The Essence of XML
(preliminary version)

Jérôme Siméon[1] and Philip Wadler[2]

[1] Bell Laboratories, Murray Hill, NJ 07974, USA
simeon@research.bell-labs.com
[2] Avaya Labs Research, Basking Ridge, NJ 07920, USA
wadler@avaya.com

Abstract. The World-Wide Web Consortium (W3C) promotes XML and related standards, including XML Schema, XQuery, and XPath. This paper describes a formalization XML Schema. A formal semantics based on these ideas is part of the official XQuery and XPath specification, one of the first uses of formal methods by a standards body. XML Schema features both named and structural types, with structure based on tree grammars. While structural types and matching have been studied in other work (notably XDuce, Relax NG, and previous formalizations of XML Schema), this is the first work to study the relation between named types and structural types, and the relation between matching and validation.

1 Introduction

There are a number of type systems for XML, including: DTDs, part of the original W3C recommendation defining XML [1]; XML Schema, a W3C recommendation which supersedes DTDs [13]; Relax NG, an Oasis standard [5]; Relax [10] and TREX [4], two ancestors of Relax NG; and the type systems of XDuce [8] and YATL [6]. All of these take a structural approach to typing, with the exception of XML Schema, which takes a named approach. (Another possible exception is DTDs, which are so restricted that the named and structural approaches might be considered to coincide.)

The W3C is responsible for three programming languages connected with XML: XSLT, a language for stylesheets [3, 9]; XQuery, an analogue of SQL for XML data [15]; and XPath, the common core of XSLT and XQuery, which is jointly managed by the working groups responsible for the other two languages [14]. All three of these are functional languages. XSLT 1.0 and XPath 1.0 became recommendations in November 1999 — they are untyped. XML Schema 1.0 became a recommendation in May 2001. XSLT 2.0, XQuery 1.0, and XPath 2.0 are currently being designed — they have type systems based on XML Schema.

This paper presents a formalization of XML Schema, developed in conjunction with the XQuery and XPath working groups. The paper presents a simplified version, treating the essential constructs. The full version is being developed as part of the XQuery and XPath Formal Semantics [16], one of the first industrial specifications to exploit formal methods. The full version treats not just XML Schema, but also the dynamic and static semantics of the XQuery and XPath.

Formal methods are particularly helpful for typing — the only complete description of the static type system of XQuery and XPath is in the formal specification. However, keeping two specifications in sync has not always been easy.

An earlier formal specification of XML Schema [2] was influenced by XDuce [8]; it ignored the named aspects of Schema and took a purely structural approach. The specification of Relax NG [5] also uses formal methods, and also is purely structural; it was influenced by the earlier work on XML Schema [2].

Matching and validation. Types in XML differ in some ways from types as used elsewhere in computing. Traditionally, a value *matches* a type — given a value and a type, either the value belongs to the type or it does not. In XML, a value *validates* against a type — given an (external) value and a type, validation produces an (internal) value or it fails.

For instance, consider the following XML Schema.

```
<xs:simpleType name="feet">
  <xs:restriction base="xs:float"/>
</xs:simpleType>
<xs:element name="height" type="feet"/>
```

In our type system, this is written as follows.

```
define type feet restricts xs:float
define element height of type feet
```

Now consider the following XML document.

```
<height>10023</height>
```

In our model, before validation this is represented as follows.

```
  <height>10023</height>
=>
  element height { "10023" }
```

And after validation it is represent as follows.

```
  validate as element height { <height>10023</height> }
=>
  element height of type feet { 10023.0 }
```

Validation has annotated the element with its type, and converted the text `"10023"` into the corresponding floating point number `10023.0`.

Our model provides both validation and matching. Validation attaches types to XML data. Unvalidated data may not match against a type. The following *does not* hold.

```
    element height { "10023" }
  matches
    element height
```

After validation, matching succeeds. The following *does* hold.

```
    element height of type feet { 10023.0 }
  matches
    element height
```

The inverse of validation is type erasure.

```
    element height of type feet { 10023.0 }
  erases to
    element height { "10023" }
```

The following theorem characterizes validation in terms of matching and erasure.

Theorem 1. *We have that*
`validate as` *Type* `{` *UntypedValue* `} =>` *Value*
if and only if
Value `matches` *Type*
and
Value `erases to` *UntypedValue*

Perhaps this theorem looks obvious, but if so let us assure you that it was not obvious to us when we began. It took some time to come to this formulation, and some tricky adjustments were required to ensure that holds.

One trick is that we model validation and erasure by relations, not functions. Naively, one might expect validation to be a partial function and erasure to be a function. That is, for a given type each untyped value validates to yield at most one typed value, and each typed values erases to one untyped value. One subtlety of the system presented here is that validation and erasure are modeled by relations. For example, the strings `"10023"` and `"10023.0"` both validate to yield the float `10023.0`, and hence we also have that the float erases to yield either string.

Shortcomings of XML and Schema. Our aim is to model XML and Schema as they exist — we do not claim that these are the best possible designs. Indeed, we would argue that XML and Schema have several shortcomings.

First, we would argue that a data representation should explicitly distinguish, say, integers from strings, rather than to infer which is which by validation against a Schema. (This is one of the many ways in which Lisp S-expressions are superior to XML.)

Second, while derivation by extension in Schema superficially resembles subclassing in object-oriented programming, in fact there are profound differences. One can typecheck code for a class without knowing all subclasses of that class (this supports separate compilation), but one cannot typecheck against a Schema type without knowing all types that derive by extension from that class (and hence separate compilation is problematic).

Nonetheless, XML and Schema are widely used standards, and there is value in modeling these standards. In particular, such models may: (i) improve our understanding of exactly what is mandated by the standard, (ii) help implementors create conforming implementations, and (iii) suggest how to improve the standards.

Relation of our model to Schema. Schema is a large and complex standard. In this paper, we attempt to model only the most essential features. These include: simple types and complex types; named and anonymous types; global and local elements; atomic, list, and union simple types; derivation by restriction; and derivation by extension. We model only two primitive datatypes, `xs:float` and `xs:string`, while Schema has nineteen primitive datatypes.

Many features of Schema that are omitted here are dealt with in the formal semantics for XQuery [16]. These include: namespaces; attributes; all groups (interleaving); text nodes; mixed content; substitution groups; `xsi:nil` attributes; and `xsi:type` attributes. There are other features of Schema that are not yet dealt with in the full formal semantics, but which we hope to model in future. These include: abstract types; default and fixed values; skip, lax, and strict wildcards; and facets of simple types.

Schema is normally written in an XML notation, but here we use a notation that is more readable and compact. The mapping of XML notation into our notation is described in the XQuery formal semantics.

There are a few aspects in which our treatment diverges from Schema. First, we permit ambiguous content models, while Schema does not. We do this because it makes our model simpler, and because ambiguity is important to support type checking, as discussed in Section 9. Second, we permit one type to be a restriction of another whenever the set of values belonging to the first type is included in the set of values belonging to the second, while Schema imposes ad hoc syntactic constraints. Again, we do this because it makes our model simpler, and because our more general model better supports type checking. Third, we only support the occurrence operators `?`, `+`, and `*`, while Schema supports arbitrary counts for minimum and maximum occurrences. This is because arbitrary counts may lead to a blow-up in the size of the finite-state automata we use to check when one type is included in another.

2 XML Schema by example

XML Schema supports a wide range of features. These include simple types and complex types, anonymous types, global and local declarations, derivation by restriction, and derivation by extension.

Simple and complex types. Here are declarations for two elements of simple type, one element with a complex type, and one complex type.

```
  define element title of type xs:string
  define element author of type xs:string
  define element paper of type paperType
  define type paperType {
    element title ,
    element author +
  }
```

Schema specifies nineteen primitive simple type types, including `xs:string` and `xs:float`.

A type declaration associates a name and a structure. The structure of a complex type is a regular expression over elements. As usual, `,` denotes sequence, `|` denotes alternation, `?` denotes an optional occurrence, `+` denotes one or more occurrences, and `*` denotes zero or more occurrences.

Validating annotates each element with its type.

```
  validate as paper {
    <paper>
      <title>The Essence of Algol</title>
      <author>John Reynolds</author>
    <paper>
  }
=>
  element paper of type paperType {
    element title of type string { "The Essence of Algol" },
    element author of type string { "John Reynolds" }
  }
```

Anonymous types. Instead of naming a type, it can defined in place without a name. Here is the `paper` element with its type expanded in place.

```
  define element paper {
    element title , element author +
  }
```

Validating now yields the following result.

```
  validate as paper {
    <paper>
      <title>The Essence of ML</title>
      <author>Robert Harper</author>
      <author>John Mitchell</author>
    </paper>
   }
=>
  element paper {
    element title of type xs:string { "The Essence of ML" },
    element author of type xs:string { "Robert Harper" },
```

```
    element author of type xs:string { "John Mitchell" }
  }
```

Now the paper element has no type annotation, because there is no type name to annotate it with. The other elements still have type annotations.

Global and local declarations. Similarly, one may include an element declaration in place. Here is the paper element with the nested elements expanded in place.

```
  define element paper {
    element title of type xs:string ,
    element author of type xs:string +
  }
```

Here the paper is declared globally, while title and author are declared locally. In this case, validation proceeds exactly as before.

Allowing local declarations increases expressiveness, because now it is possible for elements with the same name to be assigned different types in different places; see [11, 7]. An example of such a definition appears later.

Atomic, list, and union types. Every simple type is an atomic type, a list type, or a union type. The atomic types are the nineteen primitive types of Schema, such as xs:string and xs:float, and the types derived from them. List types are formed using the occurrence operators ?, +, and *, taken from regular expressions. Union types are formed using the alternation operator |, also taken from regular expressions.

Here is an example of a list type.

```
  element floats { xs:float + }
```

In XML notation, lists are written space-separated.

```
  validate as floats { <floats>1.0 2.0 3.0</floats> }
=>
  element floats { 1.0, 2.0, 3.0 }
```

Some types may be ambiguous. XML Schema specifies how to resolve this ambiguity: every space is taken as a list separator, and in case of a union the first alternative that works is chosen.

```
  element trouble { ( xs:float | xs:string )* }

  validate as trouble { <trouble>this is not 1 string</trouble> }
=>
  element trouble {"this", "is", "not", 1, "string" }
```

Ambiguous types can be problematic; this will be further discussed in Section 9.

Derivation by restriction on simple types. New simple types may be derived by restriction.

```
define type miles restricts xs:float
define type feet restricts xs:float
```

Here is an example, with two `height` elements have different types.

```
define element configuration {
  element shuttle { element height of type miles },
  element observatory { element height of type feet }
}
```

We have the `miles` and `feet` are both subtypes of `xs:float`, but neither is a subtype of the other. The following function definition is legal.

```
define function observatory_height (element configuration $c)
returns element height of type feet {
  $c/observatory/height
}
```

It would still be legal if `feet` were replaced by `xs:float`, but not if it were replaced by `miles`. In this example, `element configuration` is the type of the formal parameter `$c`, the XPath expression `$c/observatory/height` extracts the `observatory` child of the `configuration` element, and then extracts the `height` child of the `observatory` element.

Derivation by restriction on complex types. New complex types may also be derived by restriction. The following example is a simplified form of the information that may occur in a bibliographic database, such as that used by Bib-TeX.

```
define element bibliography {
  element of type publicationType *
}
define type publicationType {
  element author *,
  element title ?,
  element journal ?,
  element year ?
}
define type articleType restricts publicationType {
  element author +,
  element title,
  element journal,
  element year
}
define type bookType restricts publicationType {
  element author +,
```

```
    element title,
    element year
  }
  define element book of type bookType
  define element article of type articleType
```

Here a publication may have any number of authors, a mandatory title, and a optional journal, and year. An article must have at least one author, and a mandatory title, journal, and year. A book must have at least one author, a mandatory title and year, and no journal.

Derivation by restriction declares a relationship between two types. This relation depends on both names and structures, in the sense that one name may be derived by restriction from another name only if every value that matches the structure of the first also matches the structure of the second.

When one type is derived from another by restriction, it is fine to pass the restricted type where the base type is expected. For example, consider the following function.

```
  define function getTitle ( element of type publicationType $p )
  returns element title {
    $p/title
  }
```

Here it is acceptable to pass either an article or book element to the function `getTitle()`.

There is a type `xs:anyType` at the root of the type hierarchy. If a type definition does not specify otherwise, it is considered a restriction of `xs:anyType`.

Derivation by extension. New complex types may also be derived by extension.

```
  define type color restricts xs:string
  define type pointType {
    element x of type xs:float ,
    element y of type xs:float
  }
  define type colorPointType extends pointType {
    element c of type color
  }
  define element point of type pointType
  define element colorPoint of type colorPointType
```

When one type restricts another, one must check that the proper relation holds between the types. When one type extends another, the relation holds automatically, since values of the new type are defined to consist of the concatenation of values of the base type with values of the extension.

Again, when one type is derived from another by extension, it is fine to pass the extended type where the base type is expected. Unlike with restriction, this can lead to surprising consequences. Consider the following.

```
define function countChildren (element of type pointType $p)
returns xs:integer {
  count($p/*)
}
```

This function counts the number of children of the element `$p`, which will be 2 or 3, depending on whether `$p` is an element of type `point` or `colorPoint`.

In XQuery, type checking requires that one knows all the types that can be derived from a given type — the type is then treated as the union of all types that can be derived from it. Types derived by restriction add nothing new to this union, but types derived by extension do. This "closed world" approach — that type checking requires knowing all the types derived from a type — is quite different from the "open world" approach used in many object-oriented languages — where one can type-check a class without knowing all its subclasses.

In an object-oriented language, one might expect that if an element of type `colorPoint` is passed to this function, then the `x` and `y` elements would be visible but the `c` element would not be visible. Could the XQuery design adhere better to the object-oriented expectation? It is not obvious how to do so. For instance, consider the above function when `pointType` is replaced by `xs:anyType`.

```
define function countChildren (element of type xs:anyType $x)
  returns xs:integer {
    count($x/*)
  }
```

Here it seems natural to count all the children, while an object-oriented interpretation might suggest counting none of the children, since `xs:anyType` is the root of the type hierarchy.

3 Values and types

This section describes values and types. For brevity, we consider only 2 atomic types: *String* and *Float*.

3.1 Values

A value is a sequence of zero or more items. An item is either an element or an atomic value. An atomic value is a string or a float. Elements are optionally annotated with their type. An element with no type annotation is the same as an element with the type annotation `xs:anyType`.

$$
\begin{array}{lcl}
\mathit{Value} & ::= & \texttt{()} \\
 & | & \mathit{Item}(,\mathit{Item})* \\
\mathit{Item} & ::= & \texttt{element}\ \mathit{ElementName}\ \mathit{Annotation}?\ \texttt{\{}\ \mathit{Value}\ \texttt{\}} \\
 & | & \mathit{AtomicValue} \\
\mathit{AtomicValue} & ::= & \mathit{String}\ |\ \mathit{Float} \\
\mathit{Annotation} & ::= & \texttt{of type}\ \mathit{TypeName}
\end{array}
$$

We also write $Value_1$, $Value_2$ for the concatenation of two values.

An untyped value is a sequence of zero or more untyped items. An untyped item is either an element without type annotation or a string. Untyped values are used to described XML documents before validation. Every untyped value is a value.

$$
\begin{array}{lcl}
\mathit{UntypedValue} & ::= & \texttt{()} \\
 & | & \mathit{UntypedItem}(\texttt{,}\ \mathit{UntypedItem})* \\
\mathit{UntypedItem} & ::= & \texttt{element}\ \mathit{ElementName}\ \texttt{\{}\ \mathit{UntypedValue}\ \texttt{\}} \\
 & | & \mathit{String}
\end{array}
$$

A simple value consists of a sequence of zero or more atomic values. Every simple value is a value.

$$
\begin{array}{lcl}
\mathit{SimpleValue} & ::= & \texttt{()} \\
 & | & \mathit{AtomicValue}(\texttt{,}\mathit{AtomicValue})*
\end{array}
$$

Here are some examples of values.

```
element paper of type paperType {
  element title of type string { "The Essence of Algol" },
  element author of type string { "John Reynolds" }
}
element title of type string { "The Essence of Algol" },
element author of type string { "John Reynolds" }
```

Here are some examples of untyped values.

```
element paper {
  element title { "The Essence of Algol" },
  element author { "John Reynolds" }
}
element latitude { "20.0" },
```

Here are some examples of simple values.

```
"John Reynolds"
10023
1.0, 2.0, 3.0
```

3.2 Types

Types are modeled on regular tree grammars [12, 7]. A type is either an item type, the empty sequence (`()`), or composed by sequence (`,`), choice (`|`), or multiple occurrence – either optional (`?`), one or more (`+`), or zero or more (`*`).

$$
\begin{array}{lcl}
\mathit{Type} & ::= & \texttt{()} \\
 & | & \mathit{ItemType} \\
 & | & \mathit{Type}\ \texttt{,}\ \mathit{Type} \\
 & | & \mathit{Type}\ \texttt{|}\ \mathit{Type} \\
 & | & \mathit{Type}\ \mathit{Occurrence} \\
\mathit{Occurrence} & ::= & \texttt{?} \mid \texttt{+} \mid \texttt{*}
\end{array}
$$

An item type is an element type or an atomic type. Atomic types are specified by name; these names include `xs:string` and `xs:float`.

$$\begin{array}{lcl} \mathit{ItemType} & ::= & \mathit{ElementType} \\ & \mid & \mathit{AtomicTypeName} \\ \mathit{AtomicTypeName} & ::= & \mathit{TypeName} \end{array}$$

An element type gives an optional name and an optional type specifier. A name alone refers to a global declaration. A name with a type specifier is a local declaration. A type specifier alone is a local declaration that matches any name. The word "element" alone refers to any element.

$$\mathit{ElementType} \quad ::= \quad \texttt{element}\ \mathit{ElementName}?\ \mathit{TypeSpecifier}?$$

A type specifier either references a global type, or defines a type by derivation. A type derivation either restricts an atomic type, or restricts a named type to a given type, or extends a named type by a given type.

$$\begin{array}{lcl} \mathit{ItemType} & ::= & \mathit{TypeReference} \\ & \mid & \mathit{TypeDerivation} \\ \mathit{TypeReference} & \mid & \texttt{of type}\ \mathit{TypeName} \\ \mathit{TypeDerivation} & \mid & \texttt{restricts}\ \mathit{AtomicTypeName} \\ & \mid & \texttt{restricts}\ \mathit{TypeName}\ \texttt{\{}\ \mathit{Type}\ \texttt{\}} \\ & \mid & \texttt{extends}\ \mathit{TypeName}\ \texttt{\{}\ \mathit{Type}\ \texttt{\}} \end{array}$$

A simple type is composed from atomic types by choice or occurrence. Every simple type is a type.

$$\begin{array}{lcl} \mathit{SimpleType} & ::= & \mathit{AtomicTypeName} \\ & \mid & \mathit{SimpleType}\ \texttt{|}\ \mathit{SimpleType} \\ & \mid & \mathit{SimpleType}\ \mathit{Occurrence} \end{array}$$

We saw many examples of types and simple types in Section 2.

3.3 Top level definitions

At the top level, one can define elements, and types.

$$\begin{array}{lcl} \mathit{Definition} & ::= & \texttt{define element}\ \mathit{ElementName}\ \mathit{TypeSpecifier} \\ & \mid & \texttt{define type}\ \mathit{TypeName}\ \mathit{TypeDerivation} \end{array}$$

Global element declarations, like local element declarations, consist of a name and a type specifier. A global type declaration specifies both the derivation and the declared type. We saw many examples of definitions in Section 2.

3.4 Built-in type declarations

The two XML Schema built-in types `xs:anyType` and `xs:anySimpleType` are defined as follows.

```
define type xs:anyType restricts xs:anyType {
  xs:anySimpleType | element*
}
define type xs:anySimpleType restricts xs:anyType {
  ( xs:float | xs:string )*
}
```

4 Relationships between names

We need auxiliary judgments to describe relationships between element names and between type names.

4.1 Element name sets

An element name set is either a singleton consisting of just the given element name, or the wildcard * describing the set of all element names.

$$\mathit{ElementNameSet} \quad ::= \quad \mathit{ElementName} \mid \texttt{*}$$

The judgment

$$\mathit{ElementName}\ \texttt{within}\ \mathit{ElementNameSet}$$

holds when the element name is within the specified element name set. For example:

```
paper within paper
paper within *
```

An element name is within the set consisting of just that element name.

$$\frac{}{\mathit{ElementName}\ \texttt{within}\ \mathit{ElementName}}$$

An element name is within the set consisting of all element names.

$$\frac{}{\mathit{ElementName}\ \texttt{within}\ \texttt{*}}$$

4.2 Derives

The judgment

$$\mathit{TypeName}_1\ \texttt{derives from}\ \mathit{TypeName}_2$$

holds when the first type name derives from the second type name. For example,

```
bookType derives from publicationType
bookType derives from xs:anyType
colorPointType derives from xs:anyType
feet derives from xs:float
feet derives from xs:anySimpleType
feet derives from xs:anyType
```

This relation is a partial order: it is reflexive and transitive by the rules below, and it is asymmetric because no cycles are allowed in derivation by restriction or extension.

Derivation is reflexive and transitive.

$$\frac{}{\mathit{TypeName}\ \texttt{derives from}\ \mathit{TypeName}}$$

$$\frac{\begin{array}{c}\mathit{TypeName}_1\ \texttt{derives from}\ \mathit{TypeName}_2\\ \mathit{TypeName}_2\ \texttt{derives from}\ \mathit{TypeName}_3\end{array}}{\mathit{TypeName}_1\ \texttt{derives from}\ \mathit{TypeName}_3}$$

Every type name derives from the type it is declared to derive from by restriction or extension.

$$\frac{\texttt{define type}\ \mathit{TypeName}\ \texttt{restricts}\ \mathit{BaseTypeName}}{\mathit{TypeName}\ \texttt{derives from}\ \mathit{BaseTypeName}}$$

$$\frac{\texttt{define type}\ \mathit{TypeName}\ \texttt{restricts}\ \mathit{BaseTypeName}\ \texttt{\{}\ \mathit{Type}\ \texttt{\}}}{\mathit{TypeName}\ \texttt{derives from}\ \mathit{BaseTypeName}}$$

$$\frac{\texttt{define type}\ \mathit{TypeName}\ \texttt{extends}\ \mathit{BaseTypeName}\ \texttt{\{}\ \mathit{Type}\ \texttt{\}}}{\mathit{TypeName}\ \texttt{derives from}\ \mathit{BaseTypeName}}$$

5 Auxiliary judgments

We now define two auxiliary judgments that are used in matching and validation. Here is the rule from matching that uses these judgments.

$$\frac{\begin{array}{c}\mathit{ElementType}\ \texttt{yields}\ \mathit{ElementNameSet}\ \mathit{TypeSpecifier}\\ \mathit{TypeSpecifier}\ \texttt{resolves to}\ \mathit{BaseTypeName}\ \texttt{\{}\ \mathit{Type}\ \texttt{\}}\\ \mathit{ElementName}\ \texttt{within}\ \mathit{ElementNameSet}\\ \mathit{TypeName}\ \texttt{derives from}\ \mathit{BaseTypeName}\\ \mathit{Value}\ \texttt{matches}\ \mathit{Type}\end{array}}{\texttt{element}\ \mathit{ElementName}\ \texttt{of type}\ \mathit{TypeName}\ \texttt{\{}\ \mathit{Value}\ \texttt{\}}\ \texttt{matches}\ \mathit{ElementType}}$$

The element type yields an element name set and a type specifier, and the type specifier resolves to a base type name and a type. Then the given element matches the element type if three things hold: the element name must be within the element name set, the type name must derive from the base type name, and the value must match the type.

5.1 Yields

The judgment

$$\textit{ElementType}\ \texttt{yields}\ \textit{ElementNameSet}\ \textit{TypeSpecifier}$$

takes an element type and yields an element name set and a type specifier. For example,

```
element author yields author xs:string
element height of type feet yields height of type feet
element of type feet yields * of type feet
element yields * of type xs:anyType
```

If the element type is a reference to a global element, then it yields the the name of the element and the type specifier from the element declaration.

$$\frac{\texttt{define element}\ \textit{ElementName}\ \textit{TypeSpecifier}}{\texttt{element}\ \textit{ElementName}\ \texttt{yields}\ \textit{ElementName}\ \textit{TypeSpecifier}}$$

If the element type contains an element name and a type specifier, then it yields the given element name and type specifier.

$$\frac{}{\texttt{element}\ \textit{ElementName}\ \texttt{\{}\ \textit{TypeSpecifier}\ \texttt{\}}\ \texttt{yields}\ \textit{ElementName}\ \textit{TypeSpecifier}}$$

If the element type contains only a type specifier, then it yields the wildcard name and the type specifier.

$$\frac{}{\texttt{element \{}\ \textit{TypeSpecifier}\ \texttt{\} yields *}\ \textit{TypeSpecifier}}$$

If the element type has no element name and no type specifier, then it yields the wildcard name and the type `xs:anyType`.

$$\frac{}{\texttt{element yields * xs:anyType}}$$

5.2 Resolution

The judgment

$$\textit{TypeSpecifier}\ \texttt{resolves to}\ \textit{TypeName}\ \texttt{\{}\ \textit{Type}\ \texttt{\}}$$

resolves a type specifier to a type name and a type. For example,

```
  of type colorPoint
resolves to
  colorPoint {
    element x of type xs:float ,
    element y of type xs:float
    element c of type color
  }
```

and

```
    restricts xs:float  resolves to  xs:float { xs:float }
```

and

```
    restricts publicationType {
        element author +,
        element title,
        element year
    }
  resolves to
    publicationType {
        element author +,
        element title,
        element year
    }
```

and

```
    extends pointType {
      element c of type color
    }
  resolves to
    colorPoint {
      element x of type xs:float ,
      element y of type xs:float
      element c of type color
    }
```

If the type specifier references a global type, then resolve the type derivation in its definition, yielding a base type name and a type. Resolution returns the type name and the type (the base type name is discarded).

$$\frac{\begin{array}{c}\texttt{define type } \mathit{TypeName}\ \ \mathit{TypeDerivation}\\ \mathit{TypeDerivation}\ \texttt{resolves to}\ \mathit{BaseTypeName}\ \texttt{\{}\ \ \mathit{Type}\ \texttt{\}}\end{array}}{\texttt{of type}\ \mathit{TypeName}\ \texttt{resolves to}\ \mathit{TypeName}\ \texttt{\{}\ \ \mathit{Type}\ \texttt{\}}}$$

If the type specifier restricts an atomic type, then return the atomic type as both the type name and the type.

$$\frac{}{\begin{array}{c}\texttt{restricts}\ \mathit{AtomicTypeName}\\ \texttt{resolves to}\ \mathit{AtomicTypeName}\ \texttt{\{}\mathit{AtomicTypeName}\texttt{\}}\end{array}}$$

If the type specifier is a restriction of a non-atomic type, then return the given type name and the given type.

$$\frac{}{\texttt{restricts}\ \mathit{TypeName}\ \texttt{\{}\ \ \mathit{Type}\ \texttt{\}}\ \texttt{resolves to}\ \mathit{TypeName}\ \texttt{\{}\ \ \mathit{Type}\ \texttt{\}}}$$

If the type specifier is an extension, then resolve the name to get the base type, and return the given type name, and the result of concatenating the base type and the given type.

$$\frac{\texttt{of type } \mathit{TypeName} \texttt{ resolves to } \mathit{TypeName} \texttt{ \{ } \mathit{BaseType} \texttt{ \}}}{\texttt{extends } \mathit{TypeName} \texttt{ \{ } \mathit{Type} \texttt{ \} resolves to } \mathit{TypeName} \texttt{ \{ } \mathit{BaseType} \texttt{ , } \mathit{Type} \texttt{ \}}}$$

6 Matches

The judgment

$$\mathit{Value} \texttt{ matches } \mathit{Type}$$

holds when the given value matches the given type. For example,

```
    element author of type xs:string { "Robert Harper" },
    element author of type xs:string { "John Mitchell" }
  matches
    element author of type xs:string +
```

and

```
  10023 matches feet
```

and

```
    element colorPoint of type colorPointType {
      element x of type xs:float { 1.0 }
      element y of type xs:float { 2.0 }
      element c of type color { "blue" }
    }
  matches
    element colorPoint
```

The empty sequence matches the empty sequence type.

$$\frac{}{\texttt{() matches ()}}$$

If two values match two types, then their sequence matches the corresponding sequence type.

$$\frac{\mathit{Value}_1 \texttt{ matches } \mathit{Type}_1 \qquad \mathit{Value}_2 \texttt{ matches } \mathit{Type}_2}{\mathit{Value}_1 \texttt{ , } \mathit{Value}_2 \texttt{ matches } \mathit{Type}_1 \texttt{ , } \mathit{Type}_2}$$

If a value matches a type, then it also matches a choice type where that type is one of the choices.

$$\frac{\mathit{Value}\ \texttt{matches}\ \mathit{Type}_1}{\mathit{Value}\ \texttt{matches}\ \mathit{Type}_1\ \texttt{|}\ \mathit{Type}_2}$$

$$\frac{\mathit{Value}\ \texttt{matches}\ \mathit{Type}_2}{\mathit{Value}\ \texttt{matches}\ \mathit{Type}_1\ \texttt{|}\ \mathit{Type}_2}$$

A value matches an optional occurrence of a type if it matches either the empty sequence or the type.

$$\frac{\mathit{Value}\ \texttt{matches}\ \texttt{()}\ \texttt{|}\ \mathit{Type}}{\mathit{Value}\ \texttt{matches}\ \mathit{Type}\texttt{?}}$$

A value matches one or more occurrences of a type if it matches a sequence of the type followed by zero or more occurrences of the type.

$$\frac{\mathit{Value}\ \texttt{matches}\ \mathit{Type}\ \texttt{,}\ \mathit{Type}\texttt{*}}{\mathit{Value}\ \texttt{matches}\ \mathit{Type}\texttt{+}}$$

A value matches zero or more occurrences of a type if it matches an optional one or more occurrences of the type.

$$\frac{\mathit{Value}\ \texttt{matches}\ \mathit{Type}\texttt{+?}}{\mathit{Value}\ \texttt{matches}\ \mathit{Type}\texttt{*}}$$

A string matches an atomic type name if the atomic type name derives from `xs:string`. Similarly for floats.

$$\frac{\mathit{AtomicTypeName}\ \texttt{derives from xs:string}}{\mathit{String}\ \texttt{matches}\ \mathit{AtomicTypeName}}$$

$$\frac{\mathit{AtomicTypeName}\ \texttt{derives from xs:float}}{\mathit{Float}\ \texttt{matches}\ \mathit{AtomicTypeName}}$$

The rule for matching elements was explained at the beginning of Section 5.

$$\frac{\begin{array}{c}\mathit{ElementType}\ \texttt{yields}\ \mathit{ElementNameSet}\ \mathit{TypeSpecifier}\\ \mathit{TypeSpecifier}\ \texttt{resolves to}\ \mathit{BaseTypeName}\ \texttt{\{}\ \mathit{Type}\ \texttt{\}}\\ \mathit{ElementName}\ \texttt{within}\ \mathit{ElementNameSet}\\ \mathit{TypeName}\ \texttt{derives from}\ \mathit{BaseTypeName}\\ \mathit{Value}\ \texttt{matches}\ \mathit{Type}\end{array}}{\texttt{element}\ \mathit{ElementName}\texttt{of type}\ \mathit{TypeName}\ \texttt{\{}\ \mathit{Value}\ \texttt{\}}\ \texttt{matches}\ \mathit{ElementType}}$$

7 Erasure

7.1 Simply erases

To define erasure, we need an ancillary judgment. The judgment

$$\mathit{SimpleValue}\ \texttt{simply erases to}\ \mathit{String}$$

holds when *SimpleValue*erases to the string *String*. For example,

```
10023.0 erases to "10023.0"
"10023.0" erases to "10023.0"
"John Reynolds" erases to "John Reynolds"
(1.0, 2.0, 3.0) erases to "1.0 2.0 3.0"
```

The empty sequence erases to the empty string.

$$\frac{}{\texttt{() simply erases to ""}}$$

The concatenation of two non-empty sequences of values erases to the concatenation of their erasures with a separating space.

$$\frac{\begin{array}{ll}\mathit{SimpleValue}_1\ \texttt{simply erases to}\ \mathit{String}_1 & \mathit{SimpleValue}_1 \neq \texttt{()} \\ \mathit{SimpleValue}_2\ \texttt{simply erases to}\ \mathit{String}_2 & \mathit{SimpleValue}_2 \neq \texttt{()}\end{array}}{\mathit{SimpleValue}_1\ \texttt{,}\ \mathit{SimpleValue}_2\ \texttt{simply erases to concat}(\mathit{String}_1, \texttt{" "}, \mathit{String}_2)}$$

A string erases to itself.

$$\frac{}{\mathit{String}\ \texttt{simply erases to}\ \mathit{String}}$$

A float erases to any string that represents it.

$$\frac{}{\texttt{float-of-string}(\mathit{String})\ \texttt{simply erases to}\ \mathit{String}}$$

7.2 Erases

The judgment

$$\mathit{Value}\ \texttt{erases to}\ \mathit{UntypedValue}$$

holds when the given value erases to the untyped value. For example,

```
  element author of type xs:string { "John Reynolds" }
erases to
  element author { "John Reynolds" }
```

and

```
  element shuttle of type SpaceLocation {
    element latitude of type degrees { 20.0 },
    element longitude of type degrees { -155.5 },
    element height of type miles { 5.7 }
  }
erases to
  element shuttle {
    element latitude { "20.0" },
    element longitude { "-155.5" },
    element height { "5.7" }
  }
```

The empty sequence erases to itself.

$$\frac{}{\texttt{() erases to ()}}$$

The erasure of the concatenation of two values is the concatenation of their erasure, so long as neither of the two original values is simple.

$$\frac{\begin{array}{ll} \mathit{Value}_1 \texttt{ erases to } \mathit{UntypedValue}_1 & \mathit{Value}_1 \texttt{ not a simple value} \\ \mathit{Value}_2 \texttt{ erases to } \mathit{UntypedValue}_2 & \mathit{Value}_2 \texttt{ not a simple value} \end{array}}{\mathit{Value}_1 \texttt{ , } \mathit{Value}_2 \texttt{ erases to } \mathit{UntypedValue}_1 \texttt{ , } \mathit{UntypedValue}_2}$$

The erasure of a simple value is the corresponding string content using simpler erasure.

$$\frac{\mathit{SimpleValue} \texttt{ simply erases to } \mathit{String}}{\mathit{SimpleValue} \texttt{ erases to } \mathit{String}}$$

The erasure of an element is an element that has the same name and the erasure of the given content.

$$\frac{\mathit{Value} \texttt{ erases to } \mathit{UntypedValue}}{\begin{array}{c} \texttt{element } \mathit{ElementName} \texttt{ of type } \mathit{TypeName} \texttt{ \{ } \mathit{Value} \texttt{ \}} \\ \texttt{erases to} \\ \texttt{element } \mathit{ElementName} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \}} \end{array}}$$

8 Validation

8.1 Simply validate

The judgment

$$\texttt{simply validate as } \mathit{SimpleType} \texttt{ \{ } \mathit{String} \texttt{ \} => } \mathit{SimpleValue}$$

holds if validating the string against the simple type succeeds and returns the simple value. For example,

```
simply validate as xs:float { "10023.0" } => 10023.0
simply validate as xs:string { "10023.0" } => "10023.0"
simply validate as xs:string { "John Reynolds" } => "John Reynolds"
simply validate as xs:float* { "1.0 2.0 3.0" } => (1.0, 2.0, 3.0)
```

Simply validating a string against a choice type yields the result of simply validating the string against either the first or second type in the choice.

$$\frac{\texttt{simply validate as } \mathit{SimpleType}_1 \texttt{ \{ } \mathit{String} \texttt{ \} => } \mathit{SimpleValue}}{\texttt{simply validate as } \mathit{SimpleType}_1 \texttt{ | } \mathit{SimpleType}_2 \texttt{ \{ } \mathit{String} \texttt{ \} => } \mathit{SimpleValue}}$$

The rules for occurrences look slightly different from those in matching, because the simple types do not include the empty sequence or sequencing. Validating one or more occurrences breaks into two cases. In the first there is exactly one occurrence; in the second there is one occurrence followed by one or more occurrences, where the strings are separated by a space.

$$\frac{}{\texttt{simply validate as } \mathit{SimpleType}\texttt{? \{ "" \} => ()}}$$

$$\frac{\texttt{simply validate as } \mathit{SimpleType} \texttt{ \{ } \mathit{String} \texttt{ \} => } \mathit{SimpleValue}}{\texttt{simply validate as } \mathit{SimpleType}\texttt{? \{ } \mathit{String} \texttt{ \} => } \mathit{SimpleValue}}$$

$$\frac{\texttt{simply validate as } \mathit{SimpleType} \texttt{ \{ } \mathit{String} \texttt{ \} => } \mathit{SimpleValue}}{\texttt{simply validate as } \mathit{SimpleType}\texttt{+ \{ } \mathit{String} \texttt{ \} => } \mathit{SimpleValue}}$$

$$\frac{\begin{array}{c}\texttt{simply validate as } \mathit{SimpleType} \texttt{ \{ } \mathit{String}_1 \texttt{ \} => } \mathit{SimpleValue}_1 \\ \texttt{simply validate as } \mathit{SimpleType}\texttt{+ \{ } \mathit{String2} \texttt{ \} => } \mathit{SimpleValue}_2\end{array}}{\begin{array}{c}\texttt{simply validate as } \mathit{SimpleType}\texttt{+ \{ concat}(\mathit{String}_1, \texttt{" "}, \mathit{String}_2) \texttt{ \}} \\ \texttt{=> } \mathit{SimpleValue}_1 \texttt{ , } \mathit{SimpleValue}_2\end{array}}$$

$$\frac{\texttt{simply validate as } \mathit{SimpleType}\texttt{+? \{ } \mathit{String} \texttt{ \} => } \mathit{SimpleValue}}{\texttt{simply validate as } \mathit{SimpleType}\texttt{* \{ } \mathit{String} \texttt{ \} => } \mathit{SimpleValue}}$$

Simply validating a string against an atomic type derived from `xs:string` yields the string itself.

$$\frac{\mathit{AtomicTypeName} \texttt{ derives from xs:string}}{\texttt{simply validate as } \mathit{AtomicTypeName} \texttt{ \{ } \mathit{String} \texttt{ \} => } \mathit{String}}$$

Simply validating a string against an atomic type derived from `xs:float` yields the result of converting the string to a float.

$$\frac{\mathit{AtomicTypeName} \texttt{ derives from xs:float}}{\begin{array}{c}\texttt{simply validate as } \mathit{AtomicTypeName} \texttt{ \{ } \mathit{String} \texttt{ \}} \\ \texttt{=> float-of-string}(\mathit{String})\end{array}}$$

8.2 Validate

The judgment

$$\texttt{validate as } \mathit{Type} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} => } \mathit{Value}$$

holds if validating the untyped value against the type succeeds and returns the value. For example,

```
  validate as element of type xs:string {
    element author { "John Reynolds" }
  }
=>
  element author of type xs:string { "John Reynolds" }
```

and

```
  validate as element colorPoint {
    element colorPoint {
      element x { 1.0 }
      element y { 2.0 }
      element c { "blue" }
    }
  }
 =>
  element colorPoint of type colorPointType {
    element x of type xs:float { 1.0 }
    element y of type xs:float { 2.0 }
    element c of type color { "blue" }
  }
```

Validating the empty sequence as the empty type yields the empty sequence.

$$\frac{}{\texttt{validate as () \{ () \} => ()}}$$

Validating a concatenation of untyped values against a concatenation of types yields the concatenation of the validated values.

$$\frac{\begin{array}{c}\texttt{validate as } \mathit{Type}_1 \texttt{ \{ } \mathit{UntypedValue}_1 \texttt{ \} => } \mathit{Value}_1 \\ \texttt{validate as } \mathit{Type}_2 \texttt{ \{ } \mathit{UntypedValue}_2 \texttt{ \} => } \mathit{Value}_2\end{array}}{\begin{array}{c}\texttt{validate as } \mathit{Type}_1 \texttt{ , } \mathit{Type}_2 \texttt{ \{ } \mathit{UntypedValue}_1 \texttt{ , } \mathit{UntypedValue}_2 \texttt{ \}} \\ \texttt{=> } \mathit{Value}_1 \texttt{ , } \mathit{Value}_2\end{array}}$$

Validating a value against a choice type yields the result of validating the value as either the first or second type in the choice.

$$\frac{\texttt{validate as } \mathit{Type}_1 \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} => } \mathit{Value}}{\texttt{validate as } \mathit{Type}_1 \texttt{ | } \mathit{Type}_2 \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} => } \mathit{Value}}$$

$$\frac{\texttt{validate as } \mathit{Type}_2 \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} => } \mathit{Value}}{\texttt{validate as } \mathit{Type}_1 \texttt{ | } \mathit{Type}_2 \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} => } \mathit{Value}}$$

The validation rules for occurrences are similar to the rules for occurrences in matching.

$$\frac{\texttt{validate as } (\texttt{()} \texttt{ | } \mathit{Type}) \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} => } \mathit{Value}}{\texttt{validate as } \mathit{Type}\texttt{?} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} => } \mathit{Value}}$$

$$\frac{\texttt{validate as } (\mathit{Type} \texttt{ , } \mathit{Type}\texttt{*}) \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} => } \mathit{Value}}{\texttt{validate as } \mathit{Type}\texttt{+} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} => } \mathit{Value}}$$

$$\frac{\texttt{validate as } \mathit{Type}\texttt{+?} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} => } \mathit{Value}}{\texttt{validate as } \mathit{Type}\texttt{*} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} => } \mathit{Value}}$$

Validating a string against a simple type is defined in the previous section.

$$\frac{\texttt{simply validate as } \mathit{SimpleType} \texttt{ \{ } \mathit{String} \texttt{ \} => } \mathit{SimpleValue}}{\texttt{validate as } \mathit{SimpleType} \texttt{ \{ } \mathit{String} \texttt{ \} => } \mathit{SimpleValue}}$$

Validating an element against an element type is described by the following rule.

$$\frac{\begin{array}{c} \mathit{ElementType} \texttt{ yields } \mathit{ElementNameSet}\ \mathit{TypeSpecifier} \\ \mathit{TypeSpecifier} \texttt{ resolves to } \mathit{BaseTypeName} \texttt{ \{ } \mathit{Type} \texttt{ \}} \\ \mathit{ElementName} \texttt{ within } \mathit{ElementNameSet} \\ \texttt{validate as } \mathit{Type} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} => } \mathit{Value} \end{array}}{\begin{array}{c} \texttt{validate as } \mathit{ElementType} \texttt{ \{ element } \mathit{ElementName} \texttt{ \{ } \mathit{UntypedValue} \texttt{ \} \}} \\ \texttt{=> element } \mathit{ElementName} \texttt{ of type } \mathit{BaseTypeName} \texttt{ \{ } \mathit{Value} \texttt{ \}} \end{array}}$$

The element type yields an element name set and a type specifier, and the type specifier resolves to a base type name and a type. Then the given element matches the element type if two things hold: the element name must be within the element name set, and validating the untyped value against the type must yield a value. The resulting element has the element name, the base type name, and the validated value.

9 Ambiguity and the validation theorem

For a given type, validation takes an external representation (an untyped value) into an internal representation (a value annotated with types). For a given type, we would like each external representation to correspond to just one internal representation, and conversely. We show that this is the case if the type is unambiguous, using a characterization of validation in terms of erasure and matching.

Ambiguity. Validation is a judgment that relates a type and an untyped value to a value.

validate as *Type* { *UntypedValue* } => *Value*

In most of the examples we have seen, validation behaves as a function. That is, for a given type, for every untyped value, there is at most one value such that the above judgment holds. In this case, we say the type is *unambiguous*. But just as there is more than one way to skin a cat, sometimes there is more than one way to validate a value.

Here is an example of an ambiguous complex type:

```
  define element amb {
    element elt of type xs:float |
    element elt of type xs:string
  }

  validate as amb { <amb><elt>1</elt></amb> }
=>
  element amb { element elt of type xs:float { 1 } }

  validate as amb { <elt>1</elt> }
=>
  element amb { element elt of type xs:string { "1" } }
```

Here is an example of an ambiguous simple type:

```
  validate as xs:string* { "a b c" } => ("a b c")
  validate as xs:string* { "a b c" } => ("a b", "c")
  validate as xs:string* { "a b c" } => ("a", "b c")
  validate as xs:string* { "a b c" } => ("a", "b", "c")
```

There are well known algorithms for determining when regular expressions are ambiguous, and there are similar algorithms for regular tree grammars [12, 7]. These are easily adapted to give an algorithm for determining when a given type is ambiguous.

In Schema, the issue of ambiguity is resolved differently than here. Complex types are required to be unambiguous. Simple types have rules that resolve the ambiguity: every space is taken as a list separator, and in a union the first alternative that matches is chosen. Thus, for the first example above Schema deems the type illegal, while for the second example above Schema validation yields the last of the four possibilities.

Our formal model differs from Schema for two reasons. First, while Schema is concerned solely with validation against types written by a user, XQuery must also support type inference. And while it may be reasonable to require that a user write types that are unambiguous, it is not reasonable to place this restriction on a type inference system. For example, if expression e_0 has type `xs:boolean` and e_1 has type t_1 and e_2 has type t_2, the expression `if` (e_0) `then` e_1 `else` e_2 has type $t_1 \,|\, t_2$, and it is not reasonable to require that t_1 and t_2 be disjoint.

Second, defining validation as a relation rather than a function permits a simple characterization of validation in terms of matching and erasure, as given in the next section.

The validation theorem. We can characterize validation in terms of erasure and matching.

Theorem 1. *We have that*
`validate as` *Type* `{` *UntypedValue* `} =>` *Value*
if and only if
Value `matches` *Type*
and
Value `erases to` *UntypedValue*

The proof is by induction over derivations.

We would like to know that if we convert an external value to an internal value (using validation) and then convert the internal value back to an external value (using erasure) that we end up back where we started. This follows immediately from the validation theorem.

Corollary 1. *If*
`validate as` *Type* `{` *UntypedValue* `} =>` *Value*
and
Value `erases to` *UntypedValue'*
then
$UntypedValue = UntypedValue'$

Proof. From the first hypothesis and the validation theorem we have that
Value `erases to` *UntypedValue*
Taking this together with the second hypothesis and the fact that erasure is a function, the conclusion follows immediately. □

Similarly, we would like to know that if we convert an internal value of a given type to an external value (using erasure) and then convert the internal value back to an external value (using validation against that type) that we again end up back where we started, so long as the type is unambiguous. Again, this follows immediately from the validation theorem.

Corollary 2. *If*
Value `matches` *Type*
and
Value `erases to` *UntypedValue*
and
`validate as` *Type* `{` *UntypedValue* `} =>` *Value'*
and

Type is unambiguous
then
Value = *Value′*

Proof. By the validation theorem, we have that the first two hypotheses are equivalent to

`validate as` *Type* `{` *UntypedValue* `} =>` *Value*

Taking this together with the third hypothesis and the fact that validate is a function when the type is unambiguous, the conclusion follows immediately. □

References

1. Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation, February 1998.
2. Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. MSL - a model for W3C XML Schema. In *Proceedings of International World Wide Web Conference*, pages 191–200, Hong Kong, China, 2001.
3. James Clarke. XSL Transformations (XSLT) version 1.0. W3C Proposed Recommendation, October 1999.
4. James Clarke. TREX — Tree Regular Expressions for XML. Thai Open Source Software Center, February 2001.
5. James Clarke and Murata Makoto. RELAX NG specification. Oasis, December 2001.
6. Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion! In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, pages 177–188, Seattle, Washington, June 1998.
7. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1997.
8. Haruo Hosoya and Benjamin C. Pierce. XDuce: an XML processing language. In *International Workshop on the Web and Databases (WebDB'2000)*, Dallas, Texas, May 2000.
9. Michael Kay. XSL Transformations (XSLT) version 2.0. W3C Working Draft, April 2002.
10. Murata Makoto. Document description and processing languages – regular language description for XML (relax), October 2000.
11. Yannis Papakonstantinou and Victor Vianu. DTD inference for views of XML data. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, Dallas, Texas, May 2000.
12. Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages*. Springer-Verlag, 1997.
13. Henri S. Thompson, David Beech, Murray Maloney, and N. Mendelsohn. XML Schema part 1: Structures. W3C Recommendation, May 2001.
14. XPath 2.0. W3C Working Draft, April 2002.
15. XQuery 1.0: An XML query language. W3C Working Draft, April 2002.
16. XQuery 1.0 formal semantics. W3C Working Draft, March 2002.