Bridging XDM types in multiple native type systems

O'Neil Delpratt
Saxonica
`<oneil@saxonica.com>`

Matt Patterson
Saxonica
`<matt@saxonica.com>`

Abstract

We explore the relationship of XDM types and the native types in the host language of an XML processing system. In multi-tier language systems such as SaxonC we find that there is not always that one size fits all approach to representing data of a native type to what we require in the XDM type system. Secondly, we look at the complexities of handling numbers and strings; it seems simple to represent them across languages and within the XDM system, but this we found can get cumbersome and complicated. Thirdly, we consider the more complex XDM Map, and how issues of implicit and explicit type conversion meet issues of XDM and local idiom. Lastly, we dive into the use case of handling XDM Node objects such as traversing, cross language memory management (i.e. from Java to C++ and vice versa). On top of that we discuss how we add further complexity in layering C++ extension to support APIs in Python and PHP which operate in a managed code environment again.

Keywords: XML, XSLT, XQuery, XPath, C++, Python, PHP, GraalVM, XDM

1. Introduction

The XML processing languages XPath, XSLT and XQuery have in common and at their core the XQuery and XPath Data Model (XDM). Over the years, the XDM has been through several iterations, with increased complexity along the way. Firstly, we had the XPath data model 1.0, which primarily focused on the tree structure of the XML document. The data model was composed of seven types of nodes (root, element, text and attribute). Strings were determined from the type of the node. At the same time we had XSLT 1.0, which was based on this data model with additional features.

Secondly, we have the XQuery 1.0 and XPath 2.0 Data Model given as a specification. Here the 1.0 and the XSLT 1.0 data models were combined with additional data types to support more than just trees. These come from the XML schema simple types and what we call atomic values, primitive types such as `xs:string`, `xs:boolean`, `xs:integer`, and `xs:date`. We also have the notion of a sequence which is a collection of zero or more items. These items can be nodes, atomic values or a combination of both.

Thirdly, we have the XQuery and XPath Data Model 3.1 which is based on the 2.0 specification. What is new in this specification is the addition of array and map types, which are derived from the function type, which is also derived from an item. At the time of writing, the XQuery and XPath Data Model version 4.0 specification is currently at draft status.

In this paper, we look at the challenges of providing multiple language APIs to Saxon, in terms of both type system and API design constraints, and some of the lower-level concerns necessitated by the ways GraalVM manages bridging of managed Java objects with the unmanaged world of C/C++ and beyond.

As a background, it is important to mention here the building blocks of SaxonC. We use GraalVM, a JVM implementation that provides the ability to compile Java down to native code ahead-of-time, and allows that code to be distributed as a shared library. Effectively a tiny VM implementation runs in a thread in the library and allows your native code to call into it. We create a native library using GraalVM's native-image technology, which we call SaxonC. This is compiled from Saxon-J (a pure java implementation) so that we can make calls from other languages such as C/C++ to the Java code which is now native. The binary library comes packed with all the compiled JDK libraries, and a VM, required to run the Saxon-J image for execution. Creating a native image has the benefits of having faster startup time, low resource usage, flexibility in extending the application with other languages, and being easier to package.

2. Exploring the gap between XPath and native types

APIs that permit working directly with XDM data – that is, objects within the XDM type system – outside an XQuery/XPath processor require bridging XDM and native types. XDM has a different approach to data at a fundamental level from most of the languages that Saxon runs on, the languages you will use to bridge into XDM. This makes things hard. Apple's AppleScript programming language was designed to be easy to understand for non programmers, and leant heavily into an english-like syntax. One consequence of this was that although it was easy to read, in many ways that made it very hard to write. Author Matt Neuberg called this the 'English-likeness monster' [4]. XDM's pervasive use of Sequences is one of the main points of disconnect between the XDM type system and most native type systems. I call this the 'Sequence-likeness monster'.

2.1. The Sequence-likeness monster

The introduction of Sequences with the introduction of the XPath Data Model and XPath 2.0 is extremely important. Extending the XPath 1.0 Nodeset concept to things that weren't XML Nodes gives XPath much of its power, but also introduces the biggest gap in native and XPath type systems. Let's look quickly at a diagram showing XDM Values, Items, and Atomic Values:

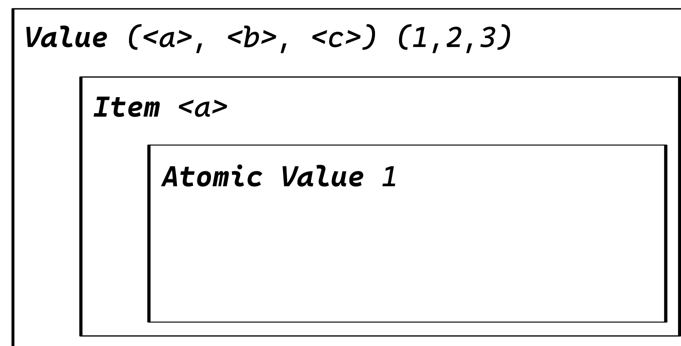

Figure 1. The XDM supertypes

A Sequence is a **Value**, an XML Node is an **Item**, and a primitive type, like a String, is an **Atomic Value**. But, Atomic Values are also Items, which are, in turn, also Values. This is a kind of inversion of more traditional object and type hierarchies, where a primitive type may be a kind of **Object**, but is certainly not a kind of meta-collection type (like Value):

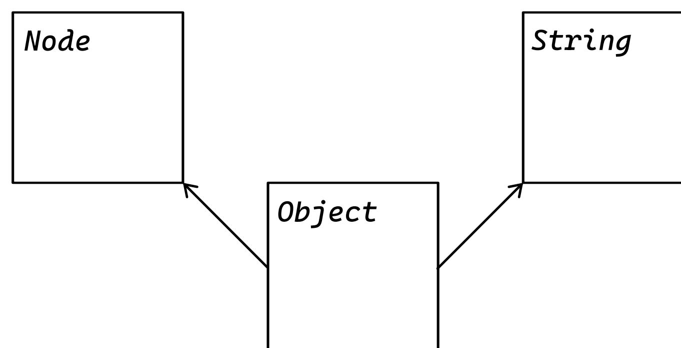

Figure 2. A 'Traditional' Object-based type system

Because of this type hierarchy all primitive types are also a kind of Value – a Sequence – even if they can only ever contain 1 item (themselves). Thought about in terms of the type system of a dynamic language like Python, all Integers in XDM are also a kind of List-ish thing containing a single member. In fact, everything in XDM is also a kind of List-ish thing, in addition to whatever else it is. In

some ways Sequences are the simplest XDM type, since, unlike a String, they don't have a concrete value, they're just a collection.

For obviously alien objects like XML Nodes, there is, perhaps, less dissonance when dealing with these objects in a non-XPath language. They are obviously unlike a String, or other native primitive type, and so it's more obvious that interacting with, and manipulating, them is unlike working with a core native type.

Dealing with creating Atomic Values (such as parameters required by an XSLT), or processing Atomic Values returned by an XPath evaluation, XQuery, or XSLT invocation is much more slippery. Unless you can guarantee what return type, perhaps from the type signature of an XPath function or XSLT template, you have to figure it out at runtime. More critically for this paper, the implementer of the native bindings for the XDM processor, have to figure it out as well, and then they need to provide meaningful mappings between XDM Atomic Values and native types.

XPath 1.0 had a significantly simpler data model, but even then this was a complex issue. JAXP's XPath evaluation methods return String or Object, and the user needs to know in advance what kind of thing their XPath will return (a node, a sequence, a primitive value) and pass it to the evaluation method as an additional argument.

Purely from the point of view of the user, XPath 3's richer data model, and increased dynamism (higher order functions and `fn:transform`, for instance) make it hard for implementers to expect users to know the return types of XPath expressions they might need to execute in advance, and, from the implementer's point of view, those same issues make it impossible to avoid dealing with the complexity of dynamic return types, whether the host language is dynamically or statically typed.

2.2. Dynamic and Static typing

XPath is a dynamically typed language. It's possible to specify strong requirements for parameter and return types, but it's not required in all situations. Many host languages, such as Java or C++, are statically typed. A situation where code in the host language is used to invoke an XSLT that is specified at runtime, and whose parameters (and their types) are therefore also not known until runtime is not an unusual one. Neither is a situation where an XSLT's return value differs in type based on the content it's processing unusual.

With a dynamically typed language there's an expectation of, and language support for, returning multiple possible types from any given function or method. A user is likely to be comfortable with the idea that they'll have to determine exactly what they got back at runtime, and process accordingly.

With a statically typed language, that's much more difficult. Without knowing the type in advance, a user must use a native type that represents an XDM super-

type like Value, and an implementer, like Saxon, can only return more specific types if they add extra type-specific invocation methods.

2.2.1. Creating XDM objects

The primary need for users of an XML processing system to create XDM objects is to hand them to the processor for processing, usually as parameters, as when invoking an XSLT transformation. Of course, parsing an XML document itself into an XDM Node is also an example of this, but we're concerned here primarily with XDM objects other than document trees.

There are two ways to create an XDM object from a native object. The XDM type can be inferred from the native type, implicit conversion. At it's simplest, this could mean converting a native String to an Atomic Value of type `xs:string`. A complex example is converting a JavaScript object to an XDM Map. Here's an example using SaxonJ's implicit conversion:

```
import net.sf.s9api.XdmAtomicValue;
new XdmAtomicValue("string"); // => An xs:string
int v = 1; new XdmAtomicValue(v); // An xs:int
double v = 42.0; new XdmAtomicValue(v); // An xs:double
new XdmAtomicValue(LocalDate.now()); // An xs:date
```

And using Saxon JS:

```
SaxonJS.atom(42.0); // An xs:double
SaxonJS.atom(true); // An xs:boolean
SaxonJS.atom("string");  // An xs:string
```

The other approach is explicit conversion, where the XDM type is specified along with the native value to be converted or a function that generates a specific type is called. Here's an example using SaxonC:

```
#include "XdmAtomicValue.h"
double d = 42;
XdmAtomicValue *value = XdmAtomicValue::makeDoubleValue(d);
```

And using SaxonC Python:

```
from saxonche import PySaxonProcessor
proc = PySaxonProcessor()
value = proc.make_double_value(42.0) # as xs:double
value = proc.make_atomic_value("date", "2024-06-08") # an xs:date
```

The last example above illustrates a variant of explicit conversion whereby the type is specified and the value is given using its XML Schema lexical string representation (like XML element or attribute content that was typed using an XML Schema).

2.2.2. Converting XDM object into native objects

Converting XDM objects into native objects is where the differences between static and dynamic typing are more apparent. Given an XPath evaluation that returns an XDM Value that we know will return a single Item, but the exact return type of the Item is not known, what can be done to convert that into a native object of an appropriate type?

With a statically typed language, about the best we can do is to provide methods on the various Value and Item objects / subclasses that report what kind of thing they are, along with methods to produce a specific native type, like a `long` or `boolean`. Then, we can use those methods in combination with branching logic to get values with correct native types out:

```
SaxonProcessor *processor = new SaxonProcessor(false);
XPathProcessor *xpathProc = processor->newXPathProcessor();
xpathProc->setLanguageVersion("3.1");
xpathProc->setContextItem((XdmValue*) input);
XdmItem *result = xPathProc->evaluateSingle(".[last()]");
if (result.isAtomic()) {
    switch(result.getPrimitiveTypeName()) {
        case "Q{http://www.w3.org/2001/XMLSchema}int":
            long num = result.getLongValue();
    }
}
```

This is complex and verbose. With a dynamically typed language there's more scope for API convenience for the user. Saxon JS is designed for very close integration in a JavaScript runtime, so calls to `SaxonJS.XPath.evaluate` and `SaxonJS.transform` will return results where the XDM objects are converted into JavaScript values, following the rules set out in its documentation (see [1]). These rules are complex, and neatly illustrate the trade off between convenience for the user and complexity for the implementer.

```
Object.is(SaxonJS.XPath.evaluate("true()"), true) // => true
SaxonJS.XPath.evaluate("map { 'a': 1 }") // => {a: 1}
SaxonJS.XPath.evaluate("map { 'a': 1 }")['a'] // => 1
```

2.3. Numbers

We've talked about the issues of moving between native types in a Saxon API host language, and native XDM types in terms of differences in the type systems, but the other issue is simply the different ways most programming languages and XDM think about how their primitive types work, and what shape they are. It's worth looking in more depth at the conversion of numeric types. Numeric types are (perhaps surprisingly) complex to deal with. XDM's numeric types owe a lot to C and Java's numeric types. As a result there are a lot of different numeric

types. The following two tables shows XDM's numeric types, and the types for Java, C++, Python, and JavaScript.

Table 1. XDM numeric types

```
xs:float
xs:double
xs:decimal
  xs:integer
     xs:nonPositiveInteger
        xs:negativeInteger
     xs:long
        xs:int
           xs:short
              xs:byte
     xs:nonNegativeInteger
        xs:unsignedLong
           xs:unsignedInt
              xs:unsignedShort
                 xs:unsignedByte
        xs:positiveInteger
```

Table 2. Other languages

Java	C++	Python	JavaScript
NumericType	bool	int	Number
IntegralType	char	float	
byte	signed char	complex	
short	unsigned char	fraction.Fraction	
int	wchar_t	decimal.Decimal	
long	short		
char	int		
FloatingPointType	long		
float	long long		
double	unsigned short		
	unsigned		
	unsigned long		
	unsigned long long		
	float		
	double		
	long double		

The main thing that sticks out is that while there's some overlap between the type systems, the XDM numeric types are more numerous even than the C++ types, if you take into account the four C++ types that are integers representing characters, and comically more numerous than JavaScript's single `Number` type. There's no simple, 100% reliable type conversion possible. Distinctions like `xs:nonNegativeInteger` and `xs:positiveInteger` are effectively impossible to capture in any other type system, and, of course, what could you do if the number you need to pass into a processor is a Python `complex` number?

Converting between numeric types of different bit length is called *widening*, when converting from a smaller to a larger type (`short` to `long`, for example), and *narrowing* when converting from a larger to a smaller type, or from a floating point to an integer type (`long` to `short`, `float` to `long`). Widening doesn't lose information, but narrowing can. Java specifies how these kinds of conversions should work (see [2]), and C++'s *Core Guidelines* have a section on avoiding narrowing conversions (see [3]). The XQuery and XPath Data Model ([5]) and XML Schema Part 2: Datatypes ([6]) specifications don't really talk about these conversions at all. Casting from one type to another numeric type is covered briefly in

([7]), but these are largely implementation dependent questions. (Casting to `xs:integer` is mentioned, but not `xs:short` or `xs:byte`.)

Many conversions would require runtime checking of values to ensure that a given non-XDM numeric object's value fits within the allowed range of, say, an `xs:positiveInteger`, or an `xs:byte` when converting from a Python integer or JavaScript number. If the values are not checked then strange behaviours in the XPath / XQuery code could happen if there are dramatic silent truncations of input values, and implementations should raise exceptions if lossy narrowings occur.

2.4. How long is a (piece of) String?

Let's continue to look at the different ways XDM and other type systems shape their primitive values. Take the example of XDM's `xs:string` Atomic Value. Strings are interesting in this case because they are a data type that has intrinsic length (the number of characters they contain).

In the XDM type system, the following statements are all true:

- The length of the XDM Atomic Value representing the string `"Hello World"` is 11.
- The length of the XDM Atomic Value representing the string `"Hello World"` is 1.
- The length of the XDM Atomic Value representing a string containing 2^10 characters is 1.

On its face, this seems ridiculous, but it shows up one of the ways XDM differs in the way it views the shape of data. In XDM, all Atomic Values are a kind of Item, but all Items are also Values, which are collections – Sequences. An Atomic Value is an XDM Value containing a single Item, a sequence of length 1, in other words. So, the Atomic Value is both a Sequence containing a single item, and a primitive value containing a number of codepoints.

In XPath this isn't a particular problem - to get the length of a string in XPath you use `fn:string-length()`, and `fn:count()` to get the length of a sequence. In host languages which are more traditionally object-oriented, you would expect to query the Atomic Value itself to find out that information.

In Python, for example, the length of a string is simply the number of codepoints it contains. There is no sense that this primitive type could also be thought of as a container collection of itself.

XDM strings are Unicode strings, which is not true in C and C++ (and used to not be true in Python). Wanting to pass an XDM string into a processor and wanting to get a non-unicode string (perhaps just the string's representation as UTF-8 encoded bytes) makes working with XDM string objects trickier and the design of an API for wrapped XDM objects which supports making use of the them as

XDM objects (iteration over sequences, perhaps) and as containers of native or native-like objects (wanting to get substrings or string lengths) needs implementers to consider the idioms of both XDM and the host language carefully.

2.5. How do you get a value from an XDM Map in Python?

Maps in XDM are also Atomic Values, and Maps are widely used in other languages. Wanting to expose an XDM Map in a host language involves making choices about what idioms to use, and which to discard. XDM Maps are immutable, unlike maps in most languages, which introduces API friction when creating an XDM map, especially in languages where creating a map all-at-once with a literal syntax is either not especially idiomatic, or just plain impossible.

When fetching a value, we need a key. What is the key? In the most trivial case, paralleling a JSON Object, the key is an XDM Atomic Value with type `xs:string`. Next, how do we construct that Atomic Value? Can we allow a Python user to call a method on the Map and pass in a Python string? Do we need to require explicit construction of an `xs:string` Atomic Value? If we allow native Python primitives, do we require separate API methods for getting values using Python types versus using XDM Atomic Values?

As we can see, even in this trivial example there are a host of questions which need answering, none of which are themselves trivial.

If we allow Python primitive values to be used in place of XDM Atomic Values in the Python API equivalent of `map:get()`, how do we convert them?

The end goal is to have Saxon construct an Atomic Value – a string – whose contents match the Python string we started with. We need to extract the bytes of the Python string, pass that C-level byte array into the Java internals using the GraalVM C API, and construct an Atomic Value from that Java string.

Once we have an Atomic Value, we can call the `get()` method of the XDM Map with the key as its argument, and hand back the resulting value.

But what is that value? When invoking an XSLT via the API it's very easy to overlook this question: the result of the transformation is written to a file, or perhaps serialized to a string, neither of which require dealing with an XDM Value. When retrieving a value from an XDM Map, the result is an XDM Value. Should this be converted to an equivalent primitive type in Python, so that an `xs:string` value becomes a Python string? What about more exotic XDM Value types? What do you do with an XDM Function Item?

If we consider Pythonic idioms, we would expect the map to respond to the `[]` operator to return values, as with `value = xdm_map[key]`. Of course, considered as a sequence, we would expect the map to respond to the `[]` operator to return items from its 1-item sequence: itself in other words.

```
xdm_map is xdm_map[0] # => true
```

This is a problem for the API designer.

XPath 1.0 had a significantly simpler data model, but even then this was a complex issue. JAXP's XPath evaluation methods return String or Object, and the user needs to know in advance what kind of thing their XPath will return (a node, a sequence, a primitive value) and pass it to the evaluation method as an additional argument. XPath 3's richer data model, and increased dynamism (higher order functions and `fn:transform`, for instance) make it harder to get away with requiring the user to have to know the return types of their expressions in advance.

2.6. Unicode strings

It's also worth looking at the issue of Unicode strings more closely. In Python and XDM we can simply say that a string is a sequence of Unicode codepoints. (This wasn't always true in Python, but with Python 3 it is). How bytes get translated into Unicode codepoints is a matter of encoding. Conceptually at least, passing strings from Python into Saxon should be straightforward.

As we mentioned above, SaxonC is implemented using GraalVM's native-image feature. One of the things that means is that to pass data between our API host language (Python in this case) and the Saxon internals we have to use GraalVM's C API. We are used to thinking of strings purely in terms of Unicode, and the encoding of those strings into bytes is an internal concern of the XML processor, or of Python, but here we are forced to deal with it: C byte arrays have no intrinsic encoding, just bytes, and while you can get lucky most of the time by assuming that the byte arrays you're passing around contain UTF-8-encoded strings, this isn't always the case, and some library functions in Java vary the encoding they use based on OS or other locale settings, and of course Unicode representations in C are heavily dependent on locate settings or environment variables.

What does that mean for the case where Python strings (Unicode) need to be passed across the GraalVM API to construct XDM strings (also Unicode)?

Effectively every C-based call needs to pass the byte array and the name of the encoding used, or the GraalVM Java API functions that reconstruct Java Strings from byte arrays passed in will assume UTF-8, even if it isn't. The main SaxonC API provides XDM wrappers as C++ classes, but even there you need to specify as the GraalVM API is C, not C++.

The intrinsic Unicode-awareness of all strings within Saxon internals (simply as a side-effect of being implemented in Java) means that even strings you might not think twice about, like filenames, that need to be passed through the GraalVM C API must have their encoding specified explicitly to avoid bugs where problems appear, but only in certain circumstances, like on one OS, or with environment variables set to certain values. These can be a nightmare to deal with because they can be so hard to reproduce for debugging purposes. Likewise,

any Saxon APIs provided in another host language, like the Python API, need to make sure they supply encoding with string data when calling across GraalVM into Saxon proper.

3. Traversing XDM Nodes via the API

The navigation of XML trees in XPath is fundamental to most things we do with XML languages like XQuery and XSLT. The non-XML syntax in XPath is both powerful and succinct, but at the same time the XPath expressions for many tasks are simple to use for navigation. For example, traversing forward in XPath can be done as follows: `/doc`, `//person`, `/doc/person[2]/firstname`. Likewise backward traversal can be done as follows: `/expr/../..`, where the expr is a valid XPath expression.

Navigation of XML trees via an API to do the same simple task as an XPath expression is verbose and error prone, but is very powerful and useful because it provides a way to integrate into other multi-tier systems, extensions for programming languages and simply the flexibility to support user requirement where performance is a criteria. Often what we are dealing with is accessing the vendor's XPath implementation API at some lower level. There are a lot of comparisons and relationships that can be drawn with when navigating XML tree between the XPath languages and APIs. See below a C++ code snippet to access the child nodes and get the string represented at that node from a parent node object. Note that in this implementation it is the users requirement to delete the associated memory when finished.

```
int childCountA = node->getChildCount();
XdmNode **childrenA = node->getChildren();
XdmNode *child = childrenA[0];
XdmNode **children = child->axisNodes(EnumXdmAxis::CHILD);
int childCount = child->axisNodeCount();
for (int i = 0; i < childCount; i++) {
    const char *childStr = children[i]->toString();
    cout << "child node:" << (childStr) << endl;
    operator delete((char *)childStr);
}
for (int i = 0; i < childCount; i++) {
    delete children[i];
}
delete[] childrenA;
delete node;
```

There are more elegant ways to traverse XML trees in APIs even in C++, but these features are often restricted or best suited to certain programming languages, such as Python, Java and C#. For example, in Java we have functional interfaces

like the following streams-based APIs (similar techniques we can find in Python APIs and Linq in C#):

```
for (XdmNode pack : testInput.select(
    child("package").where(
        attributeEq("role", "secondary"))).asListOfNodes() {
    ...
}
```

In SaxonC on Python such a query we can write as follows:

```
packs =  (pack for pack in testInput.children
      if (pack.name == 'package' and
         pack.get_attribute_value('role') == 'primary'))
```

This is equivalent to iterating over the result of XPath expression `package[@role='secondary']`, but it saves the cost of compiling the expression, which is often much greater than the execution cost. When writing XPath expressions the focus is the XML data we are querying, providing valid expression and its efficiency, but there is a lot happening under the hood by the implementation without the user needing to care much about how its done. For APIs your focus is not just on the XML data, but also efficiency in coding, memory management of the XML node objects created and how you represent the data as discussed in the previous sections on handling strings. This is in addition to the XPath and host language syntax and the actual data and paths you are navigating itself.

We now look at how we implement support for XML tree navigation in a C++ API. We chose C++ as the programming language here because its the path we have chosen for providing the basis for extension APIs in languages such as PHP and Python. The C++ API also sits on top of a Java API compiled to native using GraalVM. A traversal of an XDM Node as in a SaxonC application is in the context of XPath, XQuery and XSLT processing.

For something as simple as traversing node objects it becomes difficult to support when implemented in multi-tier languages. Multi-tier languages means interfacing with an application written in one programming language and its environment from another language. There is often further interfacing of languages. For example, in SaxonC, the porting of the application written in Java to C++ is achieved using GraalVM. PHP and Python APIs are built on top of the C++ API. We have some important questions that need answering: how do we make callbacks between the C++ and Java environments? and how do we hold on to objects in C++ that have been created in the Java space? Also, how could we prevent Java's garbage collection getting in the way when a Java object is still in use in C++?

The first question we will discuss with an example of getting a node and its child nodes. The second and third question we will discuss later as we explore the GraalVM's APIs, specifically the `ObjectHandle` and `ObjectHandles`.

3.1. Get a node, get its children

In the SaxonC C++ API, `XdmNode` objects are created by either parsing from some XML string, file or as a result of firing an XPath, XQuery or XSLT execution. As mentioned above we are calling back to the Saxon Java internals to create an in-memory representation of the XML document, which is a Java `XdmNode` object. This Java object is not directly returned to C++. Instead we use GraalVM's `ObjectHandle` API to return to C++. We will come back to talk about `ObjectHandle`, but for now, keeping things simple, we have a reference in C++, which allows us to get to the underlying Java object when we need to do some processing in the Java world. This we use to make callbacks to get child nodes in the following ways:

- `getChild(ith)` - Gets the ith child node from a current node
- `getChildren()` - Get all the child nodes from the current parent node
- `axisNodes(axis)` - Get the array of nodes reachable from this node via a given an XPath axis.

In the C++ code, for each case above we call back to Java, for example, to get the child node `getChild(ith)` we get the Java `XdmNode` child object from its parent node. Then we return a representation of that object to the C++ code (i.e. an `ObjectHandle` object from GraalVM). We have wrapper classes in C++ containing the `ObjectHandle` reference to the Java `XdmNode` object. At this stage, note that it does not matter how we represent the Java object in the C++ code, just realise that it can be complex. There are many other techniques for passing objects between languages. Some more efficient than others. For example, the Java native interface (JNI) is a common option to access the C API, but it can be slow and error prone. Another method is via native COM components, also web services via HTTP requests and other methods which are more complicated. If languages involved are in the same runtime (i.e. .NET, JVM), then it is possible to pass objects from one language to the other. For sure, implementers who are interfacing with an XPath implementations in multi-tier languages have to make the choice between these different methods.

3.2. GraalVM's ObjectHandle and ObjectHandles pool

GraalVM compiles a Java application into a native application that has its own executable. It also has the capability to create a native library which can be called from other programming languages, for example, directly using C/C++, or using the Truffle API for other programming languages such as Python, etc. For C/C++, GraalVM supports callbacks into the Java code using two different interface mechanisms. The first one is the Java native interface (JNI), which is a standard JDK API and secondly, directly through GraalVM's Native Image C API [9]. For

the latter, GraalVM exposes Java methods by marking the method with EntryPoint annotations which GraalVM interprets and creates C like interfaces to the methods. These methods are now available as export methods from the native library which can be used by C/C++ code. We make callbacks on the wrapped Java objects in the C/C++ API layer using the entry points, which we consider as being more efficient than using JNI.

We can easily make callback to the Java object from C++ and work with primitive Java/C++ types. But to manage Java objects from C++ is not so easy and preventing objects getting garbage collected (GC) by the Java JVM ahead of time before the object use is finished in the unmanaged C++ code is an issue. Java objects are held as references in an intermediate C++ interface. To achieve this we use GraalVM's ObjectHandle API. A ObjectHandle is a an opaque representation of a handle to a Java object which is given out to unmanaged code (i.e. C/C++). We also use GraalVM's ObjectHandles API as a pool of ObjectHandle, which is a managed set of ObjectHandles to keep alive the Java objects in memory and prevent them from being garbage collected.

Given our node traversal example: We get a child node in Java, create an ObjectHandle for it, add it to the ObjectHandles pool, and then return the ObjectHandle reference to C++ which will then get wrapped in a C++ XdmNode object. Another layer of wrapping will take place if we are in a Python or PHP application. This child node object will stay alive until we remove it from via ObjectHandles set.

3.3. Telling GraalVM you are not using an object

When working with XDM node objects in processors such as XSLT, XQuery, XPath and Schema Validator from C++/Python/PHP languages it is not always obvious when the object is no longer needed. This is because references can be held as external variables or structures, internally in the processors or a combination of both. As mentioned before the ObjectHandles API prevents the GC deleting the object.

We must make sure that deallocating the C++ XdmNode object which wraps an ObjectHandle cascades down to remove the associated Java object held in the ObjectHandles pool, else we have memory leaks with objects that never get deleted. Removing an object from the ObjectHandles pool will make the object available for GC. Languages such as Python and PHP have their own GC. These also need to inform the internal program when objects are to be freed.

3.4. How do you know when you're not using an object?

Processors hold and then release XDM objects that were used in execution, for example, used as a parameter or context item. In the C++ environment it is mostly

the users responsibility to deallocate the memory associated with objects they no longer need. To solves this we add internal hooks (i.e. reference counting) on objects if they are still indirectly required by processor. We then hand over the deallocation responsibility to the internals of the processor object.

In the PHP and Python world it is somewhat more complicated because we are in a managed code environment which has its own GC. Here we have to add our own reference counting techniques to ensure that PHP or Python's GC doesn't cause GraalVM's GC to either be called too early, or not to be called at all. Likewise, if for example an XDM node object is no longer used in the Python script the reference count for that object at that point should be zero. Therefore we can safely delete that object. At the same time we expect that child nodes which have been created exist independent of the parent node and therefore are relinquished to the Python GC, which will have its own reference counting checks.

4. Memory Management

Managed code, in simple terms, is code that is managed at execution time. In Java's JVM the code is translated into an intermediate language which is interpreted and executed. The entire memory management is taken care of by the runtime using the garbage collection feature.

Unmanaged code is when the code is compiled to native code or machine code and executed by the operating system. The whole memory management of the program is unmanaged; therefore it is the user's responsibility to handle the memory allocated by the program throughout its span, and delete it when no longer needed and when the program is terminated. Correct memory management is important to avoid errors and memory access violations such as segmentation faults.

The management of memory in C++ is fundamental. There are two aspects we would like to talk about here: Firstly, how we keep alive objects such as XDM Node objects in main memory. And secondly, how we create and return strings from the internal Java code which needs to be returned in C++. For example, serialising nodes, returning the XSLT transform as a string and executing an query to string.

In SaxonC, for XDM node objects we have our own memory management where we keep a reference counter in the objects for where the object is used. We use this to prevent the object from being deleted too early. This is particularly important in extension languages like Python and PHP which have their own GC. If the XDM object is internally referenced by processors and variables then we keep alive these XDM node objects using the reference counter, until they are no longer referenced.

For the textual data that we create in the Java code and which needs to be returned to the C++ API we create this data directly in the C++ memory space. In essence, we are allocating the C++ memory space from managed code (i.e. Java). This we found to be the more efficient because we are not having to make unnecessary copies of the data. The pointer to the string is returned to the user with the responsibility to deallocate the associated memory when finished with the data.

5. Conclusion

Providing meaningful API access to XML processing with Saxon requires that API users can construct and make use of XDM Values. In host languages other than Java or JavaScript, this means directly or indirectly using SaxonC, with its GraalVM-C-API-based bridge between Java and C/C++.

The XDM view of the world is very different to the way that most host languages structure data. XDM wrappers that bridge between the worlds need to be a first-class part of any API, otherwise it's extremely difficult, or impossible, to do anything other than simply invoking a transform or query. Passing parameters or arguments in to a transform or XPath evaluation as lexical XPath strings, as in the XPath 1.0 days, is no longer viable.

Providing first-class XDM wrappers to API clients in other languages for Saxon through SaxonC requires wrangling the expectations of managed code in the core Java runtime, and the managed code in the host language runtime through the unmanaged layer of GraalVM's C API and SaxonC's own C++ classes. Correctly managed, this allows, for example, a Python user to make use of XSLT 3.0, and XPath/XQuery 3.1 in a way that feels idiomatically Pythonic, and more generally expands the base of developers who can make use of modern XML technologies. We still have a lot of room to improve, and we hope that this survey of some of the higher-level challenges and lower-level engineering will be useful to other implentors and users, as well as ourselves.

Bibliography

[1] Saxonica: *Type Conversion between JavaScript and XDM*, https://www.saxonica.com/saxon-js/documentation2/index.html#!xdm/conversions

[2] James Gosling, Bill Joy, et al. *The Java Language Specification, Chapter 5. Conversions and Contexts*, https://docs.oracle.com/javase/specs/jls/se22/html/jls-5.html

[3] Bjarne Stroustrup and Herb Sutter, editors *C++ Core Guidelines, ES.46 Avoid lossy (narrowing, truncating) arithmetic conversions*, https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#es46-avoid-lossy-narrowing-truncating-arithmetic-conversions

[4] Matt Neuberg *Applescript: The Definitive Guide 2nd edition*, https://www.oreilly.com/library/view/applescript-the-definitive/0596102119/ch04s03.html

[5] Norman Walsh, John Snelson, and Andrew Coleman, editors *XQuery and XPath Data Model 3.1*, https://www.w3.org/TR/xpath-datamodel-31/

[6] Paul V. Biron and Ashok Malhotra, editors *XML Schema Part 2: Datatypes Second Edition*, https://www.w3.org/TR/xmlschema-2/

[7] Michael Kay, editor *XPath and XQuery Functions and Operators 3.1 § 19.1.2 Casting to numeric types*, https://www.w3.org/TR/xpath-functions/#casting-to-numerics

[8] *GraalVM*, https://www.Graalvm.org/

[9] Kevin Menard: *Embedding Truffle Languages*, on May 9, 2022 https://nirvdrum.com/2022/05/09/truffle-language-embedding.html

[10] Abel Braaksma (Exselt): *Writing more robust XSLT stylesheets by understanding and leveraging the XDM data model*. XMLPrague 2019, Prague, 7-9 February 2019 `https://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf` `https://youtu.be/Y2sRDh-ymBU?si=jgLeE269SwxbYd97`

[11] Rennau, Hans-Jürgen, and David A. Lee. *XDML - an extensible markup language and processor for XDM.* Presented at Balisage: The Markup Conference 2011, Montréal, Canada, August 2 - 5, 2011. In Proceedings of Balisage: The Markup Conference 2011. Balisage Series on Markup Technologies, vol. 7 (2011). `https://doi.org/10.4242/BalisageVol7.Rennau01`