An XSD 1.1 Schema Validator Written in XSLT 3.0

Michael Kay, Saxonica

Abstract

The paper describes an implementation of an XML Schema validator written entirely in XSLT 3.0. The main benefit of doing this is that the validator becomes available on any platform where XSLT 3.0 is implemented. At the same time, the experiment provides a valuable test of the performance and usability of an XSLT 3.0 implementation, and the experience gained from writing this application can feed back improvements to the XSLT processor that will benefit a large range of applications.

1. Introduction

This paper describes the outline design of an XML Schema (XSD 1.1) validator written entirely in XSLT 3.0.

There were two reasons for undertaking this project:

- Firstly, we see a demand to make implementations of the latest XML standards from W3C available on a wider variety of platforms. The base standards (XML 1.0, XPath 1.0, XSLT 1.0, XSD 1.0) that came out between 1998 and 2001 were very widely implemented, but subsequent refinements of these standards have seen far less vendor support. One can identify a number of reasons for this: perhaps the first versions of the standards met 90% of the requirements, and the market for subsequent enhancements is therefore much smaller; perhaps the early implementors found that their version 1 products were commercially unsuccessful and they therefore found it difficult to make a business case for further investment. In the case of products like libxml/libxslt produced by enthusiasts working in their spare time, perhaps the enthusiasts are more excited by a brand new technology than by version 2 of something well established. Whatever the causes, XSD 1.1 in particular only has three known implementations (Altova, Saxon, and Xerces) compared with dozens of implementations of XSD 1.0, despite the fact that there is a high level of consensus in the user community that the new features are extremely useful.

- Secondly, as editor of the XSLT 3.0 specification and developer of a leading implementation of the language, I felt a need to "get my hands dirty" by developing an application in XSLT 3.0 that would stretch the capabilities of the language. I am often called on to provide advice to XSLT 3.0 users, and even now that the language specification is complete, I find myself designing extensions to meet requirements that go beyond the base standard, and this can only be done on the basis of personal experience in using the language "in anger" for real applications. In addition, the usability of a language compiler depends in high measure on the quality of diagnostics available, and improving the quality of diagnostics depends entirely on a feedback process: you need to make real programming mistakes, spot when the diagnostics could be more helpful, and make the required changes.

More specifically, Saxonica has released an implementation of XSLT 3.0 written in Javascript to run in the browser, and an obvious next step is to extend that implementation to run under Node.js on the server. But for the server environment we need to implement a larger subset of the language, and so the question arose, should we write a version of the schema processor in Javascript? However, Javascript is not the only language of interest: we're also seeing demand for our technology from users whose programming environment is C, C ++, C#, PHP, or Python. Since (using a variety of tools) we've been trying to deliver XSLT 3.0 capability in all these environments, a natural next step is to try and deliver a schema processor that will run on any of these platforms, by writing it in portable XSLT 3.0 code.

Architecturally, there are two parts to an XSD schema processor:

- The first part is the schema compiler. This takes as input the source XSD schema documents, and performs as much pre-processing of this input as it can prior to the validation of actual XML instances against the schema. This preprocessing includes verification that the schema documents themselves represent a valid schema, and other tasks such as expanding defaults. The XSD specification from W3C describes a "schema component model" which is an abstract description of a data structure that can be used to represent a compiled schema; and it gives detailed rules for translating source schema documents to this abstract data model. The Saxon schema processor follows this approach very closely, and it also defines a concrete XML-based representation of the schema component model which realises the output of the schema compilation phase in concrete form. This output in fact contains a little more than the SCM components and their properties; it also contains a representation of the finite state machines used to implement the grammar of complex types defined in the schema.

- The second part is the instance validator. The instance validator takes two inputs: the compiled schema (SCM) output by the schema compiler, and an XML instance document to be validated. In principle its output is a PSVI (post schema validation infoset) containing the instance document as a tree, decorated with information derived from schema validation, including information identifying nodes that are found to be invalid. In practice, in the Saxon implementation, there are two outputs: a validation report, which in its canonical form is an XML document listing all the validation errors that were found, and a tree representation of the instance document with added type annotations, in the form prescribed by the XDM data model used by XPath, XQuery, and XSLT. Because the Saxon schema validator is primarily designed to meet the schema processing requirements of XSLT and XQuery, it only delivers the subset of PSVI required in that environment: this means that the output XML tree is only delivered if it is found to be valid, and the only tree decorations added by the validator are references from valid element and attribute nodes to the schema types against which they were validated. In effect, if the input is valid, then the output of the Saxon schema validator is a representation of the input XML instances with added type annotations; while if the input is invalid, then the output is a report listing validation errors.

The software described in this paper is an XSLT 3.0 implementation of the instance validator. It would certainly be possible to write an XSLT 3.0 implementation of the schema compiler, and we may well do that next, but we haven't tackled this yet.

Moreover, the only output of the instance validator described here is a validation report showing the validation errors. The software doesn't attempt to produce an XDM representation of the input document with type annotations. In fact, this isn't possible to do using standard XSLT 3.0 without extensions. XSLT 3.0 has no instructions to output elements and attributes with specific type annotations; the only way it can generate typed output is by putting the output through a schema validator, which is assumed to exist as an external component. So to write that part of the validator in XSLT 3.0, we would need to invent some language extensions.

2. The Validation Task

In this section we describe what the validator actually does.

Let's start with an example of a very simple schema, like this:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:element name="books">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="book" maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
  <xs:key name="isbn-key">
   <xs:selector xpath="book"/>
   <xs:field xpath="@isbn"/>
  </xs:key>
 </xs:element>

 <xs:element name="book">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="publisher" type="xs:string"/>
    <xs:element name="author" type="xs:string" minOccurs="1" maxOccurs="5"/>
    <xs:element name="date" type="xs:gYear"/>
    <xs:element name="price" type="moneyType"/>
   </xs:sequence>
   <xs:attribute name="isbn" type="ISBNType" use="required"/>
   <xs:assert test="if (publisher eq 'McGraw-Hill') then starts-with(@isbn, '007') else
                    if (publisher eq 'Academic Press') then starts-with(@isbn, '012')
                    else true()"/>
  </xs:complexType>
 </xs:element>

 <xs:complexType name="moneyType">
  <xs:simpleContent>
   <xs:extension base="xs:decimal">
    <xs:attribute name="currency" type="currencyType"/>
   </xs:extension>
  </xs:simpleContent>
 </xs:complexType>

 <xs:simpleType name="currencyType">
  <xs:restriction base="xs:string">
   <xs:enumeration value="USD"/>
   <xs:enumeration value="GBP"/>
   <xs:enumeration value="EUR"/>
   <xs:enumeration value="CAD"/>
  </xs:restriction>
 </xs:simpleType>

 <xs:simpleType name="ISBNType">
  <xs:restriction base="xs:string">
   <xs:pattern value="[0-9]{9}[0-9X]"/>
  </xs:restriction>
 </xs:simpleType>

</xs:schema>
```

A valid XML instance conforming to this schema might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book isbn="0070491712">
        <title>Apple PASCAL: a hands-on approach</title>
        <author>Arthur Luehrmann</author>
        <author>Herbert Peckham</author>
        <publisher>McGraw-Hill</publisher>
        <date>1981</date>
        <price currency="USD">13.95</price>
    </book>
    <book isbn="0124119700">
        <title>An Introduction to Direct Access Storage Devices</title>
        <author>Hugh Sierra</author>
        <publisher>Academic Press</publisher>
        <date>1990</date>
        <price currency="USD">72.95</price>
    </book>
</books>
```

The Saxon-EE schema compiler can be invoked to convert the source schema into an SCM file, for example with a command such as:

```
java com.saxonica.Validate xsd:books.xsd -scmout:xsd
```

Here is the resulting SCM file:

```
<?xml version="1.0" encoding="UTF-8"?>
<scm:schema xmlns:scm="http://ns.saxonica.com/schema-component-model"
            generatedAt="2018-06-04T10:39:09.702+01:00"
            xsdVersion="1.1">
   <scm:simpleType id="C0"
                   name="currencyType"
                   base="#string"
                   variety="atomic"
                   primitiveType="#string">
      <scm:enumeration value="EUR"/>
      <scm:enumeration value="CAD"/>
      <scm:enumeration value="USD"/>
      <scm:enumeration value="GBP"/>
   </scm:simpleType>
   <scm:simpleType id="C1"
                   name="ISBNType"
                   base="#string"
                   variety="atomic"
                   primitiveType="#string">
      <scm:pattern value="[0-9]{9}[0-9X]"/>
   </scm:simpleType>
   <scm:complexType id="C2"
                    name="moneyType"
                    base="#decimal"
                    derivationMethod="extension"
                    abstract="false"
                    variety="simple"
                    simpleType="#decimal">
      <scm:attributeUse required="false" inheritable="false" ref="C3"/>
   </scm:complexType>
   <scm:attribute id="C3"
      name="currency"
      type="C0"
      global="false"
      inheritable="false"
      containingComplexType="C2"/>
   <scm:element id="C4"
                name="book"
                type="C5"
```

```
            global="true"
            nillable="false"
            abstract="false"/>
<scm:complexType id="C5"
   base="#anyType"
   derivationMethod="restriction"
   abstract="false"
   variety="element-only">
   <scm:attributeUse required="true" inheritable="false" ref="C9"/>
   <scm:modelGroupParticle minOccurs="1" maxOccurs="1">
      <scm:sequence>
         <scm:elementParticle minOccurs="1" maxOccurs="1" ref="C10"/>
         <scm:elementParticle minOccurs="1" maxOccurs="1" ref="C11"/>
         <scm:elementParticle minOccurs="1" maxOccurs="5" ref="C12"/>
         <scm:elementParticle minOccurs="1" maxOccurs="1" ref="C13"/>
         <scm:elementParticle minOccurs="1" maxOccurs="1" ref="C14"/>
      </scm:sequence>
   </scm:modelGroupParticle>
   <scm:finiteStateMachine initialState="0">
      <scm:state nr="0">
         <scm:edge term="C10" to="1"/>
      </scm:state>
      <scm:state nr="1">
         <scm:edge term="C11" to="2"/>
      </scm:state>
      <scm:state nr="2">
         <scm:edge term="C12" to="3"/>
      </scm:state>
      <scm:state nr="3" minOccurs="1" maxOccurs="5">
         <scm:edge term="C12" to="3"/>
         <scm:edge term="C13" to="4"/>
      </scm:state>
      <scm:state nr="4">
         <scm:edge term="C14" to="5"/>
      </scm:state>
      <scm:state nr="5" final="true"/>
   </scm:finiteStateMachine>
   <scm:assertion xmlns:xs="http://www.w3.org/2001/XMLSchema"
      test="if (publisher eq 'McGraw Hill') then starts-with(@isbn, '007')
                  else if (publisher eq 'Academic Press') then starts-with(@isbn, '012')
                  else true()"
      defaultNamespace=""
      xml:base="file:/Users/mike/Documents/papers/markupuk2018/books.xsd"/>
</scm:complexType>
<scm:element id="C6"
            name="books"
            type="C7"
            global="true"
            nillable="false"
            abstract="false">
   <scm:identityConstraint ref="C8"/>
</scm:element>
<scm:complexType id="C7"
   base="#anyType"
   derivationMethod="restriction"
   abstract="false"
   variety="element-only">
   <scm:elementParticle minOccurs="1" maxOccurs="unbounded" ref="C4"/>
   <scm:finiteStateMachine initialState="0">
      <scm:state nr="0">
         <scm:edge term="C4" to="1"/>
      </scm:state>
```

```
         <scm:state nr="1" final="true">
            <scm:edge term="C4" to="2"/>
         </scm:state>
         <scm:state nr="2" final="true">
            <scm:edge term="C4" to="2"/>
         </scm:state>
      </scm:finiteStateMachine>
   </scm:complexType>
   <scm:key id="C8" name="isbn-key" targetNamespace="">
      <scm:selector xmlns:xs="http://www.w3.org/2001/XMLSchema"
                    xpath="book"
                    defaultNamespace=""/>
      <scm:field xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 xpath="@isbn"
                 defaultNamespace=""
                 type="#string"/>
   </scm:key>
   <scm:attribute id="C9"
      name="isbn"
      type="C1"
      global="false"
      inheritable="false"
      containingComplexType="C5"/>
   <scm:element id="C10"
                name="title"
                type="#string"
                global="false"
                containingComplexType="C5"
                nillable="false"
                abstract="false"/>
   <scm:element id="C11"
                name="publisher"
                type="#string"
                global="false"
                containingComplexType="C5"
                nillable="false"
                abstract="false"/>
   <scm:element id="C12"
                name="author"
                type="#string"
                global="false"
                containingComplexType="C5"
                nillable="false"
                abstract="false"/>
   <scm:element id="C13"
      name="date"
      type="#gYear"
      global="false"
      containingComplexType="C5"
      nillable="false"
      abstract="false"/>
   <scm:element id="C14"
                name="price"
                type="C2"
                global="false"
                containingComplexType="C5"
                nillable="false"
                abstract="false"/>
</scm:schema>
```

Let's look briefly at what this contains. The children of the `scm:schema` element represent different *schema components* such as element declarations, attribute declarations, simple and complex types, each with a unique identifier. For convenience I've rearranged these in order of the component identifier (the actual order doesn't matter).

- C0 is the simple type named `currencyType`. It is an atomic type derived from `xs:string` (the built-in type is represented as `#string`). The `scm:enumeration` elements list the permitted values.
- C1 is the simple type named `ISBNType`. It is an atomic type derived from `xs:string`, with a pattern facet constraining the permitted values.
- C2 is the complex type named `moneyType`. It is a "complex type with simple content", allowing simple content of type `xs:decimal`, and an attribute. The complex type contains an `scm:attributeUse` element which is a reference to the attribute declaration defining the attribute; like all references from one component to another, this uses the component identifier, in this case `C3`.
- C3 is the attribute declaration for `currency`; it is a local declaration (`global="false"`). The type of the attribute is defined by a reference to the simple type component C0.
- C4 is the element declaration for the `book` element; it is defined largely by a reference to the complex type C5 which defines the allowed content.
- C5 is the complex type defining the allowed content of `book` elements. The complex type itself has no name. The type is derived by restriction from `xs:anyType`, and the permitted content is defined in terms of a `modelGroupParticle` containing a sequence of `elementParticles` representing the permitted child elements: these are references to local element declarations appearing later in the SCM file. The complex type component also contains a representation of the (deterministic) finite state machine used to check instances against the grammar defined in the source schema. This defines a set of states (the initial state is 0, the final state is 5) and the permitted transitions between them. The transitions (edges) are defined by reference to the schema components for the contained element particles. Finally, the complex type component contains the XPath assertion that constrains the relationship between publishers and ISBNs. The `scm:assertion` element includes an `xml:base` attribute because it is possible (in theory) for the evaluation of the XPath assertion to depend on the base URI of the containing element in the source schema.
- C6 is the element declaration for the outermost `books` element: it refers to the complex type definition C7 and the identity constraint (the `xs:key` constraint) C8.
- C7 is the complex type definition for the outermost `books` element. Like C5, it contains a (very simple) finite state machine used to enforce the grammar.
- C8 represents the `xs:key` constraint specifying that ISBNs must be unique.
- C9 to C13 are the attribute and element declarations for the details of a book, and are all very simple. The meanings of the attributes are very closely aligned with the properties of the abstract Schema Component Model defined in the W3C specification.

Given this schema and this instance document, the task of the validator is to produce an empty validation report showing that there are no errors. The validation report becomes more interesting if the instance is invalid. For example, we can use this command:

```
java com.saxonica.Validate -xsd:books.scm -s:books-invalid.xml -report:report.xml
```

to validate this invalid instance:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book isbn="0070491712">
        <title>Apple PASCAL: a hands-on approach</title>
        <publisher>McGraw-Hill</publisher>
        <date>1981</date>
        <price currency="NZD">13.95</price>
    </book>
    <book isbn="0134119700">
        <title>An Introduction to Direct Access Storage Devices</title>
        <author>Hugh Sierra</author>
        <publisher>Academic Press</publisher>
        <date>1990-04</date>
        <price currency="USD">72.95</price>
    </book>
</books>
```

and the result is the following report:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<validation-report xmlns="http://saxon.sf.net/ns/validation"
                   system-id="file:/Users/mike/Documents/papers/markupuk2018/
                   books-invalid.xml">
   <error line="6"
          column="15"
          path="/Q{}books[1]/Q{}book[1]/Q{}date[1]"
          xsd-part="1"
          constraint="cvc-complex-type.2.4">In content of element &lt;book&gt;: The
            content model does not allow element &lt;Q{}date&gt; to appear immediately
            after element &lt;publisher&gt;. No further elements are allowed at
            this point. </error>
   <error line="7"
          column="31"
          path="/Q{}books[1]/Q{}book[1]/Q{}price[1]/@currency"
          xsd-part="2"
          constraint="cvc-complex-type.3">Value "NZD" contravenes the enumeration
          facet "EUR, USD, CAD, GBP" of the type Q{}currencyType</error>
   <error line="11"
          column="17"
          path="/Q{}books[1]/Q{}book[2]/Q{}author[1]"
          xsd-part="1"
          constraint="cvc-complex-type.2.4">In content of element &lt;book&gt;:
            The content model does not allow element &lt;Q{}author&gt; to appear
            immediately after element &lt;title&gt;. No further elements are allowed
            at this point. </error>
   <error line="13"
          column="15"
          path="/Q{}books[1]/Q{}book[2]/Q{}date[1]"
          xsd-part="2"
          constraint="cvc-datatype-valid.1">The content "1990-04" of element &lt;date&gt;
            does not match the required simple type. Cannot convert '1990-04' to a
            gYear</error>
   <error line="9"
          column="29"
          path="/Q{}books[1]/Q{}book[2]"
          xsd-part="1"
          constraint="sec-cvc-assertion.0">Element book does not satisfy assertion
            if (publisher eq 'McGraw Hill') then starts-with(@isbn, '007') else
            if (publisher eq 'Academic Press') then starts-with(@isbn, '012')
            else true()</error>
   <meta-data>
      <validator name="SAXON-EE" version="9.9.0.1"/>
      <results errors="5" warnings="0"/>
      <schema file="books.scm" xsd-version="1.1"/>
      <run at="2018-06-04T11:12:24.651+01:00"/>
   </meta-data>
</validation-report>
```

The report shown here comes from the existing Saxon-EE validator written in Java. Our task is to reproduce this report with a validator written entirely in portable XSLT.

3. Design Considerations

This section describes some of the design challenges posed.

3.1. Generic Stylesheet or Generated Stylesheet?

At the beginning of this exercise we considered two alternative designs: the validator could run as a generic XSLT stylesheet taking its rules dynamically from the SCM input document, or it could be a custom XSLT stylesheet generated from the SCM input document and dedicated to doing validation against this particular schema.

Both approaches have potential advantages, but we chose the first on the grounds of simplicity. As we will see later, there are some technical challenges where the second approach might have afforded a solution.

3.2. Subset of XSLT 3.0

We decided that, if possible, the validator should be written in the subset of XSLT 3.0 that is supported by Saxon-JS (including the use of `xsl:evaluate` which requires a later release of Saxon-JS). This decision means that we cannot use higher-order functions: these are an optional XSLT 3.0 feature which Saxon-JS does not currently support.

3.3. Streaming

In an ideal world, we would use an XSLT 3.0 streaming transformation to process the input instance document, so that it does not need to be held completely in memory.

The existing Saxon validator, written in Java, uses streamed processing whereever possible. The main case where streaming is not possible is in evaluating XSD 1.1 assertions: assertions can use arbitrary XPath expressions to process the subtree of the source document rooted at the element to which the assertion applies. The existing validator starts building an in-memory tree when it encounters such an assertion; in the absence of such assertions it is full streamed. It would be possible in principle to avoid building the subtree if the assertion uses a streamable subset of XPath, but the validator does not attempt this.

Emulating this behaviour in the new XSLT validator might be possible, but it is not easy, and in the current project we have not attempted it. One of the main reasons for this is that there are other XSD features (notably the evaluation of uniqueness and referential constraints) for which a streamed implementation is even more difficult.

The main constraint here is that `xsl:evaluate` (the XSLT 3.0 instruction to perform evaluation of a dynamically-constructed XPath expression) is not streamable, because static analysis has no access to the XPath expression in question, and streamability analysis is always done statically. Since `xsl:evaluate` is essential to enable XSD 1.1 features such as assertions and type alternatives to be evaluated, this is a stopper. We might be able to get around it by using the alternative design considered (a generated stylesheet in which the XPath expressions become statically analyzable), but we decided not to go that way.

3.4. Typed data

We have already mentioned that XSLT 3.0 can only generate a typed result tree (specifically, a node with non-trivial type annotations) by invoking a schema validator to produce the type annotations, and this precludes the possibility of writing that schema validator in XSLT 3.0. So the first obvious restriction we have to live with is that our validator will only do that part of the job concerned with detecting invalidity, not the other part concerned with augmenting the supplied instance with type information.

Unfortunately, even the task of distinguishing valid from invalid documents requires some use of type information associated with nodes. The most obvious example is that assertions (XPath expressions in `xsd:assert` declarations) are defined to operate on "semi-typed" data - that is data that has been validated using all the constraints in the schema other than assertions, and typed accordingly. For example, if `@price` and `@discount` are attributes of type `xs:decimal`, then the assertion `test="@discount lt @price"` is defined to do a decimal comparison, not a string comparison.

We don't currently have a solution to this problem. We found, however, that very few schemas are affected. With such schemas, it is currently necessary to rewrite the assertion to do explicit typing: in this case `test="xs:decimal(@discount) lt xs:decimal(@price)"`

Another case where typed data can be important is in `key` and `keyref` constraints. For example, if a uniqueness constraint applies to an attribute `@birthDate` of type `xs:date`, then values of that attribute are compared as dates, not as strings. Again, we found that this is rarely a problem in practice, but in this case we do have a solution that covers most cases: by doing static analysis of the XPath expressions used in the `selector` and `field` elements of the constraint, we can usually determine the data type of the values these expressions are selecting, and we can put this inferred type into the SCM and use it to cast the values before comparison. The only case where this approach proves inadequate are pathological cases where the type cannot be statically inferred, for example when the XPath expressions use union types or wildcards.

3.5. Support for xsi:schemaLocation

The current project is implementing a validator only, with no access to a schema compiler.

The `xsi:schemaLocation` and `xsi:noNamespaceSchemaLocation` attributes allow an instance document to reference the location of a (source) schema document containing a schema for the instance document. Because this validator has no access to a schema compiler, it cannot implement this capability.

This isn't a fundamental show-stopper. Support for these attributes isn't required for XSD conformance, and many applications avoid using them (there are a number of arguments against using them, not least that when you validate a document, it's because you don't trust it to be valid, and if you don't trust it to be valid, why should you trust it to tell you where the schema is?)

If we had a schema compiler written in XSLT, then it would of course be possible to interface this.

4. Implementation

This section of the paper is not a complete top-down documentation of the working implementation. Rather it gives a selection of snapshots into interesting aspects of the implementation, showing some of the techniques used and obstacles encountered. It is particularly focused on showing how new features in XSLT 3.0 proved valuable.

4.1. Use of Maps for Returned Values

A classic problem with functional programming is that a function can only return one result. If you want to compute two values (say a maximum and minimum) from the same input then you have two choices. You can either process the input more than once (which may involve some redundant computation), or you can return a composite result.

In this application there are many cases where we need to return a composite result.

To take an example, suppose we are validating an attribute, and we find that the declared type for that attribute is a user-defined list type, where the item type of the list is `part-number`, and `part-number` is derived from `xs:ID`. The calling code wants to know (a) whether the attribute is valid against its declared type; (b) what error messages to report if not; and (c) whether the value contains any `xs:ID` or `xs:IDREF` values that need to be added to global tables of `xs:ID` and `xs:IDREF` values for document-level validity checking at the end.

Our solution to this is that we recurse through the instance document in a tree-walk driven by `xsl:apply-templates` in the normal way, but the return value from `xsl:apply-templates` is a map containing all the information gleaned from the processing of this subtree.

Very often the information returned from several calls on `xsl:apply-templates` (for example, one call for child elements and another for attributes) will need to be combined into a single map. At the top level, when we return from the initial call on `xsl:apply-templates` on the root node, all the information that is needed to produce the validation report is present in one large map, and the final stage of processing takes this map and generates the XML report.

The maps that are produced by the different processing stages thus typically include some subset of a common set of fields. These include:

Table 1. The structure of maps used to return partial results of processing

Name	Value
valid	An `xs:boolean` indicating whether the subtree is valid
errors	A set of error objects indicating error information to be included in the validation report
value	The typed value of an element or attribute
type	The type against which a subtree was validated
lexical	The lexical form of an attribute or text node after whitespace normalization
id	A set of `xs:ID` values found in the subtree
id-map	A mapping from `xs:ID` values found in the subtree, to the nodes on which they appeared
idrefs	A set of `xs:IDREF` values found in the subtree

When two of these maps representing properties of different subtrees are combined, different rules apply to each field. For example, for the `id` and `idrefs` and `errors` fields we can take the union of the two values. For the `valid` property, we can apply a logical AND; a tree is valid only if all its subtrees are valid. For `value` and `type` we can drop the value; these fields are used only at the next level up, and do not propagate all the way to the root.

4.2. Declaring Map Types

Standard XSLT 3.0 allows maps (as described in the previous section) to be returned from templates, stored in variables, and so on, so this style of processing is perfectly possible without departing from the standard. However, the facilities for declaring the type of these maps are very weak. The closest we can get is `map(xs:string, item()*)` which is satisfied by any map whose keys are strings.

Much of the debugging process for this stylesheet involves understanding the contents of these returned maps, and it is therefore frustrating that the XSLT 3.0 type system is so poor at validating these maps and reporting type errors. Saxon therefore introduces an extension to XSLT 3.0, called tuple types. Here is a declaration of the returned structure as a tuple type:

```
  tuple( valid: xs:boolean?,          (: indicates whether the subtree is valid
                                         (default = true) :)
         errors: element(vr:error)*,  (: a list of errors found when validating the
                                         subtree :)
         value: xs:anyAtomicType*,    (: the typed value of an element or attribute :)
         type: xs:string*,            (: the types of the typed values, as component
                                         IDs :)
         lexical: xs:string?,         (: the lexical form of a value after whitespace
                                         normalization :)
         id: xs:string*,              (: any ID values found while validating an
                                         element or attribute :)
         id-map: map(xs:string, element())*,  (: a mapping from ID values to elements :)
         idrefs: xs:string*)          (: any IDREF values found while validating an
                                         element or attribute :)
```

This clearly documents the expected contents of the map much more precisely than the bland declaration `map(xs:string, item()*)`.

Tuples in Saxon are not a separate data type in the way that maps and arrays are separate data types. Rather, a tuple type is an alternative way of constraining the content of a map. It defines the (string-valued) keys that can appear in the map, and for each permitted key, the permitted type of the corresponding values. Declaring the expected type of a map in this form gives much improved static and dynamic type checking. For example, attempting to reference a non-existing field using the lookup expression `$result?Value` can generate a static error message, as can its use in an inappropriate context such as `$result?valid eq "true"`.

Because tuple type declarations are often quite lengthy, as in this example, Saxon allows them to be declared once using a type alias:

```
<saxon:type-alias name="validation-outcome" type="
  tuple( valid: xs:boolean?,          (: indicates whether the subtree is valid
                                         (default = true) :)
         errors: element(vr:error)*,  (: a list of errors found when validating the
                                         subtree :)
         value: xs:anyAtomicType*,    (: the typed value of an element or attribute :)
         type: xs:string*,            (: the types of the typed values, as component
                                         IDs :)
         lexical: xs:string?,         (: the lexical form of a value after whitespace
                                         normalization :)
         id: xs:string*,              (: any ID values found while validating an
                                         element or attribute :)
         id-map: map(xs:string, element())*,  (: a mapping from ID values to elements :)
         idrefs: xs:string*)          (: any IDREF values found while validating an
                                         element or attribute :)

 "/>
```

And the type can then be referenced wherever an `as` attribute can appear, for example:

```
<xsl:template match="*" as="map(xs:string, item()*)" saxon:as="~validation-outcome"/>
```

The syntax of `saxon:as` is an XPath `SequenceType` augmented with Saxon-specific syntax, in this case a reference to a type alias marked as such by the presence of the leading tilde (~). The semantics of `saxon:as` are that it provides type information additional to that contained in the `as` attribute. Because (under the XSLT extensibility rules) attributes in the Saxon namespace are ignored by XSLT processors other than Saxon, this whole mechanism enables Saxon to do extra compile-time and run-time type checking, without in any way sacrificing the interoperability of the stylesheet: it still functions correctly under other standards-conforming XSLT 3.0 processors.

4.3. Assessment against Complex Types using Finite State Machines

As we've seen, the finite state machines used to evaluate a sequence of elements against the grammar rules for a complex type are constructed by the schema compiler and embedded in the SCM file that is used as input to the validator.

A simplified validator for a simple finite state machine could be written like this:

```
<xsl:iterate select="$node/*">
    <xsl:param name="state" select="$initial-state" as="element(scm:state)"/>
    <xsl:on-completion>
        <xsl:if test="not($state/@final = 'true')">
            <xsl:sequence select="map{'errors':
                                      scm:error($node, 'Element content is
                                      incomplete')}"/>
        </xsl:if>
    </xsl:on-completion>
    <xsl:variable name="matching-edge" as="element(scm:edge)?"
        select="$state/scm:edge[scm:get(@term)[@name = local-name(current())
                    and string(@targetNamespace) = namespace-uri(current())]]"/>
    <xsl:variable name="matching-wildcard-edge" as="element(scm:edge)?"
        select="$state/scm:edge[scm:get(@term)[self::scm:wildcard[
                                scm:wildcard-matches($containing-type, .,
                                current())]]]"/>
    <xsl:choose>
        <xsl:when test="empty($matching-edge) and empty($matching-wildcard-edge)">
             <xsl:break select="map{'errors': scm:error(., 'Element ' || name()
                                                || ' is not allowed here')}"/>
        </xsl:when>
        <xsl:when test="empty($matching-edge)">
            <xsl:variable name="wildcard"
                          select="scm:get($matching-wildcard-edge/@term)"
                          as="element(scm:wildcard)?"/>
            <xsl:sequence select="scm:check-wildcard-match($containing-type,
                                  $wildcard, .)"/>
            <xsl:next-iteration>
                <xsl:with-param name="state"
                                select="$states[@nr =
                                        $matching-wildcard-edge/@to]"/>
            </xsl:next-iteration>
        </xsl:when>
        <xsl:otherwise>
            <xsl:variable name="decl"
                          select="scm:get($matching-edge/@term)"
                          as="element(scm:element)"/>
            <xsl:apply-templates select="." mode="explicit-decl">
                <xsl:with-param name="decl" select="$decl"/>
            </xsl:apply-templates>
            <xsl:next-iteration>
                <xsl:with-param name="state"
                                select="$states[@nr =
                                        $matching-edge/@to]"/>
            </xsl:next-iteration>
        </xsl:otherwise>
    </xsl:choose>
</xsl:iterate>
```

The way this code works is as follows:

- The `xsl:iterate` instruction is new in XSLT 3.0. It is rather like `xsl:for-each`, except that it processes the selected items strictly in sequence; the code for processing one item can set parameters for processing the next item; and it is possible to break out of the loop early. The same effect could be achieved with a recursive template, but `xsl:iterate` is often easier to understand. In this case we are iterating over the children of the element being validated.
- There is a single parameter, the current state, which is initially set (by the calling code) to the state numbered 0.
- The `xsl:on-completion` instruction is executed when we reach the end of the sequence of child elements. If the current state is a final state, we return nothing (meaning all is well, the input is valid). Otherwise we return a map containing an error value.
- There are two kinds of transition possible in a given state: named element transitions, and wildcard transitions. We first find all the matching named element transitions (the schema compiler will have ensured there can be at most one) and all the matching wildcard transitions.

- If both sets are empty, there is no legal transition for the current child element in this state, so we return an error value.
- If there is a wildcard transition possible, but no named-element transition, then we check that the wildcard transition is really allowed and that the element is valid against the wildcard (this will take account of its `processContents` attribute, and then proceed to process the next child element in the state reached by this transition.
- If there is a named-element transition possible, then we call `apply-templates` to check that the child element is valid against the required type for the named element, and then proceed to process the next child element in the state reached by this transition.

The actual logic is more complex than this. Firstly, we use a finite state machine with counters, to reduce the size of the finite state machine needed for a grammar such as `<element name="book" minOccurs="100" maxOccurs="200"/>`. Secondly, XSD 1.1 allows "open content" which allows elements matching a given wildcard to appear either (a) anywhere (interleaved content), or (b) at the end of the sequence (suffix content). The possibility of open content is not integrated into the finite state machine, but is instead handled by the validator as it arises. However, the basic principle is retained of stepping through the children using `xsl:iterate` to maintain the current state.

4.4. Checking Assertions

The code for checking input data against assertions defined in the schema is very straightforward. Here is the actual logic (no simplifications this time):

```
<xsl:function name="scm:check-assertions" as="map(*)" saxon:as="tuple(error:
            element(vr:error)*)">
    <xsl:param name="type" as="element(scm:complexType)"/>
    <xsl:param name="node" as="element()"/>
    <xsl:variable name="copy-sans-comments" as="element()">
        <xsl:apply-templates select="$node" mode="copy-sans-comments"/>
    </xsl:variable>
    <xsl:variable name="failures" as="element(vr:error)*">
        <xsl:for-each select="$type/scm:assertion">
            <xsl:try>
                <xsl:variable name="assertion-result" as="item()*">
                    <xsl:evaluate xpath="@test"
                        context-item="$copy-sans-comments"
                        namespace-context="scm:make-namespace-context(.)"
                        base-uri="{base-uri($scm)}">
                        <xsl:with-param name="value" select="$copy-sans-comments"/>
                    </xsl:evaluate>
                </xsl:variable>
                <xsl:if test="not($assertion-result)">
                    <xsl:sequence select="scm:error($node, ' must satisfy assertion '
                    || @test)"/>
                </xsl:if>
                <xsl:catch errors="*">
                    <xsl:sequence select="scm:error($node, ' must satisfy assertion '
                    || @test || '.
                            Evaluation of the assertion failed with a dynamic error: '
                            || $err:description)"/>
                </xsl:catch>
            </xsl:try>
        </xsl:for-each>
    </xsl:variable>
    <xsl:sequence select="map{'errors': $failures}"/>
</xsl:function>
```

Notes relating to this code:

- The function returns a map, but the `saxon:as` declaration reveals that there is only one field defined in this map, namely the `errors` field. If the constraint is satisfied, an empty map is returned. The reason for defining it this way is that the calling code can use its standard mechanism for combining the results of different validation processes.
- The function makes a copy of the element being validated, in which comments and processing instructions have been removed. This is prescribed by the specification. A copy of the subtree is needed to ensure that the XPath expressions in the assertion have no access to nodes in the input document that fall outside the subtree being validated.

- The assertion is evaluated using `xsl:evaluate`, a new XSLT 3.0 instruction that evaluates XPath expressions known only dynamically (in this case, an expression read from the SCM file). The instruction provides machinery to establish the static and dynamic context for evaluating the expression, here including the context item, the value of the `$value` variable, the namespace context, and the base URI.
- If the effective boolean value of the assertion result is false, the function returns an error value.
- If a dynamic error occurs while evaluating the assertion, this is caught using the new `xsl:try` instruction in XSLT 3.0, and the function returns an error value.
- The whole process is repeated for each defined assertion. If more than one assertion fails, then more than one error will be returned.

4.5. Other Complications

It's worth mentioning a few other complications that the implementation has to deal with, without going into gory detail:

- Gregorian types. XSD 1.1 introduces a new facet which allows you to specify that the timezone on a date/time value is mandatory or optional. It turns out to be easy to check this using XPath expressions for an `xs:date`, `xs:time`, or `xs:dateTime`; but there's no easy way to do it for `xs:gYear`, `xs:gYearMonth`, and friends. Similarly, XSD 1.1 allows facets to control the range of these values, for which XPath offers no support. The validator therefore includes a library of functions for handling the Gregorian types.
- Regular Expressions. The syntax for regular expressions contained in the XSD pattern facet is very similar to the syntax for the XPath `fn:matches()` function -- but not quite the same. Most of the differences are extensions in the XPath version (for example, support for back-references), and since the schema compilers has done static validation on the expression, we can ignore these differences. The remaining difficulty is the characters "^" and "$", which represent themselves in XSD, but are meta-characters in XPath. To handle this we need to pre-process the regular expression to replace occurrences of "^" and "$"(if not within square brackets) by "\^" and "\$".
- Equality semantics. The equality matching rules in XSD 1.1, used for example by the enumeration facet or in key/keyref comparison, don't correspond directly to any of the ways of testing equality in XPath. For example, in XPath the integer 3 and the double 3e0 are equal, in XSD they are not. This also makes it difficult to use XPath maps to enforce uniqueness constraints. It is therefore necessary to use a custom function for comparing atomic values, and a function `schemaComparable(x)` with the property that `schemaComparable(x) eq schemaComparable(y)` under the XPath rules if and only if `x` and `y` are equal under the XSD rules.

5. Results

This section attempts to assess the quality metrics of the completed validator.

Saxonica's quality objectives for all its software are conformance, usability, and performance, in that order. We therefore assess the validator against these criteria.

5.1. Conformance

W3C publishes a comprehensive test suite for XSD 1.1. The XSLT validator is currently passing 41330 out of 41363 tests. This is after excluding tests that are not applicable, for example, tests that rely on `xsi:schemaLocation`. This level of conformance comfortably exceeds that of many widely-used schema processors, and the failures largely involve edge cases that few users will ever encounter.

5.2. Usability

Usability is measured primarily by the quality of the error messages. This is not yet as good as the Java validator. Here is the validation report produced for the invalid booklist document supplied earlier:

```
<vr:validation-report xmlns:vr="http://saxon.sf.net/ns/validation">
    <vr:error path="/Q{}books[1]/Q{}book[1]/Q{}date[1]">Element date is not allowed
    here</vr:error>
    <vr:error path="/Q{}books[1]/Q{}book[2]/Q{}author[1]">Element author is not
    allowed here</vr:error>
    <vr:error path="/Q{}books[1]/Q{}book[2]"> must satisfy assertion
                           if (publisher eq 'McGraw Hill')
                           then starts-with(@isbn, '007')
```

```
                                else if (publisher eq 'Academic Press')
                                then starts-with(@isbn, '012') else true()</vr:error>
</vr:validation-report>
```

In comparison with the report produced by the Java validator, we see:

- There are no line and column numbers associated with the errors, only a path. This omission is because XSLT provides no standard way of obtaining the line or column number of a node. Some versions of Saxon provide extension functions to get this information, but we want to avoid using extensions; and in any case, a key target environment is Saxon-JS, where we rely on the Javascript DOM as our data model, and the Javascript DOM does not provide line and column information.
- There is no information about which constraint in the XSD specification is violated (this information is of little value to most users, but it is required by the conformance rules).
- There are only three errors reported, rather than five. This is because the XSLT validator is quicker to stop validating sibling elements once one of them has been found to be in error.

So there is some work still to be done to get the usability up to the level of the existing validator.

5.3. Performance

No serious work on optimizing (or measuring) performance has yet been done. However, it's useful to get some very preliminary data to assess whether performance is going to be a major obstacle to the feasibility of the approach.

I constructed a valid data file containing ten thousand book elements.

With the existing Saxon-EE schema validator, validation took 1.1 seconds.

With the XSLT validator, validation (using Saxon-EE on Java as the XSLT engine) took 9.4 seconds.

This represents a ballpark estimate of the relative efficiency. It's not a thorough benchmark in any way; there is no point in doing a thorough benchmark until some basic performance tuning has been done.

There are clearly many opportunities for performance improvement. Some of the obvious inefficiencies, relative to the Java validator, include:

- In evaluating items with a pattern facet, the regular expression is recompiled every time an item is validated. This is because the `fn:matches()` function precompiles the regular expression if it is known statically, but it makes no attempt to cache the compiled regular expression if the same regex is used repeatedly. The regex in this case is read from the SCM file at run-time, so no compile-time optimization is possible.
- Similarly, XPath expressions used in assertions may be recompiled every time they are used. There are some circumstances in which `xsl:evaluate` will cache compiled XPath expressions that are used repeatedly, but this doesn't appear to be happening in this stylesheet.
- Too much data is being retained from validation and passed upwards from the validation of a child to the validation of its parent. This results in bloated maps containing validation outcomes, that take a long time to combine. It's probably not difficult to find "low-hanging" optimizations in this area.

It's clear that a lot of the time is being spent creating and combining the maps that are used to pass data up the tree. The whole application relies very heavily on maps, and its performance depends on the performance of map operations such as `map:put` and `map:merge`. It's possible that it might benefit from a different implementation of maps that is tailored to the usage patterns that occur when map types are declared as tuple types. It could also benefit from changes to the application to make more selective use of maps. In particular, we seem to be incurring heavy costs inspecting and copying maps that are actually empty, because all the data is valid: there's clearly an opportunity for optimizations here.

For the moment, all we can conclude about performance is that more work needs to be done. Making the XSLT validator as fast as the existing Java validator is probably unachievable, but we should be able to get acceptably close.

6. Conclusions

Firstly, we have shown that writing an XSD 1.1 validator in XSLT 3.0 is feasible, with a few caveats: the main limitation is that XSLT 3.0 does not allow the creation of typed element and attribute nodes except by use of an XSD validator, which creates a recursive dependency. This limitation could probably be solved by means of a few simple XSLT extension functions.

Whether an XSD 1.1 validator written in this way can achieve an acceptable level of usability and performance remains an open question. We're certainly close, but we're not quite there yet.

The experiment has certainly yielded insights on how to design complex applications to take advantage of new XSLT 3.0 capabilities; and it has provided usability feedback that has led directly to improvements in the XSLT 3.0 processor, for example in the way type-checking errors with maps are reported and with features for tracing and diagnostics.

It has not so far proved possible to write a streaming validator by exploiting the streaming capabilities of XSLT 3.0. The main obstacle is that the `xsl:evaluate` instruction is intrinsically unstreamable because it cannot be statically analyzed. One way around this problem would be to generate a custome stylesheet to perform validation against a specific schema. Another approach might be to allow a stylesheet author to assert that an `xsl:evaluate` instruction is streamable, and to have the XPath compiler check this assertion when the instruction is evaluated.