MSL
A model for W3C XML Schema

Allen Brown, Microsoft, allenbr@microsoft.com
Matthew Fuchs, Commerce One, matthew.fuchs@commerceone.com
Jonathan Robie, Software AG, jonathan.robie@SoftwareAG-USA.com
Philip Wadler, Avaya Labs, wadler@avaya.com

ABSTRACT

MSL (Model Schema Language) is an attempt to formalize some of the core idea in XML Schema. The benefits of a formal description is that it is both concise and precise. MSL has already proved helpful in work on the design of XML Query. We expect that similar techniques can be used to extend MSL to include most or all of XML Schema.

1. INTRODUCTION

XML is based on two simple ideas: represent documents and data as trees, and represent the types of documents and data using tree grammars. Tree grammars are represented using DTDs [4] or XML Schema [6, 11, 3]. XML Schema is being developed under the auspices of the World Wide Web Consortium (W3C), the body responsible for HTML and XML, among other things. As of this writing, XML Schema recently entered candidate recommendation status.

XML Schema is more powerful than DTDs. Among other things, it uses a uniform XML syntax, supports derivation of document types (similar to subclassing in object-oriented languages), permits 'all' groups and nested definitions, and provides atomic data types (such as integers, floating point, dates) in addition to character data. XML Schema is also more complex than DTDs, requiring a couple of hundred pages to describe, as opposed to the thirty or so in the original specification of XML 1.0 (which included DTDs). The remainder of this paper assumes some familiarity with XML Schema.

MSL (Model Schema Language) is an attempt to formalize some of the core idea in XML Schema. The benefits of a formal description is that it is both concise and precise, although it does require some familiarity with mathematical notation.

MSL is described with an inference rule notation. Originally developed by logicians [7, 13], this notation is now widely used for describing type systems and semantics of programming languages [10]. A basic understanding of grammar rules and first-order predicate logic should be adequate to understand this paper; all other notations are defined before they are used.

We hope our work on MSL may make XML Schema easier to understand, and may aid in the process of designing other specifications and tools that build on XML Schema. In particular, MSL has already proved helpful in work on the design of XML Query, another W3C standard currently under development. We expect that similar techniques can be used to extend MSL to include most or all of XML Schema.

MSL (like XML Schema) draws on standard ideas about type systems for semistructured data as described in the literature [1, 9], notably the use of regular expressions and tree grammars. In particular, MSL closely resembles the type system in Xduce [8]. Another example of formalizing part of an XML specification can be found in [12].

Many important aspects of XML Schema are not modeled by MSL. We have focussed on the core material in XML Schema Part I (Structures), as we believe this is the most complex. Many features of XML Schema are *not* modeled. These include the following.

- Identity constraints.
- The mapping from XML Schema syntax into components.
- Skip and lax wildcard validation.
- The unambiguity restriction on content models.
- The sibling element constraint.
- The `xsi:nil` attribute.
- A check that abstract components are not instantiated.
- Support for form and form default.
- Support for final, block, use, and value.
- The Post Schema Validation Infoset.
- Atomic datatypes.

We have begun to work on modeling the first two of these. We believe that most or all of the items in the above list could be modeled with additional effort.

In addition, MSL differs from XML Schema on the definition of restriction. We believe the definition of restriction in XML Schema is unnecessarily *ad hoc*, as explained in Appendix A.

We expect an MSL document to be released by the W3C and for MSL to continue to evolve, as will XML Schema.

Copyright is held by the author/owner.
WWW10, May 1-5, 2001, Hong Kong.
ACM 1-58113-348-0/01/0005.

This paper concentrates on the key element of MSL and XML Schema, namely normalization, refinement, and validation.

The central point the reader should take away is that it is possible to create a simple model that captures the essence of these features. This was not obvious before we began, and indeed the initial version of MSL was much more complex than what is presented here.

The remainder of this paper is organized as follows.

- Section 2 gives an overview of MSL.
- Section 3 defines the basic MSL structures, including names, groups, and components.
- Section 4 describes normalization.
- Section 5 describes refinement.
- Section 6 describes validation.
- Appendix A lists problems with the current XML Schema Working Drafts, uncovered during the MSL work.
- Appendix B lists suggestions for improving XML Schema, based on the MSL work.

2. OVERVIEW

This section uses a running example to introduce the MSL representation of a schema. MSL uses a mathematical notation that is easier to manipulate formally than the XML syntax of Schema. One aspect of MSL is the use of *normalization* to provide a unique name for each component of a schema.

Figure 1 shows an example schema written in W3C XML Schema syntax, and Figure 2 shows an example XML document which validates against the given schema. These will be used as running examples throughout this section.

2.1 Normalization

MSL uses a normalized form of a schema, which assigns a unique universal name to each component of a schema, and flattens the structure. A component is anything which may be defined or declared: an element, an attribute, a simple type, a complex type, a group, or an attribute group.

Figure 3 shows the normalized universal names of the components in our sample schema. For each name, we list two forms: the long form is the name proper, while the short form is an abbreviated version we use in examples to improve readability.

The names reflect the nesting structure of the original schema. Nested elements or attributes are given the name of the element or attribute; nested types are given the anonymous name `*`. The syntax of names is chosen to be similar to XPath [5].

MSL's normalized names clearly distinguish local names from global names. Where previously it might be possible to confuse the global `a` element with the local `a` element, now these are given distinct names, `a` and `u/d/*/a` respectively. The latter indicates that the global type `u` contains a local element `d` which contain an anonymous type `*` which contains a local element `a`. Each attribute or element contains at most one anonymous type, so using the name `*` for such types leads to no confusion.

```
<xsi:schema
    targetNamespace = "http://www.foo.org/baz.xsd"
    xmlns = "http://www.foo.org/baz.xsd"
    xmlns:xsi = "http://www.w3.org/2000/10/XMLSchema"
    elementFormDefault = "qualified">
  <xsi:element name="a" type="t"/>
  <xsi:simpleType name="s">
    <xsi:list itemType="xsi:integer"/>
  </xsi:simpleType>
  <xsi:complexType name="t">
    <xsi:attribute name="b" type="xsi:string"/>
    <xsi:attribute use="optional" type="s" name="c"/>
  </xsi:complexType>
  <xsi:complexType name="u">
    <xsi:complexContent>
      <xsi:extension base="t">
        <xsi:choice>
          <xsi:element name="d">
            <xsi:complexType>
              <xsi:sequence>
                <xsi:element name="a"
                             type="xsi:string"
                             minOccurs="1"
                             maxOccurs="unbounded"/>
              </xsi:sequence>
            </xsi:complexType>
          </xsi:element>
          <xsi:element name="e" type="xsi:string"/>
        </xsi:choice>
      </xsi:extension>
    </xsi:complexContent>
  </xsi:complexType>
</xsi:schema>
```

Figure 1: Example XML Schema

```
<a xmlns = "http://www.foo.org/baz.xsd"
   xmlns:xsi =
     "http://www.w3.org/2000/10/XMLSchema-instance"
   xsi:type = "u"
   b = "zero"
   c = "1 2">
  <d xmlns = "">
    <a>three</a>
    <a>four</a>
  </d>
</a>
```

Figure 2: Example XML document

2.2 Model groups

MSL uses a mathematical notation that is more compact than XML, and more amenable to formal use. Mathematical notation is used for model groups, components, and documents.

MSL uses standard regular expression notation [2] for model groups. In what follows, g stands for a model group (as does g_1, g_2, and so on). The constructors for model groups include the following.

ϵ empty sequence

$\emptyset$ empty choice

long form	short form
http://www.foo.org/baz.xsd#element::a	a
http://www.foo.org/baz.xsd#type::s	s
http://www.foo.org/baz.xsd#type::t	t
http://www.foo.org/baz.xsd#type::t/attribute::b	t/@b
http://www.foo.org/baz.xsd#type::t/attribute::c	t/@c
http://www.foo.org/baz.xsd#type::u	u
http://www.foo.org/baz.xsd#type::u/element::d	u/d
http://www.foo.org/baz.xsd#type::u/element::d/type::*	u/d/*
http://www.foo.org/baz.xsd#type::u/element::d/type::*/element::a	u/d/*/a
http://www.foo.org/baz.xsd#type::u/element::d/type::*/element::a	u/e

Figure 3: Normalized universal names

g_1 , g_2	sequence, g_1 followed by g_2
g_1 | g_2	choice, g_1 or g_2
g_1 & g_2	interleaving, g_1 and g_2 in any order ('all' group)
g{m,n}	repetition, g repeated between minimum m and maximum n times (m a natural number, n a natural number or ∞)
mixed(g)	mixed content in group g
a[g]	attribute, with name a and content g
e[g]	element, with name e and content g
w	wildcard
p	atomic datatype (such as xsi:String)
x	component name

Here is an example of a group in MSL notation, which corresponds to an a element with content of type u in our running example.

```
a[
 (t/@b[xsi:String] &
  t/@c[xsi:Integer{0,∞}]{0,1}),
 (u/d[u/d/*/a[xsi:String]{1,∞}] |
  u/d/*/e[xsi:String])
]
```

Note that the group constructors are used uniformly in several contexts. Repetition (g{m,n}) is used for lists of atomic datatypes, to indicate whether an attribute is optional, and for elements. Similarly, interleaving (g_1 & g_2) is used for attributes and for elements.

2.3 Components

Next, we show how components are represented in MSL notation. Each component is one of seven sorts: element, attribute, simple type, complex type, attribute group, or model group. Regardless of sort, each component is represented uniformly as a record with seven fields:

sort is the sort of the component;

name is the name of the component;

base is the name of the base component of the structure;

derivation specifies how the component is derived from the base, it is one of restriction or extension;

refinement is the set of permitted derivations from this component as base, it is a subset of {restriction, extension};

abstract is a boolean, representing whether this type is abstract;

content is the content of the structure, a model group.

Here are the components of the normalized schema represented in MSL notation.

```
component(
  sort = element,
  name = a,
  base = xsi:UrElement,
  derivation = restriction,
  refinement = {restriction,extension},
  abstract = false,
  content = a[u]
)

component(
  sort = simpleType,
  name = s,
  base = xsi:UrSimpleType,
  derivation = restriction,
  refinement = {restriction},
  abstract = false,
  content = xsi:Integer{0,∞}
)

component(
  sort = complexType,
  name = t,
  base = xsi:UrType,
  derivation = restriction,
  refinement = {restriction,extension},
  abstract = false,
  content = t/@b & t/@c
)

component(
  sort = attribute,
  name = t/@b,
  base = xsi:UrAttribute,
  derivation = restriction,
  refinement = {restriction},
  abstract = false,
  content = t/@b[xsi:String]
)
```

```
component(
  sort = attribute,
  name = t/@c,
  base = xsi:UrAttribute,
  derivation = restriction,
  derivation = {restricition},
  abstract = false,
  content = t/@c[s]{0,1}
)


component(
  sort = complexType,
  name = u,
  base = t,
  derivation = extension,
  refinement = {restriction,extension},
  abstract = false,
  content = (t/@b & t/@c), (u/d | u/e)
)


component(
  sort = element,
  name = u/d,
  base = xsi:UrElement,
  derivation = restriction,
  refinement = {},
  abstract = false,
  content = u/d[u/d/*]
)


component(
  sort = complexType,
  name = u/d/*,
  base = xsi:UrType,
  derivation = restriction,
  refinement = {},
  abstract = false,
  content = u/d/*{0,∞}
)


component(
  sort = element,
  name = u/d/*/a,
  base = xsi:UrElement,
  derivation = restriction,
  refinement = {},
  abstract = false,
  content = u/d/*/a[xsi:String]
)


component(
  sort = element,
  name = u/e,
  base = xsi:UrElement,
  derivation = restriction,
  refinement = {},
  abstract = false,
  content = u/e[xsi:String]
)
```

Observe that if we start with the content of the top-level `a` node, replace the name `t` with its refinement `u`, and then expand out all names (that is, replace the names `t/@b` and `t/@c` with the contents of those attributes, and so on), then the result is the same as the model group given in the previous subsection.

Note that MSL does not nest declarations to express their scope. Instead, the scope of a declaration is reflected in its normalized name.

2.4 Documents

MSL also provides a compact notation for XML documents, both before and after normalization. Here is the original document in MSL notation.

```
a[
 @xsi:type["u"],
 @b["zero"],
 @c[1,2],
 d[
  a["three"],
  a["four"]
 ]
]
```

Note that attributes and elements are represented uniformly, as are sequences of attributes, sequences of elements, and lists of atomic datatypes.

Here is the normalized document in MSL notation.

```
a[
 @xsi:type["u"],
 t/@b["zero"],
 t/@c[1,2],
 u/d[
  u/d/*/a["three"],
  u/d/*/a["four"]
 ]
]
```

Finally, here is the normalized document in MSL notation with type information added.

```
a[
 u ∋
 t/@b[xsi:String ∋ "zero"],
 t/@c[s ∋ 1,2],
 u/d[
  u/d/* ∋
  u/d/*/a[xsi:String ∋ "three"],
  u/d/*/a[xsi:String ∋ "four"]
 ]
]
```

Unlike the `xsi:type` convention, the MSL notation allows one to uniformly express information about element and attribute types. The type of an element or attribute is indicated by writing x`[`$t \ni d$`]` where x is an attribute or element name, t is a type name, and d is the content of the attribute or element.

3. STRUCTURES

This section defines the structures used in MSL: names, wildcards, atomic datatypes, model groups, components, and documents.

3.1 Names

A *namespace* is a URI, and a *local name* is an NCName, as in the Namespace recommendation. We let i range over namespaces and j range over local names.

namespace	i	::=	URI
local name	j	::=	NCName

A *symbol space* is one of the six symbol spaces in XML

Schema. We let ss range over symbol spaces.

symbol space	ss	::=	`element`
		|	`attribute`
		|	`type`
		|	`attributeGroup`
		|	`modelGroup`
		|	`notation`

(We make no further use of `attributeGroup`, `modelGroup`, and `notation` in this document.)

A *symbol name* consists of a symbol space paired with a local name or with `*` (the latter names an anonymous component). A *name* consists of a URI followed by a sequence of one or more symbol names. We let sn range over symbol names, and x range over names.

symbol name	sn	::=	
named			ss`::`j
anonymous		|	ss`::*`
name	x	::=	i`#`sn_1`/`$\cdots$`/`sn_l

It is sometimes convenient to use a short form of names. For these, we make the simplifying assumption that there are only two symbol spaces. Symbol names in the attribute symbol space are written `@`j, all other symbol names are written j (or `*` for anonymous names). The URI may be dropped when it is obvious from context.

Example names are shown in Figure 3. The scope length of the second to last name, with short form `u/d/*/a`, is 4.

Before normalization, all names in a document have scope length equal to one. It is helpful to define functions to extract the namespace from a name, and to extract the symbol space and local name of the last symbol name. If $x = i$`#`sn_1`/`$\cdots$`/`sn_l and $sn_l = ss$`::`j then we define `namespace(`x`)` $= i$, `symbol(`x`)` $= ss$, and `local(`x`)` $= j$.

We also introduce several subclasses of names. An *attribute name* is the name of an attribute component, and similarly for *element*, *simple type*, and *complex type names*. We let a, e, s, k range over attribute, element, simple type, and complex type names.

attribute name	a	::=	x
element name	e	::=	x
simple type name	s	::=	x
complex type name	k	::=	x

A *type name* is a simple or complex type name. We let t range over type names.

type name	t	::=	s
		|	k

The class of a name must be consistent with its symbol space.

`symbol(`a`)`	=	`attribute`
`symbol(`e`)`	=	`element`
`symbol(`t`)`	=	`type`

3.2 Wildcards

A wildcard denotes a set of element names. A *wildcard item* is of the form `*:*` denoting any name in any namespace, or i`:*` denoting any name in namespace i. A *wildcard* consists of wildcard items, union of wildcards, or difference of wildcards. We let v range over wildcard items, and w range over wildcards.

wildcard item	v	::=	
any namespace			`*:*`
given namespace		|	i`:*`
wildcard	w	::=	
wildcard item			v
union		|	w_1`+`w_2
difference		|	w_1`!`w_2

A wildcard union w_1`+`w_2 stands for any name in w_1 or w_2. A wildcard difference w_1`!`w_2 stands for any name in w_1 but not in w_2. For example, the wildcard `*:*!(baz:*+xsi:String)` denotes any name in any namespace, except for names in namespace `baz`, and local name `String` in namespace `xsi`.

3.3 Atomic datatypes

MSL takes as given the *atomic datatypes* from XML Schema Part 2 [3]. We let p range over atomic datatypes, and c range over constants of such datatypes.

Typically, an atomic datatype is either a primitive datatype, or is derived from another atomic datatype by specifying a set of facets. Lists of atomic datatypes are specified using repetition, while unions are specified using alternation, as defined below.

An example of an atomic datatype is `xsi:String`, and a constant of that datatype is `"zero"`.

3.4 Model groups

We use traditional regular expression notation for model groups. Let g range over model groups.

group	g	::=	
empty sequence			ϵ
empty choice		|	$\emptyset$
sequence		|	g_1 `,` g_2
choice		|	g_1 `|` g_2
interleave		|	g_1 `&` g_2
repetition		|	g{m`,`n}
attribute		|	a`[`g`]`
element		|	e`[`g`]`
mixed content		|	`mixed(`g`)`
wildcard		|	w
atomic datatype		|	p
component name		|	x
minimum	m	::=	natural
maximum	n	::=	m
unbounded		|	∞

An example of a group appears in Section 2.2.

The empty sequence matches only the empty document; it is an identity for sequence and interleaving. The empty choice matches no document; it is an identity for choice.

$$\begin{array}{rcccl} \epsilon \texttt{ , } g & = & g & = & g \texttt{ , } \epsilon \\ \emptyset \texttt{ | } g & = & g & = & g \texttt{ | } \emptyset \\ \epsilon \texttt{ \& } g & = & g & = & g \texttt{ \& } \epsilon \end{array}$$

For use with repetitions, we extend arithmetic to include ∞ in the obvious way. For any n we have $n + \infty = \infty + n = \infty$ and $n \leq \infty$ is always true, and $\infty < n$ is always false.

3.5 Components

A *sort* is one of the six sorts of component in XML Schema. We let *srt* range over sorts.

sort	*srt*	::=	`attribute`
		|	`element`
		|	`simpleType`
		|	`complexType`
		|	`attributeGroup`
		|	`modelGroup`

(We make no further use of `attributeGroup` or `modelGroup` in this document.)

A *derivation* specifies how a component is derived from its base. We let *der* range over derivations, and *ders* range over sets of derivations. We let *b* range over booleans.

derivation	*der*	::=	`extension`
		|	`refinement`
derivation set	*ders*	::=	$\{der_1, \cdots, der_l\}$
boolean	*b*	::=	`true`
		|	`false`

A *component* is a record with seven fields.

`sort` is the sort of the component (*srt*);

`name` is the name of the component (x);

`base` is the name of the base component of the structure (x);

`derivation` specifies how the component is derived from the base (*der*);

`refinement` is the set of permitted derivations from this component as base (*ders*);

`abstract` is a boolean, representing whether this type is abstract (*b*);

`content` is the content of the structure, a model group (g).

We let *cmp* range over components.

component *cmp* ::=
```
component(
  sort = srt
  name = x
  base = x
  derivation = der
  refinement = ders
  abstract = b
  content = g
)
```

Examples of components appear in Section 2.3.

For a given schema, we assume a fixed *dereferencing map* that takes names to the corresponding components. We write `deref(`x`)` = *cmp* if name x corresponds to component *cmp*.

The dereferencing map must map attribute names to attribute components, and similarly for elements, simple types, and complex types.

```
deref(a).sort  =  attribute
deref(e).sort  =  element
deref(s).sort  =  simpleType
deref(k).sort  =  complexType
```

3.6 Constraints

Recall that the group constructs are used uniformly in several contexts. Repetition ($g\{m,n\}$) is used for lists of atomic datatypes, to indicate whether an attribute is optional, and for elements. Similarly, interleaving (g_1 `&` g_2) is used for attributes and for elements. The advantage of this approach is that the semantics of groups is uniform, and need be given only once. Thus, for instance, how repetition acts is defined once, not separately for attributes and elements.

However, it is helpful to define additional syntactic categories that specify various subsets of groups. These syntactic classes are then used to constrain the content of components, for instance, to indicate that the content of an element component should be an element, and that the content of a type component should consist of attributes followed by elements.

An *attribute group* contains only attribute names, combined using interleaving. An *element group* contains no attribute names.

attribute group	*ag*	::=	
empty sequence			ϵ
interleaving		|	ag_1 `&` ag_2
attribute name		|	a
element group	*eg*	::=	
empty sequence			ϵ
empty choice		|	$\emptyset$
sequence		|	eg_1 `,` eg_2
choice		|	eg_1 `|` eg_2
interleaving		|	eg_1 `&` eg_2
repetition		|	$eg\{m,n\}$
wildcard		|	w
element name		|	e
type name		|	t

(Interleaving in element groups might be further constrained, as in Schema Part 1, Section 5.7, All Group Limited.)

Given the above, we can specify the allowed contents of the four sort of component as follows. An *attribute content* is either an attribute or an optional attribute, where the content is a simple type name. An *element content* is an element where the content is a type name. A *simple type content* is an atomic datatype or a list of atomic datatype. A *complex type content* is an attribute group followed by either a simple type content or an element group.

attribute content	*ac*	::=	
attribute			a`[`s`]`
optional attribute		|	a`[`s`]{0,1}`
element content	*ec*	::=	
element			e`[`t`]`
union content	*uc*	::=	
atomic datatype			p
union		|	uc `|` uc
simple content	*sc*	::=	
union			uc
list		|	$uc\{m,n\}$
complex content	*kc*	::=	
simple			ag `,` sc
element		|	ag `,` eg
mixed		|	ag `,` `mixed(`eg`)`

The content of each sort of component corresponds to the syntax above. That is, the content of an attribute component is always an attribute content ac; the content of an element component is always an element content ec; the content of a simple type component is always a simple content sc; and the content of a complex type component is always a complex content kc. Further, for an attribute or element component the name of the component is the same as the name of the attribute or element in its content.

It is easy to confirm that the example components in Section 2.3 satisfy these constraints.

3.7 Documents

A *document* is a sequence of items, where each item is an atomic datatype, or an attribute (with a document as content), or an element (with a document as content). We let d range over documents.

document	d	::=
empty document		ϵ
sequence	|	d_1 , d_2
constant	|	c
attribute	|	a[d]
element	|	e[d]

Examples of documents appear in Section 2.4.

A *typed document* is a document with added type information for each element and attribute. We let td range over typed documents.

document	td	::=
empty document		ϵ
sequence	|	td_1 , td_2
constant	|	c
attribute	|	a[$s \ni d$]
element	|	e[$t \ni td$]

Examples of typed documents appear in Section 2.4. Typed documents are the result of normalization, Section 4.

An *document item* is either a constant of atomic datatype, or an attribute, or an element.

document item	di	::=
constant	|	c
attribute	|	a[c]
element	|	e[d]

Document items are used to define interleaving, Section 6.

3.8 Inference rules

Operations such as normalization and validation are described with an inference rule notation. Originally developed by logicians [7, 13], this notation is now widely used for describing type systems and semantics of programming languages [10]. In this notation, when all judgements above the line hold, then the judgement below the line holds as well. Here is an example of a rule used later on. We write $d \in g$ if document d validates against group g.

$$\frac{d_1 \in g_1 \qquad d_2 \in g_2}{d_1 \text{ , } d_2 \in g_1 \text{ , } g_2} \quad \text{(SEQUENCE)}$$

The rule says that if both $d_1 \in g_1$ and $d_2 \in g_2$ hold, then d_1 , $d_2 \in g_1$, g_2 holds as well. For instance, take $d_1 =$ `a[1]`, $d_2 =$ `b["two"]`, $g_1 =$ `a[xsi:Integer]`, and $g_2 =$ `b[xsi:String]`. Then since both

`a[1]` $\in$ `a[xsi:Integer]`

and

`b["two"]` $\in$ `b[xsi:String]`

hold, we may conclude that

`a[1] , b["two"]` $\in$ `a[xsi:Integer] , b[xsi:String]`

holds.

4. NORMALIZATION

Normalization of a document replaces each name by the corresponding normalized name, and adds type information to the document. Normalization is performed with respect to a given schema; in our formalism the schema is determined by the dereferencing map, `deref()`. Section 2.4 gives an example of a document before and after normalization.

Prior to normalization, all names in the document have exactly one symbol name. To build normalized names, we use an operation that extends a name with an additional symbol name. Let x_1 and x_2 be two names, and where the second name has only one symbol name. We write $x_1 \rhd x_2$ for the operation that extends x_1 by adding the symbol name of x_2 (if they are in the same namespace) or is the name x_2 (if they are in different namespaces).

$$\begin{array}{lcl} i\#sn_1/\cdots/sn_l \rhd i\#sn & = & i\#sn_1/\cdots/sn_l/sn \\ i\#sn_1/\cdots/sn_l \rhd i'\#sn & = & i'\#sn, \quad i \neq i' \end{array}$$

For each sort there is a root type (`AnyType`, `AnyElement`, etc), for which we define

$$x \rhd \texttt{AnyElement} \;=\; \texttt{AnyElement}.$$

In effect, the root types are in a fixed namespace, so the rule for roots is subsumed by the rule for different namespaces.

We write $x \vdash a \Rightarrow a'$ or $x \vdash e \Rightarrow e'$ to indicate that in the context specified by name x that attribute name a normalizes to a' or that element name e normalizes to e'. To normalize an attribute or element name we extend the context by the name (if extension yields the normalized name of some component; these are the 'extend' rules), or use the element name directly (otherwise; these are the 'reset' rules). We write $x \in$ `dom(deref())` and $x \notin$ `deref()` to indicate whether or not x is in the domain of the dereferencing map; that is, whether or not x names some component.

$$\frac{x \rhd a \in \texttt{dom(deref())}}{x \vdash a \Rightarrow x \rhd a} \quad \text{(EXTEND ATTRIBUTE)}$$

$$\frac{x \rhd e \in \texttt{dom(deref())}}{x \vdash e \Rightarrow x \rhd e} \quad \text{(EXTEND ELEMENT)}$$

$$\frac{x \rhd a \notin \texttt{dom(deref())}}{x \vdash a \Rightarrow a} \quad \text{(RESET ATTRIBUTE)}$$

$$\frac{x \rhd e \notin \texttt{dom(deref())}}{x \vdash e \Rightarrow e} \quad \text{(RESET ELEMENT)}$$

We write $x \vdash d \Rightarrow dt$ to indicate that in the context specified by name x that document d normalizes to typed document td. We write $\texttt{@xsi:type} \notin d$ if d does not contain an `xsi:type` attribute. Note that the type names in `xsi:type` attributes always refer to global types and need not be normalized.

$$\frac{}{x \vdash c \Rightarrow c'} \quad \text{(CONSTANT)}$$

$$\frac{}{x \vdash \epsilon \Rightarrow \epsilon} \quad \text{(EMPTY DOCUMENT)}$$

$$\frac{x \vdash d_1 \Rightarrow td_1 \qquad y \vdash d_2 \Rightarrow td_2}{x \vdash d_1 \texttt{ , } d_2 \Rightarrow td_1 \texttt{ , } td_2} \quad \text{(SEQUENCE)}$$

$$\frac{\begin{array}{c} x \vdash a \Rightarrow a' \\ \texttt{deref(}a'\texttt{).content} = a'\texttt{[}s\texttt{]} \text{ or} \\ \texttt{deref(}a'\texttt{).content} = a'\texttt{[}s\texttt{]\{0,1\}} \\ s \vdash d \Rightarrow td \end{array}}{x \vdash a\texttt{[}d\texttt{]} \Rightarrow a'\texttt{[}s \ni td\texttt{]}} \quad \text{(ATTRIBUTE)}$$

$$\frac{\begin{array}{c} x \vdash e \Rightarrow e' \\ \texttt{@xsi:type} \notin d \\ \texttt{deref(}e'\texttt{).content} = e'\texttt{[}t\texttt{]} \\ t \vdash d \Rightarrow td \end{array}}{x \vdash e\texttt{[}d\texttt{]} \Rightarrow e\texttt{[}t \ni td\texttt{]}} \quad \text{(UNTYPED ELEMENT)}$$

$$\frac{\begin{array}{c} x \vdash e \Rightarrow e' \\ \texttt{deref(}e'\texttt{).content} = e'\texttt{[}t'\texttt{]} \\ t <: t' \\ t \vdash d \Rightarrow td \end{array}}{x \vdash e\texttt{[@xsi:type[}t\texttt{] , }d\texttt{]} \Rightarrow e\texttt{[}t \ni td\texttt{]}} \quad \text{(TYPED ELEMENT)}$$

The (TYPED ELEMENT) rule uses the relation $x<:x'$, which is defined Section 5.1.

5. REFINEMENT

5.1 Base chain

We write $x<:x'$ if starting from the component with name x one can reach the component with name x' by successively following base links.

$$\frac{}{x <: x} \quad \text{(REFLEXIVE)}$$

$$\frac{x <: x' \qquad x' <: x''}{x <: x''} \quad \text{(TRANSITIVE)}$$

$$\frac{\texttt{deref(}x\texttt{).base} = x'}{x <: x'} \quad \text{(BASE)}$$

5.2 Extension

We write $g <:_{\texttt{ext}} g'$ if group g is derived from group g' by adding attributes and elements. It is specified using attribute groups ag and element groups eg as defined in Section 3.6.

$$\frac{}{ag \texttt{ , } eg <:_{\texttt{ext}} (ag \texttt{ \& } ag') \texttt{ , } eg \texttt{ , } eg'} \quad \text{(EXTEND)}$$

5.3 Restriction

We write $g <:_{\texttt{res}} g'$ if the instances of group g are a subset of the instance of group g'. That is, $g <:_{\texttt{res}} g'$ if for every document d such that $d \in g$ it is also the case that $d \in g'$.

5.4 Constraints

A schema must satisfy certain constraints on refinement to be well-formed. Define $x <:_{der} x'$ to be $x <:_{\texttt{res}} x'$ if $der = \texttt{restriction}$ and $x <:_{\texttt{ext}} x'$ if $der = \texttt{extension}$. We write $\vdash x$ to indicate that the component with name x is well-formed with respect to refinement.

$$\frac{\begin{array}{c} x' = \texttt{deref(}x\texttt{).base} \\ der = \texttt{deref(}x\texttt{).derivation} \\ der \in \texttt{deref(}x'\texttt{).refinement} \\ \texttt{deref(}x\texttt{).content} <:_{der} \texttt{deref(}x'\texttt{).content} \end{array}}{\vdash x} \quad \text{(REFINEMENT)}$$

6. VALIDATION

We write $c \in_{\texttt{p}} p$ if constant c validates against atomic datatype p. We do not specify this relation further, but simply assume it is as defined in Schema Part 2.

We write $e \in_{\texttt{v}} v$ and $e \in_{\texttt{w}} w$ if element name e validates against wildcard item v or wildcard w. We write $e \notin_{\texttt{w}} w$ if it is not the case that $e \in_{\texttt{w}} w$.

$$\frac{}{e \in_{\texttt{v}} \texttt{*:*}} \quad \text{(ANY NAMESPACE)}$$

$$\frac{\texttt{namespace(}x\texttt{)} = i}{e \in_{\texttt{v}} i\texttt{:*}} \quad \text{(GIVEN NAMESPACE)}$$

$$\frac{e \in_{\texttt{v}} v}{e \in_{\texttt{w}} v} \quad \text{(WILDCARD ITEM)}$$

$$\frac{e \in_{\texttt{w}} w_1}{e \in_{\texttt{w}} w_1 \texttt{+} w_2} \quad \text{(WILDCARD SUM 1)}$$

$$\frac{e \in_{\texttt{w}} w_2}{e \in_{\texttt{w}} w_1 \texttt{+} w_2} \quad \text{(WILDCARD SUM 2)}$$

$$\frac{e \in_{\texttt{w}} w_1 \qquad e \notin_{\texttt{w}} w_2}{e \in_{\texttt{w}} w_1 \texttt{!} w_2} \quad \text{(WILDCARD DIFFERENCE)}$$

We write $d \mapsto_{\text{inter}} d' \texttt{ ; } d''$ (read "d interleaves d' and d''") if some interleaving of d' and d'' yields d. This relates one sequence to many pairs of sequences. For example,

$$\begin{array}{lcl} \texttt{a[1],a[2],a[3]} & \mapsto_{\text{inter}} & \texttt{a[1],a[2] ; a[3]} \\ \texttt{a[1],a[2],a[3]} & \mapsto_{\text{inter}} & \texttt{a[2] ; a[1],a[3]} \\ \texttt{a[1],a[2],a[3]} & \mapsto_{\text{inter}} & \epsilon \texttt{ ; a[1],a[2],a[3]} \end{array}$$

among other possibilities. This is used in the rule for interleaving, $g_1 \texttt{ \& } g_2$.

$$\frac{}{\epsilon \mapsto_{\text{inter}} \epsilon \texttt{ ; } \epsilon} \quad \text{(INTERLEAVE EMPTY)}$$

$$\frac{d_1 \mapsto_{\text{inter}} d_1' \,;\, d_1'' \qquad d_2 \mapsto_{\text{inter}} d_2' \,;\, d_2''}{d_1 \,,\, d_2 \mapsto_{\text{inter}} d_1' \,,\, d_2' \,;\, d_1'' \,,\, d_2''}$$
(INTERLEAVE SEQUENCE)

$$\frac{}{di \mapsto_{\text{inter}} id \,;\, \epsilon}$$ (INTERLEAVE ITEM 1)

$$\frac{}{di \mapsto_{\text{inter}} \epsilon \,;\, di}$$ (INTERLEAVE ITEM 2)

We write $d \mapsto_{\text{unmix}} d'$ (read "d unmixes to d'") if d is a sequence of elements and character data and d' is the same sequence with all character data removed. For example,

$$\texttt{a[1],"x",a[2]} \quad \mapsto_{\text{unmix}} \quad \texttt{a[1],a[2]}$$

This is used in the rule for mixed content, $\texttt{mixed}(g)$.

$$\frac{}{\epsilon \mapsto_{\text{unmix}} \epsilon}$$ (UNMIX EMPTY)

$$\frac{d_1 \mapsto_{\text{unmix}} d_1' \qquad d_2 \mapsto_{\text{unmix}} d_2'}{d_1 \,,\, d_2 \mapsto_{\text{unmix}} d_1' \,,\, d_2'}$$
(UNMIX SEQUENCE)

$$\frac{}{e\texttt{[}d\texttt{]} \mapsto_{\text{unmix}} e\texttt{[}d\texttt{]}}$$ (UNMIX ELEMENT)

$$\frac{}{c \mapsto_{\text{unmix}} \epsilon}$$
(UNMIX CHARACTER DATA)

We write $d \in g$ if document d validates against model group g.

$$\frac{}{\epsilon \in \epsilon}$$ (EMPTY)

$$\frac{d_1 \in g_1 \qquad d_2 \in g_2}{d_1 \,,\, d_2 \in g_1 \,,\, g_2}$$ (SEQUENCE)

$$\frac{d \in g_1}{d \in g_1 \mid g_2}$$ (CHOICE 1)

$$\frac{d \in g_2}{d \in g_1 \mid g_2}$$ (CHOICE 2)

$$\frac{d \mapsto_{\text{inter}} d_1 \,;\, d_2 \qquad d_1 \in g_1 \qquad d_2 \in g_2}{d \in g_1 \,\&\, g_2}$$
(INTERLEAVE)

$$\frac{d_1 \in g \qquad d_2 \in g\{m,n\}}{d_1 \,,\, d_2 \in g\{m+1,n+1\}}$$ (REPETITION 1)

$$\frac{d_1 \in g \qquad d_2 \in g\{0,n\}}{d_1 \,,\, d_2 \in g\{0,n+1\}}$$ (REPETITION 2)

$$\frac{}{\epsilon \in g\{0,n\}}$$ (REPETITION 3)

$$\frac{d \in g}{a\texttt{[}d\texttt{]} \in a\texttt{[}g\texttt{]}}$$ (ATTRIBUTE)

$$\frac{d \in g}{e\texttt{[}d\texttt{]} \in e\texttt{[}g\texttt{]}}$$ (ELEMENT)

$$\frac{d \mapsto_{\text{unmix}} d' \qquad d' \in g}{d \in \texttt{mixed}(g)}$$ (MIXED GROUP)

$$\frac{d \in s}{e\texttt{[}s \ni d\texttt{]} \in e\texttt{[}s\texttt{]}}$$ (TYPED ATTRIBUTE)

$$\frac{d \in t \qquad t <: t'}{e\texttt{[}t \ni d\texttt{]} \in e\texttt{[}t'\texttt{]}}$$ (TYPED ELEMENT)

$$\frac{d \in e \qquad e \in_{\texttt{w}} w}{d \in w}$$ (WILDCARD)

$$\frac{c \in_{\texttt{p}} p}{c \in p}$$ (ATOMIC DATATYPE)

$$\frac{d \in e \qquad e <: e'}{d \in e'}$$
(ELEMENT REFINEMENT)

$$\frac{d \in g \qquad g = \texttt{deref}(x)\texttt{.content}}{d \in x}$$
(COMPONENT NAME)

The (TYPED ELEMENT) and (ELEMENT REFINEMENT) rules use the relation $x <: x'$, which is defined Section 5.1. The check that $t <: t'$ in the (TYPED ELEMENT) rule is redundant, as it is also performed during normalization.

When processing a normalized document with types, the (ATTRIBUTE) and (ELEMENT) rules are not required, the (TYPED ATTRIBUTE) and (TYPED ELEMENT) are used instead.

7. REFERENCES

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[2] Alfred V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*. MIT Press Elsevier, Cambridge, Massachusetts, 1990.

[3] Paul V. Biron and Ashok Malhotra, editors. *XML Schema Part 2: Datatypes*. World Wide Web Consortium, 2000.

[4] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, editors. *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium, 1998.

[5] James Clark and Steve DeRose, editors. *XML path language (XPath) version 1.0*. World Wide Web Consortium, 1999.

[6] David C. Fallside, editor. *XML Schema Part 0: Primer*. World Wide Web Consortium, 2000.

[7] Gottlob Frege. Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought (1879). In Jan van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*. Harvard University Press, Cambridge, Massachusetts, 1967.

[8] Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for xml. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.

[9] Tova Milo and Dan Suciu. Type inference for queries on semistructured data. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1999.

[10] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, Massachusetts, 1996.

[11] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn, editors. *XML Schema Part 1: Structures*. World Wide Web Consortium, 2000.

[12] Philip Wadler. A formal semantics of patterns in XSLT. In B. Tommie Usdin, Deborah A. Lapeyre, and C. M. Sperberg-McQueen, editors, *Proceedings of Markup Technologies*, Philadelphia, 1999.

[13] Philip Wadler. New language, old logic. *Dr Dobbs Journal*, December 2000.

APPENDIX

A. PROBLEMS WITH XML SCHEMA

- *Restriction.* The following problems were uncovered in Schema Part 1, Section 5.10, which describes restriction.
  - *Complexity.* An attempt at defining formal rules corresponding to the definition of restriction required more than 100 lines of formal rules (as compared to the one-line definition given in the current document). Many rules had five or six premises.
  - *Transitivity.* There appear to be numerous cases where type t is a restriction of type t' and type t' is a restriction of type t'', but type t is not a restriction of type t'', according to the given rules.
  - *Ad hoc.* As noted below, many of the choices in the defition of restriction seem arbitrary.
  - *Elt:Elt–NameAndTypeOK.* Why the constraint that nullable of R and B be identical? A weaker constraint is more sensible: if R is nullable then B must be nullable. That is, it is fine for R not to be nullable even if B is nullable.
  - *Elt:Elt–NameAndTypeOK.* Why is it ok for the corresponding types to be related by list or extension, but not by restriction? Clearly, restriction should be allowed.
  - *Elt:All/Choice/Sequence—RecurseAsIfGroup.* Why does Elt:All/Choice/Sequence recurse, but Wildcard:All/Choice/Sequence is forbidden? Recursion seems equally sensible for both.
  - *All:All–Recurse vs. Sequence:All–RecurseUnordered.* Why must the former maintain order while the latter need not? Losing order seems equally sensible for both.
- *Wildcards.* Say that element e is in the equivalence class of element e' (that is, that e has base e'), and that elements e and e' are in different namespaces, and that a wildcard w includes the namespace of e' but not of e. If the wildcard is strict, does element e validate against wildcard w? Intuitively, one would expect e to match w (and this is what happens in MSL), but Schema Part 1 does not appear to consider this case.

B. SUGGESTIONS FOR XML SCHEMA

- *Restriction.* Change the definition of restriction to the one used in this document. One type is a restriction of another if every instance of the first type is also an instance of the second.
- *Uniformity.* Change the name `equivClass` to `base`, to emphasize the uniformity between elements and types.
- *Wildcards.* Specify how wildcards interact with equivalence classes. (See discussion in Appendix A.)
- *Symbol spaces.* Combine all symbol spaces other than `attribute`. This allows one to use the short form of naming described in the document. For instance, in place of

  ```
  type::u/element::d/type::*/element::a
  ```

  one could write `u/d/*/a`, and in place of

  ```
  type::t/attribute::b
  ```

  one could write `t/@b`.