name="count">
            <xsl:value-of select="$c"/>
        </xsl:with-param>
    </xsl:call-template>
    </xsl:otherwise>
</xsl:choose>
</xsl:when>
<!-- if it's the same as the last, just increase count -->
<xsl:when test='$c=$last'>
    <!-- call recursively -->
    <xsl:call-template name="make-pattern">
        <xsl:with-param name="mask">
            <xsl:value-of select="$mask"/>
        </xsl:with-param>
        <xsl:with-param name="i">
            <xsl:value-of select="$i + 1"/>
        </xsl:with-param>
        <xsl:with-param name="last">
            <xsl:value-of select="$last"/>
        </xsl:with-param>
        <xsl:with-param name="count">
            <xsl:value-of select="$count + 1"/>
        </xsl:with-param>
    </xsl:call-template>
</xsl:when>
<!-- a new/different character -->
<xsl:otherwise>
    <!-- catches the fact that the user missed the backslash in
         front of a literal character -->
    <!-- output old count -->
    <xsl:if test="$count > 1">
        <xsl:text>{</xsl:text>

```

```

        <xsl:value-of select="$count"/>
        <xsl:text>}</xsl:text>
    </xsl:if>
    <!-- convert character class -->
    <xsl:variable name="out">
        <xsl:call-template name="convert-characterClass">
            <xsl:with-param name="class">
                <xsl:value-of select="$c"/>
            </xsl:with-param>
        </xsl:call-template>
    </xsl:variable>
    <!-- output unicode character classes -->
    <xsl:value-of select="$out"/>
    <!-- call recursively -->
    <xsl:call-template name="make-pattern">
        <xsl:with-param name="mask">
            <xsl:value-of select="$mask"/>
        </xsl:with-param>
        <xsl:with-param name="i">
            <xsl:value-of select="$i + 1"/>
        </xsl:with-param>
        <xsl:with-param name="last">
            <xsl:value-of select="$c"/>
        </xsl:with-param>
        <xsl:with-param name="count">
            <xsl:value-of select="1"/>
        </xsl:with-param>
    </xsl:call-template>
</xsl:otherwise>
</xsl:choose>
</xsl:when>
<xsl:otherwise>
    <xsl:if test="$count > 1">
        <xsl:text>{</xsl:text>
        <xsl:value-of select="$count"/>
        <xsl:text>}</xsl:text>
    </xsl:if>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<!-- template:    name="convert-characterClass"
function:        converts a wml character class into the proper
                  unicode character classes, as defined by <characterClasses/>
parameter:      class :wml character class
output:         matching unicode class(es)
-->
<xsl:template name="convert-characterClass">
    <xsl:param name="class"/>

    <xsl:if test='contains($classesList, $class)''>
        <xsl:variable name="i">
            <xsl:value-of select="translate($class, $classesList, $classesTrans)"/>
        </xsl:variable>

```

```

    <xsl:value-of select="document('')//this:class[$i + 1]/this:unicode"/>
  </xsl:if>
</xsl:template>

<!-- template:    name="escape-char"
      function:    escapes characters, to not have a special meaning in
                   reg exp
      parameters:  char :character to escape
      output:      save version
-->
<xsl:template name="escape-char">
  <xsl:param name="char"/>
  <xsl:choose>
    <!-- check if the character must be escaped -->
    <xsl:when test='contains($specialCharacters, $char)'\>
      <xsl:text>\</xsl:text>
      <xsl:value-of select="$char"/>
    </xsl:when>
    <xsl:otherwise>
      <!-- no escaping needed, just return $char -->
      <xsl:value-of select="$char"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!--
boolean element
-->

<!-- template:    match="boolean"
      function:    converts a boolean into a xsd:boolean
      parameters:  none
      output:      converted definition
-->
<xsl:template match="boolean">
  <!-- create definition -->
  <xsl:variable name="definition">
    <xsl:element name="xsd:simpleType">
      <xsl:choose>
        <!-- if we have closed range, or no enumeration value, we can
              create simple type -->
        <xsl:when test="@enum='closed' or
                      count(value)=0">
          <xsl:element name="xsd:restriction">
            <xsl:attribute name="base">xsd:boolean</xsl:attribute>
            <xsl:apply-templates select="value"/>
          </xsl:element>
        </xsl:when>
        <xsl:otherwise>
          <xsl:element name="xsd:union">
            <xsl:element name="xsd:simpleType">
              <xsl:element name="xsd:restriction">

```

```

        <xsl:attribute name="base">xsd:boolean</xsl:attribute>
        <xsl:apply-templates select="value"/>
    </xsl:element>
</xsl:element>
<xsl:element name="xsd:simpleType">
    <xsl:element name="xsd:restriction">
        <xsl:attribute name="base">xsd:boolean</xsl:attribute>
    </xsl:element>
</xsl:element>
</xsl:element>
</xsl:otherwise>
</xsl:choose>
</xsl:element>
</xsl:variable>

<!-- if we have a name, we can create an element or attribute,
      otherwise we're part of a union -->
<xsl:choose>
    <xsl:when test="string-length(@name) > 0">
        <!-- create element or attribute -->
        <xsl:element name="xsd:element">
            <xsl:attribute name="name">
                <xsl:value-of select="@name"/>
            </xsl:attribute>
            <!-- check common attributes -->
            <xsl:call-template name="checkAttributes"/>

            <xsl:choose>
                <!-- if we have no children, and no other restrictions,
                     we can make a short definition -->
                <xsl:when test='count(child::node())=0'>
                    <xsl:attribute name="type">xsd:boolean</xsl:attribute>
                </xsl:when>
                <xsl:otherwise>
                    <!-- dump full definition -->
                    <xsl:copy-of select="$definition"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:element>
    </xsl:when>
    <xsl:otherwise>
        <!-- just dump definition -->
        <xsl:copy-of select="$definition"/>
    </xsl:otherwise>
</xsl:choose>
</xsl:template>

<!--
number element
-->

<!-- template:    matches="number"
      function:   converts a number into an xsd:number.
-->

```

```

parameters: none
output:      converted definition
-->
<xsl:template match="number">
  <!-- create definition -->
  <xsl:variable name="definition">
    <xsl:element name="xsd:simpleType">
      <xsl:choose>
        <!-- if we have closed range, or no enumeration value, we can
              create simple type -->
        <xsl:when test="@enum='closed' or
                      count(value)=0">
          <xsl:element name="xsd:restriction">
            <xsl:attribute name="base">xsd:decimal</xsl:attribute>
            <xsl:call-template name="restrictNumber"/>
            <xsl:apply-templates select="value"/>
          </xsl:element>
        </xsl:when>
        <xsl:otherwise>
          <xsl:element name="xsd:union">
            <xsl:element name="xsd:simpleType">
              <xsl:element name="xsd:restriction">
                <xsl:attribute name="base">xsd:decimal</xsl:attribute>
                <xsl:call-template name="restrictNumber"/>
                <xsl:apply-templates select="value"/>
              </xsl:element>
            </xsl:element>
            <xsl:element name="xsd:simpleType">
              <xsl:element name="xsd:restriction">
                <xsl:attribute name="base">xsd:decimal</xsl:attribute>
                <xsl:call-template name="restrictNumber"/>
              </xsl:element>
            </xsl:element>
          </xsl:element>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:element>
  </xsl:variable>

  <!-- if we have a name, we can create an element or attribute,
        otherwise we're part of a union -->
  <xsl:choose>
    <xsl:when test="string-length(@name) > 0">
      <!-- create element or attribute -->
      <xsl:element name="xsd:element">
        <xsl:attribute name="name">
          <xsl:value-of select="@name"/>
        </xsl:attribute>
        <!-- check common attributes -->
        <xsl:call-template name="checkAttributes"/>

        <xsl:choose>
          <!-- if we have no children, and no other restrictions,
                we can make a short definition -->
          <xsl:when test='count(child::node())=0 and

```

```

                (string-length(@min)=0 or @min=$minInfinity) and
                (string-length(@max)=0 or @max=$maxInfinity) and
                (string-length(@precision)=0 or
                 @precision=$unlimitedPrecision) and
                (string-length(@scale)=0 or @scale=$unlimitedScale)')>
        <xsl:attribute name="type">xsd:decimal</xsl:attribute>
    </xsl:when>
    <xsl:otherwise>
        <!-- dump full definition -->
        <xsl:copy-of select="$definition"/>
    </xsl:otherwise>
</xsl:choose>
</xsl:element>
</xsl:when>
<xsl:otherwise>
    <!-- just dumpe definition -->
    <xsl:copy-of select="$definition"/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<!-- template:    name="restrictNumber"
function:        add erstrictions to a number
parameters:     a number context node with the following attributes
                 @max
                 @min
                 @precision
                 @scale
                 and the following child elements
                 <min>
                 <max>
                 <precision>
                 <scale>
output:         elements to represent the restrictions
-->
<xsl:template name="restrictNumber">

    <xsl:message terminate="no">@max:<xsl:value-of select="@max"/></xsl:message>
    <!-- check if max is a number -->
    <xsl:variable name="maxIsNumber">
        <xsl:call-template name="checkNumber">
            <xsl:with-param name="test">
                <xsl:value-of select="@max"/>
            </xsl:with-param>
        </xsl:call-template>
    </xsl:variable>
    <xsl:message terminate="no">$maxIsNumber:<xsl:value-of
select="$maxIsNumber"/></xsl:message>

    <!-- check if min is a number -->
    <xsl:variable name="minIsNumber">
        <xsl:call-template name="checkNumber">
            <xsl:with-param name="test">
                <xsl:value-of select="@min"/>

```

```

    </xsl:with-param>
  </xsl:call-template>
</xsl:variable>

<!-- check if precision is a number -->
<xsl:variable name="precisionIsNNI">
  <xsl:call-template name="checkNonNegInt">
    <xsl:with-param name="test">
      <xsl:value-of select="@precision"/>
    </xsl:with-param>
  </xsl:call-template>
</xsl:variable>

<!-- check if scale is a number -->
<xsl:variable name="scaleIsNNI">
  <xsl:call-template name="checkNonNegInt">
    <xsl:with-param name="test">
      <xsl:value-of select="@scale"/>
    </xsl:with-param>
  </xsl:call-template>
</xsl:variable>

<!-- write xform:max restriction -->
<xsl:if test="string-length(@max) > 0 and
             not(@max=$maxInfinity) and
             $maxIsNumber='false'">
  <xsl:attribute name="xform:maxInclusive">
    <xsl:value-of select="@max"/>
  </xsl:attribute>
</xsl:if>

<!-- write xform:min restriction -->
<xsl:if test="string-length(@min) > 0 and
             not(@min=$minInfinity) and
             $minIsNumber='false'">
  <xsl:attribute name="xform:minInclusive">
    <xsl:value-of select="@min"/>
  </xsl:attribute>
</xsl:if>

<!-- write xform:precision restriction -->
<xsl:if test="string-length(@precision) > 0 and
             not(@precision=$unlimitedPrecision) and
             $precisionIsNNI='false'">
  <xsl:attribute name="xform:precision">
    <xsl:value-of select="@precision"/>
  </xsl:attribute>
</xsl:if>

<!-- write xform:scale restriction -->
<xsl:if test="string-length(@scale) > 0 and
             not(@scale=$unlimitedScale) and
             $scaleIsNNI='false'">
  <xsl:attribute name="xform:scale">
    <xsl:value-of select="@scale"/>
  </xsl:attribute>
</xsl:if>

```

```
</xsl:attribute>
</xsl:if>

<!-- write xsd:max restriction -->
<xsl:choose>
  <xsl:when test="$maxIsNumber='true'">
    <xsl:element name="xsd:maxInclusive">
      <xsl:attribute name="value">
        <xsl:value-of select="@max"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
  <xsl:when test="count(max) > 0">
    <xsl:element name="xsd:maxInclusive">
      <xsl:attribute name="value">
        <xsl:value-of select="max[1]"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
</xsl:choose>

<!-- write xsd:min restriction -->
<xsl:choose>
  <xsl:when test="$minIsNumber='true'">
    <xsl:element name="xsd:minInclusive">
      <xsl:attribute name="value">
        <xsl:value-of select="@min"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
  <xsl:when test="count(min) > 0">
    <xsl:element name="xsd:minInclusive">
      <xsl:attribute name="value">
        <xsl:value-of select="min[1]"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
</xsl:choose>

<!-- write xsd:precision restriction -->
<xsl:choose>
  <xsl:when test="$precisionIsNNI='true'">
    <xsl:element name="xsd:precision">
      <xsl:attribute name="value">
        <xsl:value-of select="@precision"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
  <xsl:when test="count(precision) > 0">
    <xsl:element name="xsd:precision">
      <xsl:attribute name="value">
        <xsl:value-of select="precision[1]"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
</xsl:choose>
```

```

    </xsl:when>
</xsl:choose>

<!-- write xsd:scale restriction -->
<xsl:choose>
  <xsl:when test="$scaleIsNNI='true'">
    <xsl:element name="xsd:scale">
      <xsl:attribute name="value">
        <xsl:value-of select="@scale"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
  <xsl:when test="count(scale) > 0">
    <xsl:element name="xsd:scale">
      <xsl:attribute name="value">
        <xsl:value-of select="scale[1]"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
</xsl:choose>

</xsl:template>

<!--
currency element
-->

<!-- template:  matches="currency"
function:  converts a currency into a currency.
parameters: none
output:    partially converted definition
-->
<xsl:template match="money">
  <!-- create definition -->
  <xsl:variable name="definition">
    <xsl:element name="xsd:simpleType">
      <xsl:choose>
        <!-- if we have closed range, or no enumeration value, we can
create simple type -->
        <xsl:when test="@enum='closed' or
count(value)=0">
          <xsl:element name="xsd:restriction">
            <xsl:attribute name="base">xform:currency</xsl:attribute>
            <xsl:call-template name="restrictCurrency"/>
            <xsl:apply-templates select="value"/>
          </xsl:element>
        </xsl:when>
        <xsl:otherwise>
          <xsl:element name="xsd:union">
            <xsl:element name="xsd:simpleType">
              <xsl:element name="xsd:restriction">
                <xsl:attribute name="base">xform:currency</xsl:attribute>
                <xsl:call-template name="restrictCurrency"/>
              </xsl:element>
            </xsl:element>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:element>
    </xsl:variable>
  </xsl:template>

```

```

        <xsl:apply-templates select="value"/>
    </xsl:element>
</xsl:element>
<xsl:element name="xsd:simpleType">
    <xsl:element name="xsd:restriction">
        <xsl:attribute name="base">xform:currency</xsl:attribute>
        <xsl:call-template name="restrictCurrency"/>
    </xsl:element>
</xsl:element>
</xsl:element>
</xsl:otherwise>
</xsl:choose>
</xsl:element>
</xsl:variable>

<!-- if we have a name, we can create an element or attribute,
      otherwise we're part of a union -->
<xsl:choose>
    <xsl:when test="string-length(@name) > 0">
        <!-- create element or attribute -->
        <xsl:element name="xsd:element">
            <xsl:attribute name="name">
                <xsl:value-of select="@name"/>
            </xsl:attribute>
            <!-- check common attributes -->
            <xsl:call-template name="checkAttributes"/>

            <xsl:choose>
                <!-- if we have no children, and no other restrictions,
                      we can make a short definition -->
                <xsl:when test='count(child::node())=0 and
                    (string-length(@min)=0 or @min=0) and
                    (string-length(@max)=0 or @max=$maxLengthSpecial) '>
                    <xsl:attribute name="type">xform:currency</xsl:attribute>
                </xsl:when>
                <xsl:otherwise>
                    <!-- dump full definition -->
                    <xsl:copy-of select="$definition"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:element>
    </xsl:when>
    <xsl:otherwise>
        <!-- just dump definition -->
        <xsl:copy-of select="$definition"/>
    </xsl:otherwise>
</xsl:choose>

</xsl:template>

<!-- template:    name="restrictCurrency"
                  function:    add restrictions to a currency
                  parameters:  a currency context node with the following attributes
                               @max

```

```

        @min
        @mask
        and the following child elements
        <min>
        <max>
        <mask>
    output:      elements to represent the restrictions
-->
<xsl:template name="restrictCurrency">
<!-- this is currently not implemented, because it is unclear what min, max
    or mask are supposed to mean -->
</xsl:template>

<!--
money element
-->

<!-- template:  matches="money"
    function:   converts a money into a money.
    parameters: none
    output:     partially converted definition
-->
<xsl:template match="money">
    <!-- only elements of type money can be converted into Schemas -->
    <xsl:choose>
        <xsl:when test="string-length(@name) = 0">
            <xsl:message terminate="no">
                An anonymous money definition is not supported.
            </xsl:message>
            <xsl:comment> Anonymous money definition dropped </xsl:comment>
        </xsl:when>
        <xsl:otherwise>
            <xsl:element name="xsd:element">
                <xsl:attribute name="name">
                    <xsl:value-of select="@name"/>
                </xsl:attribute>
                <!-- check common attributes -->
                <xsl:call-template name="checkAttributes"/>

                <xsl:attribute name="type">xform:money</xsl:attribute>

                <!-- add allowCurrency -->
                <xsl:if test="string-length(@allowCurrency) > 0">
                    <xsl:attribute name="xform:allowCurrency">
                        <xsl:value-of select="@allowCurrency"/>
                    </xsl:attribute>
                </xsl:if>

                <!-- add min value -->
                <xsl:if test="string-length(@min) > 0 and
                    not(@min=$minInfinity)">
                    <xsl:attribute name="xform:minInclusive">
                        <xsl:value-of select="@min"/>
                    </xsl:if>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>

```

```

        </xsl:attribute>
    </xsl:if>

    <!-- add max value -->
    <xsl:if test="string-length(@max) > 0 and
                not(@max=$maxInfinity)">
        <xsl:attribute name="xform:maxInclusive">
            <xsl:value-of select="@max"/>
        </xsl:attribute>
    </xsl:if>

    <!-- add precision value -->
    <xsl:if test="string-length(@precision) > 0 and
                not(@precision=$unlimitedPrecision)">
        <xsl:attribute name="xform:precision">
            <xsl:value-of select="@precision"/>
        </xsl:attribute>
    </xsl:if>

    <!-- add scale value -->
    <xsl:if test="string-length(@scale) > 0 and
                not(@scale=$unlimitedScale)">
        <xsl:attribute name="xform:scale">
            <xsl:value-of select="@scale"/>
        </xsl:attribute>
    </xsl:if>

    </xsl:element>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<!--
date element
-->

<!-- template:    matches="date"
function:        converts a date into an xsd:date.
parameters:      none
output:          converted definition
-->
<xsl:template match="date">
    <!-- create definition -->
    <xsl:variable name="definition">
        <xsl:element name="xsd:simpleType">
            <xsl:choose>
                <!-- if we have closed range, or no enumeration value, we can
                     create simple type -->
                <xsl:when test="@range='closed' or
                                count(value)=0">
                    <xsl:element name="xsd:restriction">
                        <xsl:attribute name="base">xsd:date</xsl:attribute>
                        <xsl:call-template name="restrictDate"/>
                    </xsl:element>
                </xsl:when>
            </xsl:choose>
        </xsl:element>
    </xsl:variable>
    <xsl:element name="date" namespace="xsd" use="replace">
        <xsl:attribute name="type">{ $definition }</xsl:attribute>
    </xsl:element>
</xsl:template>

```

```

        <xsl:apply-templates select="value"/>
    </xsl:element>
</xsl:when>
<xsl:otherwise>
    <xsl:element name="xsd:union">
        <xsl:element name="xsd:simpleType">
            <xsl:element name="xsd:restriction">
                <xsl:attribute name="base">xsd:date</xsl:attribute>
                <xsl:call-template name="restrictDate"/>
                <xsl:apply-templates select="value"/>
            </xsl:element>
        </xsl:element>
        <xsl:element name="xsd:simpleType">
            <xsl:element name="xsd:restriction">
                <xsl:attribute name="base">xsd:date</xsl:attribute>
                <xsl:call-template name="restrictDate"/>
            </xsl:element>
        </xsl:element>
    </xsl:element>
</xsl:otherwise>
</xsl:choose>
</xsl:element>
</xsl:variable>

```

```

<!-- if we have a name, we can create an element or attribute,
      otherwise we're part of a union -->

```

```

<xsl:choose>
    <xsl:when test="string-length(@name) > 0">
        <!-- create element -->
        <xsl:element name="xsd:element">
            <xsl:attribute name="name">
                <xsl:value-of select="@name"/>
            </xsl:attribute>
            <!-- check common attributes -->
            <xsl:call-template name="checkAttributes"/>

            <xsl:choose>
                <!-- if we have no children, and no other restrictions,
                     we can make a short definition -->
                <xsl:when test='count(child::node())=0 and
                                string-length(@min)=0 and
                                string-length(@max)=0 and
                                (string-length(@precision)=0 or
                                 @precision="days")'>
                    <xsl:attribute name="type">xsd:date</xsl:attribute>
                </xsl:when>
                <xsl:otherwise>
                    <!-- dump full definition -->
                    <xsl:copy-of select="$definition"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:element>
    </xsl:when>
    <xsl:otherwise>
        <!-- just dump definition -->

```

```

    <xsl:copy-of select="$definition"/>
  </xsl:otherwise>
</xsl:choose>
</xsl:template>

<!-- template:    name="restrictDate"
function:    add restrictions to a date
parameters: a date context node with the following attributes
              @max
              @min
              @precision
              and the following child elements
              <min>
              <max>
              <precision>
output:      elements to represent the restrictions
-->
<xsl:template name="restrictDate">

  <!-- check if @max is a date -->
  <xsl:variable name="maxIsDate">
    <xsl:call-template name="checkDate">
      <xsl:with-param name="test">
        <xsl:value-of select="@max"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:variable>

  <!-- check if @min is a date -->
  <xsl:variable name="minIsDate">
    <xsl:call-template name="checkDate">
      <xsl:with-param name="test">
        <xsl:value-of select="@min"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:variable>

  <!-- check if precision is valid -->
  <xsl:variable name="precisionIsValid">
    <xsl:call-template name="checkDatePrecision">
      <xsl:with-param name="test">
        <xsl:value-of select="@precision"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:variable>

  <!-- write xform:max restriction -->
  <xsl:if test="string-length(@max) > 0 and
              $maxIsDate='false'">
    <xsl:attribute name="xform:max">
      <xsl:value-of select="@max"/>
    </xsl:attribute>
  </xsl:if>

```

```
<!-- write xform:min restriction -->
<xsl:if test="string-length(@min) &gt; 0 and
    $minIsDate='false'">
  <xsl:attribute name="xform:min">
    <xsl:value-of select="@min"/>
  </xsl:attribute>
</xsl:if>

<!-- write xform:precision restriction -->
<xsl:if test="string-length(@precision) &gt; 0 and
    $precisionIsValid='false'">
  <xsl:attribute name="xform:precision">
    <xsl:value-of select="@precision"/>
  </xsl:attribute>
</xsl:if>

<!-- write xsd:max restriction -->
<xsl:choose>
  <xsl:when test="$maxIsDate='true'">
    <xsl:element name="xsd:max">
      <xsl:attribute name="value">
        <xsl:value-of select="@max"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
  <xsl:when test="count(max) &gt; 0">
    <xsl:element name="xsd:max">
      <xsl:attribute name="value">
        <xsl:value-of select="max[1]"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
</xsl:choose>

<!-- write xsd:min restriction -->
<xsl:choose>
  <xsl:when test="$minIsDate='true'">
    <xsl:element name="xsd:min">
      <xsl:attribute name="value">
        <xsl:value-of select="@min"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
  <xsl:when test="count(min) &gt; 0">
    <xsl:element name="xsd:min">
      <xsl:attribute name="value">
        <xsl:value-of select="min[1]"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
</xsl:choose>

<!-- write xsd:duration restriction -->
<xsl:choose>
  <xsl:when test="$precisionIsValid='true'">
```

```

    <xsl:call-template name="writeDatePrecision">
      <xsl:with-param name="precision">
        <xsl:value-of select="@precision"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:when>
  <xsl:when test="count(precision) > 0">
    <xsl:call-template name="writeDatePrecision">
      <xsl:with-param name="precision">
        <xsl:value-of select="precision[1]"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:when>
</xsl:choose>
</xsl:template>

<!-- template:   name="writeDatePrecision"
function:       writes the definitions to matche the precision
                attribute
parameters:    $precision: one of years, months, days
output:        proper definition to allow only the requested
                precision
-->
<xsl:template name="writeDatePrecision">
  <xsl:param name="precision"/>

  <xsl:choose>
    <xsl:when test="$precision='years'">
      <xsl:element name="xsd:duration">
        <xsl:attribute name="value">P1Y</xsl:attribute>
      </xsl:element>
      <xsl:element name="xsd:pattern">
        <xsl:attribute name="value">\d*\d{4}</xsl:attribute>
      </xsl:element>
    </xsl:when>
    <xsl:when test="$precision='months'">
      <xsl:element name="xsd:duration">
        <xsl:attribute name="value">P1M</xsl:attribute>
      </xsl:element>
      <xsl:element name="xsd:pattern">
        <xsl:attribute name="value">\d*\d{4}-\d{2}</xsl:attribute>
      </xsl:element>
    </xsl:when>
  </xsl:choose>
</xsl:template>

<!--
time element
-->

<!-- template:   matches="time"
function:        converts a time into an xsd:time.

```

```

parameters: none
output:      converted definition
-->
<xsl:template match="time">
  <!-- create definition -->
  <xsl:variable name="definition">
    <xsl:element name="xsd:simpleType">
      <xsl:choose>
        <!-- if we have closed range, or no enumeration value, we can
              create simple type -->
        <xsl:when test="@range='closed' or
                      count(value)=0">
          <xsl:element name="xsd:restriction">
            <xsl:attribute name="base">xsd:time</xsl:attribute>
            <xsl:call-template name="restrictTime"/>
            <xsl:apply-templates select="value"/>
          </xsl:element>
        </xsl:when>
        <xsl:otherwise>
          <xsl:element name="xsd:union">
            <xsl:element name="xsd:simpleType">
              <xsl:element name="xsd:restriction">
                <xsl:attribute name="base">xsd:time</xsl:attribute>
                <xsl:call-template name="restrictTime"/>
                <xsl:apply-templates select="value"/>
              </xsl:element>
            </xsl:element>
            <xsl:element name="xsd:simpleType">
              <xsl:element name="xsd:restriction">
                <xsl:attribute name="base">xsd:time</xsl:attribute>
                <xsl:call-template name="restrictTime"/>
              </xsl:element>
            </xsl:element>
          </xsl:element>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:element>
  </xsl:variable>

  <!-- if we have a name, we can create an element or attribute,
        otherwise we're part of a union -->
  <xsl:choose>
    <xsl:when test="string-length(@name) > 0">
      <!-- create element -->
      <xsl:element name="xsd:element">
        <xsl:attribute name="name">
          <xsl:value-of select="@name"/>
        </xsl:attribute>
        <!-- check common attributes -->
        <xsl:call-template name="checkAttributes"/>

        <xsl:choose>
          <!-- if we have no children, and no other restrictions,
                we can make a short definition -->
          <xsl:when test='count(child::node())=0 and

```

```

        string-length(@min)=0 and
        string-length(@max)=0 and
        (string-length(@precision)=0 or
         @precision="seconds")'>
    <xsl:attribute name="type">xsd:time</xsl:attribute>
</xsl:when>
<xsl:otherwise>
    <!-- dump full definition -->
    <xsl:copy-of select="$definition"/>
</xsl:otherwise>
</xsl:choose>
</xsl:element>
</xsl:when>
<xsl:otherwise>
    <!-- just dumpe definition -->
    <xsl:copy-of select="$definition"/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<!-- template:    name="restrictTime"
function:        add erstrictions to a time
parameters:      a time context node with the following attributes
                  @max
                  @min
                  @precision
                  and the following child elements
                  <min>
                  <max>
                  <precision>
output:          elements to represent the restrictions
-->
<xsl:template name="restrictTime">

    <!-- check if max is a time -->
    <xsl:variable name="maxIsTime">
        <xsl:call-template name="checkTime">
            <xsl:with-param name="test">
                <xsl:value-of select="@max"/>
            </xsl:with-param>
        </xsl:call-template>
    </xsl:variable>

    <!-- check if min is a time -->
    <xsl:variable name="minIsTime">
        <xsl:call-template name="checkTime">
            <xsl:with-param name="test">
                <xsl:value-of select="@min"/>
            </xsl:with-param>
        </xsl:call-template>
    </xsl:variable>

    <!-- check if precision is valid -->
    <xsl:variable name="precisionIsValid">

```

```

<xsl:call-template name="checkTimePrecision">
  <xsl:with-param name="test">
    <xsl:value-of select="@precision"/>
  </xsl:with-param>
</xsl:call-template>
</xsl:variable>

<!-- write xform:max restriction -->
<xsl:if test="string-length(@max) > 0 and
             $maxIsTime='false'">
  <xsl:attribute name="xform:max">
    <xsl:value-of select="@max"/>
  </xsl:attribute>
</xsl:if>

<!-- write xform:min restriction -->
<xsl:if test="string-length(@min) > 0 and
             $minIsTime='false'">
  <xsl:attribute name="xform:min">
    <xsl:value-of select="@min"/>
  </xsl:attribute>
</xsl:if>

<!-- write xform:precision restriction -->
<xsl:if test="string-length(@precision) > 0 and
             $precisionIsValid='false'">
  <xsl:attribute name="xform:precision">
    <xsl:value-of select="@precision"/>
  </xsl:attribute>
</xsl:if>

<!-- write xsd:max restriction -->
<xsl:choose>
  <xsl:when test="$maxIsTime='true'">
    <xsl:element name="xsd:max">
      <xsl:attribute name="value">
        <xsl:value-of select="@max"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
  <xsl:when test="count(max) > 0">
    <xsl:element name="xsd:max">
      <xsl:attribute name="value">
        <xsl:value-of select="max[1]"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
</xsl:choose>

<!-- write xsd:min restriction -->
<xsl:choose>
  <xsl:when test="$minIsTime='true'">
    <xsl:element name="xsd:min">
      <xsl:attribute name="value">
        <xsl:value-of select="@min"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
</xsl:choose>

```

```

        </xsl:attribute>
    </xsl:element>
</xsl:when>
<xsl:when test="count(min) > 0">
    <xsl:element name="xsd:min">
        <xsl:attribute name="value">
            <xsl:value-of select="min[1]"/>
        </xsl:attribute>
    </xsl:element>
</xsl:when>
</xsl:choose>

<!-- write xsd:duration restriction -->
<xsl:choose>
    <xsl:when test="$precisionIsValid='true'">
        <xsl:call-template name="writeTimePrecision">
            <xsl:with-param name="precision">
                <xsl:value-of select="@precision"/>
            </xsl:with-param>
        </xsl:call-template>
    </xsl:when>
    <xsl:when test="count(precision) > 0">
        <xsl:call-template name="writeTimePrecision">
            <xsl:with-param name="precision">
                <xsl:value-of select="precision[1]"/>
            </xsl:with-param>
        </xsl:call-template>
    </xsl:when>
</xsl:choose>
</xsl:template>

<!-- template:    name="writeTimePrecision"
function:        writes the definitions to matche the precision
                  attribute
parameters:     $precision: one of hours, minutes, seconds
output:         proper definition to allow only the requested
                  precision
-->
<xsl:template name="writeTimePrecision">
    <xsl:param name="precision"/>

    <xsl:choose>
        <xsl:when test="$precision='hours'">
            <xsl:element name="xsd:duration">
                <xsl:attribute name="value">PT60M</xsl:attribute>
            </xsl:element>
            <xsl:element name="xsd:pattern">
                <xsl:attribute name="value">
                    <xsl:text>([01][0-9]|2[0-3])(:00(:00(.000)?)?)?</xsl:text>
                </xsl:attribute>
            </xsl:element>
        </xsl:when>
        <xsl:when test="$precision='minutes'">
            <xsl:element name="xsd:duration">

```

```

    <xsl:attribute name="value">PT60S</xsl:attribute>
  </xsl:element>
  <xsl:element name="xsd:pattern">
    <xsl:attribute name="value">
      <xsl:text>([01][0-9]|2[0-3])</xsl:text>
      <xsl:text>(:[0-5][0-9](:00(.000)?)?)?</xsl:text>
    </xsl:attribute>
  </xsl:element>
</xsl:when>
</xsl:choose>
</xsl:template>

<!--
duration element
-->

<!-- template:  matches="duration"
function:      converts a duration into an xsd:timeDuration.
parameters:   none
output:       converted definition
-->
<xsl:template match="duration">
  <!-- create definition -->
  <xsl:variable name="definition">
    <xsl:element name="xsd:simpleType">
      <xsl:choose>
        <!-- if we have closed range, or no enumeration value, we can
              create simple type -->
        <xsl:when test="@range='closed' or
                      count(value)=0">
          <xsl:element name="xsd:restriction">
            <xsl:attribute name="base">xsd:timeDuration</xsl:attribute>
            <xsl:call-template name="restrictDuration"/>
            <xsl:apply-templates select="value"/>
          </xsl:element>
        </xsl:when>
        <xsl:otherwise>
          <xsl:element name="xsd:union">
            <xsl:element name="xsd:simpleType">
              <xsl:element name="xsd:restriction">
                <xsl:attribute name="base">
                  <xsl:text>xsd:timeDuration</xsl:text>
                </xsl:attribute>
                <xsl:call-template name="restrictDuration"/>
                <xsl:apply-templates select="value"/>
              </xsl:element>
            </xsl:element>
            <xsl:element name="xsd:simpleType">
              <xsl:element name="xsd:restriction">
                <xsl:attribute name="base">
                  <xsl:text>xsd:timeDuration</xsl:text>
                </xsl:attribute>
                <xsl:call-template name="restrictDuration"/>
              </xsl:element>
            </xsl:element>
          </xsl:union>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:element>
  </xsl:variable>
  <xsl:element name="xsd:simpleType">
    <xsl:attribute name="base">xsd:timeDuration</xsl:attribute>
    <xsl:call-template name="restrictDuration"/>
  </xsl:element>

```

```

        </xsl:element>
    </xsl:element>
</xsl:element>
</xsl:otherwise>
</xsl:choose>
</xsl:element>
</xsl:variable>

<!-- if we have a name, we can create an element or attribute,
      otherwise we're part of a union -->
<xsl:choose>
  <xsl:when test="string-length(@name) > 0">
    <!-- create element or attribute -->
    <xsl:element name="xsd:{@as}">
      <xsl:attribute name="name">
        <xsl:value-of select="@name"/>
      </xsl:attribute>
      <!-- check common attributes -->
      <xsl:call-template name="checkAttributes"/>

      <xsl:choose>
        <!-- if we have no children, and no other restrictions,
              we can make a short definition -->
        <xsl:when test='count(child::node())=0 and
                      string-length(@min)=0 and
                      string-length(@max)=0 and
                      (string-length(@precision)=0 or
                       @precision="seconds")'>
          <xsl:attribute name="type">xsd:timeDuration</xsl:attribute>
        </xsl:when>
        <xsl:otherwise>
          <!-- dump full definition -->
          <xsl:copy-of select="$definition"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:element>
  </xsl:when>
  <xsl:otherwise>
    <!-- just dump definition -->
    <xsl:copy-of select="$definition"/>
  </xsl:otherwise>
</xsl:choose>
</xsl:template>

<!-- template:      name="restrictDuration"
      function:      add restrictions to a duration
      parameters:    a duration context node with the following attributes
                     @max
                     @min
                     @precision
                     and the following child elements
                     <min>
                     <max>
                     <precision>

```

output: elements to represent the restrictions

```
-->
<xsl:template name="restrictDuration">

  <!-- check if max is a duration -->
  <xsl:variable name="maxIsDuration">
    <xsl:call-template name="checkDuration">
      <xsl:with-param name="test">
        <xsl:value-of select="@max"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:variable>

  <!-- check if min is a duration -->
  <xsl:variable name="minIsDuration">
    <xsl:call-template name="checkDuration">
      <xsl:with-param name="test">
        <xsl:value-of select="@min"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:variable>

  <!-- check if precision is valid -->
  <xsl:variable name="precisionIsValid">
    <xsl:variable name="test">
      <xsl:call-template name="checkDatePrecision">
        <xsl:with-param name="test">
          <xsl:value-of select="@precision"/>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:variable>

    <xsl:choose>
      <xsl:when test="$test='false'">
        <xsl:call-template name="checkTimePrecision">
          <xsl:with-param name="test">
            <xsl:value-of select="@precision"/>
          </xsl:with-param>
        </xsl:call-template>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$test"/>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

  <!-- write xform:max restriction -->
  <xsl:if test="string-length(@max) > 0 and
    $maxIsDuration='false'">
    <xsl:attribute name="xform:max">
      <xsl:value-of select="@max"/>
    </xsl:attribute>
  </xsl:if>

  <!-- write xform:min restriction -->
```

```

<xsl:if test="string-length(@min) > 0 and
    $minIsDuration='false'">
  <xsl:attribute name="xform:min">
    <xsl:value-of select="@min"/>
  </xsl:attribute>
</xsl:if>

<!-- write xform:precision restriction -->
<xsl:if test="string-length(@precision) > 0 and
    $precisionIsValid='false'">
  <xsl:attribute name="xform:precision">
    <xsl:value-of select="@precision"/>
  </xsl:attribute>
</xsl:if>

<!-- write xsd:max restriction -->
<xsl:choose>
  <xsl:when test="$maxIsDuration='true'">
    <xsl:element name="xsd:max">
      <xsl:attribute name="value">
        <xsl:value-of select="@max"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
  <xsl:when test="count(max) > 0">
    <xsl:element name="xsd:max">
      <xsl:attribute name="value">
        <xsl:value-of select="max[1]"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
</xsl:choose>

<!-- write xsd:min restriction -->
<xsl:choose>
  <xsl:when test="$minIsDuration='true'">
    <xsl:element name="xsd:min">
      <xsl:attribute name="value">
        <xsl:value-of select="@min"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
  <xsl:when test="count(min) > 0">
    <xsl:element name="xsd:min">
      <xsl:attribute name="value">
        <xsl:value-of select="min[1]"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
</xsl:choose>

<!-- write xsd:duration restriction -->
<xsl:choose>
  <xsl:when test="$precisionIsValid='true'">
    <xsl:call-template name="writeDurationPrecision">

```

```

        <xsl:with-param name="precision">
            <xsl:value-of select="@precision"/>
        </xsl:with-param>
    </xsl:call-template>
</xsl:when>
<xsl:when test="count(precision) > 0">
    <xsl:call-template name="writeDurationPrecision">
        <xsl:with-param name="precision">
            <xsl:value-of select="precision[1]"/>
        </xsl:with-param>
    </xsl:call-template>
</xsl:when>
</xsl:choose>
</xsl:template>

<!-- template:    name="writeDurationPrecision"
function:        writes the definitions to matche the precision
                 attribute
parameters:     $precision: one of hours, minutes, seconds
output:         proper definition to allow only the requested
                 precision
-->
<xsl:template name="writeDurationPrecision">
    <xsl:param name="precision"/>

    <xsl:choose>
        <xsl:when test="$precision='years'">
            <xsl:element name="xsd:pattern">
                <xsl:attribute name="value">
                    <xsl:text>P(\d+(\.\d+)?|\.\d+)Y</xsl:text>
                </xsl:attribute>
            </xsl:element>
        </xsl:when>
        <xsl:when test="$precision='months'">
            <xsl:element name="xsd:pattern">
                <xsl:attribute name="value">
                    <xsl:text>P(\d+(\.\d+)?|\.\d+)Y|</xsl:text>
                    <xsl:text>P(\d+Y)?(\d+(\.\d+)?|\.\d+)M</xsl:text>
                </xsl:attribute>
            </xsl:element>
        </xsl:when>
        <xsl:when test="$precision='days'">
            <xsl:element name="xsd:pattern">
                <xsl:attribute name="value">
                    <xsl:text>P(\d+(\.\d+)?|\.\d+)Y|</xsl:text>
                    <xsl:text>P(\d+Y)?(\d+(\.\d+)?|\.\d+)M|</xsl:text>
                    <xsl:text>P(\d+Y)?(\d+M)?(\d+(\.\d+)?|\.\d+)D</xsl:text>
                </xsl:attribute>
            </xsl:element>
        </xsl:when>
        <xsl:when test="$precision='hours'">
            <xsl:element name="xsd:pattern">
                <xsl:attribute name="value">
                    <xsl:text>P(\d+(\.\d+)?|\.\d+)Y|</xsl:text>

```

```

        <xsl:text>P(\d+Y)?(\d+(\.\d+)?)|\.\d+)M|</xsl:text>
        <xsl:text>P(\d+Y)?(\d+M)?(\d+(\.\d+)?)|\.\d+)D|</xsl:text>
        <xsl:text>P(\d+Y)?(\d+M)?(\d+D)?T(\d+(\.\d+)?)|</xsl:text>
        <xsl:text>\.\d+)H</xsl:text>
    </xsl:attribute>
</xsl:element>
</xsl:when>
<xsl:when test="$precision='minutes'">
    <xsl:element name="xsd:pattern">
        <xsl:attribute name="value">
            <xsl:text>P(\d+(\.\d+)?)|\.\d+)Y|</xsl:text>
            <xsl:text>P(\d+Y)?(\d+(\.\d+)?)|\.\d+)M|</xsl:text>
            <xsl:text>P(\d+Y)?(\d+M)?(\d+(\.\d+)?)|\.\d+)D|</xsl:text>
            <xsl:text>P(\d+Y)?(\d+M)?(\d+D)?T(\d+(\.\d+)?)|</xsl:text>
            <xsl:text>\.\d+)H|</xsl:text>
            <xsl:text>P(\d+Y)?(\d+M)?(\d+D)?T(\d+H)?</xsl:text>
            <xsl:text>(\d(\.\d+)?)|\.\d+)M</xsl:text>
        </xsl:attribute>
    </xsl:element>
</xsl:when>
</xsl:choose>
</xsl:template>

<!--
uri element
-->

<!-- template: matches="uri"
function: converts a uri into an xsd:uriReference.
parameters: none
output: converted definition
-->
<xsl:template match="uri">
    <!-- create definition -->
    <xsl:variable name="definition">
        <xsl:element name="xsd:simpleType">
            <xsl:choose>
                <!-- if we have closed range, or no enumeration value, we can
                     create simple type -->
                <xsl:when test="@range='closed' or
                               count(value)=0">
                    <xsl:element name="xsd:restriction">
                        <xsl:attribute name="base">xsd:uriReference</xsl:attribute>
                        <xsl:call-template name="restrictUri"/>
                        <xsl:apply-templates select="value"/>
                    </xsl:element>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:element name="xsd:union">
                        <xsl:element name="xsd:simpleType">
                            <xsl:element name="xsd:restriction">
                                <xsl:attribute name="base">
                                    <xsl:text>xsd:uriReference</xsl:text>

```

```

        </xsl:attribute>
        <xsl:call-template name="restrictUri"/>
        <xsl:apply-templates select="value"/>
    </xsl:element>
</xsl:element>
<xsl:element name="xsd:simpleType">
    <xsl:element name="xsd:restriction">
        <xsl:attribute name="base">
            <xsl:text>xsd:uriReference</xsl:text>
        </xsl:attribute>
        <xsl:call-template name="restrictUri"/>
    </xsl:element>
</xsl:element>
</xsl:element>
</xsl:otherwise>
</xsl:choose>
</xsl:element>
</xsl:variable>

<!-- if we have a name, we can create an element or attribute,
      otherwise we're part of a union -->
<xsl:choose>
    <xsl:when test="string-length(@name) > 0">
        <!-- create element -->
        <xsl:element name="xsd:element">
            <xsl:attribute name="name">
                <xsl:value-of select="@name"/>
            </xsl:attribute>
            <!-- check common attributes -->
            <xsl:call-template name="checkAttributes"/>

            <xsl:choose>
                <!-- if we have no children, and no other restrictions,
                      we can make a short definition -->
                <xsl:when test='count(child::node())=0 and
                               string-length(@scheme)=0'>
                    <xsl:attribute name="type">xsd:uriReference</xsl:attribute>
                </xsl:when>
                <xsl:otherwise>
                    <!-- dump full definition -->
                    <xsl:copy-of select="$definition"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:element>
    </xsl:when>
    <xsl:otherwise>
        <!-- just dump definition -->
        <xsl:copy-of select="$definition"/>
    </xsl:otherwise>
</xsl:choose>
</xsl:template>

<!-- template:    name="restrictUri"
      function:    add restrictions to a uri

```

```

parameters: a duration context node with the following attributes
              @scheme
              and the following child elements
              <scheme>
output:      elements to represent the restrictions
-->
<xsl:template name="restrictUri">

  <!-- make pattern from scheme -->
  <xsl:variable name="pattern">
    <xsl:for-each select="scheme">
      <xsl:call-template name="convertScheme">
        <xsl:with-param name="scheme">
          <xsl:value-of select="."/>
        </xsl:with-param>
      </xsl:call-template>
      <xsl:if test="not(position()=last())">
        <xsl:text>|</xsl:text>
      </xsl:if>
    </xsl:for-each>

    <xsl:if test="@scheme">
      <xsl:if test="count(scheme) > 0">
        <xsl:text>|</xsl:text>
      </xsl:if>
      <xsl:call-template name="convertScheme">
        <xsl:with-param name="scheme">
          <xsl:value-of select="@scheme"/>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:if>
  </xsl:variable>

  <xsl:if test="string-length($pattern) > 0">
    <xsl:element name="xsd:pattern">
      <xsl:attribute name="value">
        <xsl:value-of select="$pattern"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:if>
</xsl:template>

<!-- template:   name="convertScheme"
function:       converts a scheme to a pattern
parameters:    scheme      : the scheme to transform
output:        a pattern with the same meaning
-->
<xsl:template name="convertScheme">
  <xsl:param name="scheme"/>

  <xsl:choose>
    <xsl:when test="contains($scheme, ' ')">
      <!-- splitt scheme -->
      <xsl:call-template name="convertScheme">

```

```

    <xsl:with-param name="scheme">
      <xsl:value-of select="substring-before($scheme, ' ')" />
    </xsl:with-param>
  </xsl:call-template>
  <xsl:text>|</xsl:text>
  <xsl:call-template name="convertScheme">
    <xsl:with-param name="scheme">
      <xsl:value-of select="substring-after($scheme, ' ')" />
    </xsl:with-param>
  </xsl:call-template>
</xsl:when>
<xsl:otherwise>
  <xsl:text>(</xsl:text>
  <xsl:choose>
    <xsl:when test="$scheme='mailto'">
      <xsl:value-of select="$scheme" />
      <xsl:text>:.*</xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$scheme" />
      <xsl:text>://.*</xsl:text>
    </xsl:otherwise>
  </xsl:choose>
  <xsl:text>)</xsl:text>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

```

```

<!--
binary element
-->

```

```

<!-- template: matches="binary"
function: converts a binary into an xsd:binary.
parameters: none
output: converted definition
-->

```

```

<xsl:template match="binary">
  <!-- create definition -->
  <xsl:variable name="definition">
    <xsl:element name="xsd:simpleType">
      <xsl:element name="xsd:restriction">
        <xsl:attribute name="base">xsd:binary</xsl:attribute>
        <xsl:call-template name="restrictBinary" />
      </xsl:element>
    </xsl:element>
  </xsl:variable>

```

```

  <!-- if we have a name, we can create an element or attribute,
otherwise we're part of a union -->
  <xsl:choose>
    <xsl:when test="string-length(@name) > 0">
      <!-- create element -->

```

```

<xsl:element name="xsd:element">
  <xsl:attribute name="name">
    <xsl:value-of select="@name"/>
  </xsl:attribute>
  <!-- check common attributes -->
  <xsl:call-template name="checkAttributes"/>

  <xsl:choose>
    <!-- if we have no children, and no other restrictions,
         we can make a short definition -->
    <xsl:when test='count(child::node())=0 and
                  string-length(@type)=0'>
      <xsl:attribute name="type">xsd:binary</xsl:attribute>
    </xsl:when>
    <xsl:otherwise>
      <!-- dump full definition -->
      <xsl:copy-of select="$definition"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:element>
</xsl:when>
<xsl:otherwise>
  <!-- just dump definition -->
  <xsl:copy-of select="$definition"/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<!-- template:   name="restrictBinary"
function:      add restrictions to a binary
parameters:   a duration context node with the following attributes
               @length
               @max
               @min
               @mediaType
               and the following child elements
               <length>
               <min>
               <max>
               <mediaType>
output:       elements to represent the restrictions
-->
<xsl:template name="restrictBinary">

  <!-- convert list of types into media attribute -->
  <xsl:attribute name="xform:mediaType">
    <xsl:for-each select="mediaType">
      <xsl:variable name="isMedia">
        <xsl:call-template name="checkMedia">
          <xsl:with-param name="test">
            <xsl:value-of select="."/>
          </xsl:with-param>
        </xsl:call-template>
      </xsl:variable>
    </xsl:for-each>
  </xsl:attribute>

```

```

    <xsl:if test="$isMedia='true'">
      <xsl:value-of select="."/>
      <xsl:if test="not(position()=last())">
        <xsl:text> </xsl:text>
      </xsl:if>
    </xsl:if>
  </xsl:for-each>

  <xsl:if test="not(string-length(@mediaType)=0)">
    <xsl:text> </xsl:text>
    <xsl:variable name="isMedia">
      <xsl:call-template name="checkMedia">
        <xsl:with-param name="test">
          <xsl:value-of select="@mediaType"/>
        </xsl:with-param>
      </xsl:call-template>
    </xsl:variable>
    <xsl:if test="isMedia='true'">
      <xsl:value-of select="@mediaType"/>
    </xsl:if>
  </xsl:if>
</xsl:attribute>

<!-- check if @length is an non negative integer -->
<xsl:variable name="lengthIsNNI">
  <xsl:call-template name="checkNonNegInt">
    <xsl:with-param name="test">
      <xsl:value-of select="@length"/>
    </xsl:with-param>
  </xsl:call-template>
</xsl:variable>

<!-- check if @max is an non negative integer -->
<xsl:variable name="maxIsNNI">
  <xsl:call-template name="checkNonNegInt">
    <xsl:with-param name="test">
      <xsl:value-of select="@max"/>
    </xsl:with-param>
  </xsl:call-template>
</xsl:variable>

<!-- check if @min as an non negative integer -->
<xsl:variable name="minIsNNI">
  <xsl:call-template name="checkNonNegInt">
    <xsl:with-param name="test">
      <xsl:value-of select="@min"/>
    </xsl:with-param>
  </xsl:call-template>
</xsl:variable>

<!-- write xform:length restriction -->
<xsl:if test="string-length(@length) > 0 and
  not(@length=$maxLengthSpecial) and
  $lengthIsNNI='false'">
  <xsl:attribute name="xform:length">

```

```

    <xsl:value-of select="@length"/>
  </xsl:attribute>
</xsl:if>

<!-- write xform:maxLength restriction -->
<xsl:if test="string-length(@max) > 0 and
             not(@max=$maxLengthSpecial) and
             $maxIsNNI='false'">
  <xsl:attribute name="xform:maxLength">
    <xsl:value-of select="@max"/>
  </xsl:attribute>
</xsl:if>

<!-- write xform:minLength restriction -->
<xsl:if test="string-length(@min) > 0 and
             not(@min='0') and
             $minIsNNI='false'">
  <xsl:attribute name="xform:minLength">
    <xsl:value-of select="@min"/>
  </xsl:attribute>
</xsl:if>

<!-- write xsd:length restriction -->
<xsl:choose>
  <xsl:when test="$lengthIsNNI='true'">
    <xsl:if test="not(@length=$maxLengthSpecial)">
      <xsl:element name="xsd:length">
        <xsl:attribute name="value">
          <xsl:value-of select="@length"/>
        </xsl:attribute>
      </xsl:element>
    </xsl:if>
  </xsl:when>
  <xsl:when test="count(length) > 0">
    <xsl:element name="xsd:length">
      <xsl:attribute name="value">
        <xsl:value-of select="length[1]"/>
      </xsl:attribute>
    </xsl:element>
  </xsl:when>
</xsl:choose>

<!-- write xsd:maxLength restriction -->
<xsl:choose>
  <xsl:when test="$maxIsNNI='true'">
    <xsl:if test="not(@max=$maxLengthSpecial)">
      <xsl:element name="xsd:maxLength">
        <xsl:attribute name="value">
          <xsl:value-of select="@max"/>
        </xsl:attribute>
      </xsl:element>
    </xsl:if>
  </xsl:when>
  <xsl:when test="count(max) > 0">
    <xsl:element name="xsd:maxLength">

```

```

        <xsl:attribute name="value">
            <xsl:value-of select="max[1]"/>
        </xsl:attribute>
    </xsl:element>
</xsl:when>
</xsl:choose>

<!-- write xsd:minLength restriction -->
<xsl:choose>
    <xsl:when test="$minIsNNI='true'">
        <xsl:if test="not(@min='0')">
            <xsl:element name="xsd:minLength">
                <xsl:attribute name="value">
                    <xsl:value-of select="@min"/>
                </xsl:attribute>
            </xsl:element>
        </xsl:if>
    </xsl:when>
    <xsl:when test="count(min) > 0">
        <xsl:element name="xsd:min">
            <xsl:attribute name="value">
                <xsl:value-of select="min[1]"/>
            </xsl:attribute>
        </xsl:element>
    </xsl:when>
</xsl:choose>
</xsl:template>

<!--
element element
-->

<!-- template:    matches="element"
function:        converts a element into an xsd:element.
parameters:     none
output:         converted definition
-->
<xsl:template match="element">
    <xsl:choose>
        <xsl:when test="string-length(@name)=0">
            <xsl:message terminate="no">
                An anonymous element definition is not supported.
            </xsl:message>
            <xsl:comment> Anonymous element definition dropped </xsl:comment>
        </xsl:when>
        <xsl:otherwise>
            <xsl:element name="xsd:element">
                <xsl:attribute name="name">
                    <xsl:value-of select="@name"/>
                </xsl:attribute>

                <!-- check common attributes -->
                <xsl:call-template name="checkAttributes"/>
            </xsl:element>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>

```

```

        <xsl:attribute name="type">
            <xsl:value-of select="@type"/>
        </xsl:attribute>
    </xsl:element>
</xsl:otherwise>
</xsl:choose>
</xsl:template>

<!--
value element
-->

<!-- template:    matches="value"
function:        converte value into an enumeration.
parameters:      none
output:          <enumeration> that represents the value
-->
<xsl:template match="value">
    <xsl:element name="xsd:enumeration">
        <xsl:attribute name="value">
            <xsl:value-of select="."/>
        </xsl:attribute>
    </xsl:element>
</xsl:template>

<!--
Named templates
-->

<!--
Datatype checks
-->

<!-- template:    name="checkXpression"
function:         checks if a given string is an Xpression or not
parameter:       test
output:          true   if it's an Xpression
                 false  otherwise
note:           this can not work, it must always return true.
                 But perhapes it can be made to work later
-->
<xsl:template name="checkXpression">
    <xsl:param name="test"/>

    <!-- we can not test this, so we must return true -->
    <xsl:text>true</xsl:text>
</xsl:template>

<!-- template:    name="checkBoolean"

```

```

function:  check if $test is a boolean
parameters: test
output:    true   if it's a boolean
           false  otherwise
-->
<xsl:template name="checkBoolean">
  <xsl:param name="test"/>

  <xsl:choose>
    <xsl:when test="true">
      <xsl:text>true</xsl:text>
    </xsl:when>
    <xsl:when test="false">
      <xsl:text>false</xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text>false</xsl:text>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!-- template:  name="checkNumber"
function:  check if $test is a number
parameters: test
output:    true   if it's a number
           false  otherwise
-->
<xsl:template name="checkNumber">
  <xsl:param name="test"/>

  <xsl:variable name="number">
    <xsl:value-of select="number($test)"/>
  </xsl:variable>

  <xsl:choose>
    <xsl:when test="$number='NaN'">
      <xsl:text>false</xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:text>true</xsl:text>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<!-- template:  name="checkNonNegInt"
function:  check if $test is a non negativ integer
parameters: test
output:    true   if it's a non negativ integer
           false  otherwise
-->
<xsl:template name="checkNonNegInt">
  <xsl:param name="test"/>

```

```

<xsl:variable name="number">
  <xsl:value-of select="number($test)"/>
</xsl:variable>

<xsl:choose>
  <xsl:when test="$number='NaN'">
    <xsl:text>>false</xsl:text>
  </xsl:when>
  <xsl:otherwise>
    <xsl:variable name="integer">
      <xsl:value-of select="round($number)"/>
    </xsl:variable>
    <xsl:choose>
      <xsl:when test="not($integer=$number)">
        <xsl:text>>false</xsl:text>
      </xsl:when>
      <xsl:otherwise>
        <xsl:choose>
          <xsl:when test="$integer < 0">
            <xsl:text>>false</xsl:text>
          </xsl:when>
          <xsl:otherwise>
            <xsl:text>>true</xsl:text>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:otherwise>
</xsl:choose>
</xsl:template>

<!-- template:   name="checkDate"
function:   checks if a given string is a date or not
parameter:  test    :the string to test
output:     true    if it's a date
            false   otherwise
note:       this does not work, it always returns false
-->
<xsl:template name="checkDate">
  <xsl:param name="test"/>

  <xsl:text>>false</xsl:text>
</xsl:template>

<!-- template:   name="checkTime"
function:   checks if a given string is a time or not
parameter:  test    :the string to test
output:     true    if it's a time
            false   otherwise
note:       this does not work, it always returns false
-->
<xsl:template name="checkTime">
  <xsl:param name="test"/>

```

```

    <xsl:text>>false</xsl:text>
</xsl:template>

<!-- template:    name="checkDuration"
    function:     checks if a given string is a duration or not
    parameter:    test    :the string to test
    output:       true    if it's a duration
                 false   otherwise
    note:         this does not work, it always returns false
-->
<xsl:template name="checkDuration">
    <xsl:param name="test"/>

    <xsl:text>>false</xsl:text>
</xsl:template>

<!-- template:    name="checkDatePrecision"
    function:     checks if a given string is a valid precision for a
                 date
    parameter:    test
    output:       true    if it is
                 false   otherwise
-->
<xsl:template name="checkDatePrecision">
    <xsl:param name="test"/>

    <xsl:choose>
        <xsl:when test="$test='years'">
            <xsl:text>>true</xsl:text>
        </xsl:when>
        <xsl:when test="$test='months'">
            <xsl:text>>true</xsl:text>
        </xsl:when>
        <xsl:when test="$test='dayss'">
            <xsl:text>>true</xsl:text>
        </xsl:when>
        <xsl:otherwise>
            <xsl:text>>false</xsl:text>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>

<!-- template:    name="checkTimePrecision"
    function:     checks if a given string is a valid precision for a
                 time
    parameter:    test
    output:       true    if it is
                 false   otherwise
-->
<xsl:template name="checkTimePrecision">
    <xsl:param name="test"/>

```

```

<xsl:choose>
  <xsl:when test="$test='hours'">
    <xsl:text>true</xsl:text>
  </xsl:when>
  <xsl:when test="$test='minutes'">
    <xsl:text>true</xsl:text>
  </xsl:when>
  <xsl:when test="$test='seconds'">
    <xsl:text>true</xsl:text>
  </xsl:when>
  <xsl:otherwise>
    <xsl:text>>false</xsl:text>
  </xsl:otherwise>
</xsl:choose>
</xsl:template>

<!-- template:   name="checkMedia"
function:   checks is a string is a valid mimetype
parameters: test      : the string to test
output:    true       : is it's a valid mimetype
           false      : otherwise
note:      does not work, always returns false
-->
<xsl:template name="checkMedia">
  <xsl:param name="test"/>

  <xsl:text>>false</xsl:text>
</xsl:template>

<!--
Common attributes
-->

<!-- template:   name="checkAttributes"
function:   check the common XForms attributes and produces
            apropiated attributes
parameters: a context node, with the following attributes
            @id      ( )
            @required (false)
            @relevant (false)
            @readOnly (false)
            @calc     ( )
            @choices  ( )
            @validate ( )
glob. var.: $maxOccursSpecial
output:     transformation of attributes
-->
<xsl:template name="checkAttributes">
  <!-- copy ID, if any -->
  <xsl:if test="string-length(@id) &gt; 0">
    <xsl:attribute name="id">

```

```

    <xsl:value-of select="@id"/>
  </xsl:attribute>
</xsl:if>

<!-- check if this element is optional, it is optional,
      if: it's not relevant or it's not required, or it's
      minOccurs = 0 -->
<xsl:choose>
  <xsl:when test="@relevant='true'">
    <xsl:choose>
      <xsl:when test="@required='true'">
        <xsl:if test="not(@minOccurs='1')">
          <!-- check if $minOccurs if a non negative integer -->
          <xsl:variable name="nn">
            <xsl:call-template name="checkNonNegInt">
              <xsl:with-param name="test">
                <xsl:value-of select="@minOccurs"/>
              </xsl:with-param>
            </xsl:call-template>
          </xsl:variable>
          <xsl:choose>
            <xsl:when test="$nn='false'">
              <!-- if it's not, the element needs not to occur in
                    a Schema -->
              <xsl:if test="not(name(..)='model' or name(..)='simple')">
                <!-- root elements must not have minOccurs -->
                <xsl:attribute name="minOccurs">foo0</xsl:attribute>
              </xsl:if>
              <!-- but preserve it in the xfm namespace -->
              <xsl:attribute name="xform:minOccurs">
                <xsl:value-of select="@minOccurs"/>
              </xsl:attribute>
            </xsl:when>
            <xsl:otherwise>
              <!-- minOccurs is a non negative Integer, so we can
                    have the Schema follow the restriction -->
              <xsl:if test="not(name(..)='model' or name(..)='simple')">
                <xsl:attribute name="minOccurs">
                  <xsl:value-of select="@minOccurs"/>
                </xsl:attribute>
              </xsl:if>
            </xsl:otherwise>
          </xsl:choose>
          <!-- there is no else, because the default for minOccurs
                in Schemas is 1 -->
        </xsl:if>
      </xsl:when>
      <xsl:otherwise>
        <xsl:if test="not(name(..)='model' or name(..)='simple')">
          <!-- root elements must not have minOccurs -->
          <xsl:attribute name="minOccurs">foo10</xsl:attribute>
        </xsl:if>
        <!-- preserver required and minOccurs in xfm namespace -->
        <xsl:attribute name="xform:required">
          <xsl:value-of select="@required"/>
        </xsl:attribute>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:when>
  <xsl:otherwise>
    <xsl:if test="not(name(..)='model' or name(..)='simple')">
      <!-- root elements must not have minOccurs -->
      <xsl:attribute name="minOccurs">foo10</xsl:attribute>
    </xsl:if>
    <!-- preserver required and minOccurs in xfm namespace -->
    <xsl:attribute name="xform:required">
      <xsl:value-of select="@required"/>
    </xsl:attribute>
  </xsl:otherwise>
</xsl:choose>

```

```

        </xsl:attribute>
        <xsl:attribute name="xform:minOccurs">
            <xsl:value-of select="@minOccurs"/>
        </xsl:attribute>
    </xsl:otherwise>
</xsl:choose>
</xsl:when>
<xsl:otherwise>
    <xsl:if test="not(name(..)='model' or name(..)='simple')">
        <!-- root elements must not have minOccurs -->
        <xsl:attribute name="minOccurs">foo20</xsl:attribute>
    </xsl:if>
    <!-- preserver relevant, required and minOccurs in xfm
        namespace -->
    <xsl:if test="not(string-length(@relevant)=0 or @relevant='false')">
        <xsl:attribute name="xform:relevant">
            <xsl:value-of select="@relevant"/>
        </xsl:attribute>
    </xsl:if>
    <xsl:if test="not(string-length(@required)=0 or @required='false')">
        <xsl:attribute name="xform:required">
            <xsl:value-of select="@required"/>
        </xsl:attribute>
    </xsl:if>
    <xsl:if test="not(string-length(@minOccurs)=0 or @minOccurs='1')">
        <xsl:attribute name="xform:minOccurs">
            <xsl:value-of select="@minOccurs"/>
        </xsl:attribute>
    </xsl:if>
</xsl:otherwise>
</xsl:choose>

<!-- check maxOccurs, it can be: "unbounded", "1" (default), a none
    negative integer, or an expression -->
<xsl:choose>
    <xsl:when test="@maxOccurs=$maxOccursSpecial">
        <!-- copy it into the Schema -->
        <xsl:if test="not(name(..)='model' or name(..)='simple')">
            <!-- root elements must not have maxOccurs -->
            <xsl:attribute name="maxOccurs">unbounded</xsl:attribute>
        </xsl:if>
    </xsl:when>
    <xsl:when test="not(@maxOccurs='1')">
        <!-- check if $maxOccurs if a non negative integer -->
        <xsl:variable name="nn">
            <xsl:call-template name="checkNonNegInt">
                <xsl:with-param name="test">
                    <xsl:value-of select="@maxOccurs"/>
                </xsl:with-param>
            </xsl:call-template>
        </xsl:variable>
        <xsl:choose>
            <xsl:when test="$nn='false'">
                <!-- if it's not, the element may occur an unlimited number
                    of times in a Schema -->

```

```

<xsl:if test="not(name(..)='model' or name(..)='simple')">
  <!-- root elements must not have maxOccurs -->
  <xsl:attribute name="maxOccurs">unbounded</xsl:attribute>
</xsl:if>
<!-- but preserve maxOccurs it in the xfm namespace -->
<xsl:if test="not(@maxOccurs='unbounded')">
  <xsl:attribute name="xform:maxOccurs">
    <xsl:value-of select="@maxOccurs"/>
  </xsl:attribute>
</xsl:if>
</xsl:when>
<xsl:otherwise>
  <!-- maxOccurs is a non negative Integer, so we can
    have the Schema follow the restriction -->
  <xsl:if test="not(name(..)='model' or name(..)='simple')">
    <!-- root elements must not have maxOccurs -->
    <xsl:attribute name="maxOccurs">
      <xsl:value-of select="@maxOccurs"/>
    </xsl:attribute>
  </xsl:if>
</xsl:otherwise>
</xsl:choose>
</xsl:when>
<!-- there is no otherwise, because the default for maxOccurs in
  Schemas is 1 -->
</xsl:choose>

<!-- readOnly is of general interest, but can only be mapped into the
  xfm namespace -->
<xsl:if test="not(string-length(@readOnly)=0 or @readOnly='false')">
  <xsl:attribute name="xform:readOnly">
    <xsl:value-of select="@readOnly"/>
  </xsl:attribute>
</xsl:if>

<!-- validate is of general interest, but can only be mapped into the
  xfm namespace -->
<xsl:if test="not(string-length(@validate)=0 or @validate='true')">
  <xsl:attribute name="xform:validate">
    <xsl:value-of select="@validate"/>
  </xsl:attribute>
</xsl:if>

<!-- calc is of general interest, but can only be mapped into the
  xfm namespace -->
<xsl:if test="string-length(@calc) > 0">
  <xsl:attribute name="xform:calc">
    <xsl:value-of select="@calc"/>
  </xsl:attribute>
</xsl:if>

<!-- choices is of general interest, but can only be mapped into the
  xfm namespace -->
<xsl:if test="string-length(@choices) > 0">
  <xsl:attribute name="xform:choices">

```

```

    <xsl:value-of select="@choices"/>
  </xsl:attribute>
</xsl:if>
</xsl:template>

</xsl:transform>

```

B.2 Required DTD for Transformation

```

<?xml version="1.0"?>
<!-- DTD for XForms -->

<!-- ENTITY definitions -->

<!-- Taken from the Schema definition:
      can be overridden in the internal subset of a xforms document to
      establish a namespace prefix -->
<!ENTITY % p          ''>
<!-- add optional namespace to all elements -->
<!ENTITY % xform      "%p;xform">
<!ENTITY % model      "%p;model">
<!ENTITY % submit     "%p;submit">
<!ENTITY % instance   "%p;instance">
<!ENTITY % bind        "%p;bind">
<!ENTITY % simple     "%p;simple">

<!-- data types -->
<!ENTITY % string     "%p;string">
<!ENTITY % boolean    "%p;boolean">
<!ENTITY % number     "%p;number">
<!ENTITY % currency   "%p;currency">
<!ENTITY % money      "%p;money">
<!ENTITY % date       "%p;date">
<!ENTITY % time       "%p;time">
<!ENTITY % duration   "%p;duration">
<!ENTITY % uri        "%p;uri">
<!ENTITY % binary     "%p;binary">
<!ENTITY % element    "%p;element">
<!ENTITY % attribute  "%p;attribute">
<!ENTITY % null       "%p>null">

<!ENTITY % datatype   "%string;|%boolean;|%number;|%currency;|%money;|
                        %date;|%time;|%duration;|%uri;|%binary;">

<!ENTITY % group      "%p;group">
<!ENTITY % union      "%p;union">
<!ENTITY % array      "%p;array">
<!ENTITY % switch     "%p;switch">
<!ENTITY % case       "%p;case">

```

```

<!-- facets -->
<!ENTITY % mask          "%p;mask">
<!ENTITY % pattern       "%p;pattern">
<!ENTITY % mediaType     "%p;mediaType">
<!ENTITY % scheme        "%p;scheme">
<!ENTITY % value         "%p;value">
<!ENTITY % allowCurrency "%p;allowCurrency">
<!ENTITY % length        "%p;length">
<!ENTITY % max           "%p;max">
<!ENTITY % min           "%p;min">
<!ENTITY % precision     "%p;precision">
<!ENTITY % scale         "%p;scale">
<!-- common datatype facets -->
<!ENTITY % cdf           "%value;">

<!-- attribute names -->
<!ENTITY % URIref       "CDATA">
<!ENTITY % bool         "(true|false)">
<!ENTITY % datesteps   "years|months|days">
<!ENTITY % timesteps   "hours|minutes|seconds">

<!ENTITY % name         "name          NMTOKEN          #IMPLIED">
<!-- NOTE: while a name is required, it can not be a required attribute
         as the union defines a name and the elements don't need a
         name in this case -->
<!ENTITY % id           "id            ID              #IMPLIED">
<!ENTITY % ref          "ref           %URIref;        #IMPLIED">
<!ENTITY % readOnly    "readOnly      CDATA           'false'">
<!ENTITY % required     "required      CDATA           'true'">
<!-- NOTE: while required is a boolean field, the real value can come
         from an expression, this is a text -->
<!ENTITY % relevant     "relevant      CDATA           'true'">
<!-- NOTE: while relevant is a boolean field, the real value can come
         from an expression, this is a text -->
<!ENTITY % enum         "enum          (open|closed) 'open'">
<!ENTITY % choices      "choices      CDATA           #IMPLIED">
<!ENTITY % calc         "calc         CDATA           #IMPLIED">
<!ENTITY % validate     "validate     CDATA           'true'">
<!ENTITY % condition    "condition    CDATA           'true'">
<!ENTITY % minOccurs   "minOccurs     CDATA           '1'">
<!ENTITY % maxOccurs    "maxOccurs     CDATA           '1'">
<!-- NOTE: The special value "unbounded" represents an unlimited
         repetition. -->
<!ENTITY % precisiona   "precision   CDATA           'unlimited'">
<!ENTITY % scalea       "scale        CDATA           'unlimited'">

<!-- common attributes -->
<!ENTITY % comatt       "%name;
                        %id;
                        %required;
                        %relevant;
                        %readOnly;
                        %validate;" >

<!-- data type attributes -->

```

```

<!ENTITY % dtattr      "%comatt;
                        %enum;" >
<!-- simple data type attributes -->
<!ENTITY % sdtattr     "%dtattr;
                        %calc;
                        %choices;
                        %minOccurs;
                        %maxOccurs;" >
<!-- NOTE: I've added minOccurs and mxOccurs while they are not explicitly
in the spec -->
<!-- NOTE: I would have liked to include min and max here, but there
are element specific defaults that can not be generalised
-->

<!-- ELEMENT definitions -->

<!ELEMENT %xform; ((%submit;)*,(%model;)*,(%instance;)*,(%bind;)*)>
<!ATTLIST %xform; action  %URIref;      #IMPLIED
                    method CDATA        #IMPLIED
                    id     ID            #IMPLIED >

<!ELEMENT %submit; ANY>
<!-- NOTE: the syntax for submit is not finalized, so I allow anything -->
<!ELEMENT %bind;   ANY>
<!-- NOTE: the syntac for bind is not finalized, so I allow anything -->
<!ELEMENT %instance; ANY>
<!-- NOTE: there's no way to tell a validating parser to ignore the
content of an element, if it has child elements, therefor
no xform with an instance can ever be valid -->

<!ELEMENT %model; (%simple;|%group;|%array;|%union;|%switch;|
                  %datatype;|%element;)*>
<!ATTLIST %model; %id;
                    %ref; >

<!ELEMENT %simple; (%group;|%array;|%union;|%switch;|%datatype;|%element;)*>
<!ATTLIST %simple; %id;
                    %ref;
                    %name; >

<!ELEMENT %group; (%group;|%array;|%union;|%switch;|%datatype;|
                  %element;|%attribute;)*>
<!ATTLIST %group; %comatt;
                    %minOccurs;
                    %maxOccurs; >

<!ELEMENT %switch; (%case;,(%case;)+)>
<!ATTLIST %switch; %comatt;
                    %minOccurs;
                    %maxOccurs; >

```

```

<!ELEMENT %case; (%datatype;|%element;)*>
<!ATTLIST %case; %name;
           %condition; >

<!ELEMENT %union; ((%datatype;)+)>
<!ATTLIST %union; %comatt;
           %minOccurs;
           %maxOccurs; >

<!ELEMENT %string; (%cdf;|%length;|%mask;|%max;|%min;|%pattern;)*>
<!ATTLIST %string; %sdtattr;
           length    CDATA    #IMPLIED
           max       CDATA    "unlimited"
           min       CDATA    "0"
           mask      CDATA    #IMPLIED
           pattern   CDATA    #IMPLIED >
<!-- NOTE: there can be any number of masks and patterns, an entry
           that matches any mask/pattern is valide
           (e.i., they are ORed) -->

<!ELEMENT %length; (#PCDATA)*>

<!ELEMENT %mask;   (#PCDATA)*>

<!ELEMENT %max;    (#PCDATA)*>

<!ELEMENT %min;    (#PCDATA)*>

<!ELEMENT %pattern; (#PCDATA)*>

<!ELEMENT %value; (#PCDATA|%null;)*>

<!ELEMENT %null; EMPTY>

<!ELEMENT %boolean; EMPTY>
<!ATTLIST %boolean; %comatt;
           %calc;
           %minOccurs;
           %maxOccurs; >

<!ELEMENT %number; (%cdf;|%max;|%min;|%precision;|%scale;)*>
<!ATTLIST %number; %sdtattr;
           min       CDATA    "minus infinity"
           max       CDATA    "plus infinity"
           %precisiona;
           %scalea;>

<!ELEMENT %precision; (#PCDATA)*>

```

```

<!ELEMENT %scale;      (#PCDATA)*>

<!ELEMENT %currency; (%cdf;|%mask;)*>
<!ATTLIST %currency; %sdtattr;
           mask      CDATA  #IMPLIED >

<!ELEMENT %money; (%cdf;|%allowCurrency;|%max;|%min;|%precision;|%scale;)*>
<!ATTLIST %money; %sdtattr;
           min      CDATA  "minus infinity"
           max      CDATA  "plus infinity"
           %precisiona;
           scale    CDATA  "2"
           allowCurrency CDATA  #IMPLIED >

<!ELEMENT %allowCurrency; (#PCDATA)*>

<!ELEMENT %date; (%cdf;|%max;|%min;|%precision;)*>
<!ATTLIST %date; %sdtattr;
           min CDATA  #IMPLIED
           max CDATA  #IMPLIED
           precision (%datesteps;) #IMPLIED>
<!-- NOTE: the special value of "now" for min and max indicate the
           current time -->

<!ELEMENT %time; (%cdf;|%max;|%min;|%precision;)*>
<!ATTLIST %time; %sdtattr;
           min CDATA  #IMPLIED
           max CDATA  #IMPLIED
           precision (%timesteps;) #IMPLIED >
<!-- NOTE: The time zone is expressed as hours relative to UTC -->

<!ELEMENT %duration; (%cdf;|%max;|%min;|%precision;)*>
<!ATTLIST %duration; %sdtattr;
           precision (%datesteps;|%timesteps;) #IMPLIED
           min CDATA  #IMPLIED
           max CDATA  #IMPLIED >

<!ELEMENT %uri; (%cdf;|%length;|%max;|%min;|%scheme;)*>
<!ATTLIST %uri; %sdtattr;
           length  CDATA  #IMPLIED
           max     CDATA  "unlimited"
           min     CDATA  "0"
           scheme  CDATA  #IMPLIED>

<!ELEMENT %scheme; (#PCDATA)>

<!ELEMENT %binary; (%cdf;|%mediaType;|%length;|%max;|%min;)*>

```

```
<!ATTLIST %binary; %sdtattr;
          length    CDATA    #IMPLIED
          max       CDATA    "unlimited"
          min       CDATA    "0"
          mediaType CDATA    #IMPLIED >

<!ELEMENT %mediaType; (#PCDATA)*>

<!ELEMENT %element; EMPTY>
<!ATTLIST %element; %comatt;
          type    CDATA    #REQUIRED
          %minOccurs;
          %maxOccurs; >
```

[previous](#) [next](#) [contents](#)

Appendix C: Sample Forms

Contents

- [C.1 XForms and XHTML](#)

C.1 XForms and XHTML

XHTML Document with Multiple XForms - Page.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xml:lang="en" lang="en"
    xmlns="http://www.w3.org/1999/xhtml"
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
    xmlns:xform="http://www.w3.org/2001/02/xforms">

<head>
  <title>Sample XForms Document</title>

  <xform xmlns="http://www.w3.org/2001/02/xforms" id="poll">
    <submitInfo target="http://example.com/app1" />
    <model>
      <simple>
        <number name="choiceCode" enum="closed">
          <value>-1</value>
          <value>10</value>
          <value>20</value>
          <value>30</value>
        </number>
      </simple>
    </model>
  </xform>

  <xform xmlns="http://www.w3.org/2001/02/xforms" id="search">
    <submitInfo target="http://example.com/app2" />
    <bind id="Query" ref="/query/searchstring"/>
    <model>
      <schema xmlns="http://www.w3.org/2000/10/XMLSchema">
        <element name="query">
          <complexType>
            <attribute name="searchstring" type="string" />
          </complexType>
        </element>
      </schema>
    </model>
    <instance model="search" xmlns="http://example.org/ns">
```

```

    <query searchstring="Enter your Query Here" />
  </instance>
</xform>
</head>

<body>
  <!-- ... -->

  <!-- Site Search Markup -->
  <xform:textbox ref="id('Query')">
    <xform:caption>Search our Web Site</xform:caption>
    <xform:help>Enter your search term here and hit "Go!"</xform:help>
  </xform:textbox>
  <xform:submit xform="search">
    <xform:caption>Go!</xform:caption>
  </xform:submit>

  <!-- ... -->

  <!-- Daily Poll Markup -->
  <xform:exclusiveSelect style="list-ui: radio" xform="poll"
    ref="choiceCode">
    <xform:caption>When do you plan to implement XForms?</xform:caption>
    <xform:item value="-1">Don't know</xform:item>
    <xform:item value="10">0-6 Months</xform:item>
    <xform:item value="20">6-12 Months</xform:item>
    <xform:item value="30">More than 12 Months</xform:item>
  </xform:exclusiveSelect>
  <xform:submit xform="poll">
    <xform:caption>Submit</xform:caption>
  </xform:submit>

  <!-- ... -->

</body>
</html>

```

This is an example of a single document that might be hand authored. This example shows an XHTML document with two separate XForms embedded using simple syntax, one with a set of initial [instance data](#).

Sample Instance Data for Poll

```
<choiceCode>30</choiceCode>
```

Here is a sample of the [instance data](#) for the Poll form.

Sample Instance Data for Site Search

```
<query xmlns="http://example.com/ns" searchstring="MP3" />
```

Here is a sample of the [instance data](#) for Site Search form.

Appendix D: Optional Function Libraries

Contents

- [D.1 Finance Libraries](#)

D.1 Finance Methods

This is an optional library that supports a variety of common financial calculations for interest rates, monthly payments etc.

`apr(n1, n2, n2)`

Returns the annual percentage rate for a loan, where `n1` is the principal amount of the loan, `n2` is the monthly payment, and `n3` is the number of months payments will have to be made. For example `"apr(35000, 269.50, 30 times 12)"` returns 0.085 (or 8.5%) for the annual interest rate on a loan of \$35,000 being repaid at \$269.50 per month over 30 years.

`cterm(n1, n2, n2)`

Returns the number of periods needed for an investment earning a fixed, but compounded, interest rate to grow to a future value, where `n1` is the interest rate per period, `n2` is the future value of the investment, and `n3` is the amount of the initial investment. For example `"cterm(.02, 200, 100)"` returns 35 as the required period for \$100 invested at 2% to grow to \$200.

`fv(n1, n2, n3)`

Returns the future value of periodic constant payments at a constant interest rate, where `n1` is the amount of each equal payment, `n2` is the interest rate per period, and `n3` is the total number of periods. For example `"fv(100, .075 over 12, 10 times 12)"` returns 17793.03 as the amount present after paying \$100 a month for 10 years in an account bearing an annual interest of 7.5%.

`ipmt(n1, n2, n2, n3, n4, n5)`

Returns the amount of interest paid on a loan over a period of time, where `n1` is the principal amount of the loan, `n2` is the annual interest rate, `n3` is the monthly payment, `n4` is the first month of the computation, and `n5` is the number of months to be computed. For example `"ipmt(30000, .085, 295.50, 7, 3)"` returns 624.88 as the

amount of interest paid starting in July (month 7) for 3 months on a loan of \$30,000.00 at an annual interest rate of 8.5% being repaid at a rate of \$295.50 per month.

`npv(n1, n2 [, ...])`

Returns the the net present value of an investment based on a discount rate, and a series of periodic future cash flows, where n1 is the discount rate over one period, n2 ... are cash flow values which must be equally spaced in time and occur at the end of each period. For example "`npv(0.15, 100000, 120000, 130000, 140000, 50000)`" returns 368075.16 as the net present value of an investment projected to generate \$100,000, \$120,000, \$130,000, \$140,000 and \$50,000 over each of the next five years and the rate is 15% per annum.

`pmt(n1, n2, n3)`

Returns the payment for a loan based on constant payments and a constant interest rate, where n1 is the principal amount of the loan, n2 is the interest rate per period, and n3 is the number of monthly payments. For example, "`pmt(30000.00, .085 over 12, 12 times 12)`" returns 333.01 as the monthly payment for a loan of a \$30,000, borrowed at a yearly interest rate of 8.5%, repayable over 12 years (144 months).

`ppmt(n1, n2, n2, n3, n4, n5)`

Returns the amount of principal paid on a loan over a period of time, where n1 is the principal amount of the loan, n2 is annual interest rate, n3 is the monthly payment, n4 is is the first month of the computation, and n5 is the number of months to be computed. For example "`ppmt(30000, .085, 295.50, 7, 3)`" returns 261.62 as the amount of principal paid starting in July (month 7) for 3 months on a loan of \$30,000 at an annual interest rate of 8.5%, being repaid at \$295.50 per month. The annual interest rate is used in the function because of the need to calculate a range within the entire year.

`pv(n1, n2, n3)`

Returns the present value of an investment of periodic constant payments at a constant interest rate, where n1 is the amount of each equal payment, n2 is the interest rate per period, and n3 is the total number of periods. For example "`pv(1000, .08 over 12, 5 times 12)`" returns 49318.43 as the present value of \$1000.00 invested at 8% for 5 years.

`rate(n1, n2, n3)`

Returns the compound interest rate per period required for an investment to grow from present to future value in a given period, where n1 is the future value, n2 is the present value and n3 is is the total number of periods. For example "`rate(110, 100, 1)`" returns 0.10 as what the rate of interest must be for and investment of \$100 to grow to \$110 if invested for 1 term.

`term(n1, n2, n3)`

Returns the number of periods needed to reach a given future value from periodic constant payments into an interest bearing account, where n1 is the payment amount made at the end of each period, n2 is the interest rate per period, and n3 is the future value. For example "`term(475, .05, 1500)`" returns 3 as the number of months for an investment of \$475, deposited at the end of each period into an account bearing 5%

compound interest, to grow to \$1500.00.

[previous](#) [next](#) [contents](#)

Appendix E: References

Contents

- [E.1 Normative](#)
- [E.2 Non-normative](#)

E.1 Normative References

[CSS2]

Bert Bos, Håkon Wium Lie, Chris Lilley, and Ian Jacobs. Recommendation: *Cascading Style Sheets, level 2 (CSS2) Specification*. Available at: <http://www.w3.org/TR/REC-CSS2>. 1998.

[DOM2 Events]

Tom Pixley. Recommendation: *Document Object Model (DOM) Level 2 Events Specification*. Available at: <http://www.w3.org/TR/DOM-Level-2-Events>. 2000.

[ISO 4217]

International Organization for Standardization (ISO). *ISO Standards for Currency Names*. 1999.

[ISO 8601]

International Organization for Standardization (ISO). *Representations of dates and times*. Available at: <http://www.iso.ch/markete/8601.pdf>. 1988.

[RFC 2045]

N. Freed and N. Borenstein. *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. Available at: <http://www.ietf.org/rfc/rfc2396.txt>. 1996.

[RFC 2396]

Tim Berners-Lee, et. al. *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*.

Available at: <http://www.ietf.org/rfc/rfc2045.txt>. 1998.

[Unicode]

Joan Aliprand, Julie Allen, Joe Becker, Mark Davis, Michael Everson, Asmus Freytag, John Jenkins, Mike Ksar, Rick McGowan, Lisa Moore, Michel Suignard, and Ken Whistler. *The Unicode Standard, Version 3.0*, Reading, Mass.: Addison-Wesley Developers Press. 2000.

[WML1.3]

Wireless Application Protocol Forum, Ltd. *Wireless Application Protocol Wireless Markup Language Specification Version 1.3*. Available at: <http://www1.wapforum.org/tech/documents/WAP-191-WML-20000219-a.pdf>. 2000.

[XForms Req]

Micah Dubinko, Dave Ragget, Sebastian Schnitzenbaumer, and Malte Wedel. Working Draft: *XForms Requirements*. Available at: <http://www.w3.org/TR/xhtml-forms-req>. 2000.

[XHTML Events]

Ted Wugofski. Working Draft: *XHTML" Events - An updated events syntax for XHTML*. Available at: <http://www.w3.org/TR/xhtml-events>. 2000

[XML 1.0]

Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Recommendation: *Extensible Markup Language (XML) 1.0 (Second Edition)*. Available at: <http://www.w3.org/TR/REC-xml>. 2000.

[XML-Names]

Tim Bray, Dave Hollander, and Andrew Layman. Recommendation: *Namespaces in XML*. Available at: <http://www.w3.org/TR/REC-xml-names>. 1999.

[XPath]

James Clark and Steve DeRose. Recommendation: *XML Path Language (XPath) Version 1.0*. Available at: <http://www.w3.org/TR/xpath>. 1999.

[XSchema-1]

Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. Candidate Recommendation: *XML Schema Part 1: Structures*. Available at: <http://www.w3.org/TR/xmlschema-1>. 2000.

[XSchema-2]

Paul V. Biron and Ashok Malhotra. Candidate Recommendation: *XML Schema Part 2: Datatypes*. Available at: <http://www.w3.org/TR/xmlschema-2>. 2000.

E.2 Non-Normative References

[ANSI X3-274]

American National Standards Institute (ANSI). *Information Technology - Programming Language REXX*. Document Number: ANSI X3.274-1996. 1996.

[ECMA 262]

European Computer Manufacturers' Association (ECMA). *ECMA-262: ECMAScript Language Specification*. Available at <ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf>. 1999.

[RFC 2141]

R. Moats. *URN Syntax*. Available at: <http://www.ietf.org/rfc/rfc2141.txt>. 1997.

[XHTML 1.0]

Steven Pemberton, et. al. Recommendation: *XHTML" 1.0: The Extensible HyperText Markup Language - A Reformulation of HTML 4 in XML 1.0*. Available at: <http://www.w3.org/TR/xhtml1>. 2000.

[XSchema-0]

David C. Fallside. *XML Schema Part 0: Primer*. Available at: <http://www.w3.org/TR/xmlschema-0>. 2000.

[XSLT]

James Clark. Recommendation: *XSL Transformations (XSLT) Version 1.0*. Available at: <http://www.w3.org/TR/xslt>. 1999.

Appendix F: Changes

Changes from the 19 December 2000 revision

Document chapters have been rearranged for greater logical progression. The headings below are given in terms of the new chapter numbers.

Editorial notes throughout the document have been reformatted and clarified as numbered "Editor's Feedback Requests" or "Editor's Notes".

Various paragraphs throughout the document have been revised for spelling, grammar, and flow.

The convention of `xform:` is used to represent the XForms namespace.

Chapter 1 - About the XForms 1.0 Specification

In section 1.4, Document Conventions have been updated for greater consistency. The rest of the document has been made to match the new document conventions.

Chapter 3 - Terminology

Removed terms "Data Structures Component" and "XForms Extensions Component" as these were used only in the non-normative concepts chapter.

Chapter 4 - Datatypes

Overall - changed facet `scheme` to `uriScheme`.

The `min` and `max` facets no longer serve as shortcuts to the `minLength` and `maxLength` facets (but do still refer to `minExclusive` and `maxExclusive`).

4.1 - For clarity, removed term "built-in" datatypes.

4.1 - Added "XForms Validation Rule:" to sentence describing potentially conflicting facet values.

Chapter 5 - XForms Model

The term "XForms Specific Property" has been changed to "model item property".

Though the syntax and usage are the same, `id` attributes are no longer classified as model item properties.

5.2.5 The `calc` property is now `calculate`.

3.2.6 New model item property - `priority`.

5.3.6 New example for `switch`

Chapter 6 - Dynamic Constraint Language

Renamed Chapter to be consistent with Table of Contents.

Renamed 6.3 to "Binding Expressions".

Added new subsection 6.3.1 "Root and Context Nodes".

Added new subsection 6.3.2 "Canonical Binding Expressions".

Chapter 8 - Binding

Removed references to multiple instances with a single Model.

Section 8.3 has been reorganized to cover all aspects of direct binding.

Section 8.4 has been reorganized to cover all aspects of indirect binding.

Section 8.5 The `model::` and `instance::` axis specifiers have been unified into a single separate `xform` attribute.

Chapter 9 - Including XForms in Other Documents

This chapter has been moved towards the end of the specification, as well as internally reorganized.

The syntax for the elements declared has been simplified (in particular, the `<instance>` element no longer needs a `model` attribute.

9.2.1 - Typo, changed element `href` to attribute.

9.2.3 - Renamed extra `<submit>` element to `<submitInfo>`.

Chapter 10 - Processing Model and Conformance

This chapter has been rewritten, based on the latest discussions within the Working Group.

Appendix A - Schema for XForms Model

Updated with new version of the XForms Model Schema.

Appendix B - XSLT

This appendix has not been significantly updated, and therefore is not synchronized with the rest of the document.

Appendix C - Sample Forms

The example has been updated in accord with the rest of XForms.

Appendix E - References

New normative reference to DOM Level 2 Events.

[previous](#) [next](#) [contents](#)