Programming with Circles, Triangles and Rectangles

Keywords: API, C#, Data interchange, Data representation, Database, Datatypes, DTD, Integration, Java, JDBC, JAXP, Objects, Query language, Relational database, RELAX NG, Schema, SOAP, SQL, Type systems, Validation, W3C XML Schema, XML, XPath, XQuery

Erik Meijer
Technical Lead
Microsoft Webdata
Redmond
WA
USA
emeijer@microsoft.com
http://www.research.microsoft.com/~emeijer

Biography

> Erik Meijer is a technical lead in the WebData group at Microsoft where he currently works on language design and type-systems for data integration in programming languages. Prior to joining Microsoft he was an associate professor at Utrecht University and adjunct professor at the Oregon Graduate Institute. Erik is one of the designers of the standard functional programming language Haskell98.

Wolfram Schulte
Researcher
Microsoft Research
Redmond
WA
USA
schulte@microsoft.com
http://www.research.microsoft.com/~schulte

Biography

> Wolfram Schulte is a researcher at Microsoft. He is interested in all aspects of software design and development for reliable systems. Currently he leads a research project focused on advanced technologies for software testing. He has also made technical contributions to Microsoft's data-oriented programming languages. Before Microsoft, Wolfram was assistant professor for Computer Science at the University of Ulm and at the Technical University Berlin.

Gavin Bierman
University Lecturer
University of Cambridge Computer Laboratory
Cambridge
England
United Kingdom

Gavin.Bierman@cl.cam.ac.uk
http://www.cl.cam.ac.uk/~gmb/

Biography

Gavin Bierman is a University Lecturer (Associate Professor) at the University of Cambridge Computer Laboratory. His research interests include the semantics of programming languages, type systems, database query languages, semi structured data, and proof theory.

Abstract

This paper proposes extending popular object-oriented programming languages such as C#, VB or Java with native support for XML. In our approach XML documents or document fragments become first class citizens. This means that XML values can be constructed, loaded, passed, transformed and updated in a type-safe manner. The type system extensions, however, are not based on XML Schemas. We show that XSDs and the XML data model do not fit well with the class-based nominal type system and object graph representation of our target languages. Instead we propose to extend the C# type system with new structural types that model XSD sequences, choices, and all-groups. We also propose a number of extensions to the language itself that incorporate a simple but expressive query language that is influenced by XPath and SQL. We demonstrate our language and type system by translating a selection of the XQuery use cases.

Table of Contents

APIs and Programming Languages

Modern programming languages now come in two parts: the programming language proper, and a set of APIs. The relationship between the language and the API is an interesting one. Whilst the language is in some senses master and the API slave, a closer look reveals that in fact they form a symbiosis; an API only makes sense in the context of a programming language and a programming language becomes interesting if it has access to a rich set of APIs.

Just like biological artifacts, this symbiosis is subject to evolution. A heavily used part of the API or common programming pattern is a candidate for promotion to the host programming language. We are all familiar with this situation: we can't imagine a language where numerals, strings, or arrays were available purely via APIs!

New application areas are suggesting new candidates for the first-class world of the programming language. Consider the language Rebol (http://www.rebol.com): besides the usual numerals, strings and booleans, the core language supports literals for a wide range of other types such as URLs, dates, money, etc. SQL has at its heart the table data type, but it also has a range of first-class data types relevant for business database applications, such as date, timestamp and interval. Even a general-purpose language such as C# **[C# Specification]** has integrated ideas such as garbage collection, objects, properties, indexers, events, and iterators into the language instead of exposing these concepts via APIs such as malloc/free, COM, or using programming patterns such as `GetXXX`/`SetXXX`.

To be concrete, consider the following example from C#. Assume that we have a collection type that implements `System.IEnumerable`. Given a collection object, `ts`, of such a type we find ourselves frequently writing code such as the following which executes some common piece of code, here written `s`, on each element of the collection:

```
IEnumerator e = ((IEnumerable)ts).GetEnumerator();
try {
   while(e.MoveNext()) {  T t = (T)e.Current; s; }
} finally {
   IDisposable d = e as System.IDisposable;
   if(d != null) d.Dispose();
}
```

This programming pattern appears so often that C# has, and Java will soon have, a first-class language construct, `foreach(T t in ts)s`, which is intended to denote the same code. Not only does this make code more compact and readable, it also reduces sources of potential bugs and frees the compiler to make possible optimizations.

The problem facing the programming language designer is not only identifying the "heavily used APIs" and "common programming patterns" mentioned earlier but also tastefully realizing them as first-class language features or constructs. Bolting on random features leads to language bloat, potentially destroying any coherence the language may have had. In addition one hopes that by promoting a feature from an API it will be better supported given its existence in the language. We will see examples of this later. Clearly this is a tough problem which requires serious research (first) as well as some experimentation.

In our opinion three areas that are ripe for liberation from their lowly API status are (a) data-access, (b) concurrency and (c) security. In this paper we concentrate on the first area: data-access. We discuss briefly concurrency in the future work section. For an interesting take on security we recommend the model of Fournet and Abadi **[Fournet and Abadi]** .

We shall fix our attention on object-oriented programming languages, exemplified by C# and Java **[Java Language Specification]** . Objects are the "circles" in our title. Thus when considering liberating an API, it will be into a host language that is object-oriented. In fact, in this paper the host language is C#, but what we say applies equally to Java, Visual Basic, or any statically typed object-oriented language.

The rest of the paper considers the promotion of data-access APIs into C#. The data we are interested in is XML. This is the "triangle" in the title, reflecting the tree/graph-like structure of XML documents. First we motivate why XML support should be a first-class feature of a language. Then we argue that this support can not be offered to arbitrary XML documents. We then introduce Xen, which is our hypothetical extension of C# with first-class XML support. We give both details of the language and its type system, as well as a number of examples to illustrate its elegance and expressive power.

The simple example program below gives a flavor of our language. It defines two types, `card` and `logo`, that together model simple business cards (taken from http://www.brics.dk/~amoeller/XML/schemas/). It then creates a new card instance, `c`, using an XML object literal and uses a path-expression to print all the fields of the card.

```
public class card {
  sequence{
    string  name;
    string  title;
    string  email;
    string? phone;
    logo?   logo;
  };
}

public class logo {
  attribute string url;
}

public class Test {

    static void Main() {
      card c = <card>
                 <name>John Doe</name>
                 <title>CEO, Widget Inc.</title>
                 <email>john.doe@widget.com</email>
                 <phone>(202) 456-1414</phone>
                 <logo url="widget.gif"/>
               </card>;
      c.*.{ Console.WriteLine(it); };
    }
}
```

Programming XML using APIs

One of the problems with the current API-based approach of programming against XML is that these APIs tend to abuse strings to represent the literals of the various (non-string) types such as documents and queries that the API exposes. As an example let us consider how to program against the XML data model using the `System.Xml` library of .NET. In the code below the `SelectNodes` method of the `XmlDocument` class takes a string argument that represents an XPath expression that will be executed against the document instance. Similarly, the `SelectSingleNode` method on `XmlElement` takes a string which is an XPath expression intended to select `XmlNodes`:

```
int GetTotalByZip(XmlDocument d, int zip) {
    int total = 0;
    string q = string.Format("orders/order[@zip={0}]/item", zip);
    foreach (XmlElement x in d.SelectNodes(q)) {
      XmlNode price = x.SelectSingleNode("price");
      XmlNode qty = x.SelectSingleNode("quantity");
      int p = Decimal.Parse(price.InnerText);
      int q = Decimal.Parse(qty.InnerText);
      total += p*q;
    }
    return total;
}
```

Using strings to represent XPath programs is not only pretty clumsy (i.e. using `String.Format` to pass parameters from C# to XPath) but also annihilates all possibilities for static checking. Perhaps more importantly passing strings that represent programs is often a security risk ("script code injection"). Also note that because result values are also passed as strings, `XmlDocument` programming involves a lot of expensive string to type coercions.

Hence, as a rule of thumb, we suggest that should one see an API that uses strings as a representation for some other important artifact, then this API requires re-designing and/or consideration for promotion to the programming language.

Using this rule of thumb, the Java programmer using JDBC **[JDBC]** or the C# programmer using ADO.Net **[ADO .Net]** will quickly conclude, as we do, that support for database tables and queries could be better provided within the programming language, as opposed to the rather poor support provided by the JDBC or ADO.Net APIs. The reader will notice that the proposals in this paper also provide first-class language support for database tables and queries -- the "rectangles" in the title.

Integrating XML into a host language

There is a wide spectrum of possibilities for moving XML support from the API level into a first-class language feature. Currently the most popular technique is to use some form of mapping from XML to the host language. There exist a plethora of such databinding tools (see http://homepages.inf.ed.ac.uk/wadler/xml/ for a list of links), the Microsoft .NET environment contains a tool called xsd.exe that can translate between XSD schema and CLR types **[xsd.exe]** ; for Java there is the JAXB tool **[JAXB]** . BEA XMLBeans is an interesting hybrid that allows the API approach and the databinding approach to coexist **[XMLBeans]** .

Using these tools we can, for example, map an XSD schema for `orders` and `items` into a corresponding set of classes and then program against them in the following way:

```
int GetTotalByZip(orders d, int zip) {
  ArrayList items = new ArrayList();
  foreach(order o in d){ if(o.zip == zip) items.Add(o.item); }
  int total = 0;
  foreach (item x in items) total += x.price*x.quantity;
  return total;
}
```

The main advantage is that via the mapping XML values are represented in the strongly typed world of C#. It is perhaps also worth pointing out that this code would be further improved with the use of generic collections. These will soon be added to both C# and Java.

This mapping technology works reasonably well, but in practice we find that there are some annoying mismatches that makes this technology harder to use than might have been expected. In essence this mapping technique is dodging the fundamental impedance mismatch between XML and the host language. As we will see later, there is often no natural counterpart in C# (or Java) for many XSD constructs and vice versa. In fact the XPath/Infoset data-model and the usual object data-models have very little in common.

One final problem: even after the mapping of XSD schema into classes, we can see clearly in the example code that we have nothing as high-level and expressive as XQuery/XPath in C#. Thus queries have to be written using lower-level loops and conditionals. Whilst C# ensures that this is done type correctly, the resulting code is verbose and hard to maintain. Hence even if there was a perfect mapping between XML and objects at the type-level, we still would lack the expressive power of path expressions in the object-oriented programming language.

In conclusion, we can see that this "databinding" approach boils down to hiding XML instead of embracing it. We are interested in finding a clearer solution. In order to successfully promote XML support from the API to the language level we shall first compare both models carefully. After this comparison we will argue that the inherent impedance mismatch between the XML and object data-models suggests that we should both modify the host object language to enhance XML support and compromise full XML fidelity. This leads to an elegant language design that incorporates the strengths of both data-models within a coherent whole: a language ideal for circles, triangles and rectangles.

The mismatch between XML and object data-models

In this section we compare the XML and object data-models concentrating on the deep mismatches between them. We give details of several substantial differences between the models that we contend are inherent and thus represent serious obstacles to the promotion of full XML support to an object-oriented programming language.

In what follows we will assume that the reader is familiar with the basic details of XML Schema (a useful primer can be found at http://www.w3.org/TR/xmlschema-0/) and the XPath 2.0 Data Model (http://www.w3.org/TR/xpath-datamodel/).

Edge-labelled vs. Node-labelled

Probably the deepest and most fundamental difference between the XML and the object data-model is that the former assumes node-labelled trees/graphs (see page 33 of **[Data On The Web]**) while the latter assumes edge-labelled trees/graphs. This clearly represents a profound dichotomy between the two worlds, yet at best many people seem unaware of this difference, and at worst confuse the two.

To make the clash concrete, consider the following simple schema for a `Point` element that contains `x` and `y` child elements:

```
<element name="Point">
  <complexType>
    <sequence>
      <element name="x" type="integer" />
      <element name="y" type="integer" />
    </sequence>
  </complexType>
</element>
```

Assuming the XML data-model, selecting the `x` child of a `Point` element `p` returns an element node with name `x`, parent `p`, type `string` and a single text node as a child node. Note that the type of the result is not `string`, but an element node that has as one of its properties the type of its *content*. Elemements are also types, types that describe the structure of nodes. It is a common misconception not to consider elements as proper types.

Most people, including the authors of section 5 of the original SOAP specification (http://www.w3.org/TR/SOAP/), believe that the above element declaration for `Point` corresponds to the following class declaration in the object world:

```
class Point {
  public int x;
  public int y;
}
```

In this case, selecting the `x` field of a `Point` instance `p` returns an integer that has no name, parent, or any of the other properties that the "corresponding" element node has in the XML world. Also note that the top level element `Point` is mapped to a corresponding type `Point`, whereas the nested elements `x` and `y` are mapped to fields `x` and `y`, which are not types by themselves.

In general, the different node types such as document, element, attribute, text, processing instructions, comments, and the properties of such nodes such as base-uri, node-name, parent, type, children, attributes, namespaces, nilled as exposed in the XPath data-model do not necessarily correspond to anything in the familiar object data-model. In other words triangles can't easily be made into circles.

Attributes versus elements

In the XML world, there is an important distinction between elements and attributes. That distinction simply does not exist in the object world. For example, consider adding an attribute `<attribute name="Color" type="string"/>` to our example element `Point` above. This becomes an item in the attributes collection of a `Point` element `p`, which is distinct from the children collection and is accessed in XPath using the attribute instead of the child axis.

In the corresponding mapped class, `Color` would simply be represented using another field, no different from `x` and `y`. In particular it is accessed using normal member access; there is no equivalent of the axis. To ensure correct representation the fields denoting attributes should be restricted to primitive (simple) types, but that would require a rather ad-hoc extension to the object data-model. Finally, concepts like global attributes and attribute groups seem difficult to model if we assume that attributes are modelled as class members.

Elements versus complex and simple types

In the XML data-model a distinction is often made between elements and attributes on the one hand and simple and complex types on the other hand. However this is a little misleading because element and attribute declarations actually define types. Perhaps a clearer and more meaningful distinction is that XSD has stratified the world into two parts: elements and attributes to describe the nodes in the data model, and simple and complex types to describe the content of those nodes.

In the object-oriented world, this distinction is not usually made; indeed there is no mechanism to describe the structure of the members of a class declaration in isolation. Hence this is another difference between the two data-models.

Unfortunately this difference leads to further problems with the mapping tools. As we noted earlier global element declarations and complex type declarations are mapped to classes, and local element declarations to field declarations inside these classes. The problem is now how to handle element references; they have no natural counterpart in the object data-model.

```
<element name="Line">
  <complexType>
    <sequence>
      <element ref="Point" />
      <element ref="Point" />
    </sequence>
  </complexType>
</element>
```

The rather clumsy solution chosen by databinding tools is to replace the element references with a copy of the actual element declaration it refers to.

Multiple occurrences of the same child element

In the XML data-model, a complex type is permitted to have several occurrences of the same child element. Consider, for example, the following declaration of a `Verse` element which has two `A` and two `B` children in a specific order:

```
<element name="Verse">
  <complexType>
    <sequence>
      <element name="A" type="string" />
      <element name="B" type="string" />
      <element name="B" type="string" />
      <element name="A" type="string" />
    </sequence>
  </complexType>
```

```
    </element>
```

At this point the mapping of elements to fields breaks down as most object-oriented languages do not allow duplicated fields. The automatic mapping tools generally pick some ad-hoc way out -- for example, the xsd.exe tool will "invent" arbitrary new field names for the duplicate elements. Clearly this is a fragile and unsatisfactory solution.

Anonymous types

XSD allows element declarations to use anonymous types, rather than insisting that they all use named types. This immediately begs the question of how to resolve type equivalence of two elements using anonymous types: is it by structural equivalence or nominal equivalence? Rather depressingly the various XML specifications have not been terribly consistent nor stable on this matter. For example, the May 2003 XQuery and XPath data-model specification implies that every occurrence of an anonymous type is assigned an implementation-defined, globally unique type name.

```
   <element name="Foo">
      <complexType>
        <sequence>
          <element name="Bar" >
            <complexType>
              <sequence>
                <element name="x" type="integer"/>
              </sequence>
          </complexType>
        </element>
        <element name="Baz" >
          <complexType>
            <sequence>
              <element name="x" type="integer"/>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
```

Given the declarations above, the type properties of a `Bar` node and of a `Baz` node would be different as each would be assigned a separate globally unique type name.

From a programming language perspective this is an unusual choice. A more meaningful approach would be to employ structural type equivalence for anonymous types. Of course, if the data-model only supports nominal type equivalence then we are left in a difficult situation when dealing with anonymous types. However, our opinion is that a solution that generates unique type names on the fly is an unsatisfactory one. In that case it is better to disallow anonymous types from the start.

Substitution groups vs derivation and the closed world assumption

XSD supports two forms of "inheritance": substitution groups for elements, and derivation by extension/restriction for complex and simple types. These two forms of inheritance are strangely intertwined where elements can only be in the same substitution group if their content types are related by derivation. This restriction reflects, as we have stated earlier, that complex/simple types describe the content of nodes.

Most object-oriented languages only support what would roughly amount to derivation by extension: a subclass can add new fields and methods. However most languages also permit a subclass to hide or override members of the superclass. This unfortunately has no counterpart in the XML data-model; it is another mismatch.

XSD distinguishes between simple types and complex types. At first glance simple types seem to correspond to primitive types in Java or C#. However, simple types in XSD can be derived from other simple types in order to further restrict the possible values of that type. Unfortunately primitive types in most object-oriented languages are `sealed` (`final`), and cannot be further specialized. The correspondence is again imperfect.

Another fundamental difference between the XML and the object data-models is that the former adopts a closed-world assumption with respect to substitution groups and derivation by extension/restriction. For example the XQuery typing rules assume that all elements of a substitution group are known when performing (static) type-checking of child access [The Essence of XML] . On the other hand an important feature of the object data-model is that it adopts an open-world assumption. When type-checking member access only the static type of the receiver type is assumed.

Namespaces, namespaces as values

In the XPath data-model, namespace nodes exist as first-class values (although subsequently XPath 2.0 has deprecated the use of name-spaces as first class values). This is not the case in the usual object data-model. For example in C# namespaces are purely syntactic sugar; from the point of view of the underlying runtime they don't exist.

In so far as the notion of namespaces exists in C#, the places where namespaces can be introduced, the places where qualified names can be used, and the actual first class status of namespaces, are much more restricted than in XML. For instance in C# or Java field names are never qualified. Again a feature in both models that at the surface appeared related has turned out to be different.

Occurence constraints part of container instead of type

In XSD occurrence constraints appear on element declarations, not on the underlying type of the children of that element. For instance, the element declaration for element `A` below states that `Foo` has at most 10 children named `A`:

```
<element name="Foo">
   <complexType>
     <sequence>
       <element name="A" type="string" maxOccurs="10"/>
       <element name="B" type="string"/>
     </sequence>
   </complexType>
 </element>
```

This is a natural thing to do when you consider elements as types. However, there is nothing that directly corresponds to this in the object data-model given that local element declarations are mapped to fields.

Instead `Foo` would be expected to be mapped to a class that has a single field `A` of type `string[]`:

```
class Foo {
  string[] A;
  string B;
}
```

This means something completely different! Class `Foo` has one member whose type is an array of `string` instances. As well as the obvious mismatch, there is nothing to restrict the size of the array, and `Foo` can have any number of `A` children.

The mismatch becomes even more pronounced once we start putting occurrence constraints on the `sequence`, `choice`, or `all` particles, such as in the example below. There are no simple counterparts in the object data-model.

```
<element name="Foo">
  <complexType>
    <sequence maxOccurs="10">
      <element name="A" type="string"/>
      <element name="B" type="string"/>
    </sequence>
  </complexType>
</element>
```

Mixed content

The idea of mixed content reflects the schizophrenic nature of XML as data versus XML as documents.

```
<element name="Bar">
  <complexType mixed="true">
    <sequence>
      <element name="A" type="integer" maxOccurs="10"/>
    </sequence>
  </complexType>
</element>
```

The children of the element instance `<Bar>aaa<A>0</A>bbb<A>1</A>ccc</Bar>` would be a text node, an element node A, a text node, an element node A and another text node. Since objects represent data not documents, there is no natural interpretation for mixed content in the object world.

The Xen data-model

XML as syntax for object literals

The detailed examples in the previous section demonstrate that at a foundational level there is a significant gulf between the XML and object data-models **[Impedance Mismatch]** . In our opinion the impedance mismatch is too big to attempt a complete integration.

Given these problems, how are we to proceed? Our approach is to first take as our starting point the object data-model of our programming language. This is the model that programmers are familiar with, and that is supported by the underlying execution engine. Rather than trying to blindly integrate the complete XML data-model and the full complexity of schemas, we shall take a simpler design goal: We consider XML 1.0 **[XML 1.0]** as simply syntax for serialized object instances. We shall see in the rest of this paper the significant design advantages that follow from such a simple intuition.

Take the following XML fragment:

```
<Point>
  <x>46</x>
  <y>11</y>
</Point>
```

We consider this to simply denote the XML serialization of an object of type `Point`, say `p`, that is constructed via the series of assignments `Point p = new Point(); p.x = 46; p.y = 11.` Even though the XML fragment is a valid instance of an `Point` element according to the XSD schema we gave earlier, none of the XML data-model properties will be exposed at the level of the programming language; the programmer just sees an instance of class `Point`.

Even though we have adopted a simple view of XML, in fact the object data-model and the corresponding type-system is not powerful enough to handle the rich scenarios that we envisage. In order to consume and generate a wide enough set of documents we need to enrich the type system with new type constructors such as unions, sequences and streams. These have been carefully chosen and designed so that they integrate coherently with the existing type system. Interestingly, the resulting types look very similar to Relax NG compact notation (http://www.oasis-open.org/committees/relax-ng/compact-20020607.html), or DTDs written using XSD keywords.

We define XML fidelity as being able to serialize and deserialize as many possible documents that are expressible by some given XML schema language (http://www.xml.com/pub/a/2001/12/12/schemacompare.html), not as how closely we match one of the XML datamodels in our programming language once we have parsed an XML document. Although Xen by design doesn't support the entirety of the full XML stack, we believe that our type system and language extensions are rich enough to support many potential scenarios. For example we are able to cover the complete set of XQuery Use Cases **[XQuery Use Cases]** , and we have written several large applications, some of them up to 50,000 lines, without running into significant fidelity problems.

Xen type system

In this section we give some details of the Xen type system. Despite its considerable expressive power we have carefully designed this type system as a small, yet coherent extension of the C# type system. For reasons of space we can not describe all of the language in this paper, but it does support the entirety of the C# language.

First we consider Xen class declarations. Instead of having zero or more field declarations inside a class, we allow classes to have one content type and zero or more attributes, in addition to zero or more method declarations as usual. Let `C` denote class declarations, `I` identifiers, `T` types, `A` attribute declarations, and `M` method declarations. A Xen class declaration has the following general form:

```
C ::= class I : I...I { T; A* M*  }
```

An attribute declaration is like a field declaration, but is decorated by the `attribute` keyword

```
A ::= attribute T I;
```

Attributes have no special semantics except for influencing the serialization format of types. In particular we have no special "axis" to distinguish between child and attribute access.

A type is either a sequence type (corresponding to the XSD `<sequence>` particle, the DTD `(...,...)` construct), a choice type (corresponding to a C-style `union`, the XSD `<choice>` particle, or the DTD `(..|...)` construct), an all type (corresponding to the XSD `<all>` particle), a possibly empty stream type (corresponding to the XSD `minOccurs="0", maxOccurs="unbounded"` occurrence constraint, or the DTD `*` construct), a nonempty stream type (corresponding to the XSD `minOccurs="1", maxOccurs="unbounded"` occurrence constraint, or the DTD `+` construct), an option type (corresponding to the XSD `minOccurs="0", maxOccurs="1"` occurrence constraint, or the DTD `?` construct) or a named type that was introduced by a content class. Types are given by the following grammar.

```
T ::= sequence{ ... T I? ; ...}
   |  choice{ ... T I? ; ...}
   |  all{ ... T I?; ...}
   |  T* | T+ | T?
   |  I
```

We use the phrase `T I` *?* to indicate that field names in Xen are optional.

Subtyping relationships between these new types are defined based on their structure and not only by their name. Structural subtyping is nothing new. In a limited form it is already available in C# or Java. Suppose for example, that class `ColorPoint` is a subtype of class `Point`, then an array of `ColorPoints`, (written `ColorPoint[]`) is a subtype of an array of `Points` (written `Point[]`). In the same fashion, in Xen a stream of `ColorPoints` (written as `ColorPoint*`) is a subtype of a stream of `Points` (written as `Point*`). Similar covariance rules apply to the other structural types. It is beyond the scope of this paper to explain Xen's subtyping relationships in greater detail. Instead we refer interested readers to two accompanying papers (**[Unifying Tables, Objects and Documents]** , and **[The Meaning of Xen]**). These papers also give some details on related programming languages that embrace XML such as Xtatic **[Xtatic]** and CDuce **[CDuce]** .

To illustrate both the type system and the syntax of Xen we will take examples from the XQuery Use Cases. This defines a bibliography of books given by the following DTDs:

```
<!ELEMENT bib  (book* )>
<!ELEMENT book (title,(author+ | editor+), publisher, price)>
<!ATTLIST book  year CDATA  #REQUIRED >
```

In fact the DTD for `book` is a nice example illustrating where current object-oriented languages fall short in defining rich content models. For example there is no equivalent notion of union types in Java or C# and no way to indicate that a collection should contain at least one element.

Below we give the Xen class declaration for `books`. It states that a `book` has either at least one `editor` or at least one `author`, a `publisher` and a `price`, and a `year` attribute:

```
public class book {
  sequence{
    string title;
    choice{
      sequence{ editor editor; }+;
      sequence{ author author; }+;
    }
    string publisher;
    int price;
  }
  attribute int year;
}
```

Object literals

When serializing values of type `book` as XML, they will conform to the given DTD for `book`. However, why would you only use XML serialization outside the language? Xen internalizes XML serialized objects into the language, allowing programmers to use XML fragments as object literals. For instance, we can create a new instance of a `book` object using the following XML object literal:

```
book b = <book year="2004">
           <title>C# Concisely</title>
           <author>
             <first>Judith</first><last>Bishop</last>
           </author>
           <author>
             <first>Nigel</first><last>Horspool</last>
           </author>
           <publisher>Addison-Wesley</publisher>
           <price>68.00</price>
         </book>;
```

The Xen compiler contains a validating XML parser that analyzes the XML literal and "deserializes" it at compile time into code that will construct the correct book instance. This allows Xen programmers to treat XML fragments as first-class expressions in their code.

Embedded expressions

In addition to first-class XML object literals, Xen also allows arbitrary embedded code, using curly braces as escape syntax (where curly braces themselves are escaped using `\{` and `\}` respectively). As Xen is statically typed, the type of the embedded expression must be of an "allowable" type (see section **The Xen validation rules** for more details). In the following example we splice in the embedded expression `LookupPrice("C# Concisely")` to construct the price of our book.

```
  book b = <book year="2004">
             <title>C# Concisely</title>
```

```
                 <author>
                   <first>Judith</first><last>Bishop</last>
                 </author>
                 <author>
                   <first>Nigel</first><last>Horspool</last>
                 </author>
                 <publisher>Addison-Wesley</publisher>
                 <price>{LookupPrice("C# Concisely")}</price>
               </book>;
```

Of course embedded expressions can be used for attributes also, so we could use `<book year={e}/>` in the example above provided `e` is some expression of type `int` that computes the year of our book.

The validation rules in Xen allows us to write particularly concise code when dealing with embedded expressions. Consider the following declaration of a `book` whose authors are computed by an embedded expression:

```
book b = <book year="2004">
             <title>C# Concisely</title>
             { authors }
             <publisher>Addison-Wesley</publisher>
             <price>68.00</price>
           </book>;
```

According to the content type of `book`, the expected type for the embedded expression `{ authors }` is

```
choice{
  sequence{ editor editor; }+
  sequence{ author author; }+
}
```

We'd like to be able to use the following declaration for `authors`.

```
author* authors = <author>
                   <first>Judith</first><last>Bishop</last>
                  </author>
                  <author>
                    <first>Nigel</first><last>Horspool</last>
                  </author>;
```

The variable `authors` has type `author*`. This type is considered to allowable instead of the expected type according to the Xen validation rules because the set of documents described by the type `author*` is a subset of the set of documents described by the original union type. Thus the declaration is permitted with no explicit type conversions needed.

The Xen validation rules

The validation rules are defined using a relation *is allowable* between types. Let `S` and `T` be two types. We say that `S` is allowable for `T` if the set of serialized XML documents of type `S` is a subset of the set of serialized documents for type `T`. It is the case that if `S` is implicitly convertible to `T` then `S` is allowable for `T`, but not necessarily the other way around. For instance the serialized values of type `sequence{ A A }*` are the same as the serialized values of the type `A*`, but the type `sequence{ A A }*` is not implicitly convertible to the type `A*`.

We do not put any restrictions on our class declarations and in particular we do not impose any non-ambiguity constraints. For example the following class declaration `A` is perfectly legal in Xen, even though it is obvious that the content model is ambiguous:

```
class a { int; }

class As {
  sequence{
    choice{
      sequence{ a a; a a; }
      sequence{ a a; }
    }
    choice{
      sequence{ a a; a a; }
      sequence{ a a; }
    }
  }
}
```

The validator will attempt to validate a given object literal in all possible ways, and only flag an error if it did not find exactly one parse. When using ambiguous context-free grammars this is similar to the idea of using a general purpose parser such as GLR or Early [ASF/SDF] and demanding that each particular parse is unambiguous.

In this case, the literal below is ambiguous, and the validator will generate a compile-time error

```
As a = <As><a>1</a><a>2</a><a>3</a></As>;
```

To disambiguate we can use embedded expressions and write

```
As a = <As>{<a>1</a><a>2</a>}<a>3</a></As>;
```

to indicate to the validator that we intend the first choice to be the two `a`'s and the second just one `a`. Of course we should be careful with ambiguous content models when values have to be serialized to the outside world where we cannot use embedded expressions.

New syntax

In the previous sections we have seen how we can construct object instances using XML literals. However, for a language that processes XML it is just as important to be able to deconstruct, or more generally query values. For this purpose, we will introduce path expressions in Xen. These are inspired by both XPath [XPath] and SQL, but adapted to our extended object data-model.

Generators and Iterators

Streams of values play an important role in many XML content types, for instance a `book` can have one or more `authors` or `editors`. In Xen stream types `?`, `+`, and `*` all implement the `IEnumerable` interface and hence we can iterate over any stream using a `foreach` loop:

```
    book b = ...
    author* authors = b.author;
    foreach(author a in authors) Console.WriteLine(a);
```

Note that even though the content type of `book` says that a `book` has one or more `authors`, the result type of `b.author` is `author*` because it could be that the `book` has `editors` not `authors`. (To be precise, the return type of `b.author` is `author+?`, which is according to Xen's type equivalences the same as `author*`. See our companion papers [Unifying Tables, Objects and Documents] and [The Meaning of Xen] for a more detailed account of Xen's type system.

Whilst it is easy to traverse streams using `foreach` loops, it is still a little verbose compared to XPath. In the next couple of sections we discuss some of the features that we have added to Xen to make iteration over streams even simpler. Before describing them we first need to tackle another problem, namely how to make it easy to *generate* streams.

The most convenient way to generate finite streams is to use the fact that the Xen type system defines an implicit conversion between sequences and streams. For example Xen considers `sequence{ Author; Author;}` to be a subtype of `Author*`. The following code is then type correct (where `new(..)` is the Xen sequence constructor).

```
  author* authors = new(author1, author2);
```

Of course, this only works for finite streams; it does not help us to generate more complex or infinite streams. For this we introduce `yield` statements. A method body that contains `yield` statements can "return" multiple times, each time yielding the next element of a stream. For example the method `FromTo(n,m)` below generates a stream of integers *n, n+1, ..., m* by yielding *(m-n)+1* times:

```
  // FromTo(n,m) ==> n, n+1, ...,m
  int* FromTo(int n, int m){
    for(int i=n; i=<m; i++) yield i;
  }
```

The concept of methods that return multiple times is also available in XQuery, which (confusingly) uses the `return` keyword instead of `yield`. It is also supported by several other programming languages including Icon, Clu, and Python, and will be in the next version of C#.

It turns out that especially for generators it is convenient to allow statement blocks as expressions. So for instance instead of having to define a separate method `FromTo` we can just use the block expression `{ for(int i=n; i=<m; i++) yield i; }` of type `int*`.

Lifting

The most important difference between path expressions and ordinary member selection as found in object-oriented languages is the way path expressions lift (homomorphically extend) over structural types such as streams, sequences, and unions.

Suppose that we have a `bibliography` that contains zero or more `books` as its content model.

```
public class bibliography {
  book* books;
}
```

and that given an instance `bib` of type `bibliography` we want to select all the `titles` of all the `books`. The most natural way to write this query is by using the following path expression:

```
string* titles = bib.books.title;  // Xen code
```

Note that the result type of the sub-expression `b.books` is `book*`. In C# or Java, the subsequent `title` member access would not be allowed as it would be assumed to be on the stream itself, rather than the elements of the stream. In Xen the notion of member access has been generalized to support this style of programming. Member access is defined to be lifted (mapped) over the stream and so the code above is valid and behaves as expected. In contrast we would have to write the following C# code to achieve the same effect.

```
string* getTitles(books bs){
  foreach(book it in bs) yield it.title;
}

string* titles = getTitles(bib.books); // C# code
```

Note how convenient the Xen code `b.books.title` is in comparison to the somewhat clumsy C# code. This is an example of where promoting XML access to the language level has allowed us to provide elegant support for a typical programming pattern.

A similar situation happens when we have an instance `b` of a choice type say `choice{ string; Button; }`. The most natural way to select the `BackColor` field of `b` is to write `b.BackColor`, which either returns `null` if `b` happens to be of type `string` or a value of type `Color` in case `b` is indeed of type `Button`. This is precisely what happens in Xen. Without such lifting we would be forced to define a new method that performs an explicit instance check to determine the actual type of `b`:

```
Color? getBackColor(choice{string; Button; } b){
  if(b is Button) return ((Button)b).BackColor;
  return null;
}
```

and then use `getBackColor(b)` in place of Xen's more elegant `b.BackColor`.

Finally, it nearly goes without saying that lifting of member access over sequence types is also extremely convenient. In fact, Xen automatically lifts member-access (including method calls) over all its structural types. As we have seen this yields elegant and compact code that is familiar to the XPath programmer.

Filtering

Besides lifting, filters are another example of an XPath feature that is indispensable for writing concise code. Again we have promoted this vital programming pattern to a first-class language feature. Given a stream expression `e`, the Xen expression `e[p]` denotes the stream resulting from applying the filter `p` to each element. In the boolean predicate `p` the stream element can be referenced using the implicit variable `it`.

For example given a `bibliography` of books, `bib`, we can select all books whose publisher is Addison-Wesley that are published after 1991 by the following filter expression.

```
    book* AWbooks =
      bib.book[it.publisher == "Addison-Wesley" && it.year > 1991];
```

Apply-to-all

Often we want to lift operations other than member-access and method invocation over a structural type. For example, suppose we want to collect the title and year of all books, or we want to print out all the titles of the books. Again we have promoted this common programming pattern to a first-class language feature; the apply-to-all block.

Given a stream expression `e` and a code-block `{ s }` where `s` is a Xen statement, the apply-to-all expression `e.{ s }` generates a stream resulting from executing the code-block for each element in the original stream (again `s` can contain the identifier `it` to denote the current stream element).

To collect all titles and publication dates of a stream of Addison-Wesley books (henceforth called `AWbooks`) we apply the method body `{ return new(it.title, it.year); }` to the `AWbooks` stream as follows.

```
  sequence{string; int;}* bs
   = AWBooks.{return new(it.title, it.year);};
```

In this case the return type of the apply-to-all block is `sequence{ string; int;}`. The result of the whole expression is a stream of type `sequence{ string; int;}*` that is built lazily.

Apply-to-all blocks need not just be used to construct new streams. Printing out all the titles of the `AWbooks` books is simply a one liner:

```
  AWbooks.{ Console.WriteLine(it); }
```

Note that the return type of `{ Console.WriteLine(it); }` is `void`; therefore the result of the whole expression is `void`. Obviously `void` results cannot be consumed, thus it makes no sense to postpone the evaluation of `void` apply-to-all blocks. As a consequence, Xen evaluates the effect of `void` apply-to-all blocks eagerly.

XQuery use cases

At this point we have described enough of the features of Xen to be able to express all the XQuery Use Cases [XQuery Use Cases] . Due to lack of space we will restrict ourselves to just a few of the more interesting ones in this paper.

The first use case was covered earlier. It asks to list books published by Addison-Wesley after 1991, including their year and title. We solved it using a simple filter expression.

```
    book* AWbooks =
      bib.book[it.publisher == "Addison-Wesley" && it.year > 1991];
```

The second use case asks to create a list of all the title-author pairs, with each pair enclosed in a `result` element. In other words, the output is a stream of result elements where result is defined by the type `result`:

```
  class result {
    sequence{
      string title;
      author author;
    }
  }
```

This query can be expressed in several different ways, but the most concise one uses two nested apply-to-all blocks. For each `book` in the bibliography it remembers the `title`, and then for each `author` of the book, the nested apply-to-all block generates the required result element:

```
  result* authors_and_titles =
    bib.book.{ title = it.title;
               return it.author.{
                 <result>
                   <title>{title}</title>
                   <author>{it}</author>
                 </result>;}
             };
```

Conceptually, the return type of the query is a nested stream `result**`. Similar to XQuery, this is the same as `result*` according to the Xen typing rules.

The fifth XQuery use case is interesting because it requires a join (in the sense of relational algebra) of results from two sources. This is where the rectangles of the title finally come into the picture.

The reader familiar with relational databases may have already realized that the Xen data-model subsumes the (nested) relational model. For example the SQL declaration:

```
  CREATE TABLE Customer
   ( name string NULL,
     custid int);
```

corresponds to the following Xen declaration of a stream of sequence types:

```
sequence{ string? name; int custid; }* Customer;
```

In addition to the path queries that we have used up to now, Xen also supports first-class SQL-style select statements. SQL-style select expressions work on any stream, not only those that originate from an actual relational database.

We can now handle the fifth use case, which asks to list for each book found at both booksellers `A` and `BN`, the `title` of the book and its `price` from each source. So the first task is to get the list of all books that are available at both `BN` and `A`. In SQL this is easily done using an (inner) join. Using a `select` statement, we can easily select the `A` and `BN` books whose titles are the same, and produce a stream of the requested information:

```
A_BN =
  select
    <book-with-prices>
      <title>{a.title;}</title>
      <price-A>{a.price}</price-A>
      <price-BN>{bn.price}</price-BN>
    </book-with-prices>
  from
    a  in Abooks.book,
    bn in BNbooks.book
  where
    a.title == b.title
```

The XQuery use case actually uses a different schema for the two shops.

Future Work

In the introduction we mentioned that besides data access, we believe that concurrency and security are the two other obvious candidates for promotion to full language support. The current story for both concurrency and security is as disheartening as it is for data access. Languages such as C# or Java provide only crude support for concurrent programming. Their `lock(x)s` statement is (roughly) just syntactic sugar for `Monitor.Enter(x); try{s}finally{Monitor.Exit(x);}`, but apart from that the programmer is left unsupported and has to explicitly create and start threads, maintain reader/writer locks, mutexes, apartment states, lock cookies, etc.

Unfortunately today's applications are increasingly dealing with complicated concurrent scenarios; for example, complex orchestration code for webservices. Clearly we need better, high-level support for concurrency and security to facilitate writing of such code. Interestingly there are several proposals for orchestration languages that are based on an abstract model of concurrent communication called the Pi-calculus **[Pi Calculus]** ; examples include BPEL4WS **[BPEL4WS]** and WSCI **[WSCI]** .

Both these languages unfortunately use XML for their concrete syntax and lack many of the other desired features of a concrete programming language. An interesting experiment is to add support for join patterns (the join-calculus **[Join Calculus]** is a programming oriented dialect of the Pi-calculus) to Xen, based on the Polyphonic C# proposal **[Polyphonic C#]** . The resulting language begins to approach the kind of programming language that we envisage is necessary to program advanced webservices.

Conclusions

In this paper we have argued that by generalizing the type system and language syntax, it is possible for a modern object-oriented (*circles*) language to provide first-class support for manipulating both relational (*rectangles*) and hierarchical data (*triangles*) in a sound and statically typed manner. We have demonstrated this by describing Xen, our hypothetical extension of C#.

Our approach is to build on top of current object data-models and in particular we do not try to simply bolt-on naively the entire XML data model or XSD "type-system". We have argued carefully in this paper why such an approach is hampered by serious incompatibilities between the data-models. In contrast we consider XML 1.0 basically as syntax for serialized object graphs that can be used inside the language as a convenient way to denote object instances, or externally as a wire format for data interchange. In the latter case we assume that the other end will have some independent way to describe the set of allowable documents.

Acknowledgements

We should like to acknowledge the hard work of the WebData language team consisting of William Adams, Joyce Chen, Kirill Gavrylyuk, David Hicks, Steve Lindeman, Chris Lovett, Frank Mantek, Wolfgang Manousek, Neetu Rajpal, Herman Venter, and Matt Warren, and the MSR Cambridge Programming Principles and Tools group, in particular Luca Cardelli, Nick Benton, Claudio Russo, Cedric Fournet and Alain Frisch. Special thanks to Chris and Steve for the original version of the XQuery Use Cases, and to Dare Obasanjo for his insightful comments. Gavin Bierman was supported by EU projects Appsem II and Pepito.

Bibliography

[C# Specification]
http://download.microsoft.com/download/0/a/c/0acb3585-3f3f-4169-ad61-efc9f0176788/CSharp.zip

[Fournet and Abadi]
Cédric Fournet and Martin Abadi. *Access Control based on Execution History*. Proceedings of the 10th Annual Network and Distributed System Security Symposium. February 2003. http://research.microsoft.com/~fournet/papers/access-control-based-on-execution-history-ndss.pdf

[Java Language Specification]
http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html

[JDBC]
http://java.sun.com/products/jdbc/

[ADO .Net]
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconoverviewofadonet.asp

[xsd.exe]
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpconxmlschemadefinitiontoolxsdexe.asp

[JAXB]

http://java.sun.com/xml/jaxb/

[XMLBeans]
http://dev2dev.bea.com/technologies/xmlbeans/index.jsp

[Data On The Web]
S. Abiteboul, P. Buneman and D. Suciu. *Data On The Web*. Morgan Kaufmann, 2000.

[XPath]
http://www.w3.org/TR/xpath

[XML 1.0]
http://www.w3.org/TR/REC-xml

[XQuery Use Cases]
http://www.w3.org/TR/xquery-use-cases/

[Impedance Mismatch]
Dave Thomas. *The Impedance Imperative Tuples + Objects + Infosets = Too Much Stuff!*. JOT, 2003 Vol. 2, No. 5, September-October 2003. http://www.jot.fm/issues/issue_2003_09/column1.pdf

[The Essence of XML]
Jerome Simeon and Philip Wadler. *The Essence of XML*. Proceedings of POPL 2003, New Orleans, January 2003. http://homepages.inf.ed.ac.uk/wadler/topics/xml.html#xml-essence

[ASF/SDF]
http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment

[Unifying Tables, Objects and Documents]
Erik Meijer and Wolfram Schulte. *Unifying Tables, Objects and Documents*. Proceedings DP-COOL 2003. http://www.research.microsoft.com/~emeijer/Papers/XS.pdf

[The Meaning of Xen]
Erik Meijer, Wolfram Schulte, and Gavin Bierman. *The Meaning of Xen*. Paper in preparation.

[Xtatic]
http://www.cis.upenn.edu/~bcpierce/xtatic/

[CDuce]
http://www.cduce.org/

[BPEL4WS]
ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf

[WSCI]
http://www.w3.org/TR/wsci/

[Pi Calculus]
R. Milner. *Communicating with Mobile Agents: The pi-Calculus*. Cambridge University Press, Cambridge, 1999.

[Join Calculus]
http://pauillac.inria.fr/join/

[Polyphonic C#]
Nick Benton, Luca Cardelli and Cedric Fournet. *Modern Concurrency Abstractions for C#*. In B. Magnusson (Ed.), Proceedings ECOOP 2002. LNCS 2374, Springer-Verlag. http://research.microsoft.com/~nick/polyphony/PolyphonyECOOP.A4.pdf