XS

Draft 1.7
July 27, 2004

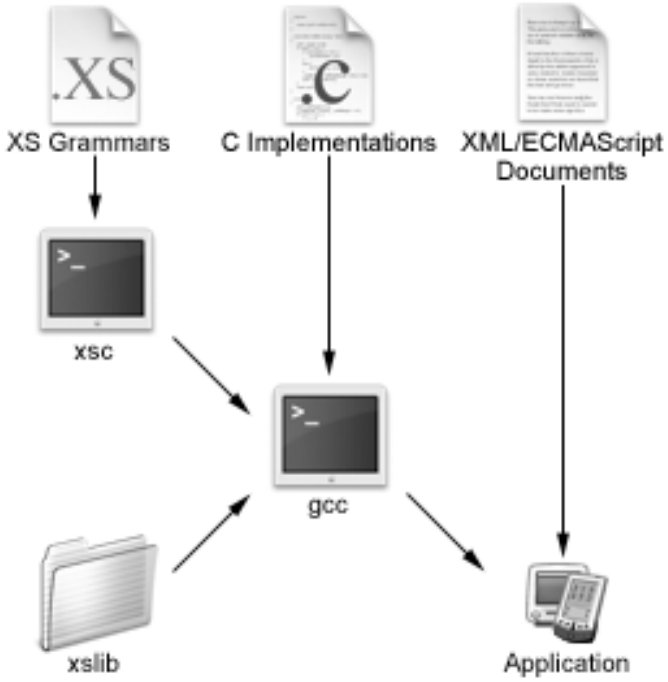
1 Introduction

XS is designed to develop tools and runtimes for standard based networked interactive multimedia applications on various devices. Applications are defined by XML and ECMAScript documents. Tools and runtimes are built in C to process and interpret such documents.

1.1 Context

By design, XML is "just" a markup language and ECMAScript is "just" a script language. XML processors and ECMAScript interpreters, like for instance James Clark's expat and Mozilla's javascript, expect to be used by tools and runtimes through C calls and callbacks. Tools and runtimes are semantically in charge, they are obviously responsible of what works and what does not work.

While most XML and ECMAScript efforts are focused on domain dependant and vendor independent specifications, the purpose of XS is "just" to help to build tools and runtimes. XS grammars bind XML descriptions, ECMAScript behaviors and C interfaces. XS grammars are themselves defined by XML and ECMAScript files. C implementations are defined by separate C files.



XS is both a command and a library. The core of the xslib library is of course an XML processor and an ECMAScript interpreter. The xsc command, like the yacc and antlr commands, transforms grammars into C code, data and interfaces that will be compiled and linked with C implementations and xslib to build tools and runtimes.

1.2 Background

XML and ECMAScript are well known standards but, in order to present XS, it is useful to highlight some notations...

1.2.1 XML

An XML document is a tree of element, attribute, processing instructions and text nodes. An element node has a name and can contain element, attribute processing instructions and text nodes. Attribute and processing instructions node have a name and contains one text node. An XML document has one and only one root which is an element node.

Example:

```
<root>
	<element attribute="text of attribute">
		text of element
	</element>
</root>
```

Like all markup languages, XML has always to be transformed into something else to make sense for such or such application. A significant part of the work on XML has been related to such transformations, for instance XSL applies templates to XML documents to convert them into other XML documents.

Example:

```
<xsl:template match:person>
	<p>
		<xsl:value-of select="name"/>
	</p>
</xsl:template>
```

1.2.2 ECMAScript

In ECMAScript, there are several types of value: undefined, null, boolean, date, number, regexp, string, function, and array or object. Arrays contain items: values accessed by index. Objects contain properties: values accessed by name. Globals are properties of an anonymous object.

Example:

```
gObject = {
	pBoolean: true,
	pNumber: 101,
	pRegexp: /[0-9]+/,
	pString: "hello",
	pFunction: function() { trace("hello") },
	pArray: [ "item 0", "item 1" ],
	pObject: { /* and so on */ }
}
```

ECMAScript is a prototype-based language: an object can inherit properties from another object, and so on... The inheriting object is the instance. The inherited object is the prototype.

The new operator invokes a constructor to build an instance. A constructor is like a function but has a property named "prototype" that becomes the prototype of the instances that the constructor builds.

Example

```
function P() {}
P.prototype = { p: 0 }
i = new P()
trace(i.p) // 0
```

1.3 Grammars

XS grammars are themselves defined by XML and ECMAScript files.

The two faces of XS emphasize the two programming techniques relevant to ECMAScript and XML: programming with prototypes and programming with templates.

1.3.1 Framework

Structurally, what XS builds from a grammar is a framework of ECMAScript prototypes. Such a framework is the global scope of the application.

For most ECMAScript properties, the name and the value attributes of the XS element become the name and the value of the ECMAScript property.

For ECMAScript functions, the corresponding XS element contains the text that becomes the sequence of instructions of the ECMAScript function.

For ECMAScript object, the corresponding XS element contains XS elements that become the properties of the ECMAScript object.

All prototype-based language but ECMAScript allow to define the hierarchy of prototypes explicitly instead of implicitly through constructors. XS allows both.

For ECMAScript functions, the prototype attribute of the corresponding XS element build the constructor. For ECMAScript objects, the prototype attribute of the corresponding XS element build the hierarchy of prototypes.

1.3.2 Parser and Serializer

Functionnally, what XS builds from a grammar are a parser and a serializer. The parser transforms an XML document into an ECMAScript instance, the serializer transforms an ECMAScript instance into an XML document.

XS elements have a name and a pattern attribute in order to define templates, the name is an ECMAScript identifier, the pattern is an XML element or attribute identifier.

XML documents are parsed from the root to the leaves. When the pattern of an XS element matches an XML node, the parser applies the template to the matched XML node to convert it into an ECMAScript property.

ECMAScript instances are serialized recursively. When the name of an XS element matches an ECMAScript property, the parser applies the template to the matched ECMAScript property to convert it into an XML node.

Once grammars are compiled by xsc and linked with xslib, the parser and the serializer are available in ECMAScript and in C.

In ECMAScript, the global object “xs” has two properties “parse” and “serialize” to convert an XML document into an ECMAScript instance and vice versa. The XML document is an ECMAScript string.

Example:

```
xml = xs.serialize(root)
root = xs.parse(xml)
```

In C there are macros to convert an XML document into an ECMAScript instance and vice versa from and to a C buffer or a C stream, see XS in C for details.

1.4 Sandbox

To provide useful features or fixes, for instance to update the system or the applications of a device, part of the distributed code needs to be trusted. The mechanism to secure the distribution of trusted code depends on the platform and usually relies on encryption and signature.

However, part of the distributed code does not need to be trusted if it is executed in a sandbox: an environment designed by the system or the applications to prevent untrusted code to harm the device. XS allows a system or applications to build such an environment.

1.4.1 Framework and Scripts

The xslib library is an ECMAScript interpreter: text nodes of XML documents can be scripts to be executed. And, in C, there are macros to execute a script from a C buffer or a C stream, see XS in C for details.

Most ECMAScript interpreters run in a framework developed with a separate language. The framework decides which features are available to scripts: structurally ECMAScript code is always executed in a sandbox

In XS, the framework itself is mainly developed with ECMAScript. Grammars define which properties are visible or invisible to scripts with the script attribute. Properties available to scripts build the sandbox, the programming interface between the framework and scripts.

Example:

```
<!-- grammar -->
<object name="foo">
      <string name="visible" value="foo"/>
      <string name="invisible" value="foo" script="false"/>
</object>

// script
trace(foo.visible) // foo
trace(foo.invisible) // undefined
```

The xslib library extends the specification of the object prototype to allow the framework to test if a property can be used by scripts with Object.prototype.isScriptable().

Example:

```
<function name="test">
      if (this.isScriptable("foo")
            trace("foo can be used by scripts”)
</function>
```

1.4.2 Runtime properties

Since ECMAScript allows the creation of properties at runtime, a script could create, with or without intent, a property that would clash with an invisible property of the framework.

To avoid dependencies between the framework and scripts beyond their programming interface, XS allows scripts to create such properties but they are invisible to the framework.

So it is possible for an instance to have two properties with the same name, one invisible to scripts, one invisible to the framework.

Example:

```
<!-- grammar -->
<object name="foo">
      <string name="visible" value="foo"/>
      <string name="invisible" value="foo" script="false"/>
</object>

// script
foo.visible = "goo"// the visible property is shared
foo.invisible = "goo"// foo has two invisible properties…
```

The xslib library extends the specification of the object prototype to allow the framework to use properties like if it was a script. It is sometimes convenient, for instance to install properties at runtime that are meaningful only to scripts.

Example:

```
<function name="test" params="it">
      trace(it.sandbox.invisible)
      it.sandbox.invisible = "goo"
      delete sandbox.invisible
      for (var i in it.sandbox)
            trace(it.sandbox[i])
</function>
```

The value of Object.prototype.sandbox is a handle that allows the framework to call, delete, enumerate, get and set runtime properties created by scripts. In C there is an equivalent macro, see XS in C for details.

1.4.3 Runtime calls

So the xlib library can execute code out of the sandbox (framework code) or in the sandbox (script code). Through function or constructor properties, framework code can call or invoke script code and vice versa.

At runtime, framework code can test if it has been called by script code, directly or indirectly. The result of the test is a number, 0 means called by framework code, 1 means called directly by script code, more than 1 means called indirectly by script code. In C there is an equivalent macro, see XS in C for details.

Example:

```
<function name="test">
      if (xs.script())
            trace("called by script code")
      else
            trace("called by framework code")
</function>
```

2 Packages

The root element of an XS grammar is the package element.

2.1 Import

A package can import another package with the import element. An imported package can import another package, and so on, but a package cannot import itself, directly or indirectly.

The value of the link attribute is “static” or “dynamic”. It is “static” by default.

xsc links implementations of all statically imported packages together. Dynamically imported packages are only interfaces and will be linked at runtime, see XS in C for details.

Example:

```
<package>
	<import href="stream.xs" link="dynamic"/>
	<import href="soap.xs"/>
</package>
```

2.2 Namespace

A package can define namespaces with the namespace element. The namespace element binds a prefix to a uri with its prefix and uri attributes. Without prefix attribute, the namespace element defines the default namespace.

In an XML document, the default namespace does not apply to attribute nodes. Similarly, in an XS grammar, the default namespace for patterns does not apply to patterns that match attribute nodes.

The parser uses the uri attribute of the namespace to match the uri of the namespace of XML documents. The serializer uses the prefix and the uri attributes to declare namespaces and qualify names of attributes, elements and processing instructions.

Example:

```
<package>
	<namespace uri="http://www.w3.org/2001/svg/"/>
	<namespace prefix="smil" uri="http://www.w3.org/2001/SMIL20/"/>
	<!-- animated vector graphics -->
</package>
```

2.3 Patch

A package can change the prototype of another package with the patch element. The patch element can add, remove and modify properties.

Example:

```
<!-- main.xs -->
<package>
	<object name="foo">
		<string name="b" pattern="@b"/>
		<string name="c" pattern="@c"/>
	</object>
</package>

<!-- change.xs -->
<package>
	<import href="main.xs" link="dynamic"/>
	<patch prototype="foo">
		<string name="a" pattern="@aa"/> <!-- added -->
		<undefined name="b"/> <!-- removed -->
		<string name="c" pattern="@cc"/> <!-- modified -->
	</patch>
</package>
```

The patch element changes the framework but it also changes the parser and the serializer.

2.4 Program

With the program element, a package can contain sequences of ECMAScript instructions to execute when the package is initialized.

The sequence of ECMAScript instructions can be in the package itself or in a separate file with the href attribute.

The sequence of ECMAScript instructions can be replaced by a C function with the c attribute.

Example:

```
<package>
      <program>
            trace("wow");
      </program>
      <program href="wow.s"/>
      <program c="Wow"/>
</package>
```

2.5 Target

The target element and the –t option of xsc allow to include or exclude part of a grammar into or from a build.

The name attribute can be a boolean expression with !, &&, || and parenthesis.

The target element can contain patch, program and property elements but neither import nor namespace elements.

Example:

```
<!-- use xsc -t EDITOR to build the editor -->
<package>
      <target name="EDITOR">
            <function name="save" params="document" c="save"/>
      </target>
</package>
```

3 Properties

Except for the package, import, namespace, patch, program and target elements, all elements of XS are property elements: they prototype, parse and serialize ECMAScript properties. All property elements have name, pattern, delete, enum, set and script attributes.

The name attribute contains the name of the ECMAScript property that the property element prototypes and instantiates. If the property element is a child of the package element, the name attribute contains the name of an ECMAScript global.

The pattern attribute contains the pattern to be matched in an XML document to instantiate the ECMAScript property. By default the value is inherited from the prototype of the instance that owns the property. If there is no prototype, by default the value is “”, which means that the property is neither parsed nor serialized.

The delete, enum and set attributes contain the value of the attributes of the ECMAScript property. By default the value is inherited from the prototype of the instance that owns the property. If there is no prototype, by default the value is true, which means that the property can be deleted, enumerated and set.

The script attribute defines if the property is visible or invisible to scripts. By default the value is inherited from the prototype of the instance that owns the property. If there is no prototype, by default the value is true, which means that the property can be used by scripts.

3.1 Examples...

In the examples hereunder, the **Grammar** paragraph presents the grammar itself, the **Prototype** paragraph presents the ECMAScript instructions corresponding to what the grammar prototypes, the **Document** paragraph presents a compatible XML document and the **Instance** paragraph presents the ECMAScript expressions corresponding to what the grammar instantiates from the XML document.

The **Grammar** and **Document** paragraphs are examples of code in XML and ECMAScript. For the sake of terseness, the XML headers and the package elements are omitted.

On the other hand, the **Prototype** and **Instance** paragraphs are provided only to explain what the grammar prototypes and instantiates, they are no examples of code, sometimes they are just comments because ECMAScript has no corresponding instructions or expressions.

3.2 Object

The object element is the main building block of XS. It is the only property element that can contain other property elements.

Grammar:

```
<object name="envelop" pattern="/Envelop">
      <object name="header" pattern="Header"/>
      <object name="body" pattern="Body"/>
</object>
```

Prototype:

```
envelop = { header: {}, body: {} }
```

Document:

```
<Envelop>
      <Header>
      </Header>
      <Body>
      </Body>
</Envelop>
```

Instance:

```
{     /* proto: envelop */
      header: { /* proto: envelop.header */ },
      body: { /* proto: envelop.body */ }
}
```

3.3 Function

The function element is a property element: it prototypes an ECMAScript function property and contains the sequence of ECMAScript instructions to execute when the function is called.

When the function element instantiates an ECMAScript property, the text node of the matched node becomes the sequence of ECMAScript instructions to execute when the function is called.

The params attribute contains the parameters of the function.

Grammar:
```
<function name="onClick" params="where, when" pattern="/click">
      trace("onClick " + where + " " + when)
</function>
```

Prototype:
```
onClick = function(where, when) {
      trace("onClick " + where + " " + when)
}
```

Document:
```
<click>
      this.goToEnd()
</click>
```

Instance:
```
function(where, when) {
      this.goToEnd()
}
```

To edit function instances, tools need to access and assign their source code.

The Function constructor allows to create function instances, its parameters set the arguments and the body of the function. The xslib library extends the specification of the function prototype to allow getting the arguments and the body of the function separately.

The value of Function.prototype.arguments is a string that represents the arguments of the function. The value of Function.prototype.body is a string that represents the body of the function. These properties cannot be deleted, cannot be enumerated, and cannot be set.

Example:
```
<object name="page" pattern="/page">
      <function name="onClick" params="x, y" pattern="onClick"/>
</object>
<function name="Page" params="body" prototype="page">
      this.onClick = new Function(this.onClick.arguments, body)
</function>
<script>
      aPage = new Page("trace('onClick')"))
</script>
```

The serializer converts the body of the function into a text node. Nothing is serialized if the function property is owned by a prototype instead of an instance.

3.4 Host Object, Host Function

The object element prototypes a host object if it has a c attribute. The c attribute contains the name of C function to execute when the object is destroyed by the garbage collector.

The function element prototypes a host function if it has a c attribute. The c attribute contains the name of a C function to execute when the function is called. In that case, the function element must be an empty element.

A host object is a special kind of object with data that can only be handled in C. Host objects and host functions are often used together to provide system features to applications. See "XS in C" for details.

Grammar:

```
<object name="window" c="DestroyWindow">
	<function name="close" params="" c="CloseWindow"/>
	<function name="move" params="x, y" c="MoveWindow"/>
	<function name="resize" params="w, h" c="ResizeWindow"/>
	<function name="setTitle" params="s" c="SetWindowTitle"/>
</object>
```

Prototype:

```
window = {
	/* data: host data */
	/* close: host function */
	/* move: host function */
	/* resize: host function */
	/* setTitle: host function */
}
```

3.5 Constructor, Getter, Setter

Constructors, getters and setters are variants of function or host function.

The function element can have a prototype attribute that contains the name of an object property element. When invoked by the new operator, the constructor builds an instance of such a prototype.

If the name of the function element begins by "get " or "set ", the function element prototypes a getter or a setter. The end of the name of the function element becomes the name of the property. The function will be executed when the property is accessed or assigned.

Grammar:

```
<object name="screen">
	<function name="get width" c="ScreenGetWidth"/>
	<function name="get height" c="ScreenGetHeight"/>
</object>
<function name="Screen" prototype="screen"/>
<script>
	var s = new Screen();
	trace("width=" + s.width);
	trace("height=" + s.height);
</script>
```

Prototype:

```
screen = {
	/* get width: native function */
	/* get height: native function */
}
Screen = function() {}
Screen.prototype = screen
var s = new Screen();
trace("width=" + s.width);
trace("height=" + s.height);
```

3.6 Boolean, Date, Number, Regexp, String

The boolean, date, number, regexp and string elements are literal property elements: they prototypes and instantiates ECMAScript literal properties.

The value attribute contains the value of the ECMAScript property that the element prototypes. The default values are "false", "01 Jan 1970 00:00:00 GMT", "0", "//" and "".

When the element instantiates an ECMAScript property, the text node of the matched node becomes the value of the ECMAScript property.

Grammar:

```
<object name="user" pattern="/user"/>
      <string name="name" value="anonymous" pattern="@name"/>
      <boolean name="nice" pattern="@nice"/>
      <number name="age" value="-1" pattern="age"/>
      <regexp name="selection" pattern="selection"/>
</object>
```

Prototype:

```
user = {
      name: "anonymous",
      nice: false,
      age: -1,
      selection: //
}
```

Document:

```
<user name="Flore" nice="true">
      <age>9</age>
</user>
```

Instance:

```
{     /* proto: user */
      name: "Flore",
      nice: true,
      age: 9
}
```

The serializer calls the toString function of the value to convert it into a text node. Nothing is serialized if the literal property is owned by a prototype instead of an instance.

3.7 Custom

The custom element is a special kind of literal property element. Its io attribute contains the name of an object property element.

The parser calls the "parse" property of the io to convert both the text node of the value attribute for the prototyped property and the text node of the matched node for the instantiated property

Grammar:

```
<object name="colorIO">
      <function name="parse" params="text" c="colorIOParse"/>
      <function name="serialize" params="color" c="colorIOSerialize"/>
</object>
<custom name="color" value="#000000" pattern="/color" io="colorIO"/>
```

Prototype:

```
color = { r: 0, g: 0, b: 0 }
```

Document:
```
<color>#FF8000</color>
```

Instance:
```
{ r: 255, g: 128, b: 0 }
```

The serializer calls the "serialize" property of the io to converts the value into a text node. Nothing is serialized if the custom property is owned by the prototype instead of an instance.

3.8 Array

The array element is a property element: it prototypes and instantiates an ECMAScript array property.

The contents attribute contains the names of other property elements that will be used to instantiate items of the ECMAScript array property. Names are separated by commas.

Grammar:
```
<object name="family" pattern="/Family">
      <string name="member" pattern="Member"/>
      <array name="members" contents="family.member" pattern="."/>
</object>
```

Prototype:
```
family = { member = "", members = [] }
```

Document:
```
<Family>
      <Member>Flore</Member>
      <Member>Veronique</Member>
      <Member>Patrick</Member>
</Family>
```

Instance:
```
{ /* proto: family */
  members = [ "Flore", "Veronique", "Patrick" ]
}
```

To serialize the items of the array property, the serializer iterates from zero to the length of the array property. Nothing is serialized if the array property is owned by the prototype instead of an instance or if the array property is empty.

3.9 Reference

The reference element is a property element: it prototypes and instantiates any ECMAScript property by reference.

The contents attribute contains the names of other property elements that will be used to prototype and instantiate the ECMAScript property. Names are separated by commas.

Grammar:
```
<object name="speaker" pattern="speaker">
      <number name="volume" pattern="@volume"/>
</object>
<object name="stereo" pattern="/stereo">
      <reference name="left" contents="speaker" pattern="left"/>
      <reference name="right" contents="speaker" pattern="right"/>
```

```
</object>
```

Prototype:
```
speaker = { volume: 0 }
stereo = { left: speaker, right: speaker }
```

Document:
```
<stereo>
      <left>
            <speaker volume="10"/>
      </left>
      <right>
            <speaker/>
      </right>
</stereo>
```

Instance:
```
{     /* proto: stereo */
      left: { /* proto: speaker */ volume: 10 },
      right: { /* proto: speaker */ }
}
```

3.10 Inheritance and Conformance

An object element can have a prototype attribute, which must be the identifier of another object element, its prototype.

An object element inherits from its prototype, which means it contains implicitly the properties of its prototype. An object element overrides a property if it contains explicitly a property element with the same name.

An object element conforms to its prototype, which means it can be contained by the array or reference elements that contain its prototype, the prototype of its prototype, and so on...

A prototype can inherit from and conform to another prototype, and so on, but a prototype cannot inherit from and conform to itself, directly or indirectly.

Grammar:
```
<array pattern="items" contents="item" pattern="/items"/>

<object name="item">
      <number name="x" pattern="@left"/>
      <number name="y" pattern="@top"/>
      <number name="width" pattern="@width"/>
      <number name="height" pattern="@height"/>
</object>

<object name="box" prototype="item" pattern="box">
      <custom name="color" pattern="@color" c="ParseColor"/>
</object>

<object name="img" prototype="item" pattern="img">
      <string name="uri" pattern="@src"/>
</object>
```

Prototype:
```
items = []
item = { x: 0, y: 0, width: 0, height: 0 }
box = { /* proto: item */ color: undefined }
```

```
img = { /* proto: item */ uri: "" }
```

Document:
```
<items>
      <box color="#808080" width="320" height="240">
      <img src="a.png" left="80" top="60" width="160" height="120"/>
</items>
```

Instance:
```
[
      {     /* proto: box */
            color: { r: 128, g: 128, b: 128 },
            width: 320, height: 240
      },
      {     /* proto: img */
            url: "a.png",
            x: 80, y: 60, width: 160, height: 120
      }
]
```

4 Patterns

Patterns use a strict subset of XPath. Such a subset has been selected to allow to stream XML documents instead of requiring to load XML documents entirely in order to randomly access their nodes.

4.1 Root Pattern

The pattern attribute matches the root of the tree if it is starts with / (slash). Only object elements can have a root pattern.

It is necessary to have an object element with a root pattern to match the root of the parsed document. However such an object can still be part of the contents of an array or reference element, the root pattern is then used like an element pattern, without the slash.

Grammar:
```
<object name="smil" pattern="/smil">
      <object name="head" pattern="head"/>
      <object name="body" pattern="body"/>
</object>
```

Prototype
```
smil = { head: {}, body: {} }
```

Document:
```
<smil>
      <head>
      </head>
      <body>
      </body>
</smil>
```

Instance:
```
{ head: {}, body: {} }
```

4.2 Element Pattern

The pattern attribute matches an element node if it starts with a letter. All property elements can have an element pattern.

Grammar:
```
<object name="page" pattern="/page">
      <number name="width" pattern="width"/>
      <number name="height" pattern="height"/>
</object>
```

Prototype
```
page = { width: 0, height: 0 }
```

Document:
```
<page>
      <width>320</width>
      <height>240</height>
</page>
```

Instance:
```
{ width: 320, height: 240 }
```

4.3 Attribute Pattern

The pattern attribute matches an attribute node if it is starts with @ (at). Only literal elements can have an attribute pattern.

Grammar:
```
<object name="user">
      <string name="name" pattern="@name"/>
</object>
<array name="users" contents="user" pattern="/users"/>
```

Prototype
```
user = { name: "" }
 users = [];
```

Document:
```
<users>
      <user name="Flore"/>
</users>
```

Instance:
```
[
       {    /* proto: user */
            name: "Flore"
       },
]
```

4.4 Processing Instruction Pattern

The pattern attribute matches a processing instruction node if it is starts with ? (question mark). Only literal elements can have a processing instruction pattern.

Grammar:
```
<object name="command">
      <function name="execute" pattern="?execute"/>
</object>
<array name="commands" contents="command" pattern="/commands"/>
```

Prototype

```
command = { execute: function() {} }
commands = [];
```

Document:
```
<commands>
      <command>
            <?execute debugger?>
</command>
</commands>
```

Instance:
```
[
       {    /* proto: command */
            execute: function() {debugger }
       },
]
```

4.5 Current Pattern

The pattern attribute matches the current node if it is a dot. All property elements can have an element pattern.

The current pattern is useful to insert object, array or reference elements in the absence of relevant nodes.

Grammar:
```
<string name="item" pattern="item"/>
<object name="collection" pattern="/collection">
      <array name="items" contents="item" pattern="."/>
</object>
```

Prototype
```
item = ""
collection = { items: [] }
```

Document:
```
<collection>
      <item>Flore</item>
      <item>Patrick</item>
      <item>Veronique</item>
</collection>
```

Instance:
```
{ items: [ "Flore", "Patrick", "Veronique" ] }
```

4.6 Nested Pattern

Element, attribute and processing instruction patterns can be paths to match nodes further in the tree. It is useful to skip irrelevant nodes.

Grammar:
```
<object name="page" pattern="/page">
      <number name="width" pattern="head/layout/@width"/>
      <number name="height" pattern="head/layout/@height"/>
</object>
```

Prototype
```
page = { width: 0, height: 0 }
```

Document:

```
<page>
	<head>
		<layout width="320" height="240"/>
	</head>
</page>
```

Instance:
```
{ width: 320, height: 240 }
```

4.7 Empty Pattern

The pattern attribute matches no node if it is empty. All property elements can have an empty pattern.

The empty pattern is useful to cancel an inherited pattern.

Grammar:
```
<object name="user">
	<string name="name" pattern="@name"/>
</object>
<object name="self" prototype="user">
	<string name="name" pattern="" value="self"/>
</object>
<array name="users" contents="user" pattern="/users"/>
```

Prototype
```
user = { name: "" }
self = { name: "self" }
users = [];
```

Document:
```
<users>
	<user name="Flore"/>
	<self/> <!-- no name attribute allowed-->
</users>
```

Instance:
```
[
	{	/* proto: user */
		name: "Flore"
	},
	{	/* proto: self */
	}
]
```

5 Reference

5.1 Names

Properties names are ECMAScript identifiers. They can be used by ECMAScript expressions and instructions in XML documents.

Like ECMAScript, XS does not scope identifiers by package. All identifiers share the global scope. Object properties can be used to structure the global scope.

Nested properties names can be referenced in the contents and prototype attributes with the ECMAScript dot notation.

5.2 Patterns

Patterns are like paths. If a pattern begins with slash (/), it is an absolute path and it is matched from the root, else it is a relative path and it is matched from the current node. One dot (.) means the current node. "foo" means the element node with the name foo. Patterns have two special forms: "@foo" means the attribute node with the name foo and "?foo" means the processing instruction node with the name foo.

A grammar can define several object elements with root patterns that would match the same XML document. But if several properties elements have patterns that would match the same node of an XML document, the grammar is ambiguous.

Properties elements with pattern can be nested or referenced by properties elements without pattern, in that case their pattern is useless.

5.3 xs:array

The xs:array element is a property element that prototypes, parses and serializes an array property from and to an element node.

Attributes

`name, required`
: The name of the property.

`contents, required`
: The name of another property element that parses and serializes the items of the array.

`delete, inherited, default="true"`
`enum, inherited, default="true"`
`set, inherited, default="true"`
: The attributes of the property.

`script, inherited, default="true"`
: If the script attribute is "false", the property cannot be scripted: the property can only be used by grammars.

`pattern, inherited, default=""`
: The pattern of an element node. If the pattern attribute is empty, the property is neither parsed nor serialized.

Elements

None.

Text

None.

5.4 xs:boolean

The xs:boolean element is a property element that prototypes, parses and serializes a boolean property from and to a text node.

Attributes

`name, required`
: The name of the property.

`value, optional, default="false"`
: The value of the property.

`delete, inherited, default="true"`
`enum, inherited, default="true"`
`set, inherited, default="true"`
: The attributes of the property.

`script, inherited, default="true"`
: If the script attribute is "false", the property cannot be scripted: the property can only be used by grammars.

`pattern, inherited, default=""`

The pattern of a node that contains a text node. If the pattern attribute is empty, the property is neither parsed nor serialized. Else the text node becomes the value of the property and vice versa.

Elements
None
Text
None.

5.5 xs:custom

The xs:custom element is a property element that prototypes, parses and serializes a property from and to a text node with custom "parse" and "serialize" functions.

Attributes

`io, required`
The name of an object with a "parse" function that converts a text node into a value and a "serialize" function that converts a value into a text node.

`name, required`
The name of the property.

`value, optional, default=""`
The value of the property, once converted by the "parse" function of the io.

`delete, inherited, default="true"`
`enum, inherited, default="true"`
`set, inherited, default="true"`
The attributes of the property.

`script, inherited, default="true"`
If the script attribute is "false", the property cannot be scripted: the property can only be used by grammars.

`pattern, inherited, default=""`
The pattern of a node that contains a text node. If the pattern attribute is empty, the property is neither parsed nor serialized. Else the text node becomes the value of the property, once converted by the "parse" function of the io, and vice versa, once converted by the "serialize" function of the io.

Elements
None
Text
None.

5.6 xs:date

The xs:date element is a property element that prototypes, parses and serializes a date property from and to a text node.

Attributes

`name, required`
The name of the property.

`value, optional, default="01 Jan 1970 00:00:00 GMT"`
The value of the property.

`delete, inherited, default="true"`
`enum, inherited, default="true"`
`set, inherited, default="true"`
The attributes of the property.

`script, inherited, default="true"`
If the script attribute is "false", the property cannot be scripted: the property can only be used by grammars.

`pattern, inherited, default=""`

The pattern of a node that contains a text node. If the pattern attribute is empty, the property is neither parsed nor serialized. Else the text node becomes the value of the property and vice versa.

Elements
None
Text
None.

5.7 xs:function

The xs:function element is a property element that prototypes, parses and serializes a function property from and to a text node.

Attributes

`name, required`
The name of the property. If it begins by "get " or "set " the function is a getter or a setter and its end is the name of the property.

`params, optional, default=""`
The parameters of the function.

`c, optional`
The name of a C function to execute when the function is called. If the c attribute exists, the xs:function must be an empty element.

`delete, inherited, default="true"`
`enum, inherited, default="true"`
`set, inherited, default="true"`
The attributes of the property.

`script, inherited, default="true"`
If the script attribute is “false”, the property cannot be scripted: the property can only be used by grammars.

`prototype, optional, default="Object.prototype"`
The name of an object property element. When invoked by the new operator, the function creates an instance of such a prototype.

`pattern, inherited, default=“”`
The pattern of a node that contains a text node. If the pattern attribute is empty, the property is neither parsed nor serialized. Else the text node becomes the sequence of ECMAScript instructions to execute when the function is called.

Elements
None
Text
None if the c attribute exists, else the sequence of ECMAScript instructions to execute when the function is called.

5.8 xs:import

The xs:import element imports a package.

Attributes

`href, required`
The URI of the package to import. Importation is transitive but acyclic.

`link, optional, default="static"`
If the link attribute is “static”, code is generated to define the properties of the package. If the link attribute is “dynamic”, no code is generated, the properties are assumed to be available at runtime.

Elements
None
Text
None

5.9 xs:number

The xs:number element is a property element that prototypes, parses and serializes a number property from and to a text node.

Attributes

`name, required`
: The name of the property.

`value, optional, default="0"`
: The value of the property.

`delete, inherited, default="true"`
`enum, inherited, default="true"`
`set, inherited, default="true"`
: The attributes of the property.

`script, inherited, default="true"`
: If the script attribute is “false”, the property cannot be scripted: the property can only be used by grammars.

`pattern, inherited, default=“”`
: The pattern of a node that contains a text node. If the pattern attribute is empty, the property is neither parsed nor serialized. Else the text node becomes the value of the property and vice versa.

Elements

None

Text

None.

5.10 xs:null

The xs:null element is a property element that prototypes a null property.

Attributes

`name, required`
: The name of the property.

`delete, inherited, default="true"`
`enum, inherited, default="true"`
`set, inherited, default="true"`
: The attributes of the property.

`script, inherited, default="true"`
: If the script attribute is “false”, the property cannot be scripted: the property can only be used by grammars.

Elements

None.

Text

None.

5.11 xs:object

The xs:object element is a property element that prototypes, parses and serializes an object property from and to an element node.

Attributes

`name, required`
: The name of the property.

`delete, inherited, default="true"`
`enum, inherited, default="true"`
`set, inherited, default="true"`
: The attributes of the property.

`script, inherited, default="true"`
: If the script attribute is “false”, the property cannot be scripted: the property can only be used by grammars.

`c, optional`

The name of a C function to execute when the object is destroyed by the garbage collector. If the c attribute exists, the xs:object prototypes and instantiates a host object.

`prototype, optional, default="Object.prototype"`

The name of another object property. The object property inherits from its prototype and conforms to its prototype. Inheritance and conformance are transitive but acyclic.

`pattern, inherited, default=“”`

The pattern of an element node. If the pattern attribute is empty, the property is neither parsed nor serialized. Else the element node becomes the value of the property and vice versa.

Elements

Zero or more property, program or target elements.

Text

None.

5.12 xs:namespace

The xs:namespace element defines a namespace for patterns in the package.

Attributes

`prefix, optional, default=""`

The prefix to bind to the uri for patterns in the package. If the prefix attribute is empty, the xs:namespace defines the default namespace.

`uri, required`

The uri to bind to the prefix for patterns in the package.

Elements

None

Text

None

5.13 xs:package

The xs:package element is the root element.

Attributes

None

Elements

Zero or more import, namespace, patch, program and property elements.

Text

None

5.14 xs:patch

The xs:patch element contains property elements to modify, add to or remove from an object property.

Attributes

`prototype, required`

The name of the object property to patch.

Elements

Zero or more property elements.

Text

None

5.15 xs:program

The xs:program element contains a sequence of ECMAScript instructions to execute when the package is initialized.

Attributes

`c, optional`

The name of a C function to execute when the package is initialized. If the c attribute exists, the xs:program must be an empty element without href attribute.

`href, optional`

The URI of a file that contains the sequence of ECMAScript instructions. If the href attribute exists, the xs:program must be an empty element without c attribute.

Elements

None

Text

None if the c or href attribute exists, else the sequence of ECMAScript instructions.

5.16 xs:reference

The xs:reference element is a property element that prototypes, parses and serializes any property by reference.

Attributes

`name, required`

The name of the property.

`contents, required`

The name of another property element that prototypes, parses and serializes the value of the property.

`delete, inherited, default="true"`
`enum, inherited, default="true"`
`set, inherited, default="true"`

The attributes of the property.

`script, inherited, default="true"`

If the script attribute is “false”, the property cannot be scripted: the property can only be used by grammars.

`pattern, inherited, default=“”`

The pattern of an element node.

Elements

None.

Text

None

5.17 xs:regexp

The xs:regexp element is a property element that prototypes, parses and serializes a regexp property from and to a text node.

Attributes

`name, required`

The name of the property.

`value, optional, default="//"`

The value of the property.

`delete, inherited, default="true"`
`enum, inherited, default="true"`
`set, inherited, default="true"`

The attributes of the property.

`script, inherited, default="true"`

If the script attribute is “false”, the property cannot be scripted: the property can only be used by grammars.

`pattern, inherited, default=“”`

The pattern of a node that contains a text node. If the pattern attribute is empty, the property is neither parsed nor serialized. Else the text node becomes the value of the property and vice versa.

Elements

None

Text

None.

5.18 xs:string

The xs:string element is a property element that prototypes, parses and serializes a string property from and to a text node.

Attributes

`name, required`
: The name of the property.

`value, optional, default=""`
: The value of the property.

`delete, inherited, default="true"`
`enum, inherited, default="true"`
`set, inherited, default="true"`
: The attributes of the property.

`script, inherited, default="true"`
: If the script attribute is “false”, the property cannot be scripted: the property can only be used by grammars.

`pattern, inherited, default=“”`
: The pattern of a node that contains a text node. If the pattern attribute is empty, the property is neither parsed nor serialized. Else the text node becomes the value of the property and vice versa.

Elements

None

Text

None.

5.19 xs:target

The xs:target element includes the elements it contains in the grammar if its name attribute evaluates to true.

Attributes

`name, required`
: A boolean expression with names, !, &&, || and parenthesis. Names that match a –t option of xsc are true.

Elements

Zero or more patch, script and property elements.

Text

None.

5.20 xs:undefined

The xs:undefined element is a property element that prototypes an undefined property.

Attributes

`name, required`
: The name of the property.

`delete, inherited, default="true"`
`enum, inherited, default="true"`
`set, inherited, default="true"`
: The attributes of the property.

`script, inherited, default="true"`
: If the script attribute is “false”, the property cannot be scripted: the property can only be used by grammars.

Elements

None.

Text

None.

6 xsc

Here is a summary of the parameters of xsc

file.xs
: The grammar to build. All grammars statically imported by the grammar are built too.

-b
: Binary. Instead of generating byte codes into the .c file, xsc generates byte codes into a binary file with the .xsb extension.

-d
: Debug. xsc generates byte codes to allow xsbug to follow the execution in the source code.

-i directory
: Input. Directory where to search imported grammars. There can be several –i options.

-o directory
: Output. Directory where to create the .c, .h and .xsb files.

-t name
: Target. Name to be matched by target elements to include the elements they contain. There can be several –t options.

-v
: Verbose.

---

[i] This document is part of xslib.