Using XQuery with Financial Messages

Michael Kay

Introduction

In this article I'll start by examining why a new query language is needed for XML, and how you might use it. I'll show a couple of examples of queries applied to data in the financial area (I shall try to find a compromise between making the examples simple enough to understand, and showing how real life is actually rather complicated, and how XQuery can help you to handle the complexity). I'll contrast XQuery with SQL, and with another XML-based processing language, XSLT.

Finally, and perhaps most importantly, I'll give a twist to the story, by showing how, through the use of C24's Integration Objects (IO), XQuery can also be used to handle non-XML data, such as comma delimited files, EDIFACT and SWIFT messages. This opens up the possibility of a new standard in message transformation and eases the burden on a large enterprise looking for standards in its Entreprise Service Bus (ESB) and Service-Oriented Architecture (SOA).

SQL and XML

For many years, the standard query language that everyone uses has been SQL. SQL is used to get data from relational databases, and it works very well in that role. It's supported by all the major vendors, and although there are considerable variations between the different dialects, there's a core that every database product recognizes. Even if you aren't using SQL yourself, you've probably seen queries like this one, which gets summary details of all the bank accounts with a postcode in a particular area:

```
SELECT A.ACCOUNT_NUMBER, A.BALANCE FROM ACCOUNTS A, BRANCH B
WHERE A.ACCOUNT_TYPE="SAVINGS"
      AND A.BRANCH = B.SORTCODE AND B.POSTCODE LIKE "RG4%"
```

Increasingly, though, we have to deal with data that doesn't sit comfortably in relational databases, with their rigid rows and columns. There are a number of reasons for this:

- The focus of attention has shifted from the database as the way applications are integrated, to the design of standard messages used to interchange information. This is because the pressure is no longer merely to integrate applications used within a single organization, but rather to exchange data with business partners to improve the efficiency of the entire value chain, and those partners rarely want to share data directly through a database. Web services have emerged to meet this need.
- Data models that satisfy the needs of an entire industry tend to be a lot more complex than data models used in-house within a single company. In particular, they often allow for many optional items of information which in most actual messages will be absent.
- In the past, there was a distinction between formal, structured data (what I call the ledger-book data) and unstructured documentary information. But it's not satisfactory to handle these separately: if you look at a company's annual report to shareholders, you see tables of numbers mixed in with text. There has therefore been a move towards technologies that can handle the full spectrum from highly-structured to very loosely structured information.

At the heart of this change is XML. XML came from the world of document preparation and publishing, but it has been embraced by the IT world because it meets the changing requirements for information management. Reflecting the three points above:

- XML is primarily designed for information interchange (messages) rather than for information storage. This has a number of implications: for example fields in a message are often optional not because the data is unknown or inapplicable, but simply because the intended recipient already has the information. In addition, databases are designed with update in mind, whereas messages are generally immutable. Relational design methodology is based on the concept of normalization. The primary object of

normalization is to avoid update anomalies, and without the need to perform updates, this approach loses its importance. As a result, XML messages will often be unnormalized.

- XML, with its hierarchic data model, can handle arbitrary complexity. It's particularly strong in situations where messages have many optional or repeatable parts, because you can simply leave out the parts that aren't applicable in your particular situation. (As with the relational model, XML can also handle cross-relationships by means of primary and foreign keys, though the details are slightly different.)
- XML handles documentary information as well as structured tabular data, and it allows the two to be freely mixed. This makes it ideal for documents such as insurance claims or contracts.

Associated with XML, a new query language is coming along called XQuery. It will be a while before this threatens to displace SQL, and as we will see in this article, it has a somewhat different role. But many people believe that it will quickly become as important as SQL has been for the last 25 years. One of those people is Don Chamberlin of IBM, Don was one of the original designers of SQL all those years ago, and is now the lead editor of the XQuery specification.

Why does XML need a Query Language?

Since XML is primarily a syntax for messages, rather than a database model, one might ask why it needs a query language. It's also worth asking why existing query languages (such as SQL) don't meet the requirement.

There are two ways XQuery can be used with XML messages. It can be used to extract information from a single message, or it can be used to search a collection of messages.

Either way, there are a number of tasks you can perform with XQuery. These include:

- Finding the data you need, and perhaps extracting this data selectively to another tool
- Analyzing and aggregating the data: finding totals, averages, trends
- Transforming the data into another format. In this last role, XQuery overaps with another XML processing language, XSLT, which was designed more specifically for this purpose. We'll look more closely at XSLT in another article.

One difference between SQL and XQuery (in version 1.0, at any rate) is that XQuery is a read-only query language. This again reflects the fact that XML focuses on messages rather than long-term persistent data. Modifications to messages and documents are generally made by transforming an input document to an output document, and this is reflected in the design of XQuery, which allows creation of a new document, but does not allow the original document to be changed.

Let's show what we mean with some examples. Here's an example of an XML message representing a "transfer-in confirmation". This message is the confirmation of the transfer, on November 13, 2004, of 2,500 units of the fund with ISIN GB123456789 from account 789xyz to account 123abc. 1,000 units out of the 2,500 are group 1 units and 1,500 units are group 2 units, and the trade date was November 09, 2004.

```
<?xml version="1.0" encoding="UTF-8"?>
<Document xmlns="urn:iso:std:iso:20022:xsd:sese.007.001.01">
<sese.007.001.01>
  <RltdRef>
    <Ref>MsgRefTransferIn02</Ref>
    <MsgNm>sese.005.001.01</MsgNm>
  </RltdRef>
  <TrfDtls>
    <TrfConfRef>TransfConf02</TrfConfRef>
    <TrfRef>TransferIn02</TrfRef>
    <FctvTrfDt>
      <Dt>2004-11-13</Dt>
    </FctvTrfDt>
    <TradDt>2004-11-09</TradDt>
    <TtlUnitsNb>
      <Unit>2500</Unit>
    </TtlUnitsNb>
    <UnitsDtls>
      <UnitsNb>
        <Unit>1500</Unit>
      </UnitsNb>
      <Grp1Or2Units>GRP2</Grp1Or2Units>
    </UnitsDtls>
    <UnitsDtls>
      <UnitsNb>
        <Unit>1000</Unit>
      </UnitsNb>
      <Grp1Or2Units>GRP1</Grp1Or2Units>
    </UnitsDtls>
    <OwnAcctTrfInd>false</OwnAcctTrfInd>
  </TrfDtls>
  <FinInstrmDtls>
    <Id>
      <ISIN>GB123456789</ISIN>
    </Id>
  </FinInstrmDtls>
  <AcctDtls>
    <AcctId>
      <Prtry>
        <Id>123abc</Id>
      </Prtry>
    </AcctId>
    <AcctSvcr>
      <BICOrBEI>TRANGB2L</BICOrBEI>
    </AcctSvcr>
  </AcctDtls>
  <SttlmDtls>
    <SttlmPtiesDtls>
      <DlvrrDtls>
        <AcctId>
          <Prtry>
            <Id>789xyz</Id>
          </Prtry>
        </AcctId>
      </DlvrrDtls>
      <DlvrgAgtDtls>
        <PtyId>
          <BICOrBEI>TRANGB2L</BICOrBEI>
        </PtyId>
      </DlvrgAgtDtls>
      <PlcOfSttlmDtls>
        <PtyId>
          <BICOrBEI>TRANGB2L</BICOrBEI>
        </PtyId>
      </PlcOfSttlmDtls>
    </SttlmPtiesDtls>
    <PhysTrfInd>false</PhysTrfInd>
  </SttlmDtls>
</sese.007.001.01>
</Document>
```

Example 1: extracting data from a message

Here's a query that you might do in order to extract basic details of this message:

```
declare namespace in = "urn:iso:std:iso:20022:xsd:sese.007.001.01";
<summary>
  <transferDate>{data(//in:FctvTrfDt/in:Dt)}</transferDate>
  <tradeDate>{data(//in:TradDt)}</tradeDate>
  <units>{data(//in:TtlUnitsNb/in:Unit)}</units>
  <from>{data(//in:DlvrrDtls/in:AcctId/in:Prtry/in:Id)}</from>
  <to>{data(//in:AcctDtls/in:AcctId/in:Prtry/in:Id)}</to>
</summary>
```

And here is the output:

```
<summary>
   <transferDate>2004-11-13</transferDate>
   <tradeDate>2004-11-09</tradeDate>
   <units>2500</units>
   <from>789xyz</from>
   <to>123abc</to>
</summary>
```

I chose a real message for this example: it comes from ISO 20022, the XML-based Universal Financial Industry message scheme, also known as UNIFI. This is a comparatively recent specification and it's not yet as widely used as the traditional SWIFT protocols. We'll see later how we can process non-XML messages that are in common use, such as SWIFT ISO 15022 messages. However, a consequence of using a real message rather than a toy one is that we're already exposed to a lot of the complexity of handling real-world data. Let's bring out a few points:

- Names like FctvTrfDt are not exactly memorable. The designers of this specification have tried to find a middle ground between meaningless codes such as F372, and long-winded names such as EffectiveTransferDate. They didn't always choose the abbreviation that you might have chosen. When you write a query, you have to get these names exactly right.

- The tags often seem more deeply nested than seems necessary. Why, for example, does FctvTrfDt (unlike TradDt) contain a Dt element, rather than containing the date directly? There can be a number of reasons for this: perhaps the designer wanted to make sure the messages could accommodate future change; perhaps they wanted to keep the message design faithful to an underlying abstract object model; or perhaps they were following design rules and guidelines that were designed to achieve consistent good practice, but which can sometimes lead to simple cases carrying baggage that's only really needed for more complex cases. Whatever the reasons, good or bad, such messages often have very many optional fields, and when most of these are omitted you can be left with a message that seems to contain a lot of scaffolding in relation to its payload.

  To address an element deep in the structure you can use a full path such as Document/sese.007.001.01/TrfDtls/FctvTrfDt/Dt. But this can be tedious, so it's often useful to skip levels by using "//" in a path, which searches the subtree for elements with the right name. When you do this, however, you need to watch out for name conflicts: for example, you couldn't use //Dt without ambiguity. (Some people will also tell you to avoid "//" for performance reasons, but that kind of advice might be right for one product and wrong for another.)

- All the tag names in this example are in the namespace urn:iso:std:iso:20022:xsd:sese.007.001.01. Although this is only written once at the start of the document, it's technically part of the name of each element, and you can't query the document without taking this into account. A document, and indeed a query, can define a default namespace, and if you want to use a name in any namespace other than the default, you need to assign a prefix and use it to refer to the namespace. In our example query, we wanted to produce output that wasn't in any namespace, so unprefixed names are used for the output. This means that when we refer to names in the input message, we need to use a prefix that tells the system which namespace we are talking about.

This particular query is what I call a "fill-in-the-blanks" query. It's constructing a simple XML document by extracting a few of the fields in the input XML document. The syntax for constructing the new document mimics the XML syntax of the target document, though it's actually XQuery syntax rather than XML syntax. The "blanks" are denoted by curly braces, and the value to be placed in the blanks is computed using an expression. This can in theory be any query, though in practice (as in our example) the expressions are usually paths that address particular locations in the source document. However, you will often find some simple arithmetic, or perhaps an if-then-else conditional, something like this:

```
<totalValue>
  {if (//isCredit='true') then sum(//credit) else sum(//debit)}
</totalValue>
```

The data() function, by the way, extracts the value of an element. If we hadn't used it, then an expression such as

```
<tradeDate>{//in:TradDt}</tradeDate>
```

would insert the whole input element, tags and all, into the result. Sometimes, of course, this can be useful.

Example 2: Aggregating data across messages

Now let's suppose we have a whole collection of similar messages, and we want to create some kind of summary report. Let's say, for example, that for each receiving account, we want to list all the accounts from which transfers were made over the previous year, with the total number of units transferred from each such account. Here's the query to do this:

```
declare namespace in = "urn:iso:std:iso:20022:xsd:sese.007.001.01";
declare variable $db := collection('sese.007.messages')
  [.//in:TradDt > current-date() - xs:yearMonthDuration('P1Y')];
<summary> {
  for $a in distinct-values( $db//in:AcctDtls/in:AcctId/in:Prtry/in:Id)
  return
    <receivingAccount id="{$a}"> {
       let $am := $db[.//in:AcctDtls/in:AcctId/in:Prtry/in:Id = $a]
       for $s in distinct-values(
            $am//in:DlvrrDtls/in:AcctId/in:Prtry/in:Id)
       let $sm := $am[.//in:DlvrrDtls/in:AcctId/in:Prtry/in:Id = $s]
       return
         <sendingAccount
              id="{$s}"
              totalUnits="{sum($sm//in:TtlUnitsNb/in:Unit)}"/>
    }</receivingAccount>
} </summary>
```

The result, depending of course on the data in the collection, might be something like this:

```
<summary>
   <receivingAccount id="123abc">
      <sendingAccount totalUnits="2500" id="789xyz"/>
      <sendingAccount totalUnits="4500" id="789pqr"/>
   </receivingAccount>
   <receivingAccount id="123def">
      <sendingAccount totalUnits="4000" id="789pqr"/>
   </receivingAccount>
</summary>
```

I'm not going to give a complete account of everything in this query. There are other tutorials available, such as my article on the Stylus Studio site [http://www.stylusstudio.com/xquery_primer.html]. But here are some salient points:

- We start by assigning a variable $db to represent the set of documents in the input collection whose trading date is later than a date one year before the current date. The square brackets act like a "where" clause: they filter the items in the collection, retaining only those that satisfy the predicate inside the square brackets. The expression xs:yearMonthDuration('P1Y') represents a duration of one year, and we can subtract this from the current date and compare the result with the date in the input message.
- The body of the query then consists of a <summary> element, into which we insert a number of <receivingAccount> elements. Specifically, we create one <receivingAccount> element for every distinct value of in:AcctDtls/in:AcctId/in:Prtry/in:Id found in any of the selected input messages.
- The construct id="{$a}" is another fill-in-the-blanks construct, only this time we are creating an XML attribute rather than an XML element.
- Within the <receivingAccount> element, we create one or more <sendingAccount> elements. The logic here is a bit complicated, but essentially we are first calculating the set of messages ($am) for the receiving party $a; we then determine the set of distinct sending parties ($s) appearing in these messages, and the set of messages ($sm) for each of these sending parties. Then we construct the <sendingAccount> element, with an attribute identifying the account, and another attribute totalUnits whose value is obtained by using the sum() function

to total all the values of TtlUnitsNb across the messages ($sm) representing transactions between these two parties.

XQuery and SQL

We could actually have done most of this, with the notable exception of constructing new XML elements and attributes, using SQL, assuming that the data was stored in suitable form. That's because the data in these messages, despite the deep element nesting, is actually rather flat (in the sense that there are few repeating elements). A notable difference from SQL, however, is that the data doesn't have to be so flat. It's possible for many of the elements in a message like this to repeat, and XQuery is designed to handle this.

In a relational database, data is stored in normalized tables. That means that whenever something can occur more than once, you need to create another table, and link it to the main table with some kind of identifier. For example, if a person can have more than one phone number, then the phone number can't be a column of the person table; instead, you have to create a new table containing just the person identifiers and phone numbers, with one entry for each phone number; and when someone does a query, they need to join data from the two tables using a SQL query like the one at the top of this article. The more flexible your data model becomes, the more you tend to find that almost any data can be optional or repeated: for example, a customer can have more than one account, an account can have more than one contact person, a contact person can have more than one address, an address can even, in the case of a large site, have more than one postal code. With SQL, every one-to-many relationship means another table, and every extra table adds complexity to the queries, even when you're interested in data that doesn't actually take advantage of any of the flexibility.

The XML data model is fundamentally different. Repeated data is the norm; data that can't repeat is treated as a special case. The operators in XQuery are designed to make it easy to work with repeated data. For example (let's stray from the complexities of financial messages for a while), the following XQuery finds the ISBNs and titles of all books that have Michael Kay as one of their authors:

```
//book[author='Michael Kay']/(title, ISBN)
```

This works for books that have multiple authors (because that's the way "=" is defined in XQuery), and it also works for books that have multiple ISBNs (hardback and paperback editions, for example). I'm not aware of any books with more than one title, but the query would still work.

An aside: many queries can be written in the concise notation I'm using here called a path expression. In fact, there's a subset of XQuery called XPath that always uses this concise notation. But many XQuery users prefer a more verbose but perhaps more readable style called the FLWOR expression. In this style, the query becomes:

```
for $b in //book
where $b/author='Michael Kay'
return ($b/title, $b/ISBN)
```

Either way, however, the query author doesn't need to worry too much about whether books can have multiple authors, titles, or ISBNs.

To do the same thing in SQL, you definitely need to know which data can be repeated, because it will affect the table structure. In this example, if books can have multiple authors and multiple ISBNs then we're likely to find ourselves writing a query like this:

```
SELECT B.TITLE, I.ISBN
FROM BOOK B, ISBN I, AUTHOR A
WHERE A.NAME = 'Michael Kay'
AND A.BOOK_ID = B.ID AND I.BOOK_ID = B.ID
```

The deep nesting of the UNIFI message we looked at earlier reflects the flexibility built into the message design. Although the data in this case is largely flat (the only field which actually repeats in the message is <UnitsDtls>), the message structure is designed to accommodate a great deal of variety, and not all messages will be this simple. To some extent it is true in XQuery as well as in

SQL that the query writer is exposed to the potential complexity implied by the message schema, not only to the actual complexity present in this instance. But there are plenty of short-cuts (like the "//" notation, and the special meaning of "=" to match any one of a set of values) that are designed to make the task of the query writer easier in this kind of environment. By contrast, in SQL, writing joins between a dozen or more tables is no fun at all.

Handling Data Types

One thing that XQuery and SQL have in common is a set of built-in data types for describing values such as integers, strings, and dates. They aren't exactly the same set of types, but the differences aren't that significant.

The original relational model didn't recognize that there could also be complex types: composite objects such as addresses, personal names, or amounts of money (currency plus numerical quantity). These are complex in that they have an internal structure. Recent versions of the SQL standard do have such a concept, which allows you for example to create two tables having the same structure (perhaps one for current employees and one for ex-employees). Many SQL developers never create types explicitly, but in theory whenever you create a table, you are really defining two things: a complex type (represented by the table's definition as a set of columns), and a container for instances of that type. More modern languages (including object-oriented models, the XML model, and recent versions of SQL) try to separate the definition of the type from the definition of a collection of instances.

XQuery can work either on typed or untyped data. In the financial world, it's likely that most data will be strongly typed, which basically means that the documents you are dealing with have a schema that defines the details of the structure. Untyped documents (documents without a schema) are more likely to be encountered in relatively unstructured environments. The biggest advantage of working with typed data is that your query can be checked for correctness against the schema. Without a schema, the query `$am//in:DlvrDtls/in:AcctId/in:Prtry/in:Id` is likely to return nothing (an empty set) as its result; with a schema, there's a good chance the system will be able to tell you that `DlvrDtls` should have been spelt `DlvrrDtls`.

With XQuery, the collection of instances that you want to query might be found in a single document, or it might span a collection of documents. The details of how collections of documents are handled (for example, the way that you create such a collection, and then add documents to it) depends on the query processor that you are using. Some products (my own Saxon product is an example) might allow you to query a set of files held in a Unix or Windows directory. This is often very useful, but it doesn't scale up for applications that need to search terabytes of data. For that you need an XML database, which will typically index each document as it is loaded into the database.

XQuery and XSLT

The capabilities of XQuery overlap the capabilities of another more established XML processing language, XSLT. XSLT was designed for performing document transformations — often from XML into a presentation format such as HTML or PDF, but also between different XML vocabularies or to modify the message content while still using the same vocabulary. We'll be talking more about XSLT in another article. In summary, the main differences are:

- XQuery has a much more concise syntax, meaning that simple queries can be written in 3 lines rather than 30
- XQuery is a smaller language, making it easier to learn and remember if you aren't using it every day

Of course the fact that XQuery is a smaller language also means that there are many features in XSLT (especially the latest version, XSLT 2.0) that are missing from XQuery, and which tend to be needed for larger applications.

Handling non-XML messages

In practice, there's lots of data being sent around the world in message formats that predate XML by many years. In the financial world, a prominent example of such a message set is represented by the SWIFT protocols. Here's an example of such a message:

```
{1:F01FRNYUS30AXXX4218327520}{2:O1030947040127FRNYUS33AXXX4218183425040127094 7N
}{3:{108:MARKETS/07740}}{4:
:20:MT103-10c
:23B:CRED
:32A:050407USD999999999999,99
:50K:MINECOFIN,KIGALI/RWANDA
:57A:/054001314
FRNYUS33
:59:/0043571601
THE INSTITUTE FOR PUBLIC-PRIVATE
PARTNERSHIPS
:70:/RFB/PARTICIPANT TUITION IN THE
//WORKSHOP''1325 INCENTIVE-BASED
//REGUL.(WDC)''FOR MS.GAJU ANITA,
//AUG 18-19,2003
:71A:SHA
-}{5:{MAC:808FA405}{CHK:5DB6725CDCE2}{PDE:}}
```

This is an MT103 message from the SWIFT ISO 15022 standard: a Single Customer Transfer Text Block. To quote the spec: *This message is used to convey a funds transfer instruction in which the ordering customer or the beneficiary customer, or both, are non-financial institutions from the perspective of the Sender*.

As with XML, this message essentially has a hierarchic structure. At the top level, it has five sections, labelled {1:...} to {5:...}. Within each of these sections there are subfields, marked either by further curly braces, or by tags such as :23B:. The tags themselves are pretty cryptic; it would take inspired guessing to work out that 23B is a Bank Operation Code identifying the type of operation, where the code CRED indicates a credit transfer with no specified SWIFT service level. In fact, there is some structure in this message that's not marked by explicit tags or punctuation at all. For example, field 32A is a composite field containing a 6-digit date in YYMMDD format (yes, a two-digit year!), a three-character currency code from the international standard ISO 4217, and then the value of the transfer. The decimal point is represented by a comma, and the number of digits after the comma depends on the currency.

Take the first field, called Block 1, the Basic Header. The first character is called the Application ID, and is always A, F, or L. The next two characters are the Service ID, which is always 01, 03, or 21. Then comes a four-character bank code, a two-character country code, a two-character location code, and so it goes on. And that's just Block 1. Remember punched cards, anyone?

In fact, there's a mixture here of fixed length fields, fields marked by punctuation and identifying tags, and fields delimited by end of line.

Bear in mind too that this is an instance document. The document defining the SWIFT schema-equivalent is vastly more complex and doesn't exist in electronically readable format.

But these differences from XML are essentially superficial. The structure of this message, as distinct from its surface syntax, is equivalent to an XML message that looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<MT103i xmlns="http://www.c24.biz/IO/SWIFT2005/MT1nn">
    <Block1>
        <ApplicationID>F</ApplicationID>
        <ServiceID>01</ServiceID>
        <LTAddress>
            <BankCode>FRNY</BankCode>
            <CountryCode>US</CountryCode>
            <LocationCode>
                <LocationCode1>3</LocationCode1>
                <LocationCode2>0</LocationCode2>
            </LocationCode>
            <LogicalTerminalCode>A</LogicalTerminalCode>
            <BranchCode>XXX</BranchCode>
```

```
        </LTAddress>
        <SessionNumber>4218</SessionNumber>
        <SequenceNumber>327520</SequenceNumber>
    </Block1>
    <Oblock2>
        <InputOutputIdentifier>O</InputOutputIdentifier>
        <MessageType>103</MessageType>
        <InputTime>0947</InputTime>
        <MessageInputReference>
            <InputDate>040127</InputDate>
            <LTAddress>
                <BankCode>FRNY</BankCode>
                <CountryCode>US</CountryCode>
                <LocationCode>
                    <LocationCode1>3</LocationCode1>
                    <LocationCode2>3</LocationCode2>
                </LocationCode>
                <LogicalTerminalCode>A</LogicalTerminalCode>
                <BranchCode>XXX</BranchCode>
            </LTAddress>
            <SessionNumber>4218</SessionNumber>
            <SequenceNumber>183425</SequenceNumber>
        </MessageInputReference>
        <OutputDate>040127</OutputDate>
        <OutputTime>0947</OutputTime>
        <MessagePriority>N</MessagePriority>
    </Oblock2>
    <Block3>
        <Field108>MARKETS/07740</Field108>
    </Block3>
    <Block4>
        <SenderRef>
            <Default>
                <Reference>MT103-10c</Reference>
            </Default>
        </SenderRef>
        <Field13aDateAndTime>
            <C>
                <TimeIndicationCode>SNDTIME</TimeIndicationCode>
                <TimeHHMM>01:00:00</TimeHHMM>
                <SignPlusOrMinus>+</SignPlusOrMinus>
                <TimeHHMM1>01:00:00</TimeHHMM1>
            </C>
        </Field13aDateAndTime>
        <Field23BBankOperationOrOptionCode>
            <B>
                <BankOperationOrOptionCode>CRED</BankOperationOrOptionCode>
            </B>
        </Field23BBankOperationOrOptionCode>
        <Field32aDatesAndAmounts>
            <A>
                <DateYYMMDD>2005-04-07</DateYYMMDD>
                <CurrencyAmount>
                    <Currency>USD</Currency>
                    <Amount>9.9999999999999E11</Amount>
                </CurrencyAmount>
            </A>
        </Field32aDatesAndAmounts>
        <Field50aOrderingCustomerOrAdditionalParty>
            <K>
                <NameAndAddress>
                    <Line35x>MINECOFIN,KIGALI/RWANDA</Line35x>
                </NameAndAddress>
            </K>
        </Field50aOrderingCustomerOrAdditionalParty>
        <Field57aAccountWithInstitution>
            <A>
                <PartyLine1>
                    <PartyIdentifier>054001314</PartyIdentifier>
                </PartyLine1>
                <BIC>
                    <BankCode>FRNY</BankCode>
                    <CountryCode>US</CountryCode>
                    <LocationCode>
                        <LocationCode1>3</LocationCode1>
                        <LocationCode2>3</LocationCode2>
                    </LocationCode>
                </BIC>
            </A>
        </Field57aAccountWithInstitution>
        <Field59Or59a>
            <Field59BeneficiaryCustomer>
                <Default>
                    <PartyIdentifier1>0043571601</PartyIdentifier1>
                    <NameAndAddress>
                        <Line35x>THE INSTITUTE FOR PUBLIC-PRIVATE</Line35x>
                        <Line35x>PARTNERSHIPS</Line35x>
                    </NameAndAddress>
                </Default>
            </Field59BeneficiaryCustomer>
        </Field59Or59a>
        <Field70DetailsOfPayment>
            <Default>
                <DetailsOfPayment>
                    <Line35x>/RFB/PARTICIPANT TUITION IN THE</Line35x>
                    <Line35x>//WORKSHOP''1325 INCENTIVE-BASED</Line35x>
                    <Line35x>//REGUL.(WDC)''FOR MS.GAJU ANITA,</Line35x>
                    <Line35x>//AUG 18-19,2003</Line35x>
                </DetailsOfPayment>
```

```
            </Default>
        </Field70DetailsOfPayment>
        <DetailsOfCharges>
            <A>
                <DetailsOfChargesCodes>SHA</DetailsOfChargesCodes>
            </A>
        </DetailsOfCharges>
    </Block4>
    <Block5>
        <MACPACCode>
            <MacTrailer>
                <MACCode>808FA405</MACCode>
            </MacTrailer>
        </MACPACCode>
        <ChkTrailer>
            <Checksum>5DB6725CDCE2</Checksum>
        </ChkTrailer>
        <PdeTrailer/>
    </Block5>
</MT103i>
```

That's quite a difference in size! – as bandwidth becomes cheaper, protocol designers find ways of using more of it. But the advantage is that all the individual fields are now tagged and named, which makes the message much more amenable to processing using general-purpose tools such as XQuery.

This means that you could translate the messages to XML, and back again, without losing information, assuming of course you know the original format of the SWIFT message. You could then easily use XML tools such as XQuery (or XSLT, if preferred) to process the data. This is important because although the SWIFT message formats are widely used, there isn't a lot of software around that makes it easy to process the data in its native form, and what there is is generally expensive. For example, no-one to my knowledge has created a database system and query language for SWIFT messages in their native form. John Davies, (CTO of C24) in his paper [http://www.c24.biz/download/c24_white_paper-parsing_a_swift_message.pdf] shows why: parsing these messages is not something you can take lightly. By contrast, the world is awash with XML software tools, and many of them are free.

Here's an example of the kind of processing that becomes possible. Given a collection of messages like the one above, we could find all "beneficiary customers" featured in messages for the current day with the query:

```
for $m in collection('messages')
where $m//Field32aDatesAndAmounts/A/DateYYMMDD = current-date()
return $m//Field59BeneficiaryCustomer
```

Compare this with the difficulty of extracting the data from the original compact SWIFT message.

We could probably end the story there: once you can translate all your SWIFT messages into XML, all your troubles are over. Except that in practice, that's not necessarily what you want to do. Managing large quantities of data in multiple formats causes all sorts of headaches: how do you ensure that the two copies of the data are always in sync? Fortunately there's another twist in the XQuery story: although XQuery was designed to handle XML, the data doesn't actually have to be real, physical XML for XQuery to process it. It can be virtual, pretend XML instead.

The key to this magic is that XQuery doesn't work directly on physical XML syntax, that is, XML-as-angle-brackets. Instead, it relies on the XML being parsed into a tree structure. You can regard this as a data structure in memory, in which the angle brackets have disappeared and all you are left with is named pieces of data related to each other by pointers. The XQuery specification of course describes it in much more abstract terms, but that's just to give implementers maximum scope to exercise their creativity. But the net effect is that the input doesn't actually have to be XML: any message format that can be visualized as a tree of named data objects will do just as well. And as we've seen, SWIFT messages fit well into that model. All you need is a parser that understands the raw syntax and builds the tree representation, and you can then run XQuery on it directly, with no angle brackets in sight.

Many people in fact regard this hierarchical data model as the true essence of XML, and the lexical representation (the angle brackets) as just a convenient way of sending it down the wire between one application and another. In fact, people are now working on alternative representations of XML that are more efficient to store and transmit — the angle brackets have served us well, because it's very convenient to have the data in a form where it can be directly displayed and edited in human-

readable form on screen, but it does use an extravagant number of bits to convey the information, and bits are not always free, especially with mobile devices.

Java Objects or XML?

We've seen briefly how you can view a SWIFT message as if it were XML. That means you can process it as an XML document, using high-level languages such as XQuery and XSLT. But what if you want to write applications in Java?

There are basically two ways of processing XML documents in Java. One way is to use a generic object model. The most common one, standardized by the W3C, is the Document Object Model or DOM. Some people prefer alternatives such as JDOM, DOM4J, or XOM. Viewed from ten thousand feet, all these models are much the same: they present the programmer with a collection of Element and Attribute objects, with methods such as getFirstChild() and getNextSibling() to navigate your way around the structure. When you have as much scaffolding as we saw in the UNIFI message, finding your way to the useful data can be hard work, and it's easy to get lost.

The alternative approach is called *data binding*. In this approach, instead of using generic objects like Element and Attribute, the Java programmer works with objects that relate to the business model: classes with names like Field26TTransactionTypeCodeOptionT. These are in fact the same names as we've seen used as XML tags in the XML representation of the message. (Not really the most friendly of names, but this is real life, not XML for dummies). The benefit of using classes that relate to the business model is that programs are much more robust: when you navigate to the third child of the fifth child of the fourth child of the root node, the Element object you end up at could be anything; with specific names, there's no risk of processing the wrong kind of object. Once you have found your way to the Field26TTransactionTypeCodeOptionT object, you can call the method getTransactionTypeCode() to get the actual value, and Java type checking ensures that you can't call this method if you've found the wrong element.

Translating the description of a message type into a set of Java class definitions is of course tedious work, but it doesn't have to be done by hand: typically, tools will translate automatically from a description of the message (in the form of an *XML Schema*) into the Java class definitions. There are a number of data binding tools that can do this for XML; a major advantage of C24's Integration Objects (IO) is that it can also do the same thing for a variety of non-XML formats including CSVs and SWIFT messages for example.

For the MT103 object (as used above), the top level class that's generated is called MT103iMessage. You can create one of these objects by calling a simple LoadMessage() method with the name of an input file as the parameter. This reads the file, parses it, and generates the Java structure in memory. Then you can call methods such as getBlock1() to get the first header block (an object of class BasicHeaderBlock1), and getBlock4() to get the message payload, an object of class MT103TextBlock. You can of course chain method calls to navigate through the structure, so you can call

```
getField36ExchangeRate().getDefault().getExchangeRate();
```

to find the exchange rate used for the transaction (returned as a type-safe Java double).

Although this follows the data binding approach whereby specific fields of the message are mapped to specific Java classes, C24 IO goes a step further, and gives you a generic XML view of the data as well. This is the XML view of the message that we looked at earlier generated by a SWIFT MT103 IO object.

With a few simple lines of code a complex SWIFT message can be formatted as XML and worked on using XQuery and XSLT. The *pièce de résistance* however is that we can apply the XQuery and XSLT directly to the IO object without having to convert it to XML. A SWIFT message can, for example, be transformed without having to pass by an intermediate XML format. This saves time (latency) and complexity in integration. An XQuery could be written to process a comma-delimited file or an XSL transform could be used to display details from a SWIFT message.

The way this works is that all the application-specific classes (like Field36ExchangeRateOptionDefault) actually extend a generic class called ComplexDataObject, which is essentially the same as the Element class of object models like DOM and JDOM. And like the generic Element class, it has generic methods such as getAttr() and getChild() that allow you to navigate the structure. This makes the data

accessible to query processors and other tools. Such tools have to be able to process any document structure, so they can't use application-specific classes, but instead prefer to work with the generic classes and methods.

What this amounts to is that you can play with the same underlying data in three different representations. The original SWIFT message format is useful because that's what the world's banking system actually recognizes, and because when you're sending millions of messages around the world, the compact representation is actually very efficient. The XML representation is much more verbose, but the advantage of tagging every field with its full name is that the data becomes accessible for integration. Lastly there's a representation in terms of Java objects, which you can use to write your own custom applications, for example applications which process data coming into your organization and update the relevant operational databases. All three formats are interchangeable though C24's Integration Objects.

Summing Up

As everyone knows, XML is the fashionable choice for information interchange between applications, both within an organization and for trading between business partners. In practice, the more advanced an industry is in doing business electronically, the harder it is to displace established protocols and replace them with XML formats. This is especially true in the financial industry, where standards such as SWIFT have been established for years and CSVs are still in wide use.

The dominance of XML means that there's a new set of powerful technologies for processing messages, one of which is the query language XQuery. And in turn, it's the availability of these tools (often at very modest cost) that fuels the adoption of XML.

To an extent, you can handle XML messages by extracting the data and storing it in a relational database. But the more complex the messages become, the harder this is to achieve. And messages are becoming more complex, because they have to meet the needs of a wider range of industries and countries. It gets to the point where you're spending all your development efforts converting data from one format to another (from XML to SQL and vice versa, for example), which in the end is not an activity that adds value, given tools like XQuery and XML databases that can hold the data in its native form.

I have tried to show in this article that XML is more than just a syntax for messages. It's a data model, and a raft of associated tools and technologies, and the fact that you're using message protocols like SWIFT that predate XML doesn't mean you're left out from all the benefits that XML brings. With technologies such as C24's Integration Objects, you can view your non-XML messages as Java objects or XML documents but in both cases you can also program in XQuery or XSLT.

Sounds too good to be true? Give it a try: I don't think you'll be disappointed.