XPath 3.1 in the Browser

John Lumley
jwL Research, Saxonica
<john@jwlresearch.com>

Debbie Lockett
Saxonica
<debbie@saxonica.com>

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

This paper discusses the implementation of an XPath3.1 processor with high levels of standards compliance that runs entirely within current modern browsers. The runtime engine Saxon-JS, written in JavaScript and developed by Saxonica, used to run pre-compiled XSLT3.0 stylesheets, is extended with a dynamic XPath parser and converter to the Saxon-JS compilation format. This is used to support both XSLT's `xsl:evaluate` *instruction and a JavaScript API* `XPath.evaluate()` *which supports XPath outside an XSLT context.*

1. Introduction

XSLT1.0 was originally developed primarily as a client-side processing technology, to increase data-adaptability and presentational flexibility with a declarative model supported in the browser. By 2006 all the major desktop browsers supported it, but the rise in importance of mobile client platforms and their code footprint pressures then sounded the death-knell. However, remnants of support for the XPath 1.0 component of XSLT can be found in most, but not all, of the current desktop browsers.

In the meantime, spurred mostly by unexpected takeup of XSLT and XQuery in server-side XML-based architectures, the technologies have progressed through the "2.0" stage (temporary variables, type model and declarations, grouping, canonical treatment of sequences, extensive function suites...) to the current candidate recommendation standards for 3.0/3.1 in XSLT, XQuery and XPath. At this stage support for maps, arrays and higher-order functions join let constructs, increased standard function libraries and others to provide a core set of common functionality based on XPath. XQuery uses this with its own (superset

of XPath) syntax to support query-based operation and reporting. XSLT adds a template-based push model, within an XML syntax, which is designed to be able to support streaming processing in suitable use cases.

These developments are sufficiently robust and powerful that, other circumstances permitting, exploiting XPath 3.1 to process over XML data is highly attractive. But at present this can only be performed in server-side situations – no browser developers could contemplate either the development effort or the necessary memory footprints needed for the "2.0" level compilers for what they consider niche (i.e. non-mobile) applications, let alone that required for 3.0+.

What *has* been developed extensively in browsers are *JavaScript* processors, both internal compilers, JIT runtimes and development libraries, such that very significant programmes can be executed in-browser[1]. And in general the level of conformance of and interoperability between implementations in different browsers is reasonable.

Exploiting this JavaScript option for supporting XSLT/XPath /XQuery has been explored in a number of cases:

- Saxon-CE[1] cross-compiled a stripped-down XSLT2.0 Saxon compiler from Java into JavaScript using Google's GWT technology. This worked, but the loaded code was large, very difficult to test and debug and very exposed to GWT's cross-browser capabilities.
- There are a small number of developers working on open-source implementations in native JavaScript for XPath 2.0 (e.g.Ilinsky[2]) and XQuery (*XQIB*[2]), though it is very unclear their level of standards conformance. Other JavaScript-based implementations for XPath 1.0 include *Wicked good XPath*[3], supporting the DOM Level 3 subset and *XPath-js*[4] which supports the full XPath 1.0 standard.
- During 2016 Saxonica developed *Saxon-JS* [3], a runtime environment for XSLT 3.0, implemented in JavaScript. This engine interprets compiled execution plans, presented as instruction trees describing the execution elements of an XSLT3.0 stylesheet. The execution plan is generated by an independent compiler, which performs all necessary parsing, optimisation, checking and code generation. Consequently the runtime footprint is modest (~ 200kB in minimised form) and programme execution incurs no compilation overhead.

1.1. Saxon-JS Runtime – dynamic evaluation

The Saxon-JS runtime environment is written entirely in JavaScript and is intended to interpret and evaluate XSLT program execution plans and write results

[1] In the process pushing client-based Java towards oblivion too.
[2] http://www.xqib.org/index.php
[3] https://github.com/google/wicked-good-xpath
[4] https://github.com/andrejpavlovic/xpathjs

into the containing web page. These execution plans are presented as instruction trees describing the execution elements of an XSLT3.0 stylesheet in terms of sets of templates and their associated match patterns, global variables and functions, sequence constructors consisting of mixtures of XSLT instructions and XPath expression resolution programs and other static elements such as output format declarations.

XSLT programs for Saxon-JS are compiled specifically for this target environment, using *Saxon-EE* to generate an XML tree as a *Stylesheet Export File* (SEF)[5]. This tree is loaded by a `SaxonJS.transform()` within a web page, which then executes the given stylesheet, with a possible XML document as source and other aspects of dynamic context, writing results to various sections of the web page via `xsl:result-document` directives. The engine supports specialist XSLT modes (e.g. `ixsl:on-click`) to trigger interactive responses, based on a model originally developed for *Saxon-CE*.

Saxon-JS supports a very large proportion of XSLT3.0 and XPath3.1 functionality. As all the static processing (parsing, static type checking, adding runtime checks.…) of the presented XSLT stylesheet is performed by Saxon-EE during its compilation operation, programs running in Saxon-JS should benefit from the very high levels of standards compliance that the standard Saxon product achieves. At runtime Saxon-JS only has to check for dynamic errors and often these are programmed in the execution plan as specific check instructions. The consequences are that

- The Saxon-JS code footprint is very much smaller, consisting only of a runtime interpreter.
- Execution starts immediately after loading the instruction tree – there is no compilation phase
- *But* Saxon-JS assumes *the code is correct*, as all static errors have been removed and necessary dynamic checks have been added.
- As there is no runtime compiler, or XPath parser, there is no implicit support for dynamic evaluation of XPath expressions (through the `xsl:evaluate` instruction), nor indeed runtime definition and evaluation of functions.

Within some related work on streamability analysis, one of the authors had been working on parsing XPath expressions using XSLT code to generate reduced parse trees and manipulating and analysing their properties. One of the possibilities was to recast this work to run entirely in Saxon-JS. But it also opened the possibility of extending Saxon-JS to both parse the XPath and generate equivalent execution plans (i.e. *Stylesheet Export File*) and hence support dynamic evaluation.

[5]Apart from some additional Saxon-JS attributes, the tree is identical to that used for linking separately-compiled packages for execution by a Java-based server-side Saxon engine – hence it shares the same degree of compilation "correctness" as any other Saxon-processed stylesheet.

This paper describes the design, construction and testing of such an extension.

2. Overall design

The route from XPath expression to evaluated result involves threee major steps. Firstly the expression string must be parsed against an XPath grammar to check that i) it is correct and ii) what XPath instructions are present. This means that matched grammar productions and variable tokens must be identified and grouped. This is most smoothly reported as a tree. Secondly this representation of the XPath expression must be converted into a set of suitable instructions for the the Saxon-JS, which will be cast as an XML tree. Finally the instruction tree needs to be interpreted by the Saxon-JS runtime, to produce the given result.

The main facility is written as an additional JavaScript object `XPathJS` added to the `SaxonJS` runtime[6]. There are three significant phases as shown in the figure

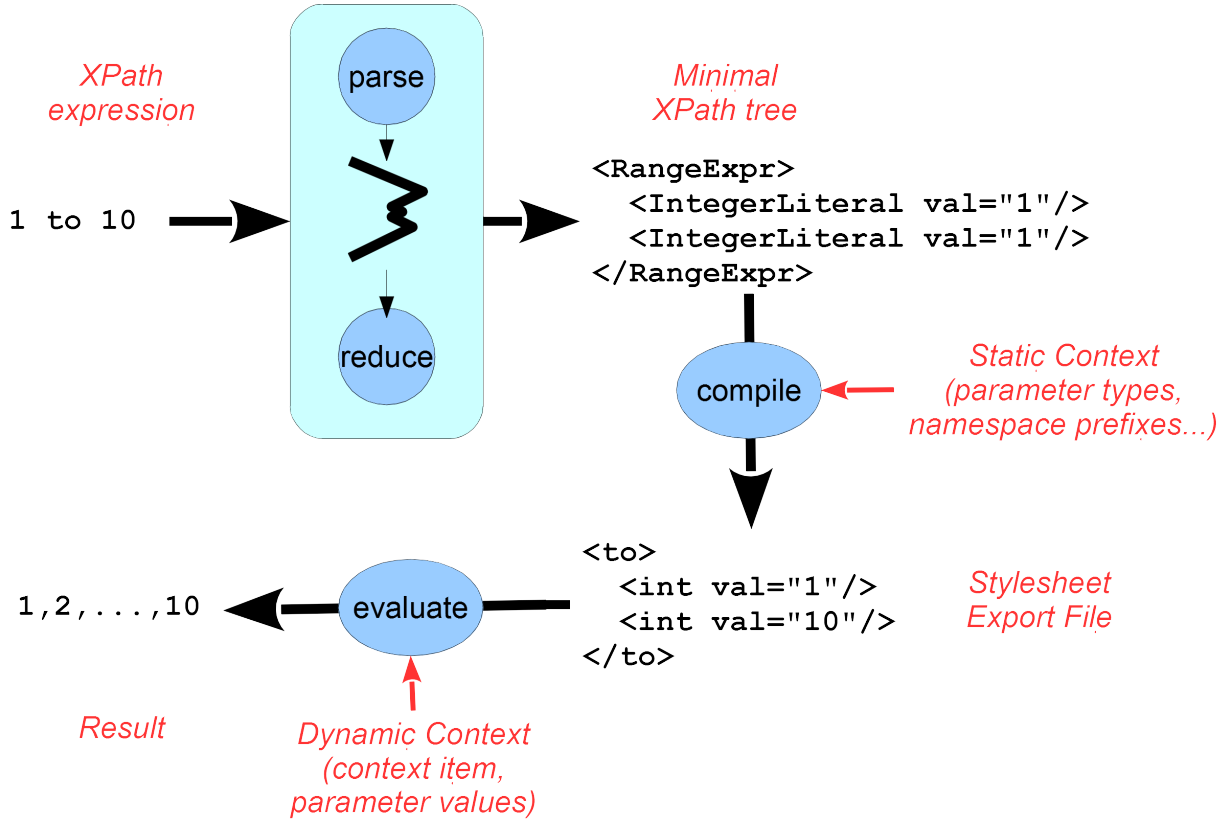


Figure 1. Processing phases

- `parse(xpath)` produces a (reduced) parse tree of the XPath expression, or an error if appropriate. This has similarities with Pemberton's *Invisible XML*,

[6]It has been arranged that the code for these features (which has a memory footprint as large as Saxon-JS itself) is loaded dynamically when first needed. Thus no additional overhead is incurred if a stylesheet does not use these features.

retaining only productions of interest, and decorating them with suitable properties (such as the `op` of a `MultiplicativeExpr`). It also converts sections defined gramatically as repeats such as `X op X (op X)*` (e.g. `let ….`) into nested trees of constant arity such that later phases only deal with strictly canonical forms.

- `compile(parseTree,staticContext)` generates a suitable SEF tree[7]. The parse tree is recursively converted to elements and attributes of the resulting SEF and static type checking is performed. More details below.
- `evaluate(SEF,contextItem,params)` interprets the execution plan with an optional context item, and given parameter value bindings. This uses the Saxon-JS evaluation engine. The result is then converted or wrapped to an appropriate final type, such as an iterator around XDM types for use in `xsl:evalute` or plain or arrayed Javascript types for a Javascript invocation.

2.1. Parsing the expression

XPath expressions are parsed by a two-step process: firstly the expression is checked for correctness and a full parse tree generated; secondly this parse tree is reduced to the essential details and converted to a canonical form.

A parser built from the XPath 3.1 EBNF grammar by Gunther Rademacher's REx parser-generator [5] is used. This is coded in JavaScript, using callback functions for starting and ending non-terminal productions, detected whitespace and terminal character phrases, which are indexed into the original input string. These callbacks are currently used to generate a DOM tree, which can be *very* large. For example the simple XPath `1 to 10` generates the following tree (which is up to 27 levels deep), some of whose closing tags have been elided:

```
<XPath>
 <Expr>
  <ExprSingle>
   <OrExpr>
    <AndExpr>
     <ComparisonExpr>
      <StringConcatExpr>
       <RangeExpr>
        <AdditiveExpr>
         <MultiplicativeExpr>
          <UnionExpr>
           <IntersectExceptExpr>
            <InstanceofExpr>
             <TreatExpr>
              <CastableExpr>
```

[7] In this case the tree is retained in memory and *not* serialized and written as an external file.

```
                <CastExpr>
                 <ArrowExpr>
                  <UnaryExpr>
                   <ValueExpr>
                    <SimpleMapExpr>
                     <PathExpr>
                      <RelativePathExpr>
                       <StepExpr>
                        <PostfixExpr>
                         <PrimaryExpr>
                          <Literal>
                           <NumericLiteral>
                            <IntegerLiteral>1</IntegerLiteral>
                           </NumericLiteral>
                          …
        </AdditiveExpr>
        <TOKEN>to</TOKEN>
        <AdditiveExpr>
         <the same…>
                           <NumericLiteral>
                            <IntegerLiteral>10</IntegerLiteral>
                           </NumericLiteral>
                          …
        </AdditiveExpr>
       </RangeExpr>
      </StringConcatExpr>
     </ComparisonExpr>
    </AndExpr>
   </OrExpr>
  </ExprSingle>
 </Expr>
 <EOF/>
</XPath>
```

As Pemberton[4] has pointed out, such trees, while correct, aren't very efficient, so the second phase reduces them to only the minimal essential elements. This is written as a recursive function `reduce()` which generally switches on the node name, with the following strategies:

- Literals have their value written as an attribute on the production.
- By default an element that has only one element child is reduced to the reduction of that child, e.g.

```
<AdditiveExpr>
  … <NumericLiteral>
      <IntegerLiteral>1</IntegerLiteral>…
→ <IntegerLiteral value="1"/>
```

- Tokens that convey variation meaning are added as suitable attributes to the main element. Non-essential tokens (which are constant for the given production) are deleted, e.g.

```
1 * 4 to ceiling(10.1)→
… <RangeExpr>
        …<MultiplicativeExpr>
          …  <IntegerLiteral>1</IntegerLiteral> …
          <TOKEN>*</TOKEN>
          …  <IntegerLiteral>4</IntegerLiteral> …
         </MultiplicativeExpr>...
        <TOKEN>to</TOKEN>
        … <FunctionCall>
             <FunctionEQName>
              <FunctionName>
               <QName>ceiling</QName>
              </FunctionName>
             </FunctionEQName>
             <ArgumentList>
              <TOKEN>(</TOKEN>
              <Argument>
               … <DecimalLiteral>10.1</DecimalLiteral> …
              </Argument>
              <TOKEN>)</TOKEN>
             </ArgumentList>
            </FunctionCall> …
       </RangeExpr>
…
→ <RangeExpr>
    <MultiplicativeExpr op="*">
     <IntegerLiteral value="1"/>
     <IntegerLiteral value="4"/>
    </MultiplicativeExpr>
    <FunctionCall name="ceiling">
     <DecimalLiteral value="10.1"/>
    </FunctionCall>
   </RangeExpr>
```

- Constructs that can have indefinite repetitions of subsections, such as `let $a:=1, $b:=2 return $a+$b` are regularised into nested trees (that can be either left or right associative) with constant arity, so that subsequent phases are presented with a canonical form, e.g.

```
<LetExpr>
 <SimpleLetBinding var="a">
  <IntegerLiteral value="1"/>
 </SimpleLetBinding>
```

```
 <LetExpr>
  <SimpleLetBinding var="b">
   <IntegerLiteral value="2"/>
  </SimpleLetBinding>
  <AdditiveExpr op="+">
   <VarRef name="a"/>
   <VarRef name="b"/>
  </AdditiveExpr>
 </LetExpr>
</LetExpr>
```

A few other sections of specialist code conversion are carried out at this stage, including replacing path shortcuts (`//`, `..` and `@`*`name`*) with equivalent full forms. With this reduced tree, code generation can then start[8].

2.2. Generating the execution plan

Many of the SEF instructions are in close correspondence with the XPath grammar productions and their "arguments" correspond to the results of evaluating their child subtrees, so the (unoptimised) code generation problem is basically to map the parse tree into a generally similar SEF tree, checking for static errors and adding runtime instructions to check for dynamic errors. This code generator is written entirely in JavaScript[9].

The bulk of the recursive compiling converter (`prepare(`*`node`*`,`*`context`*`)`) is a switch based on the XPath expression production types, e.g. `RangeExpr` for the *`x`* `to` *`y`* range generator. The `context` argument contains the static context, such as namespace prefix mappings, decimal formats and so forth, as well as semi-static information, such as the inferred type of the context item and the names, allocated storage slots and (declared or inferred?) types of the external parameters and local variables that are in-scope for the expression node being processed. (As usual name scoping follows the `following-sibling::*/ descendant-or-self::*` compound axis.)

For many of the productions the conversion is generally to produce an equivalent SEF instruction element (in this case `to`) with its two children (which could of course be anything from an `IntegerLiteral` to a full-blown computation tree) being processed recursively to produce their value-generating instruction trees.

For example the XPath `1 to 2 * 10` has a reduced parse tree of:

[8]It should be possible to perform some of this reduction during the original creation of the parse tree using smarter and language-sensitive callback functions. The authors haven't yet had an opportunity to explore this.

[9]It could have been written entirely XSLT and cross-compiled to produce an additional SEF tree that was used as a programme to generate another SEF tree for the XPath expression. The first author feels he learned more doing it "the hard way" and a rewrite in XSLT to support a more portable position is attractive, but it certainly won't be as fast as the native JavaScript version.

```
<RangeExpr>
  <IntegerLiteral value="1"/>
  <MultiplicativeExpr op="*">
    <IntegerLiteral value="2"/>
    <IntegerLiteral value="10"/>
  </MultiplicativeExpr>
</RangeExpr>
```

which is converted to an instruction tree:

```
<to type="xs:integer*">
  <int val="1" type="xs:integer"/>
  <arith op="*" calc="i*i" type="xs:integer">
    <int val="2" type="xs:integer"/>
    <int val="10" type="xs:integer"/>
  </arith>
</to>
```

where the type of arithmetic to be performed on the two inner values (integer times integer) is encoded in the `@calc` attribute – the runtime calculator is directed by this. (The `@type` annotations help show the determined type of the result during the compilation process – see below. They are not interpreted by Saxon-JS.)

In some cases, such as `LetExpr` and `ForExpr`, the context has to be altered to add a suitable variable binding and slot allocation to hold its value. In path expressions such as `RelativePathExpr` and `AxisStep`, the mapping is rather more complex and also involves the generation of specific JavaScript code, attached to the instruction element, to recognise candidate nodes in execution of the step.

The `FunctionCall` production can be used for many cases, apart from calls to core functions. Casting constructors for the `xs:atomic` types (e.g. `xs:double('NaN')`) are detected and converted to suitable `cast` instructions. Some function calls, such as `true()` are converted into direct instructions. For calls to the more regular core functions, the function signature is retrieved from a table and the arity of the call can be checked.

2.3. Static analysis and typechecking

Much of the power of XPath/XSLT comes from the ability to analyse the static context of sections of the expression and either determine errors (e.g. `1 + 'foo'`), find optimisations (e.g. `@foo/bar` will always yield the empty sequence `()`) or determine whether dynamic checks will be required (e.g. `string(foo)` will require the `child::foo` step to be checked for returning a result with a cardinality of zero or one.)

To do this requires a complete system to perform principally static type checking, and with the associated issues of type inference in operations such as arithmetic and determining when atomisation is needed, constitutes the hardest part of the development.

A static context is passed down through the recursive compilation, and all results have a type/cardinality computed for them. (The computed types are added as a JavaScript property to the elements – additionally writing their string values as `@type` attributes during development helps debugging as they appear in serializations of the instruction tree, but these are ignored by Saxon-JS.) Some instructions will need to refer to the type of the *current context item*; others may alter what the current context item is (e.g. `forEach`) and consequently alter its type for some of the children. Functions have definitive signatures which apply type constraints on their arguments and provide types for their result.

All these point to a requirement for a generic static type check mechanism which, given an instruction node and the required type from its parent context, either returns the construct if considered "type-safe"[10] or returns it surrounded with a suitable cast instruction if such is needed and permitted or detects and reports irreconcilable errors or wraps potentially errant instruction subtrees with suitable runtime check directives. This was achieved by transcribing the Java-based typechecker used in *Saxon-HE*[11] to a JavaScript equivalent, which uses many of the constants and tags in exactly the same form – this reduced transcription errors considerably.

Saxon-JS itself requires a type hierarchy model for runtime to satisfy `instance of` queries. This principally involves the built-in atomic types (`xs:NCName`, `xs:dayTimeDuration`…) and is defined alongside Javascript objects `Atomic.`*`typeName`* which acts as wrappers around suitable XDM implementations.

For static analysis however, more detail is required, particularly adding cardinality, so a compound `Type` object is used (wrapping a pointer to the singleton base type and a cardinality object). This is then used to generate an ensemble of type objects dynamically, such that many of the assertions and checks can be made using identity tests. For example `t = makeType("xs:string?")` produces a `Type` object such that `t.baseType == BuiltinAtomicType.STRING` and `t.card == StaticProperty.ALLOWS_ZERO_OR_ONE`. With this mechanism we can compute type assertions and checks quite smoothly.

2.4. Evaluation

With the execution plan now constructed as an SEF tree, then the expression can be evaluated, just like any other XPath subtree found within an XSLT stylesheet.

[10] Some operations effectively operate polymorphically.

[11] Perhaps one of the most critical sections of that product in terms of standards comformance.

The dynamic context must first be initialised. This involves setting the initial context item (if any) and setting up the values of the supplied parameters into their appropriate value storage slots. (When run under `xsl:evaluate` generating instructions for these are present as named subtree children.) Then the internal `SaxonJS.Expr.evaluate(SEF,context)` process is called. This will return an iterator over the results (or throw an error!), which will be treated as the result of `xsl:evaluate` and futher processed in the normal way.

3. Pure JavaScript XPath evaluation

The bulk of the work above was geared to supporting dynamic evaluation of XPath expressions within XSLT stylesheets run under Saxon-JS, through the `xsl:evaluate` instruction. But in the process we have constructed all the machinery to support evaluation from JavaScript itself, effectively through the compound:

```
evaluate(compile(parse(xpath),staticContext),contextItem,params)
```

This has been implemented as a function

```
XPath.evaluate(xpath,contextItem?,options?)
```

which will carry out the evaluation of the given XPath expression, with an optional binding to the initial context item, and a set of options which describe aspects both of the static (e.g. `xpathDefaultNamespace`) and the dynamic (e.g. `params: {conference : "XMLPrague", year: 2017}`) context, as well as controlling the result format[12].

This then means that with the Saxon-JS runtime loaded, but no XSLT stylesheet, it will be possible to evaluate XPath3.1 expressions. As an example here is a webpage on Figure 2 which sets the time on a number of contained clocks, both digital and analogue.

This is the JavaScript within that web page:

```
var thisDoc = window.document;

var timezones = {
  "London": "PT0H",
  "New York": "-PT5H",
  "San Francisco" : "-PT8H",
  "Delhi" : "PT5H30M",
  "Tokyo" : "PT9H"
};

function setClocks() {
```

[12]How a sequence of result items is presented - consistently as an array, "smart" (null, singleton or array) or as a (potentially lazy) iterator.

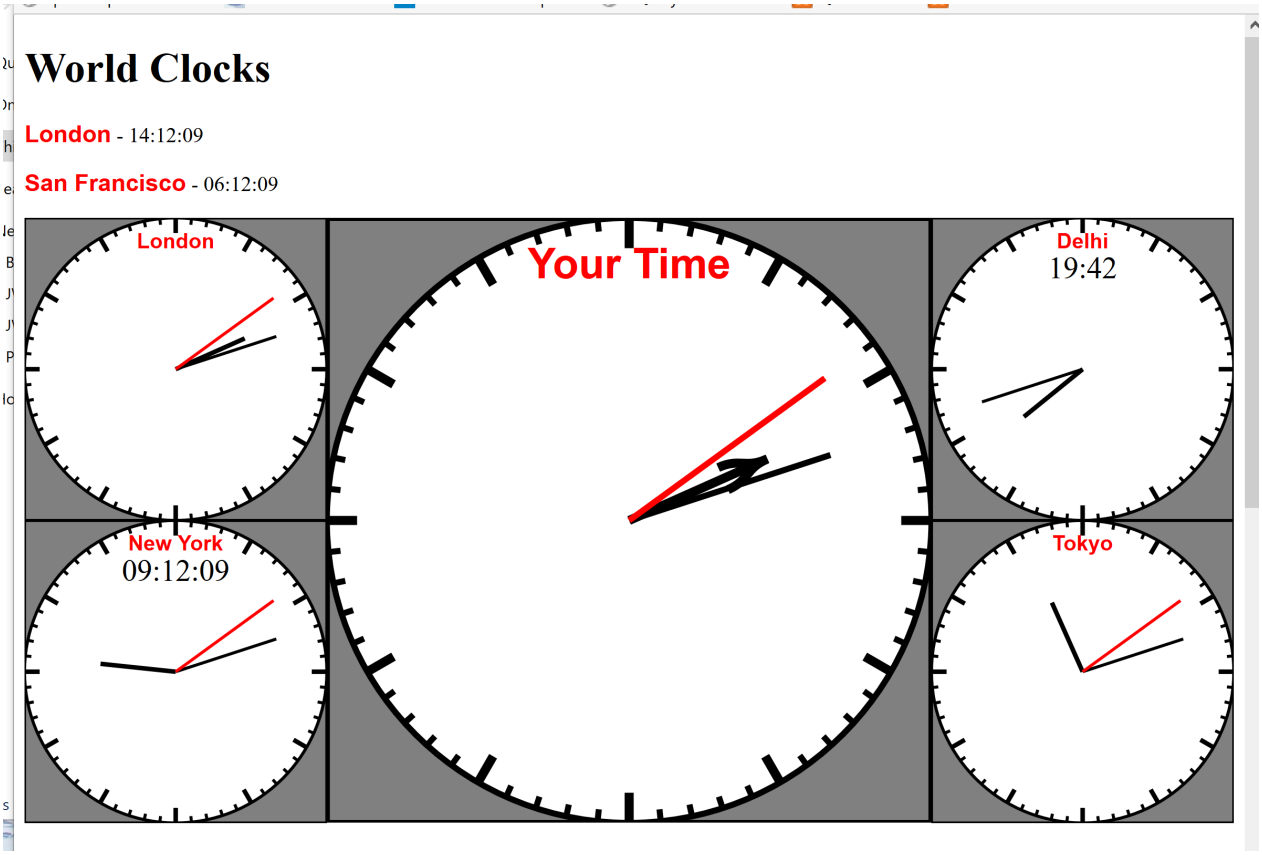


Figure 2. Clocks set by XPath

```
  var clocks = SaxonJS.XPath.evaluate(
    ".//*[tokenize(@class)='clock']",thisDoc,{resultForm:"array"});
  for(var i =0; i &lt; clocks.length; i++) {
    setClock(clocks[i]);
  }
}

function setClock(clock) {
  var findTime = "let $loc := normalize-space(.//*[@name='city']), "+
    "$t := if($loc = ('Local Time','')) then current-time() " +
    "else adjust-time-to-timezone(current-
time(),xs:dayTimeDuration(map:get($timezones,$loc)))" +
    "return map{'hour':hours-from-time($t), 'minute' : minutes-from-
time($t)," +
    " 'second' : floor(seconds-from-time($t))}";
  var time = SaxonJS.XPath.evaluate(findTime, clock,
                                    {params: {"timezones": timezones}});

  var findIndicators = "map:merge(for $class
                                  in ('hour','minute','second') " +
      "return map:entry($class, array {.//*[@class=$class]}))";
```

```
    var timeIndicators = SaxonJS.XPath.evaluate(findIndicators,clock);

    var computeAngles = "let $m := .?minute " +
      "return map{'hour':(.?hour mod 12) * 30 + $m div 2,
                  'minute' : $m * 6,
                  'second' : .?second * 6}";
    var angles = SaxonJS.XPath.evaluate(computeAngles, time);

    for(var part in timeIndicators) {
      var nodes = timeIndicators[part];
      for(var i = 0; i &lt; nodes.length; i++) {
        var node = nodes[i];
        if(clock.namespaceURI == "http://www.w3.org/2000/svg" &amp;&amp;
           !(node.localName == "tspan" || node.localName == "text" )) {
          node.setAttribute("transform","rotate("+ angles[part] + ")");
        } else {
          var t = time[part];
          node.textContent = t &lt; 10 ? "0"+t : t;
        }
      }
    }
  }
```

Each of the clocks (both HTML and SVG forms) are identified as being in class `clock` and each may contain an element with an attribute `name="city"`. For each clock the possible timezone offset is found from the name of that city (assumed the text content of the naming element) and the appropriate adjusted current time computed as separate hour, minute and second components. Then the indicators within that clock are found through an XPath returning a map of discovered nodes (which might be multiple and have `@class` either hour, minute or second) with suitable component label keys. Finally the time for each indicator is set: SVG non-text items are rotated by the appropriate angle using a `transform` attribute; all others have their text content set to the given number.

4. Testing

Very early work merely took an XPath expression and generated the candidate SEF, which was serialized and compared against what Saxon-EE would produce for the same expression. This was a very effective development method used throughout the work on the reasonable assumption that Saxon produces very highly compliant execution and that the Saxon-JS runtime had been tested by running with a server-side JavaScript interpreter such as Nashorn.

But there are many many aspects to XPath3.1, for which the dynamic compiler would have to be checked, so some (semi-)automated approach was attractive, or more likely essential. Obviously the QT3 testsuite has a large number of tests for

XPath and XQuery (more than 18,000 for XPath alone). Could we build a suitable test harness that ran some of these *entirely within the browser*? Yes we could and it proved to be highly effective, both in testing the parse/compile process and also in exercising and debugging the Saxon-JS runtime within the browser over more of the obscure facets of the QT3 test suite.

4.1. QT3 testing in the browser

This was the stimulus for getting some implementation of `xsl:evaluate` going very early in the project. It enabled an XSLT stylesheet to be written that processed the QT3 testsuite, passing each required test to `xsl:evaluate` and checking the result assertions and reporting the results *entirely within a webpage*. Development then became a question of running appropriate test sets by loading the web page, which loaded up the repertoire of QT3 test sets, clicking on those (groups) to be run and examining the results (assuming the JavaScript hadn't crashed!) In the case of failure of a specific test we can also display a serialisation of the compiled code to aid debugging.

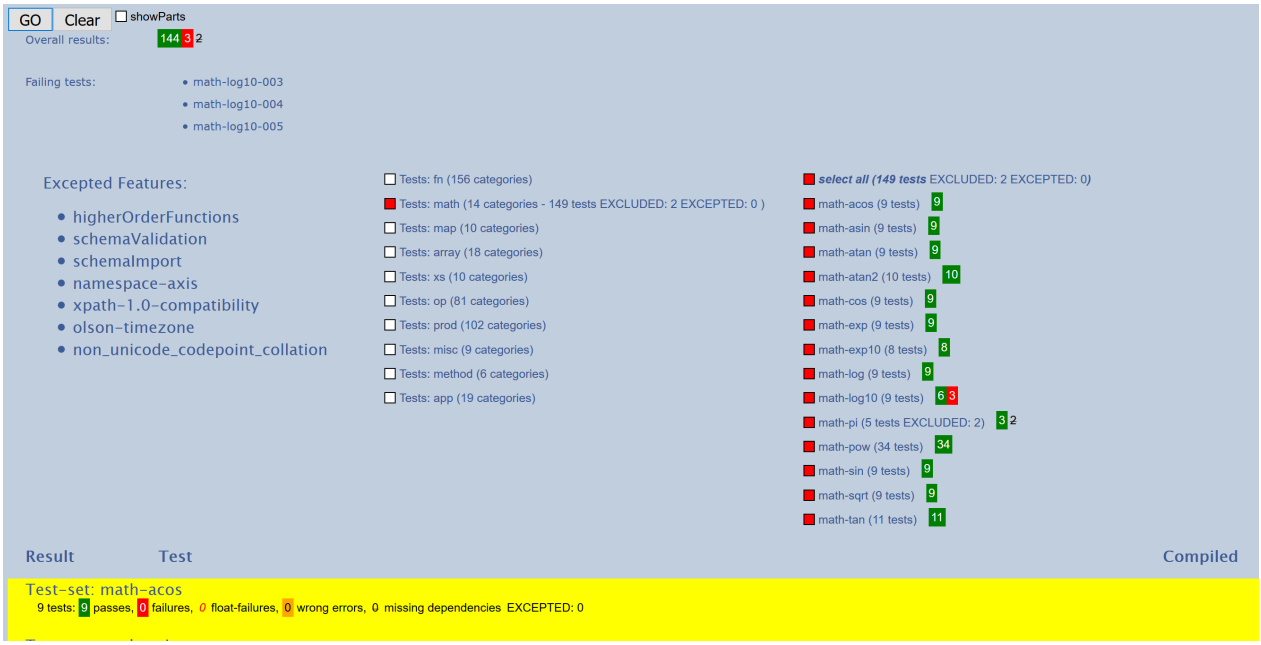


Figure 3. Testing the math: functions

This approach was used for almost all the development of the dynamic evaluation. As of writing some 18,000+ of the tests are passed with only between 110 and 190 failures, depending upon the browser used[13]. (These cover even tests for recent features such as arrays, maps, lookup operators, arrow application operators and so forth.) Here is an example of processing the `math:` function tests.

[13]There are some 4,000+ additional tests that are excluded because their dependencies include either XQuery or other features (such as higher-order functions or UCA collation) not supported by Saxon-JS

4.2. Comparing browser coverage

One of the advantages of running the QT3 tests entirely within a browser-borne setting is that performance can be checked for a variety of browsers, merely by loading the web page into the browser to be tested, clicking a few buttons and waiting for the result reports to be displayed. If we then arrange to save the final web page (`Save As...`)[14] then we have a machine-readable record of the results. By writing an XSLT stylesheet that reads in these pages[15] a comparison in coverage between the browsers can be generated. Here is the top-level comparison in terms of total errors[16]:

QT3tests - Saxon JS - browser comparison

Browsers tested:

Browser	Results					
Chrome	18347	116	173	~~4212~~	*49*	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (K
Edge	18325	138	173	~~4212~~	*49*	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (K
Unknown(IE?)	18303	188	173	~~4212~~	*21*	Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; .NET4.0C; .NI
Firefox	18344	117	175	~~4212~~	*49*	Mozilla/5.0 (Windows NT 10.0; WOW64; rv:50.0) Gecko/20100101 I
Opera	18347	116	173	~~4212~~	*49*	Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KH
Safari	18315	177	172	~~4212~~	*21*	Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/534.57.2 (KH

Excluded features

- higherOrderFunctions
- schemaValidation
- schemaImport
- namespace-axis
- xpath-1.0-compatibility
- olson-timezone
- non_unicode_codepoint_collation

Excepted test-sets

Test set	Reason
fn-collection	
fn-default-language	
fn-element-with-id	Crashes in Edge - Schema not detected?
fn-error	
fn-id	

Figure 4. Comparing browser results

---

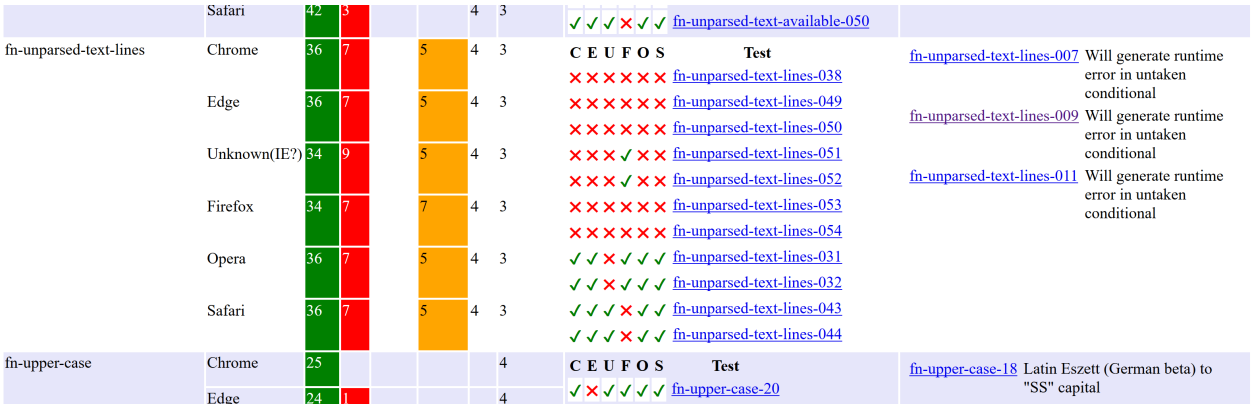
at present. A smaller number of tests (perhaps <100) are excluded specifically because they impose unrealistic demands (e.g. a map with 500,000 entries, which crashes Safari, but interestingly not the other main browsers) or are subject to some dispute (particularly so in case of statically inferred errors that dynamically will not occur)

[14]For the Edge and Internet Explorer browsers, the menu save is only the original source XHTML: to save for these browsers involves an "Inspect Element " action and a copy-and-paste of the whole active webpage DOM.

[15]Safari excluded, these are well-formed XML, so `doc()` suffices. Safari uses a binary `.webarchive` format, which can be read and decoded using EXPath File and Binary extension functions and `parse-xml()` on the resulting string.

[16]Green denotes successful tests, red failures, orange expected failures but with the wrong error code. Strike through are the number of tests not run, either by specific exception or excluded by unsupported features. Red italics indicate a failure where the XPath expression used the `xs:float` type – this is coerced to a JavaScript `Double` in Saxon-JS so results are computed to higher precision than anticipated or permitted and exact equality comparisons in the test assertions can fail.

Detailed comparison on failing tests by browser is also reported, as shown here, where different successes for tests of `unparsed-text-lines()` are revealed. (The last column lists tests that were not run deliberately and the reason why: in this case the test assertion of error is a pessimistic static assumption.)



	Safari	42	3			4	3	✓✓✓✗✓✓ fn-unparsed-text-available-050	
fn-unparsed-text-lines	Chrome	36	7		5	4	3	C E U F O S Test	fn-unparsed-text-lines-007 Will generate runtime error in untaken conditional
								✗✗✗✗✗✗ fn-unparsed-text-lines-038	
	Edge	36	7		5	4	3	✗✗✗✗✗✗ fn-unparsed-text-lines-049	fn-unparsed-text-lines-009 Will generate runtime error in untaken conditional
								✗✗✗✗✗✗ fn-unparsed-text-lines-050	
	Unknown(IE?)	34	9		5	4	3	✗✗✗✓✗✗ fn-unparsed-text-lines-051	
								✗✗✗✓✗✗ fn-unparsed-text-lines-052	fn-unparsed-text-lines-011 Will generate runtime error in untaken conditional
	Firefox	34	7		7	4	3	✗✗✗✗✗✗ fn-unparsed-text-lines-053	
								✗✗✗✗✗✗ fn-unparsed-text-lines-054	
	Opera	36	7		5	4	3	✓✓✗✓✓✓ fn-unparsed-text-lines-031	
								✓✓✗✓✓✓ fn-unparsed-text-lines-032	
	Safari	36	7		5	4	3	✓✓✓✗✓✓ fn-unparsed-text-lines-043	
								✓✓✓✗✓✓ fn-unparsed-text-lines-044	
fn-upper-case	Chrome	25					4	C E U F O S Test	fn-upper-case-18 Latin Eszett (German beta) to "SS" capital
	Edge	24	1				4	✓✗✓✓✓✓ fn-upper-case-20	

Figure 5. Detailed error comparison

4.3. Testing the JavaScript API

Testing the JavaScript API required a slightly different approach. If we assume the QT3 tests run through `xsl:evaluate` mechanism has proved both the XPath→SEF and SEF(args)→result paths are correct, we just need to test the calling options for `SaxonJS.XPath.evaluate()`. This is most conveniently carried out by constructing a web page with tabular entries containing argument components of XPath expression, context item and options, with a simple JavaScript that iterates across table rows, accumulating these components from textual values of cells, then calling the dynamic evaluator and writing the results (possibly serialized) into another result-holding cell.

5. Performance

There are two aspects of performance to consider. The first is accuracy of execution and the degree of conformance to standards. As has been indicated in the previous section, we've managed to achieve a very high degree of such as measured by the QT3 test suite, failing ~0.5% of the tests. Some errors are inevitable (e.g. the use of `Double` for `xs:float`), others are in very obscure cases (uncaught errors that actually can never be triggered), a few are just wrong and will eventually get corrected. In practice we consider the product has high enough conformance for production use.

The second of course is execution speed. This we have not measured directly yet, but the best indicator we have so far is the execution of sections of the QT3 test suite. As an example, on a high-end (2016) Windows laptop, the 3446 tests for the operators (`op-except`, `op-numeric-add` etc) are processed completely in 27

seconds under Firefox and 15 seconds under Opera, from original "click" to finished results, including execution of all the surrounding XSLT test harness[17]. As for each of the tests i) it needs to be compiled and executed and ii) its result assertions tested (some of which contain XPath expressions that must be evaluated, thus needing additional compilation), there are >3446 XPath evaluations required. This suggests that each evaluation takes somewhere in the 2-5 ms region. As we anticipate most use of these dynamic features will involve small numbers of XPath expressions, with "a human in the loop", we do not forsee significant problems with compilation/execution performance.

6. Future Developments & Conclusions

Like the Stylesheet Export File, we use XML trees (in this case in-browser DOM trees) to represent the intermediate parse results and generated code. But the manipulation of these trees involved is not complex, mostly concerned with testing node name, child and/or attribute existence and iterating over children. It has been suggested (in [3]) that the export format might also have a JSON-based alternate. If this was the case then much of the compiliation code described here could be recast relatively easily to use a JSON representation as the main data type e.g.

```
1 to 2 * 10  →
{ name : "RangeExpr",
  children : [
    { name: "IntegerLiteral", value: 1 },
    { name: "MultiplicativeExpr", op: "*",
      children: [
         { name: "IntegerLiteral", value: 2 },
         { name: "IntegerLiteral", value: 10 }
               ] }
            ]
}
```

We've suggested earlier that the reduction phase of the initial parse might be improved by a smarter use of callbacks within the parse-tree former. Obviously during the compilation process there will be many more opportunities to optimise the resulting code. The easiest would be the complete evaluation of literal subexpressions, i.e. those for which there is no dependency on execution context within the subtree. In such cases the compiled subtree can be evaluated and its sequence result projected as a sequence of suitable instructions, based on the literal forms. Whether it is worth doing this however is a moot point, as, unless the same (dynamic) XPath is going to be executed repeatedly, little extra would be

[17]Most browsers show similar performance within perhaps a factor of 2, with the noteable exception of Internet Explorer which seems to be consistently about 5x slower than the rest.

gained and indeed additional time would be taken up in determining *whether* an evaluation could be performed.

This leads on to the issue of whether an intermediate compiled output is desirable. Earlier versions of Saxon (within a Java environment) used a pair of functions `saxon:expression()` and `saxon:eval()`, with the former producing a stored expression that the latter would use to evaluate against a given dynamic context. A similar partitioning within JavaScript could be provided easily.

6.1. Conclusion

This paper has shown that when an XPath "instruction execution engine" is available within a browser context, it is possible to add a dynamic XPath evaluation facility provided that i) an accurate and efficient XPath parser is also available and ii) very accurate code for static type checking, coercion and casting is developed. Development of such a feature is aided considerably by the existence both of an external "oracle" to demonstrate what a correct "execution plan" should be for a given XPath expression (in this case using Saxon-EE and examining generated SEF) and the early construction of a test harness to exercise the QT3 test suite (in this case as an XSLT stylesheet invoking via the `xsl:evaluate` instruction.)

In the case of Saxon-JS and the `XPath.js` additional module we have developed an implementation with very high levels of conformance to the XPath 3.1 specification, demonstrated by rapid running of the the QT3 test suite *entirely within the browser*. We have also been able to demonstrate and record the (small) differences in conformance between the half-dozen major browsers.

References

[1] O'Neil Delpratt and Michael Kay. *Multi-user interaction using client-side XSLT.*. 2013. `http://archive.xmlprague.cz/2013/files/xmlprague-2013-proceedings.pdf`

[2] Sergey Ilinsky. *XPath 2.0 implementation in JavaScript* . 2016. `https://www.openhub.net/p/xpath-js`

[3] Debbie Lockett and Michael Kay. *Saxon-JS: XSLT 3.0 in the Browser.*. 2016. `http://www.balisage.net/Proceedings/vol17/html/Lockett01/BalisageVol17-Lockett01.html`

[4] Steven Pemberton. *Invisible XML.*. 2013. `http://www.balisage.net/Proceedings/vol10/html/Pemberton01/BalisageVol10-Pemberton01.html`

[5] Gunther Rademacher. *REx Parser Generator*. 2016. `http://www.bottlecaps.de/rex/`

[6] *XQuery in the Browser*. 2016. `http://www.xqib.org/index.php`