Implementation of Portable EXPath Extension Functions

Adam Retter

Evolved Binary

<adam@evolvedbinary.com>

Abstract

Various XPDLs (XPath Derived Languages) offer many high-level abstractions which should enable us to write portable code for standards compliant processors. Unfortunately the reality is that even moderately complex applications often need to call additional functions which are non-standard and typically implementation provided. These implementation provided extension functions reduce both the portability and applicability of code written using standard XPDLs. This paper examines the relevant existing body of work and proposes a novel approach to the implementation of portable extension functions for XPDLs.

Keywords: XQuery, Portability, EXPath, Haxe

1. Introduction

High-level XML processing/programming languages such as XQuery, XSLT, XProc and XForms have long held the promise of being able to write portable code that can execute on any W3C compliant implementation. Unfortunately the specification of these languages leave several issues to be "implementation defined"; Typically a pragmatic necessity, most often occurring where the language must interact with a lower-level interface, e.g. performing I/O or integrating with the environment of the host system.

If we put to one-side the potential "implementation defined" incompatibilities, which in reality are often few and can likely be worked around, there is another issue which hinders the creation of portable code, and that is the issue of implementation provided extension functions. XQuery, XSLT, XProc and XForms are all built atop XPath, which defines a Standard Library in the form of the F+O specification (XPath and XQuery Functions and Operators). XQuery 3.0 and XSLT 3.0 provide F+O 3.0 [1], and whilst XProc 1.0 and XForms 2.0 provide the older F+O 2.0 [2] it is most likely that new versions of those specifications will also adopt F+O 3.0.

Whilst F+O 3.0 offers some 164 distinct functions and 71 operators, it is predominantly focused on manipulating XML, JSON and text, unfortunately for creating complex processes or applications with XPDLs (XPath Derived Languages) e.g XQuery, XSLT, XProc and XForms, these functions by themselves are not enough. To fill this gap, many implementations have provided their own modules of extension functions to their users; for instance eXist 2.2 provides some 53 modules [1] of extension functions for XQuery, and similarly MarkLogic 8 provides some 55 modules [2] of extension functions for XQuery.

The aim of this paper is that through examining the existing approaches to portable extension functions for XPDLs, an new approach is developed for their implementation which should enable them to be reused by any XPDL processor with the minimum of effort.

1.1. Extension Function Costs

Initially these extension functions are most welcome as they enable the user to quickly and easily perform additional operations which would be impossible (or costly) to implement in an XPDL. Unfortunately over time these extension functions add a burden with regards to portability [3] [4], which typically manifests itself in two distinct ways: directly and indirectly.

1.1.1. Directly

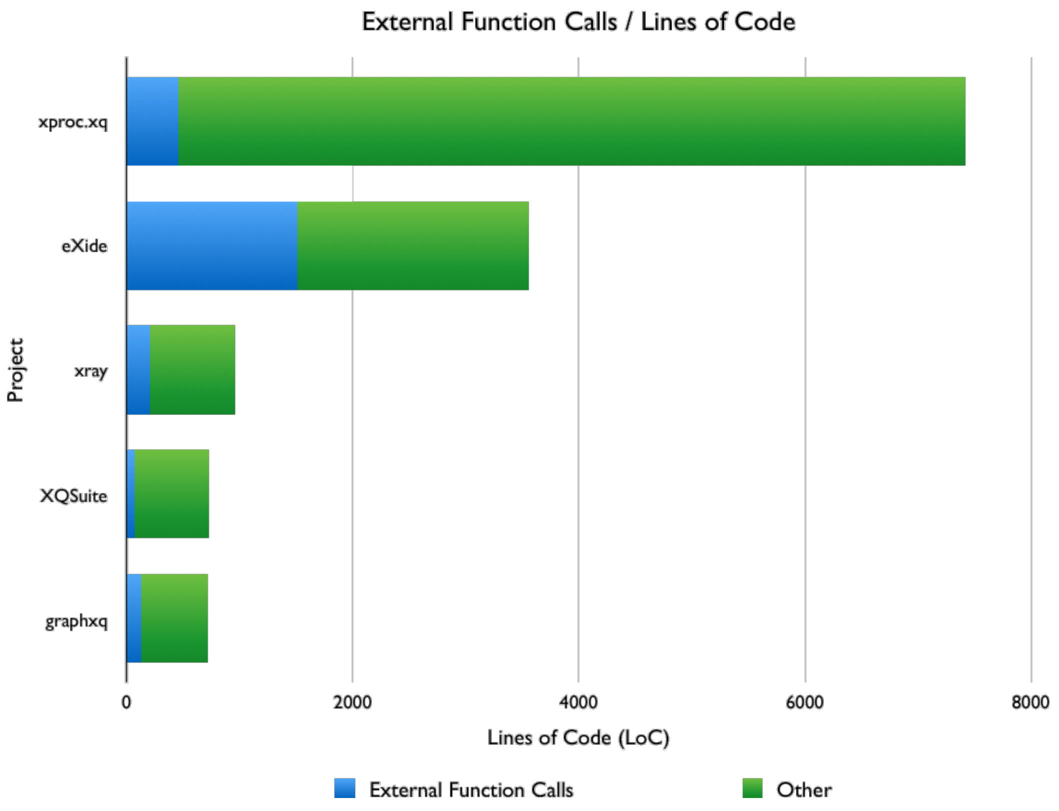
--- Restricting User Freedom

The use of proprietary implementation extension functions can adversely restrict the ability of a user to freely move between implementations or reuse their

[1] eXist XQuery extension modules were counted by examining the eXist source code at https://github.com/eXist-db/exist/tree/eXist-2.2
[2] MarkLogic XQuery extension modules were counted by examining the MarkLogic documentation at https://docs.marklogic.com/all

existing code across implementations. An examination of several Open Source projects (eXide[1], graphxq[2], xproc.xq[3], xray[4] and XQSuite[5]) which are implemented in XQuery reveals that the impact of this is typically a function of the size of the code base as illustrated in Figure 1, and the variety of extension functions that have been used as illustrated in Figure 2 and Figure 3.

Figure 1. External Function Calls / Lines of Code



[1] https://github.com/wolfgangmm/exide revision 07207a2 (12 April 2015)
[2] https://github.com/apb2006/graphxq revision 0b19756 (8 March 2015)
[3] https://github.com/xquery/xproc.xq revision f0f0697 (13 December 2014)
[4] https://github.com/robwhitby/xray revision dc03243 (25 April 2015)
[5] https://github.com/eXist-db/exist/tree/develop/src/org/exist/xquery/lib/xqsuite revision c32784a (5 May 2015)

Figure 2. Total Function Calls

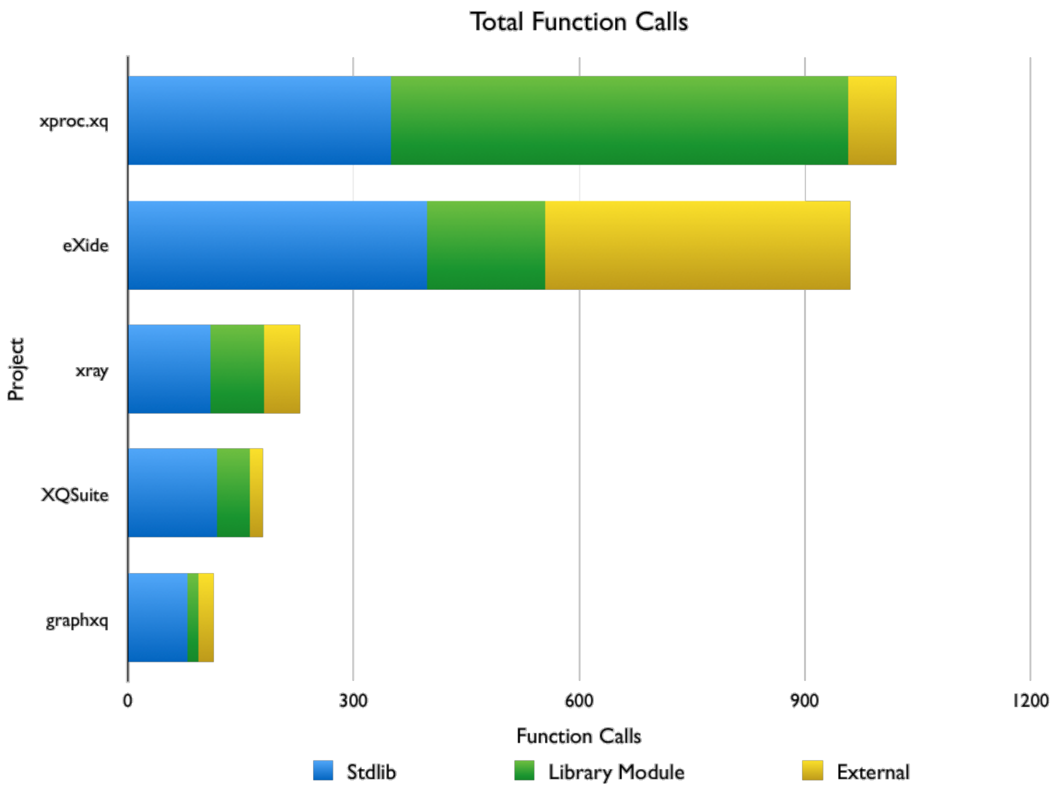
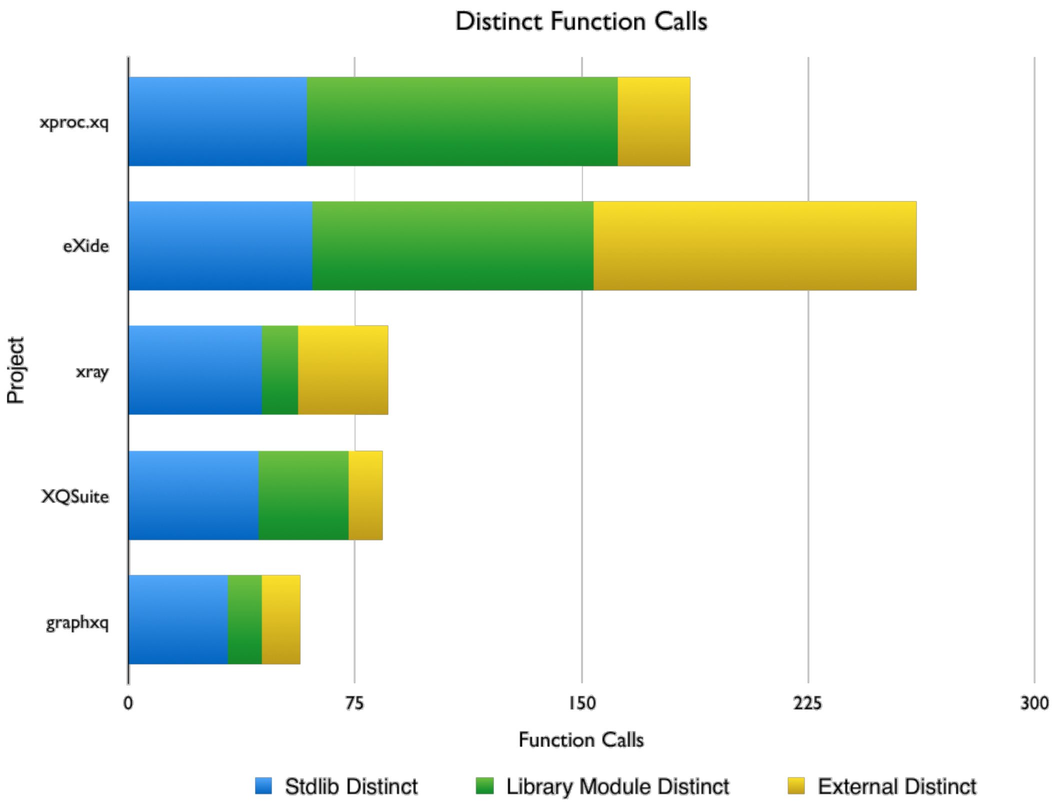


Figure 3. Distinct Function Calls



The most extreme example of this impact is often felt by XQuery framework providers (e.g. XRay and xproc.xq, xqmvc[1], etc.) who often have to attempt to abstract out various implementation extension functions to be able to provide frameworks which will work on more than one implementation. We therefore conclude that implementation specific extension functions restrict freedom by impairing code reuse.

1.1.2. Indirectly

--- Fragmenting the Community

In comparison to the C++ or Java communities, the XPDL communities are considerably smaller. The TIOBE Programming Community Index [5] for May 2015 shows that Java is the most popular programming language and that C++ is third, no XPDL languages appear in the top 100. Likewise, the PYPL Index [6] for May 2015, shows Java and C++ to hold first and fifth positions respectively, with no XPDL languages appearing in the top 16. The Redmonk Programming Language Ratings [7] for January 2015 place Java in second and C++ in joint fifth position in terms of popularity rank across StackOverflow and GitHub. From the plot produced by Redmonk we can infer that in comparison XSLT has ~57% and ~77% of the popularity rank on GitHub and Stack Overflow respectively, whilst XQuery has just ~14% and ~50%.

The last 10 years has produced an exponential growth in Open Source projects; Deshpande and Riehle reported in 2008 [8] from analysing statistics for Open Source projects over the previous 10 years that Open Source growth was doubling about every 14 months. With recent social coding services such as GitHub and BitBucket and physical events facilitated by meetup.com and others, there is likely a much greater tendency to publish even small snippets of code or utilities as open source for others to reuse.

However, when publishing XPDL code projects, if those projects depend on implementation specific extension functions, then it is often non-trivial for a user of a differing implementation to adapt the code. Even if a user can adapt the code to their implementation, if they then wish to improve it, the ability to contribute these changes back upstream is also impaired as the code bases have most likely diverged; As such further forking is implied. We conclude from this that implementation specific extension functions further fragment the XPDL communities into smaller implementation specific sub-communities by restricting portability and code sharing.

2. Prior Art

This paper is not the first work to look at improving the portability of XPDLs. In this section, previous efforts in the area of improving the portability of extension functions within one or more XPDLs are examined.

2.1. EXSLT

The EXSLT project [9] which first appeared in March 2001, at the time focused on extension functions and elements for XSLT 1.0. Arguably, XSLT 1.0 had a very limited standard library provided by the core function library of XPath 1.0 [10], with just 27 functions, augmented with 7 additional functions. EXSLT recognised that much of the XSLT community required additional extension functions and elements and that it would be desirable if such functions and elements were the same across all XSLT implementations to ensure the portability of XSLT code. EXSLT specified a set of 8 modules which include extension functions allowing XSLT developers to write portable code for tasks that were not covered by the XSLT 1.0 specification. EXSLT itself did not provide an implementation of the functions, rather it tightly defined the XSLT signatures and operational expectations and constraints of its extension functions. Any vendor may choose to implement the EXSLT modules within their XSLT implementation, however the standards set out by the EXSLT project ensure that their invocation and outcome must be the same across all implementations.

The last update to EXSLT was in October 2003, and whilst still used by many XSLT developers its relevance has decreased since the release of XSLT 2.0 [11] which expanded its standard library by adopting F+O 2.0 which provides 114 functions and 71 operators, many of which were likely inspired by EXSLT. The utility of EXSLT will likely be further reduced by the upcoming release of XSLT 3.0 which adopts F+O 3.0.

2.2. XSLT 1.1

XSLT 1.1 [12] of which the last public working draft was published in August 2001 (although the first working draft appeared in December 2000), had the stated primary goal to “improve stylesheet portability”, and included a new and comprehensive mechanism for working with extension functions in XSLT.

XSLT 1.1 like XSLT 1.0 permits the use of extension functions which are implementation defined and whose presence is testable through the use of the fn: function-

[1] The XQMVC projects' attempt at supporting both MarkLogic and eXist XQuery processors - https://code.google.com/p/xqmvc/source/browse/#svn%2Fbranches%2Fdiversify%2Fsystem%2Fprocessor%2Fimpl

available function. However, XSLT 1.1 went much further than its predecessor by introducing the `xsl:script` element which made possible the implementation of an extension function within the XSL document itself either directly in program code or by URI reference. When two distinct programming languages interact, there is always the issue of type mapping, to solve this XSLT 1.1 specified explicit DOM2 core model and argument type mappings for ECMAScript, JavaScript and Java. Extension function implementation was not limited to just ECMAScript, JavaScript or Java, however bindings and mappings for other languages were considered outside of the scope of XSLT 1.1 and were left to be implementation defined.

Providing a user of XSLT 1.1 had used extension functions implemented in either ECMAScript, JavaScript or Java, and those functions were either implemented inside an `xsl:script` element or available from a resolvable URI on the Web, then it was entirely possible to consume and/or create portable extension functions for XSLT.

The addition of `xsl:script` in XSLT 1.1 was highly controversial [13] with opponents on both sides of the debate [14][15][16]. Unfortunately, before XSLT 1.1 was finished, it was considered unworkable for several reasons by the W3C XSLT Working Group [17], and was permanently suspended to be superseded by XSLT 2.0 [11]. XSLT 2.0 adds little more than XSLT 1.0 in the area of extension functions and altogether abandons the type mapping from XSLT 1.1, clearly stating that: "The details of such type conversions are outside the scope of this specification".

2.3. FunctX

FunctX [18] released by Priscilla Walmsley in July 2006 provides a library of over 150 useful common functions for users of XQuery and XSLT. The purpose of this library is to remove the need for users to each implement their own approaches to common tasks and to provide a code set that beginners could learn from.

FunctX provides two implementations, one in XQuery 1.0 and the other in XSLT 2.0; neither require any implementation specific extensions and as such are entirely portable and useable with any W3C compliant XQuery or XSLT processor.

The availability of FunctX has almost certainly reduced the amount of duplicated effort that otherwise would have been spent by developers working with XPDLs and also removes the temptation for vendors to provide proprietary alternatives to assist their users.

2.4. EXQuery

The EXQuery project [19] which started in October 2008 as a collaborative community effort set out with the initial goal of raising awareness of the portability problems that could result from the use of non-standard vendor extensions in XQuery. Focused solely on XQuery, the non-standard extensions which could causes issues were set out as including extension functions, indexing definitions, collections, full-text search and the URI schemes used for the XPath `fn:doc` and `fn:collection` functions.

The EXQuery project firstly approached the problem of non-standard implementation specific extension functions for XQuery, with the desire to define standard function signatures and behaviour for similar XQuery functions which appeared across several implementations.

The EXQuery project shortly abandoned its work on defining standard function signatures for XQuery extension modules in favour of the EXPath project (see Section 2.5, "EXPath") which appeared in 2009, instead focusing on XQuery specific portability issues like server-side scripting resulting in RESTXQ [4].

The EXQuery project goes further than just defining standards documents that define intention and behaviour of a specific system, it also provides source code for a common implementation that may be adopted as the base for any implementation [20]; Although currently limited to Java the project has also expressed interest in producing C++ implementations.

2.5. EXPath

The EXPath project [21] started in January 2009 whilst independent had many similar goals to the EXQuery project. Critically, with regards to extension functions, it is recognised that defining standards for these at the lower XPath level as opposed to the XQuery or XSLT level would make them more widely applicable to any XPDL.

The EXPath project provides two types of specification for XPDLs, the first looks at the broader ecosystem of delivering XPDL applications (e.g. Application Packaging and Web), whilst the second and more widely adopted, focuses on defining extension function modules. It is this second specification type of extension function modules that are of interest to this paper.

The EXPath project to date has released three specifications for standard extension modules for XPDLs: Binary Data Handling, File System API and HTTP Client. In addition, at the time of writing there are another five extension module specifications under development which focus on: File Compression, Cryptography, Geospatial and NoSQL database access. The EXPath project like the EXSLT project focuses on defining function signatures and behaviour, albeit at the XPath as opposed to the XSLT level; again the goal being that any vendor may implement an extension module standard and that users will benefit from code portability across all implementations that support the EXPath specifications.

Whilst the EXPath project has predominantly focused on defining standards documents that specify the intention and behaviour of a number of modules of related XPath extension functions, there have been some related efforts [22][23][24] to produce common implementation code for JVM (Java Virtual Machine) based implementations.

3. Analysis

The review of prior art in Section 2, "Prior Art", uncovers three distinct approaches to reduce the impact of non-portable extension functions in XPDLs:

1. Function Standardisation

   Specifying function libraries and the exact behaviour of those functions so that vendors may each implement the same functions. EXSLT, EXQuery and EXPath all take this approach, although EXQuery and EXPath also have some support for reducing the overhead of implementing (for the JVM) by providing common code.
2. Function Distributions

   Providing libraries of ready-to-use common functions that are implemented in a language known to every implementation. This is the approach taken by FunctX, whose implementations are provided in pure XQuery or XSLT.
3. Implementation Type Mapping

   Tightly defining the function interface and type mapping between the host language and the extension function language. This is the approach taken by XSLT 1.1, which when restricting implementation to ECMAScript, JavaScript or Java, would have enabled the creation and use of libraries of portable extension functions for XSLT. Arguably XSLT 1.1 also overlaps with the Function Distributions approach as it allows the implementation of the extension function to be embedded within the XSLT itself.

Function Standardisation is a great start, but without a majority of significant implementations [25], adoption is likely to remain a problem. Implementation can be assisted by reducing the overhead for vendors to achieve this, one such mechanism is providing common code; however, this must be inclusive to languages other than those atop the JVM (See Section 3.2, "XPDL Implementation Survey").

Ignoring source-level interoperability for the moment, one issue with providing common code is that each implementation almost certainly has a different type system and approach to representing the XDM [26] types (amongst others). The Implementation Type Mapping approach taken by XSLT 1.1 demonstrates an interesting mechanism for solving this by explicitly laying out a type model and mappings from XSLT to the implementation language. Both the EXQuery and EXPath projects have also made embryonic attempts at defining mappings for XDM types, however both are restricted to the JVM through their use of Scala [27] and Java [28] respectively.

Function Distributions of Standardised Functions is the ultimate goal; The ability to distribute extension functions for XPDLs that will interoperate with any implementation. However, without Implementation Type Mapping and standard interfaces it is certainly impossible that an implementation of an XPDL extension function would work with an unknown vendors XPDL implementation.

Implementation Type Mapping should be considered as the foundation layer for any form of interoperability between an XPDL extension function and varying XPDL implementations. Without this every implementation of an XPDL extension function for a specific XPDL platform would require re-implementation.

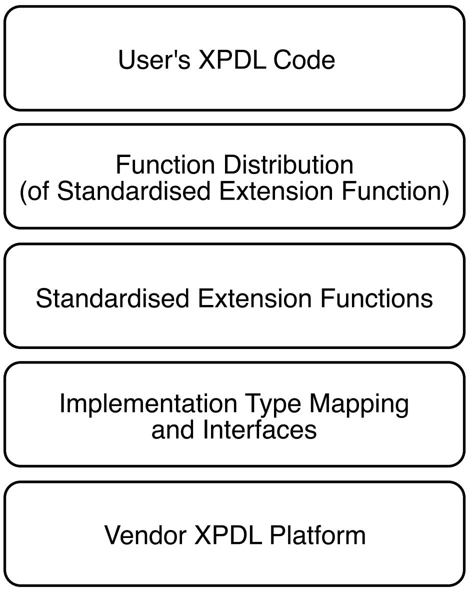
If we want to solve the problem of portable extension functions for XPDLs then it would seem that we must adopt a layered approach where we combine aspects of all three existing approaches:

1. An Implementation Type mapping needs to be created which is either at a level of abstraction that is not specific to any particular implementation language or can be losslessly implemented in a specific language, yet is still specific enough to constrain implementations to extension function standard specifications.

   Function Standardisation for extension functions needs to take place at the XPath level so as to ensure that the functions are applicable to the widest range of XPDLs.

   Standardised Functions need to be implemented according to an Implementation Type Mapping to form a Function Distribution, but in a language that allows them to be distributed in either source or binary form for any vendor implementation regardless of platform.

Figure 4. Layered Approach to Portable XPDL Extension Functions


User's XPDL Code
Function Distribution (of Standardised Extension Function)
Standardised Extension Functions
Implementation Type Mapping and Interfaces
Vendor XPDL Platform

3.1. Commonality of EXPath Standardised Extension Functions and Implementation Type Mapping

Whilst the EXPath project has provided definitions for several modules of Standardised Functions for XPDL extension functions, there has been little work by EXPath or others [29] in reducing the duplication of effort across vendors who wish to implement these functions, i.e. by exploring Implementation Type Mapping.

Consider the signature of the `file:exists` function (as shown in Example 1, “file:exists function signature”) which is just one of the Standardised Functions from the EXPath File Module [30].

Example 1. file:exists function signature

```
file:exists($path as xs:string) as xs:boolean
```

When we examine the three known implementations of this for BaseX 8.1.1 [31], eXist 2.2 [32] and Saxon 9.6.0.5 [33] we find that each implementation is very similar; Each implements a host interface which represents an XPDL function, and within that implements a host function which has access to the arguments and context of the XPDL function call. A *simplified* representation of the interfaces of these processors is extracted:

Example 2. BaseX Extension Function Interface

```
interface StandardFunc {
  Item item(QueryContext qc, InputInfo ii)
      throws QueryException;
}
```

Example 3. eXist Extension Function Interface

```
interface BasicFunction {
  Sequence eval(Sequence[] args,
      Sequence contextSequence)
      throws XPathException;
}
```

Example 4. Saxon Extension Function Interface

```
interface ExtensionFunctionCall {
  SequenceIterator call(
    SequenceIterator[] arguments,
    XPathContext context) throws XPathException;
}
```

Whilst there is currently no non-Java implementation of the EXPath File Module, if we examine a similarly simple function such as XPath's `fn:year-from-date` in XQilla [34] (a C++ implementation) then we can again extract a *simplified* function interface:

Example 5. XQilla Function Interface

```
class XQFunction {
  public:
    Sequence createSequence(DynamicContext*
        context, int flags=0) const;
};
```

The similarity of these interfaces leads us to conclude that there is further room for common abstraction and that specifying a standard Implementation Type Mapping and interfaces could lead to a reduction in duplicated effort for implementers of these EXPath extension functions and therefore any XPDL extension functions.

3.2. XPDL Implementation Survey

To achieve the broadest appeal between implementers of XPDL extension functions, it cannot be assumed that primary support for Java, ECMAScript or JavaScript in itself will be acceptable to the larger community; As partially demonstrated by the failure of XSLT 1.1 (see Section 2.2, “XSLT 1.1”). Therefore, any Implementation Type Mapping or Function Distribution should be applicable to any platform and most likely not just limited to the JVM [27][28]. To inform how such a Mapping or Distribution may be implemented, we should first understand the variety of source languages of existing XPDL processors. The results of a survey of XPDL processors is presented in Table 1, "Survey of XPDL Implementations".

Table 1. Survey of XPDL Implementations

C	C++	Haskell	Java	JavaScript	.NET	Objective-C	Pascal
libxml2[a]	Berkley DBXML (libxquery-devel) [a][b]	Haskell XML Toolbox[a]	Altova Raptor XML[a][b][c]	Frameless[a] [c]	Exselt[a][c] (F#)	GDataXML[a]	Xidel[a] [b]
libxslt[c]	Intel SOA Expressway XSLT[c]	HXQ[a][b]	Apache VXQuery[b]	Saxon/ CE[c][d]	.NET Standard Library `XmlNode`[a]	NSXML[a][b] [c]	
Saxon/ C[c][d]	MarkLogic[a][b][c]		BaseX[a][b]	xpath NPM[a]	XMLPrime[a] [b][c] (C#)	Panthro[a][b]	
	pugixml[a]		DataDirect XQuery[b]	XQIB[b]	xsltc[c] (C#)		
	QtXmlPatterns[a] [b][c]		EMC Documentum[a] [b][c]				
	Sedna[a][b]		eXist-db[a][b][c]				
	Sablotron[a][c]		GNU Qexo[a][b] [c]				
	TinyXPath[a]		IBM WebSphere Application Server Feature Pack for XML[a] [b][c]				
	Xalan-C++[a][c]		Qizx[a][b][c]				
	XQilla[a][b]		Saxon[a][b][c]				
	Zorba[a][b][c]		Xalan-J[a][c]				

[a] Implements XPath
[b] Implements XQuery
[c] Implements XSLT
[d] Source-level port of Saxon from Java

The survey was produced from aggregating the W3C XML Query list of implementations [35], the EXPath CG list of XPath engines [36] and relevant Google searches. The aggregate list was then reduced to those implementations for which information was still available and up-to-date. The list of programming languages for the survey was chosen based on the available implementations, and the native language of that implementation; For example with the Go programming language the approach appears to be to call xsltproc [37] (a wrapper around libxslt which itself is implemented in C), and the common approach from Python seems to be to use the lxml python wrapper [38] for libxml2 and libxslt (both themselves implemented in C).

From the survey it is clear that there is no lack of native programming language implementations of XPDL processors. The majority of implementations are written in Java and C++, which could likely be justified by the size of the C++ and Java communities (as briefly discussed in Section 1.1.2, "Indirectly").

4. Portable XPDL Extension Function Implementation

From an analysis of the current state of the art in regard to extension functions for XPDLs it can be determined that if we want to reduce the effort to implement a standardised extension function then we need to provide an Implementation Type Mapping; This allows the implementer of a standardised extension function to code to a standard interface without worrying about vendor specifics. However, such an Implementation Type Mapping needs to take into account the implementation language of the vendors XPDL processor, and this could potentially lead to an issue of fanout with many similar Implementation Type Mappings, one for each implementation language, which is far from ideal.

In addition, we have seen that providing common code can help to reduce the effort which is duplicated by each vendor implementing the same XPDL extension functions. Unfortunately this further compounds the fanout issue, as it would be very time consuming to provide common implementation code for each XPDL extension function in every known XPDL platform implementation language.

4.1. Implementation Portability

Ideally we would like to be able to specify a single Implementation Type Mapping and implement any XPDL extension function just once according to that mapping and have it execute with any vendors XPDL implementation.

Sun Microsystems coined the phrase "Write Once, Run Anywhere" (WORA) around 1996 in relation to Java [39]. Java is a high-level language which avoids platform specific implementation details by compiling to byte-code which is then executed by a virtual machine. The ability to distribute an XPDL extension function as byte-code has several attractions, such as the user not having to compile any code. However, the promise is somewhat shallow as executing Java requires a JVM to be installed on the target platform, without that the byte-code cannot be executed. For those vendors whose implementations are themselves not written in Java, they could still execute an XPDL extension function written in Java via JNI (Java Native Interface), however it may not be desirable to also force their users to install a JVM on their systems. A WORA experience for XPDL extension functions could eliminate the fanout cost of implementation, however Java is not suitable for all implementations.

If we can't achieve WORA we could instead consider falling back to a WOCA (Write Once, Compile Anywhere) approach where we distribute the Implementation Type Mapping and any common implementation source code in a single language that can be compiled on any platform. At first, C or C++ would seem a suitable choice for WOCA due to the fact that many XPDL processors are implemented in C or C++ and any XPDL processor implemented in Java could call a C or C++ implementation of an XPDL extension function via JNI. Through SWIG [40] we could also make any C or C++ XPDL extension function applicable to XPDL processors implemented in many other languages. Whilst C and C++ have many desirable properties, such as instruction set portability, compiler availability, and interoperability, the code is often highly hardware (e.g. big-endian vs little-endian), Operating System specific (e.g. Win32 API vs Posix API) and library specific (e.g Std vs Boost vs Qt etc), thus imposing a great deal of constraints to actually achieve WOCA; Therefore we would most likely still require several C or C++ variants for different systems.

Having identified issues with both, WORA where we would distribute a compiled intermediate byte-code for a VM (Virtual Machine), and WOCA where we would distribute source code which could be compiled to machine code, we are naturally led to investigate Source-to-source compilation. Source-to-source compilation allows us to take source code expressed in one language and translate it into a different target language. Regardless of the language of our initial source code, based on the results of our survey (see Table 1, "Survey of XPDL Implementations") we know that we would need to generate code for at least C++ and Java targets.

An examination of the available source-to-source compilers leads us to the Haxe Cross-platform Toolkit which fits our requirements well as it has targets for C++, C#, Java and JavaScript amongst others [41], with targets in development for C and LLVM [42]. Haxe uses a single source language also called Haxe which is similar to ECMAScript but with influences from ActionScript and C#. The Haxe toolkit also provides a cross-target standard library for the Haxe language. With Haxe it seems entirely possible that we can entirely eliminate the fanout issue of implementation by: 1) specifying an Implementation Type Mapping between XDM and the Haxe Language and 2) going further than providing common code for the implementation of an XPDL extension function, instead implement the entire function according to the Implementation Type Mapping in the Haxe language itself. The vendor of an XPDL processor could then take the Haxe code and compile it to the implementation language of their processor to produce a distribution of standardised extension functions; This role could also perhaps also be taken by an intermediary such as the EXPath project.

4.2. Implementation Type Mapping for Haxe

We have developed a partial Implementation Type Mapping between XDM and Haxe (the source code is available from the EXQuery GitHub repository [43]) that provides enough functionality to allow implementation of a single EXPath extension function: the `file:exists` function (as discussed in Section 3.1, "Commonality of EXPath Standardised Extension Functions and Implementation Type Mapping"). In addition to implementing Type Mappings for the XDM, we also need to implement interfaces to map the XPath concept of calling a function and passing arguments.

To produce interfaces for mapping the concept of an XPDL extension function, which is effectively an externally declared function in terms of the XPath specification, we need to understand both how a function is declared and subsequently called. A function call in XPath 3.0 [44]is made up of the several constructs expressed in EBNF (Extended Backus-Naur Form) as reproduced in Example 6, "XPath 3.0 Function Call EBNF".

Example 6. XPath 3.0 Function Call EBNF

```
FunctionCall ::= EQName ArgumentList
ArgumentList ::= "(" (Argument ("," Argument)*)? ")"
```

XPath only specifies how to call a function, it does not specify how to define a function, so here we have opted to follow the XQuery 3.0 specification which does specify how to define a function [45]. A function definition in XQuery 3.0 is made up of the EBNF constructs as reproduced in Example 7, "XQuery 3.0 Function Declaration EBNF".

Example 7. XQuery 3.0 Function Declaration EBNF

```
FunctionDecl        ::=    "function" EQName "(" ParamList? ")"
                           ("as" SequenceType)? (FunctionBody | "external")
ParamList           ::=    Param ("," Param)*
Param               ::=    "$" EQName TypeDeclaration?
FunctionBody        ::=    EnclosedExpr

TypeDeclaration     ::=    "as" SequenceType
SequenceType        ::=    ("empty-sequence" "(" ")") | (ItemType OccurrenceIndicator?)
OccurrenceIndicator ::=    "?" | "*" | "+"

EQName              ::=    QName | URIQualifiedName
```

As our XPDL extension functions will always be external in nature, we can transform the `FunctionDecl` construct, to extract a `FunctionSignature`. As our functions are always external we can also ignore the `FunctionBody` construct as this will instead be implemented in Haxe code. As our extension functions are always the target of a function call, we can reduce `EQName` to `QName`. All of the other constructs can be translated into interfaces for our Implementation Type Mapping to the Haxe language.

Example 8. Function Type Mapping for Haxe

```
package xpdl.extension.xpath;

interface Function {
  public function signature() : FunctionSignature;
  public function eval(
    arguments: Array<Argument>,
    context: Context) : Sequence;
}

class FunctionSignature {
  var name: QName;
  var returnType: SequenceType;
  var paramLists: Array<Array<Param>>;

  public function new(name, returnType, paramLists)
  {
    this.name = name;
    this.returnType = returnType;
    this.paramLists = paramLists;
  }
}
```

Example 8, "Function Type Mapping for Haxe" shows part of our Implementation Type Mapping for functions (full code in Appendix A, *Function Type Mapping in Haxe*). The mapping is direct enough that anyone with a knowledge of the relevant EBNF constructs of XPath and XQuery can understand the simplicity of the function type mapping between an XPDL extension function and Haxe.

Yet, being able to specify the interface for a function is not enough; we also need to create Implementation Type Mappings for the XDM types. Whilst ultimately we need to map all XDM types, within this paper we focus exclusively on the types required for our partial implementation, i.e. those types needed by the function signature of the `file:exists` function (see Example 1, "file:exists function signature").

The signature of `file:exists` shows how we only need to create type mappings for `xs:string` and `xs:boolean` to appropriate Haxe types. We take the approach to encapsulate the Haxe types inside representations of the XDM types, as we believe that this will provide greater flexibility for future changes.

Example 9. Haxe Implementation Type Mapping for `xs:string` and `xs:boolean`

```
package xpdl.xdm;

import xpdl.HaxeTypes.HString;

interface Item {
  public function stringValue() : xpdl.xdm.String;
}

interface AnyType {
}

interface AnyAtomicType extends Item extends AnyType
{
}

class Boolean implements AnyAtomicType {
  var value: Bool;

  public function new(value) {
    this.value = value;
  }

  public function stringValue() {
    return new xpdl.xdm.String(Std.string(value));
  }

  public function haxe() {
    return value;
  }
}

class String implements AnyAtomicType {
  var value: HString;

  public function new(value) {
    this.value = value;
  }

  public function stringValue() {
    return this;
  }

  public function haxe() {
    return value;
  }
}
```

There is certainly an argument concerning whether we should actually implement the `xs:string` and `xs:boolean` XDM types in Haxe by providing classes, or whether we should simply provide interfaces for a vendor to implement. Further research through a survey of vendor requirements would be required to answer this definitively. For the purposes of this paper, classes have been implemented for these basic atomic types.

4.3. Implementation of a portable `file:exists`

Given the function type mapping and Implementation Type Mapping that we have defined in Section 4.2, "Implementation Type Mapping for Haxe" we can now implement our first truly portable XPDL extension function by making use of Haxe.

Example 10. Implementation of the `file:exists` function in Haxe

```
class ExistsFunction implements Function {
  private static var sig = new FunctionSignature(
    new QName(
      "exists",
       "http://expath.org/ns/file",
       "file"),

    new SequenceType(
      Some(new ItemOccurrence(Boolean))),
      [
        [ new Param(new QName("path"),
          new SequenceType(Some(
            new ItemOccurrence(
              xpdl.xdm.Item.String))))
        ]
      ]
  );

  public function new() {}

  public function signature() {
    return sig;
  }

  public function eval(
    arguments : Array<Argument>,
    context: Context)
  {
    var path = arguments[0].getArgument().
               iterator().next().
               stringValue().haxe();
    var exists = FileSystem.exists(path);
    return new ArraySequence( [
                            new Boolean(exists) ] );
  }
}
```

Example 10, "Implementation of the `file:exists` function in Haxe" shows the main concern of our implementation of file:exists (full code in Appendix B, *`file:exists` implementation in Haxe*).

4.4. XPDL Processor Vendor Implementation

We have defined both an Implementation Type Mapping for XDM and associated interfaces for functions in the Haxe language, and subsequently created an implementation of an XPDL Extension Function, the EXPath File Module's `file:exists` function in Haxe written for the type mapping and interfaces. However, such an XPDL extension function implementation is still not useful without vendor support, as the Haxe code must be compiled to the XPDL processors implementation language and made available to the XPDL from the processor.

As a proof-of-concept we have compiled the Haxe code to both Java source and byte code using the Haxe compiler and modified eXist-db to support XPDL Extension Functions (the source code is available from the eXist GitHub repository [46]). Modifying eXist to recognise any XPDL Extension Function Module and make its functions available as extension functions in XQuery was achieved in approximately 300 lines of Java code; For the partial implementation, only support for the XDM types `xs:string` and `xs:boolean` was required, but we recognise that the amount of code required will increase as further XDM types are mapped.

Whilst modifying eXist to support XPDL Extension Function Modules, we recognised that there were several different approaches that could be taken to implement a mapping between eXists own XDM model and our Haxe XDM model. These approaches, whilst not exhaustive, will most likely also apply to other XPDL processors, and so we briefly enumerate them here for reference:

1. Mapping of Haxe XDM types to eXist XDM types and vice-versa. This could be achieved either statically or dynamically, or through a combination of both approaches. A static implementation would be coded in source, whereas a dynamic mapping would be generated as needed at runtime.
2. Modify eXists XDM classes to implement the Haxe XDM interfaces. This would allow us a single XDM model and we could transparently pass eXists XDM types into the Haxe compiled functions.
3. Inversion of Responsibility, using byte-code generation at runtime to have the Haxe XDM interfaces implement the eXist XDM interfaces. This would make the Haxe XDM model compatible with the eXist XDM model, so that Haxe XDM types could be used transparently by eXist.

For expediency in creating the proof-of-concept modifications in eXist, we used a static mapping of XDM types in combination with a dynamic mapping of functions. For the dynamic mapping of functions we used byte-code generation to generate classes at runtime to bridge between eXist's concept of an extension function and our Haxe XPDL extension function.

5. Summary and Conclusion

Having explicitly laid out the issues with portability of XPDLs in regard to non-standard extension functions (see Section 1, “Introduction”), we have reviewed both the past and current works on improving the status-quo (see Section 2, “Prior Art”), and performed a critical analysis of these approaches (see Section 3, “Analysis”). From our critical analysis we have identified three common approaches to improving portability: Function Standardisation, Function Distributions and Implementation Type Mappings. To resolve the issue of portability with respect to extension functions for XPDL users, we argue that there have to be solutions in place for all three approaches and that these must work together holistically.

Function Standardisation is already well supported by the EXPath project, a community oriented organisation which is vendor agnostic and has already proven itself capable of coordinating stakeholders to define modules of common XPDL extension functions and their behaviour.

Function Distributions require implementations of extension functions which they can then make available. These extension functions themselves however need to be portable, so that the resultant XPDL code that uses them is also portable. Arguably the FunctX distribution was successful because its extension functions were portable, as they were written in XSLT and XQuery, making them useable on any vendors XQuery or XSLT processor. For more complex extension functions which cannot be expressed in an XPDL, a portable Implementation Type Mapping is a required enabler to creating Function Distributions.

We have presented a solution for a portable Implementation Type Mapping through the use of source-to-source compilation (section Section 4.2, “Implementation Type Mapping for Haxe”), and implemented what we believe to be the first truly portal extension function for an XPDL whilst using a non-XPDL to implement the function (section Section 4.3, “Implementation of a portable `file:exists`”). Further, we have created a proof-of-concept by integrated support for the Implementation Type Mapping into a real-world XPDL processor (section Section 4.4, “XPDL Processor Vendor Implementation”).

The use of Haxe for source-to-source compilation is an interesting and novel approach towards solving the issue of portable extension functions for XPDLs. Whilst it does not eliminate the need of some effort by XPDL processor vendors to support it, it greatly reduces the work to a one-off exercise to support a portable Implementation Type Mapping. In this manner an XPDL extension function written once in Haxe, when compiled will work on any XPDL processor (in a target language supported by Haxe) which implements the Implementation Type Mapping. For authors of portable XPDL extension functions, rather than just creating a standardisation of a function module through the EXPath project and waiting for each vendor to implement this, they can now also write a single implementation which can be adopted quickly by the widest possible audience.

5.1. Future Work

The Implementation Type Mapping and the proof-of-concept currently only implement the basic XDM types required for this paper, a full XDM Implementation Type Mapping in Haxe is desirable and would likely provide new insights into creating a portable Implementation Type Mapping.

The target code generated by the Haxe compiler can be somewhat verbose and even confusing to the consuming developer. It is possible to tune the code generation by tightly controlling DCE (Dead Code Elimination) and native vs reflective generation. The use of various Haxe language annotations should be investigated to achieve the generation of cleaner target code.

The options for implementation approach discussed in Section 4.4, “XPDL Processor Vendor Implementation” are likely coupled to the observation at the end of Section 4.3, “Implementation of a portable `file:exists`” over how concrete the XDM Implementation Type Mapping should be. Further research is required in this area, likely informed by creating more proof-of-concept integrations with several other XPDL processors.

Whilst Haxe does not favour Java as a target above any other, a non-Java proof-of-concept would reinforce our argument that Haxe allows us to create a portable implementation. A C++ integration for the Zorba XQuery processor could perhaps serve as a suitable reinforcement.

A. Function Type Mapping in Haxe

```
package xpdl.extension.xpath;

interface Function {
  public function signature() : FunctionSignature;
  public function eval(arguments: Array<Argument>, context: Context) : Sequence;
}

interface Context {
}

class FunctionSignature {
  var name: QName;
  var returnType: SequenceType;
  var paramLists: Array<Array<Param>>;

  public function new(name, returnType, paramLists) {
    this.name = name;
    this.returnType = returnType;
    this.paramLists = paramLists;
  }
}

class QName {
  public static var NULL_NS_URI = "";
  public static var DEFAULT_NS_PREFIX = "";

  var localPart : String;
  var namespaceUri(default, null) : String;
  var prefix(default, null) : String;

  public function new(localPart, ?namespaceUri, ?prefix) {
    this.localPart = localPart;
    this.namespaceUri = (namespaceUri == null) ? NULL_NS_URI : namespaceUri;
    this.prefix = (prefix == null) ? DEFAULT_NS_PREFIX : prefix;
  }
}

class SequenceType {
  var type: Option<ItemOccurrence>; //None indicates empty-sequence()

  public function new(type) {
    this.type = type;
  }
}

class ItemOccurrence {
  var itemType: Class<Item>;
  var occurrenceIndicator: OccurrenceIndicator;

  public function new(itemType, ?occurenceIndicator) {
    this.itemType = itemType;
    this.occurrenceIndicator = (occurrenceIndicator == null) ?
                               OccurrenceIndicator.ONE : occurrenceIndicator;
  }
}

enum OccurrenceIndicator {
  ZERO_OR_ONE;    // ?
  ONE;            // implementation detail
  ONE_OR_MORE;    // +
  ZERO_OR_MORE;   // *
}

class Param {
  var name: QName;
  var type: SequenceType;

  public function new(name, type) {
    this.name = name;
    this.type = type;
  }
}

interface Argument {
  public function getArgument() : Sequence;
}

interface Module {
  public function name() : String;
  public function description() : String;
  public function functions() : List<Class<Function>>;
}
```

B. `file:exists` implementation in Haxe

```
package example.expath.file;

import xpdl.extension.Module;
import xpdl.extension.xpath.*;
import xpdl.extension.xpath.SequenceType.ItemOccurrence;
import xpdl.xdm.Sequence;
import xpdl.xdm.Item.Item;
import xpdl.xdm.Item.Boolean;
import sys.FileSystem;

class ExistsFunction implements Function {

  private static var sig = new FunctionSignature(
    new QName("exists", FileModule.NAMESPACE, FileModule.PREFIX),
    new SequenceType(Some(new ItemOccurrence(Boolean))),
    [
      [ new Param(new QName("path"), new SequenceType(Some(new ItemOccurrence(xpdl.xdm.Item.String)))) ]
    ]
  );

  public function new() {}

  public function signature() {
    return sig;
  }

  public function eval(arguments : Array<Argument>, context: Context) {
    var path = arguments[0].getArgument().iterator().next().stringValue().haxe();
    var exists = FileSystem.exists(path);
    return new ArraySequence( [ new Boolean(exists) ] );
  }
}
```

```
class ArraySequence implements Sequence {
  var items: Array<Item>;

  public function new(items: Array<Item>) {
    this.items = items;
  }

  public function iterator() {
    return new ArraySequenceIterator(items.iterator());
  }
}

class ArraySequenceIterator implements xpdl.support.Iterator<Item> {
  var it: Iterator<Item>;

  public function new(it) {
    this.it = it;
  }

  public function hasNext() {
    return it.hasNext();
  }

  public function next() {
    return it.next();
  }
}

class FileModule implements Module {
  @final public static var NAMESPACE = "http://expath.org/ns/file";
  @final public static var PREFIX = "file";

  public function name() {
    return "FileModule.hx";
  }

  public function description() {
    return "Haxe implementation of the EXPath File Module";
  }

  public function functions() : List<Class<Function>> {
    var lst = new List<Class<Function>>();
    lst.add(ExistsFunction);
    return lst;
  }
}
```

Bibliography

[1] *XPath and XQuery Functions and Operators 3.0*. W3C. 8 April 2014.
http://www.w3.org/TR/xpath-functions-30/

[2] *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)*. W3C. 14 December 2010.
http://www.w3.org/TR/xpath-functions/

[3] *Unifying XSLT Extensions*. xml.com. Leigh Dodds. 29 March 2000.
http://www.xml.com/pub/a/2000/03/29/deviant/index.html

[4] *RESTful XQuery*. Standardised XQuery 3.0 Annotations for REST. XML Prague. . XML Prague. Adam Retter. 12 February 2012.
http://archive.xmlprague.cz/2012/files/xmlprague-2012-proceedings.pdf

[5] *TIOBE Programming Community Index*. TIOBE Software. May 2015.
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[6] *PYPL PopularitY of Programming Language Index*. Pierre Carbonnelle. May 2015.
http://pypl.github.io/PYPL.html

[7] *Redmonk Programming Language Ratings*. RedMonk. January 2015.
https://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/

[8] *The Total Growth of Open Source*. Amit Deshpande and Dirk Riehle. SAP Research, SAP Labs LLC. The Fourth Conference on Open Source Systems (OSS 2008). . Springer Verlag. 197-209. 2008.

[9] *The EXSLT Project*.
http://www.exslt.org

[10] *XML Path Language (XPath) Version 1.0*. W3C. 16 November 1999.
http://www.w3.org/TR/xpath/

[11] *XSL Transformations (XSLT) Version 2.0*. W3C. 23 January 2007.
http://www.w3.org/TR/xslt20/

[12] *XSL Transformations (XSLT) Version 1.1*. W3C. 24 August 2001.
http://www.w3.org/TR/xslt11/

[13] *XSLT Extensions Revisited*. xml.com. Leigh Dodds. 14 February 2001.
http://www.xml.com/pub/a/2001/02/14/deviant.html

[14] *Re: [xsl] XSLT 1.1 comments*. W3C xsl-editors Mailing List. Michael Kay. 11 February 2001.
https://lists.w3.org/Archives/Public/xsl-editors/2001JanMar/0087.html

[15] *Re: [xsl] XSLT 1.1 comments*. xsl-list Mailing List. Steve Muench. 12 February 2001.
http://markmail.org/message/5fpk5gecmslzepdy

[16] *Petition to withdraw xsl:script from XSLT 1.1*. xml-dev Mailing List. Uche Ogbuji. 1 March 2001.
http://markmail.org/thread/tquj4ozsax3pjkm2

[17] *Minutes of the Face-to-face meeting of the W3C XQuery Working Group in Bangkok*. W3C XQuery Working Group. January 2001.
https://lists.w3.org/Archives/Member/w3c-xsl-wg/2001Feb/0083.html

[18] *FunctX*. Datypic. Priscilla Walmsley. July 2006.
http://www.functx.com

[19] *EXQuery*. Collaboratively Defining Open Standards for Portable XQuery Applications. EXQuery. October 2008.
http://www.exquery.org

[20] *EXQuery Common Implementation Source Code*. The EXQuery Project.
https://github.com/exquery/exquery

[21] *EXPath*. Collaboratively Defining Open Standards for Portable XPath Extensions. EXPath. January 2009.
http://www.expath.org

[22] *EXPath HTTP Client Module Common Implementation Source Code*. Florent Georges.
https://github.com/fgeorges/expath-http-client-java

[23] *EXPath File Module Common Implementation Source Code*. The EXQuery Project. Adam Retter.
https://github.com/exquery/exquery/tree/master/expath-file-module

[24] *EXPath File Module Common Implementation Source Code*. Florent Georges.
https://github.com/fgeorges/expath-file-java

[25] *Implementations of EXPath Modules*. W3C EXPath Community Group. 8 May 2015.
https://www.w3.org/community/expath/wiki/Modules#Implementation

[26] *XQuery and XPath Data Model 3.0*. W3C. 8 April 2014.
http://www.w3.org/TR/xpath-datamodel-30/

[27] *API for XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)*. The EXQuery Project. Adam Retter. 12 February 2015.
https://github.com/exquery/exquery/tree/xdm-model/xdm

[28] *XML Model for Java*. The EXPath Project. Florent Georges. 6 January 2015.
https://github.com/expath/tools-java

[29] *Minutes of Face-to-face meeting of the W3C EXPath Community Group in Prague*. W3C EXPath Community Group. 12 February 2015.
https://lists.w3.org/Archives/Public/public-expath/2015Feb/0005.html

[30] *File Module 1.0*. W3C EXPath Community Group. 20 February 2015.
http://expath.org/spec/file

[31] *BaseX 8.1.1 implementation of EXPath file:exists function*. BaseX. 9 January 2015.
https://github.com/BaseXdb/basex/blob/8.1.1/basex-core/src/main/java/org/basex/query/func/file/FileExists.java

[32] *eXist implementation of EXPath file:exists function*. Adam Retter. 21 February 2015.
https://github.com/adamretter/exist-expath-file-module/blob/master/src/main/scala/org/exist/expath/module/file/FileModule.scala

[33] *Saxon implementation of EXPath file:exists function*. Florent Georges. 16 January 2015.
https://github.com/fgeorges/expath-file-java/blob/master/file-saxon/src/org/expath/file/saxon/props/Exists.java

[34] *XQilla implementation of XPath fn:date-from-year function*. XQilla. 16 November 2011.
http://xqilla.hg.sourceforge.net/hgweb/xqilla/xqilla/file/6468e5681607/include/xqilla/functions/FunctionYearFromDate.hpp

[35] *W3C XML Query*. Implementations. W3C XQuery Working Group.
http://www.w3.org/XML/Query/#implementation

[36] *W3C EXPath Community Group Wiki*. XPath Engines. W3C EXPath Community Group. 8 May 2015.
https://www.w3.org/community/expath/wiki/Engine

[37] *Using XSLT with Go*. William Kennedy. 3 November 2013.
http://www.goinggo.net/2013/11/using-xslt-with-go.html

[38] *lxml*. XML Toolkit for Python.
http://lxml.de/

[39] *Write once, run anywhere*. Wikipedia, The Free Encyclopedia.
https://en.wikipedia.org/wiki/Write_once,_run_anywhere

[40] *SWIG*. Simplified Wrapper and Interface Generator.
http://www.swig.org/

[41] *Haxe Compiler Targets*.
http://haxe.org/documentation/introduction/compiler-targets.html

[42] *Experimental C and LLVM Targets for Haxe*.
https://github.com/waneck/haxe-genc

[43] *Source code for Implementation Type Mapping of XPDL Extension Functions in Haxe*. The EXQuery Project. Adam Retter. 12 May 2015.
https://github.com/exquery/xpdl-extension-lib

[44] *XML Path Language (XPath) 3.0*. Static Function Calls. W3C. 8 April 2014.
http://www.w3.org/TR/xpath-30/#id-function-call

[45] *XQuery 3.0: An XML Query Language*. Function Declaration. W3C. 8 April 2014.
http://www.w3.org/TR/xquery-30/#FunctionDecln

[46] *Source code of XPDL Extension Functions integration with eXist*. Adam Retter. 12 May 2015.
https://github.com/eXist-db/exist/tree/xpdl-extensions/src/org/exist/xpdl