XQueryP: An XML Application Development Language

Don Chamberlin, IBM Almaden Research Center
Michael J. Carey, BEA Systems
Mary Fernández, AT&T Labs
Daniela Florescu, Oracle
Giorgio Ghelli, University of Pisa
Donald Kossmann, ETH
Jonathan Robie, DataDirect Technologies
Jérôme Siméon, IBM T.J. Watson Research Center

Abstract

The three main parts of a typical business application are storage, logic, and presentation. XML is playing an increasing role in storage (using XML databases) and presentation (using AJAX and similar web technologies). Application logic, however, still relies to a large extent on traditional programming languages such as Java. Operating on XML data with a traditional programming language leads to a serious "impedance mismatch" because the type system of the programming language is not based on XML. This paper explores ways of eliminating this impedance mismatch by extending XQuery, an XML query language, to implement business logic. This approach can potentially support XML-based applications with a unified type system and programming environment.

1. Introduction

The advantages of XML as a storage and exchange format for semi-structured information are well known. Increasingly, modern business applications use XML to represent data that is heterogeneous, or sparse, or has an intrinsic order. These applications are often distributed over the Web, exchanging XML data in the form of SOAP messages. XML is also often used by business applications to store persistent data such as contracts, medical records, and incident reports. Examples of XML-intensive applications include:

- Implementation of web services
- Data transformation and integration of heterogeneous data sources
- Processing RSS feeds and other message streams
- Coordination of services in an SOA environment
- Data cleansing or normalization
- Complex manipulations of persistent XML data

Today, the actual business logic of a typical XML application is still usually written in a traditional programming language such as C or Java. XML data from other applications or from a database must be transformed into programming language objects before it can be operated on by the business logic. Results of computations must be transformed back into XML in order to be exchanged with other applications, where the process is often repeated. These transformations add complexity to applications and inhibit global optimization. The resulting loss of efficiency is generally referred to as "impedance mismatch."

The impedance mismatch problem has existed in the relational database world for many years. One historically successful approach to this problem was to extend SQL with control logic, resulting in stored procedure languages such as PL/SQL [1], Transact-SQL [2], and SQL/PL [3]. These procedural versions of SQL have been used extensively in commercial applications of

relational databases such as salesforce.com [4], and a version called PSM [5] was adopted as an ANSI/ISO Standard in 1996.

In this paper, we explore the possibility of extending XQuery to implement the business logic of XML-based applications, much as SQL has been extended for relational applications. XQuery operates on XML in its native data model, using the type system of XML Schema [6]. Like SQL, XQuery is declarative and functional, which makes it well-suited for automatic optimization. The considerations that led to the extensions of the SQL language apply equally to XQuery. However, the case for extending XQuery in a similar way is perhaps stronger due to the large mismatch between programming language type systems and XML Schema, which includes features such as complex types, attributes, multiple kinds of inheritance, and recursive hierarchies.

The principal features added to SQL by PSM are as follows:

- The notion of sequential execution of statements, with the side effects of each statement visible to the next statement
- Assignment statements that allow an application to maintain state information during processing.
- A "compound statement" consisting of some local variable declarations and an ordered sequence of statements, connected by semicolons; the compound statement can optionally be declared "atomic".
- Ability to associate "condition handlers" with a compound statement, for intercepting and handling exceptional conditions
- Control flow statements for conditional branching, iteration, early exit from loops, etc.
- A SIGNAL statement to raise an exception
- Enhancements to function definitions, including a RETURN statement and declarations of DETERMINISTIC and NONDETERMINISTIC properties

Similar features are needed to support application development in XQuery. Some of these features (such as iteration, conditional branching, and raising an exception) are already defined in XQuery 1.0, and others are specified in this paper.

As a starting point for designing an XML application development language, we begin with XQuery 1.0, which is currently a W3C Proposed Recommendation [7] and is expected to become a Recommendation early in 2007. We also base our work on the XQuery Update Facility, currently published as a W3C Working Draft [8]. We refer to the combination of these two specifications as XQuery with Update. Our design goal is to find a small set of upward-compatible extensions to XQuery with Update that can enable the development of XML-based applications. In an earlier paper, we outlined a set of procedural extensions to XQuery with Update, called XQueryP [9]. This paper expands on the XQueryP proposal, adding more details, additional features, and some examples that illustrate use of the extended language in various environments. As in the earlier paper, we assume a basic familiarity with the concepts, syntax, and terminology of XQuery with Update.

This paper does not describe a finished language design, but a work in progress, published to encourage dialog. For this reason, we introduce new language features using illustrative examples rather than a formal syntax. Although some of the authors are members of the XML Query Working Group, this paper represents the work of the individual authors and not a position of the Working Group.

2. Sequential Execution, Blocks, and Assignments

XQuery, like SQL, is a functional language in which data flows from one expression to another by function calls and returns, without side effects. XQuery 1.0 has no notion of state, and no

defined order for the evaluation of expressions except where one expression serves as an operand of another. In XQuery with Update, a given query may contain several updating expressions, but each of these expressions operates on the data in its initial state, and all updates are held in a pending list and applied together at the end of query processing.

Most business applications, on the other hand, are written in a more imperative style, in which a state is maintained in the form of a set of variables to which values can be assigned. Expressions are evaluated in a well-defined order, and the side effects of each expression are visible to the next expression to be evaluated. Our earlier paper [9] proposed a set of XQuery extensions to support this programming paradigm, consisting of the following basic features:

- Sequential execution mode: In order for side-effecting expressions to have a deterministic result, a well-defined order must be imposed on their execution. Since side-effecting expressions (such as insert, delete, and replace) can be nested inside control-flow expressions (such as for-return and if-then-else), the execution order must apply to all kinds of expressions. In [9], this was accomplished by defining a "sequential execution mode," declared in the application prolog. In sequential execution mode, for example, each iteration of a for-expression is evaluated and its side effects are made effective before the next iteration begins. The other features listed below rely on a well-defined execution order and are defined only in sequential execution mode.
- Blocks, which permit the declaration of local variables and contain a sequence of expressions separated by semicolons. Blocks are defined mainly for consistency of style with popular procedural languages.
- Assignments, which are side-effecting expressions that assign values to variables. Assignment expressions maintain a state consisting of all the variables that are in scope during the evaluation of a given expression.
- Removal of the restriction against mixing updating with non-updating expressions. A block, for example, is permitted to update a value and to return the updated result. However, certain restrictions are retained on where updating expressions can appear. For example, updating expressions cannot be used in predicates (e.g., in an if-clause or a where-clause), as operands of arithmetic expressions, or as operands of other updating expressions. XQueryP does allow an updating expression on the right-hand-side of an assignment (this is a correction to [9].)

The following example, taken from [9], illustrates a block containing a local variable and an assignment:

```
declare execution sequential;
{ declare $total-cost as xs:decimal := 0;
    for $p in /projects/project[year eq 2005]
    return
       {   set $total-cost := $total-cost + $p/cost;
           <project>
                <name>{$p/name}</name>
                <cost>{$p/cost}</cost>
                <cumulative-cost>{$total-cost}</cumulative-cost>
           </project>
        }
}
```

Sequential execution mode is declared at the level of a module and applies to all the functions declared in the module. In non-sequential mode, an updating function may return a pending

update list. If such a function is called from an expression that is executing in sequential mode, all the pending updates of the function are applied at the time the function returns.

3. Control Flow

XQuery 1.0 defines constructs for conditional branching (if-then-else) and iteration (the for-clause of a FLWOR expression). XQueryP introduces several additional kinds of expressions that are commonly found in procedural languages, listed below. All these expressions are valid only in sequential execution mode.

- A `while` expression, which executes its body repeatedly as long as a test expression is true
- A `break` expression, which terminates the nearest enclosing FLWOR expression or `while` expression
- A `continue` expression, which advances to the next iteration of the nearest enclosing FLWOR or `while` expression
- A `return` expression, which terminates execution of a function body and returns a result

The following example illustrates the `while`, `break`, and `return` expressions. The example consists of a function named `local:related-terms` that returns a bounded list of terms that are related to a given term according to a thesaurus.

Suppose that the variable `$thesaurus` is bound to a tree of elements with the following structure:

```
<thesaurus>
     <entry>
        <term>prolific</term>
        <related-term>abundant</related-term>
        <related-term>copious</related-term>
        ... (more related-terms)
     </entry>
     ... (more entries)
  </thesaurus>
```

The function call `local:related-terms($input-term, $limit)` returns a list of up to `$limit` terms that are related to `$input-term`, either directly or indirectly. For example, the function call `local:related-terms("prolific", 5)` might return the following list:

```
<term>abundant</term>
<term>copious</term>
<term>profuse</term>
<term>plentiful</term>
<term>ample</term>
```

The definition of the function is as follows:

```
declare function local:related-terms
   ($input-term as xs:string, $limit as xs:integer)
      as element(term)*
   {
   declare $candidate-terms, $candidate, $output-list;
   set $output-list := ();
   set $candidate-terms := $input-term;
   (: the loop ends when the list contains $input-term + $limit
      related terms, or when no more candidates are found :)
```

```
while (fn:count($output-list) < $limit + 1)
   {
   if (fn:empty($candidate-terms)) then break()
   else
      {
      set $candidate := $candidate-terms[1];
      set $candidate-terms := $candidate-terms[fn:position()>1];
      if (fn:not($candidate = $output-list)) then
         {   (: this is a new output term :)
         set $output-list := $output-list, $candidate;
         set $candidate-terms := ( $candidate-terms,
             $thesaurus/entry[term = $candidate]/related-term )
         }
      else ()   (: candidate is already on the list :)
      }
   }
   return ($output-list[fn:position()>1])   (: remove $input-term :)
}   (: ends body of local:related-terms :)
```

4. Exception Handling

For each operator that constructs a value of a particular type, most programming languages provide an inverse operator for deconstructing a value of that type. For example, in XQuery, item sequences are constructed with the "," operator and are deconstructed with the "for" expression. Element values are constructed by the two kinds of element-constructor expression and deconstructed with a path expression. This symmetry is a principle of good language design, and is necessary for proving important properties of the language such as compositionality.

In XQuery, all types of values have symmetric constructor and deconstructor operators, with the exception of the error value. An error value has three parts: a QName error code, an optional string description, and an optional value that can be used to encapsulate query state when the error is constructed. Many XQuery expressions may construct, or "raise", an error. For example, the error `err:XPTY0004` is raised by the "cast" expression when an atomic value cannot be cast to a given type. User-defined errors can be constructed by calling the `fn:error` function, as shown in this example:

```
fn:error(err:USER0005, "Value out of range", $value)
```

No XQuery expression can take the error value as input--therefore when an error is raised, control flow returns to the application environment. The ability to detect and recover from errors is a requirement for any language intended for large-scale application development.

To satisfy this requirement, XQueryP provides a "try-catch" expression for catching, deconstructing, and recovering from dynamic errors. A try-catch expression consists of a "try clause" enclosing a target expression, zero or more "catch clauses", and one "default clause". The default clause is optional if there is at least one catch clause.

Each catch clause contains a NameTest (for identifying a type of error), three optional variables (to be bound to the three parts of the error), and a return expression (for handling the error). The default clause has a similar structure, except that it has no NameTest because it applies to all types of errors.

The parts of a try-catch expression can be informally illustrated as follows. (Identifiers beginning with $ represent variable-names, and identifiers in *italics* represent general expressions. The as-clause is optional and, if present, may contain one, two, or three variables. A single variable is interpreted as $name, and two variables are interpreted as $name, $desc.)

```
try ( target-expr )
catch ( NameTest1 as $name, $desc, $value ) return handler-expr1
catch ( NameTest2 as $name, $desc, $value ) return handler-expr2
...
default ( $name, $desc, $value ) return general-handler-expr
```

The try-catch expression first evaluates its target expression. If the target expression does not raise an error, then the value of the try-catch expression is the value of the target expression. If the target expression does raise an error, the first catch clause is considered. If the error "matches" this clause, as defined below, the catch clause deconstructs the error value, evaluates its "return" expression, and returns the resulting value. If the error does not match, each subsequent catch clause is evaluated similarly, in order. If no catch clause matches the error, the try-catch expression re-raises the error.

An error matches a catch clause if the error's QName code matches the clause's NameTest as defined in XQuery 1.0 [see http://www.w3.org/TR/xquery/#node-tests]. If an error is matched, the optional variables in the catch clause are bound, in order, to the error's code, description, and value. These variables are then in scope in the return expression of the catch clause, just as the variables in a case clause of a typeswitch expression are in scope in the case's return expression.

The default clause does not include a NameTest; thus it matches all errors. Like a catch clause, it deconstructs an error by binding its optional variables to the error's code, description, and value, evaluates its return expression, and returns the resulting value.

An important use of try-catch is to recover from internal XQuery errors. For example, the following expression catches error `err:XQTY0024`, which is raised if the content sequence of an element constructor contains an attribute node following a node that is not an attribute node.

```
let $x := Expr
return
   try ( <a>{ $x }</a> )
   catch (err:XQTY0024)
      return
         <a>
           {$x[self::attribute()], $x[fn:not(self::attribute())]}
         </a>
```

Assuming that Expr returns a sequence containing attributes and other nodes in mixed order, the above expression recovers from this error by first selecting the attribute nodes in the sequence, followed by the remaining nodes.

Recovering from internal errors is so important that XQuery provides a special expression for this purpose. The "castable" expression checks whether an atomic value can be cast to a given type, in order to avoid an error when "cast" is evaluated. The following expression illustrates a common use of "castable":

```
if (Expr_1 castable as AtomicType)
then Expr_1 cast as AtomicType
else Expr_2
```

The above idiom can be expressed more concisely with try-catch:

```
try ( Expr_1 cast as AtomicType )
catch (err:XPTY0004) return Expr_2
```

More importantly, try-catch can be used to detect and handle *any* error, including application-defined error codes that represent exceptional conditions such as values that are outside an expected range.

As a last example, we can use try-catch together with `fn:trace` to report useful error messages before re-raising the errors to the application environment. Assume that the processing environment has a variable that maps XQuery error codes to descriptive messages:

```
declare variable $allerrs :=
  <errors>
     <error code="err:FOCA0001">
        Input value too large for decimal
     </error>
     ...
  </errors>
```

The following expression catches any XQuery error raised in the target expression, finds the error's corresponding message, reports the error by calling `fn:trace`, then re-raises the error with the descriptive text:

```
try (Expr)
catch (err:* as $code) return (
  let $msg := $allerrs[@code=$code]
  return
  (fn:trace((), fn:concat($code, " ", $msg)),
   fn:error($code, $msg))
)
```

5. Building Data Structures with References

Most application development languages provide a means to combine primitive objects into structures. In XQuery, this functionality is provided by the element constructor expression, which creates a new element whose content is a sequence of objects.

Constructor expressions are different from other XQuery expressions in that they make copies of their operands. All other kinds of XQuery expressions return nodes by reference rather than by copy. To illustrate this point, consider the following:

```
let $x := $order/item[1]
return $x/..
```

The let clause binds `$x` to a reference to an item element. The path expression in the return clause uses this reference to find the original element and navigates to its parent. The result of the expression is the same node that is bound to `$order`.

In contrast, consider the following constructor:

```
<myPurchase> {$order/item[1]} </myPurchase>
```

The content of the constructed node is a copy of the selected item node. The copy has a new nodeid and a different parent. If a path expression were to navigate to the parent of the copied item node, that parent would be the constructed `<myPurchase>` element rather than the node that is bound to `$order`.

The behavior of constructor expressions in making copies of their operands has the side effect of losing some of the properties of the original node, such as its identity and its parent. Occasionally it is desirable to create a data structure containing multiple nodes while preserving the identities and parent properties of all the nodes. For this purpose, XQueryP provides two new functions called `fn:ref` and `fn:deref`.

The first function, `fn:ref`, takes a node as its argument and returns a reference to the node, in the form of a URI. The URI returned by `fn:ref` can be used as the operand of the inverse

function, `fn:deref`, which dereferences the URI and returns the original node. The URI returned by `fn:ref` need not be meaningful to any process other than the `fn:deref` function. A URI representation is chosen for the result of `fn:ref` in order that query results can be serialized. Some implementations may choose to use XPointer conventions for URIs that represent references.

The following example returns a list of all the toasters found in a catalog. Each toaster is represented by a `<product-listing>` element containing a reference to the original product element found in the catalog, together with an average rating computed for the toaster by scanning reviews found in a separate document.

```
for $p in fn:doc("catalog.xml")//product[type="Toaster"]
let $ar := fn:avg(fn:doc("reviews.xml")
              //item[sku = $p/sku]/review/rating)
order by $ar descending
return
   <product-listing>
      <link uri="fn:ref($p)">{$p/name}</link>
      <avg-rating> {$ar} </avg-rating>
   </product-listing>
```

The `fn:deref` function can be used with the result of this query to retrieve one of the original product elements and to navigate to its parent in the catalog document. For example, if `$Q` is bound to the result of the query above, the following expression returns the parent element of the highest-rated toaster:

```
fn:deref($Q[1]/link/@uri)/..
```

The `fn:ref` and `fn:deref` functions provide a flexible mechanism for constructing various kinds of data structures. The following example illustrates how a set of functions might use `fn:ref` and `fn:deref` to implement a stack:

```
declare function stack:create()
   { <stack/> };
declare updating function stack:push($x,$s)
   { let $r := ref($x)
     return do insert <item>{$r}</item> as last into $s };
declare function stack:peek($s)
   { deref($x/item[last()]) };
declare updating function stack:pop($s)
   { let $r := $x/item[last()]
     return (do delete $r, deref($r)) };
```

6. Invoking and Coordinating Web Services

In order to facilitate the development of Web applications, XQueryP provides the ability to access web services using a service import declaration in the prolog, in the spirit of [10]. This import takes advantage of the natural correspondence between web services and XQuery modules.

As background, an XQuery module definition groups together a set of global variables and functions, which can then be imported and used in another module. The module definition declares a namespace which is shared by the variables and functions in the module. The following example defines a module containing a global variable and a function in the namespace `http://addr.example.net`, abbreviated by the prefix `addr`.

```
module namespace addr = "http://addr.example.net";
declare variable $addr:book as document-node() := doc("mybook.xml");
declare function addr:getContact ($n as xs:string) as element()
       { $addr:book//entry[@name=$n] };
```

An application can gain access to the variables and functions of a module by means of a module import in its prolog. For example, the following code imports the module defined above and invokes its `addr:getContact` function:

```
import module namespace addr = "http://addr.example.net";
addr:getContact("Andrew Eisenberg")
```

In addition to module imports, XQueryP supports service imports, which allow an application to import a web service as though it were an XQuery module. Because XQuery and WSDL-based web services both rely on XML as a data model and XML Schema as a type system, this import

provides a simple interface to web services without the need to generate a complex mapping between XML Schema and a programming language type system such as JAXB [11]. The following example illustrates a service import that might be found in the prolog of an application:

```
import service
   namespace cs="http://credit.bureau.com" name="CreditService"
```

The service import identifies a web service by its target namespace. The target namespace, or an optional location hint, may be used to retrieve the corresponding WSDL description that contains the necessary information to implement the service import. The name of the service must be provided as well, and optionally the port name (in case there are multiple ports defined for that service). In case there are several ports, each of them may be imported independently. Each port corresponds to one XQuery module.

From a user point of view, the service import behaves the same as an XQuery module import that provides access to one or more functions and/or types defined by the web service. If a function imported from a web service has side effects, it is treated as an updating function, and is invokable only where updating expressions are permitted. The way in which an implementation recognizes an imported function as side-effecting is implementation-defined. The specific ways in which web service operations correspond to XQuery functions, including how function signatures are derived from WSDL descriptions of a web service, are described in [10].

In the following example, an application imports three web services named OrderService, CreditService, and PricingService, binding them to the namespace prefixes `os`, `cs`, and `ps` respectively. The function `local:processOrder` then coordinates these three services to process an order consisting of several items. First, the function calls `ps:getPrice` to get prices of the individual items, applies applicable taxes, and computes the total cost of the order. Then the function calls `cs:reserveFunds` to check the customer's credit and place a credit hold on the amount of the order. The value returned by `cs:reserveFunds` may indicate that the customer has insufficient credit; in this case, processing of the order terminates here with an appropriate return code. If the credit check is successful, the function calls `os:placeItemOrder` to handle each individual item in the order, collecting a list of exceptions for conditions such as no-such-item, out-of-stock, etc. Finally, the order processing function returns either a successful result code or a list of exceptions. This example illustrates how XQueryP can be used to coordinate several external side-effecting services that exchange messages in XML format.

```
import service
   namespace os="http://orders.myinc.com" name="OrderService"
   (: defines os:order, os:placeItemOrder, os:ok :)

import service
   namespace cs="http://credit.bureau.com" name="CreditService"
   (: defines cs:reserveFunds, cs:insufficientFunds :)

import service
   namespace ps="http://prices.myinc.com" name="PricingService"
   (: defines ps:getPrice :)

declare updating function local:processOrder($o as element(os:order))
   as element()
   {
   declare $taxrate, $total, $creditCheck, $exceptions, $outcome;
   (: step 1: call ps:getPrice for each item and compute total price :)
   set taxrate := 1.0825;
   set $total := $taxrate * fn:sum(
      for $i in $o/items/item return $i/qty * ps:getPrice($i/sku));

   (: step 2: call cs:reserveFunds to check credit and reserve funds :)
   set $creditCheck := cs:reserveFunds($o/cardInfo, $total);
   if ($creditCheck//cs:insufficientFunds)
      then return(<cs:insufficientFunds>) (: exit here if bad credit :)
      else
         {
         (: step 3: call os:placeItemOrder to place order for each
            item and collect exceptions :)
         set $exceptions := ( );
         for $i in $o/items/item return
            { set $outcome := os:placeItemOrder(
                 $i/sku, $i/qty, $o/customerInfo, $o/cardInfo);
              if ($outcome != <os:ok/>)
              then set $exceptions := fn:concat($exceptions, $outcome)
              else ( );
            }
         (: step 4: return success code or list of exceptions :)
         if ($exceptions)
         then return ($exceptions)
         else return <os:ok/>
         }
   }   (: ends function body of local:processOrder :)
```

7. Implementation Experience

Two partial open source implementations of XQueryP are currently available.

The first implementation, constructed at ETH Zurich, is available at [12]. This implementation extends the Java-based, open source MXQuery engine, a partial implementation of XQuery with Update. The MXQuery engine shows that XQueryP can be implemented and integrated into an existing XQuery engine in a straightforward way. Sequential execution mode is implemented by implicitly applying the Pending Update List after the execution of each updating expression. Blocks and variable assignment are implemented by extending the implementation of the dynamic context of an XQuery engine. A simple garbage collection mechanism is needed because local and global variables can refer to the same XML data. While-loops are implemented

in essentially the same way as in an imperative programming language. Atomic blocks require an extension of the Pending Update List data structure in order to store UNDO information (i.e., before-images). For the MXQuery engine it took about six weeks for a student to implement these XQueryP extensions, which added about 70 KB to the footprint of the MXQuery engine.

Another partial XQueryP implementation, built as part of the Galax processor, is available at [13]. Addition of XQueryP to Galax was very simple. Galax already supported sequential execution and immediate application of updates [16]. Support for variable assignment is implemented directly as an extension to the Galax algebra and run-time. Blocks are treated as syntactic sugar. While-loops are compiled into tail-recursive functions. Some special care is needed in the Galax compiler to ensure that the optimizer does not rewrite query plans in a way that changes the semantics of sequential execution mode. This is done by checking either purity or commutativity conditions [14] between sub-plans involved in algebraic rewritings. Galax does not currently implement try-catch blocks or the `break` and `continue` expressions.

8. Related Work

XQueryP attacks the impedance mismatch between query languages and programming languages by extending a query language with programming features in order to reduce or eliminate its dependence on a host programming language. An obvious alternative approach is to extend a programming language to manipulate persistent XML data, eliminating the need for a separate query language. In the relational database world, this approach led to proposals such as Pascal-R [15]. Similar approaches are being actively pursued to add XML data manipulation to specific programming languages [16, 17]. Each of these approaches automates the transform between XML data and the type system of a specific programming language. Yet at some level the transform still exists, because the types of XML Schema (such as recursive hierarchies of complex elements) do not map directly onto the type system of any existing programming language. XQueryP attempts to eliminate this transform by operating on XML data directly in its native type system. The XQueryP approach also permits optimization tasks such as query transforms and index selection to be carried out independently of any specific programming language environment.

Florescu, Gruenhagen, and Kossmann have proposed and implemented an XML programming language called XL [18] that is similar in concept to XQueryP. Like XQueryP, XL extends XQuery to include ordered execution, assignment statements, blocks, exception handlers, and while-loops. XL also includes more ambitious features such as expressions for parallel execution, condition-based triggers, and synchronizing features such as "wait on event."

Ghelli, Re, and Siméon have also proposed and implemented a set of XQuery extensions called XQuery! [19] that permit side-effecting expressions and impose an order on expression evaluation. Unlike XQueryP, XQuery! places no restrictions on where side-effecting expressions can appear, allowing (for example) an update to be used in a predicate or nested inside another update. Also, XQuery! does not include programming features such as assignments or while-expressions.

An XQuery implementation released by Mark Logic Corporation [20] supports a number of extensions for application development, mostly in the form of an extended function library that has been used to implement several commercial applications.

Other features and approaches for XML programming have been suggested by various writers. For example, [21] proposes higher-order functions and parametric polymorphism, and [22] proposes a new declarative language for generating multi-tier applications. Although all these approaches may ultimately prove useful, they are beyond the scope of the current "minimalist" proposal.

9. Conclusions and Future Work

XQueryP is a proposal for extending XQuery to support development of XML-based applications without relying on a host programming language. The extensions needed for this purpose are relatively small and are similar those used in SQL Persistent Stored Modules (PSM).

We believe that the proposed language extensions can dramatically reduce the amount of code required to develop web services and other XML applications. This simplification is made possible by operating on XML in its native data model rather than transforming it into a different type system. The XQueryP approach makes it possible to tightly integrate all three parts of an application (storage, logic, and presentation) by basing them all on XML.

As noted earlier, this paper represents a snapshot of a work in progress. User experience with experimental XQueryP implementations will be valuable in refining the language design.

Optimization of XQuery has been studied by Florescu [23] and others. Many XQuery optimizations apply equally to XQueryP. However, exploring the optimization of XQueryP expressions in detail is an important avenue for future research.

One area in which additional work is needed is that of providing mechanisms for XQueryP applications to interact with their external environments. Since XQueryP can be used in a variety of different environments, this issue may be best addressed by providing specialized function libraries for different kinds of environmental interactions. We will briefly consider two examples of such function libraries.

The first example deals with support for end-user interactions. In the PHP programming language, for example, inputs from HTML forms are made available to application programs in the form of an associative array named `$_REQUEST` that contains a set of (key, value) pairs. Since XML data is self-describing, the natural representation for user input in XQueryP is an XML element hierarchy. For example, a call to the function `fn:form-input()` might return the following data, collected from an HTML form:

```
<form-input method="post">
   <name>M. Gulliver</name>
   <phone>555-378-4627</phone>
   <selection>item 25</selection>
   <submit/>
</form-input>
```

A second example of a useful function library might deal with management of the persistent objects that are created or manipulated by XQueryP. XQuery operates on instances of the XQuery and XPath Data Model (XDM) [24], which represents XML data as a hierarchy of nodes. However, XQuery itself does not specify where these instances come from or how new instances may be made persistent. Function libraries could be developed to create, destroy, and organize collections of persistent XML objects from within XQueryP applications. A good starting point for one such function library might be interaction with content repositories that support the JSR-170 interface defined by the Java Community Process [25].

10. Acknowledgments

The authors wish to acknowledge many helpful discussions with members of the XML Query Working Group and with Scott Boag, Kevin Beyer, and Fatma Ozcan at IBM, Daniel Engovatov and Vinayak Borkar at BEA, and Zhen Hua Liu, Muralidhar Krishnaprasad, and Anguel Novoselsky at Oracle.

References

[1] Steven Feuerstein. *Oracle PL/SQL Programming, 4th Ed.* O'Reilly & Associates, 2005.

[2] Microsoft Corp. *Transact-SQL Reference*. See http://msdn2.microsoft.com/en-us/library/ms189826.aspx.

[3] Z. Janmohamed, C. Liu, D. Bradstock, R. F. Chong, M. Gao, F. McArthur, and P. Yip. *DB2(R) SQL PL : Essential Guide for DB2 UDB on Linux, UNIX, Windows, i5/OS, and z/OS.* IBM Press, 2004.

[4] Salesforce.com. See http://www.salesforce.com/company

[5] *Database Languages--SQL--Part 4: Persistent Stored Modules (SQL/PSM).* ANSI/ISO/IEC 9075-4-1999.

[6] W3C Schema Working Group (H. Thompson et al, editors). *XML Schema, Parts 0, 1, and 2*. See http://www.w3.org/TR/xmlschema-0, -1, and -2.

[7] W3C XML Query Working Group (S. Boag et al, editors). *XQuery 1.0: An XML Query Language* (21 Nov 2006). See http://www.w3.org/TR/xquery.

[8] W3C XML Query Working Group (D. Chamberlin et al, editors). *XQuery Update Facility* (8 May 2006). See http://www.w3.org/TR/xqupdate.

[9] D. Chamberlin, M. Carey, D. Florescu, D. Kossmann, and J. Robie. "XQueryP: Programming with XQuery". *Proceedings of XIME-P 2006: Third International Workshop on XQuery Implementation, Experience, and Perspectives*, Chicago, June 2006.

[10] N. Onose and J. Simeon, "XQuery at your Web Service". *Proceedings of WWW2004*, Tokyo, Japan, 2004.

[11] *Java Architecture for XML Binding (JAXB).* See http://java.sun.com/xml/jaxb/.

[12] M. Braun, I. Carabus, P. Fischer, D. Graf, D. Kossmann, T. Kraska, and R. Temosevicius. *MXQuery: a Low-Footprint, Extensible XQuery Engine*. See http://www.mxquery.org.

[13] Galax processor version 0.6.8. See http://www.galaxquery.org/distrib.html.

[14] G. Ghelli, K. Rose, and J. Simeon, "Commutativity Analysis in XML Update Languages". *Proc. ICDT 2007.* Barcelona, Spain (to appear).

[15] M. Jarke and J. Schmidt. "Query Processing Strategies in the Pascal/R Relational Database Management System". *Proc. ACM SIGMOD Conference,* Orlando, FL, June 1982.

[16] *XLinq: .NET Language Integrated Query for XML Data.* Microsoft Corp., Sept. 2005. See http://download.microsoft.com/download/c/f/b/cfbbc093-f3b3-4fdb-a170-604db2e29e99/XLinq%20Overview.doc.

[17] R. Bordawekar, M. Burke, I. Peshansky, and M. Raghavachari. "Simplify XML Processing with XJ". *IBM DeveloperWorks*, June 2005. See http://www-128.ibm.com/developerworks/java/library/x-awxj.html

[18] D. Florescu, A. Gruenhagen, and D. Kossmann. "XL: An XML Programming Language for Web Service Specification and Composition". *Proceedings of WWW 2002*, Honolulu, HI, May 2002.

[19] G. Ghelli, C. Ré, and J. Siméon. "XQuery!: An XML Query Language with Side Effects." *Second International Workshop on Database Technologies for Handling XML Information on the*

Web (DataX 2006) March 2006, Munich, Germany. To appear in in *Lecture Notes in Computer Science,* Springer-Verlag.

[20] Mark Logic Corporation. *Mark Logic Server, XQuery API Documentation*. See http://xqzone.marklogic.com/pubs/3.0/apidocs/UpdateBuiltins.html and http://xqzone.marklogic.com/pubs/3.0/apidocs/Extension.html.

[21] M. Fernandez and J. Simeon. "Growing XQuery". *Proc. ECOOP 2003.* See http://www-db.research.bell-labs.com/user/simeon/ecoop2003.pdf.

[22] P. Wadler, University of Edinburgh. *Links: Linking Theory to Practice for the Web--Case for Support.* See http://homepages.inf.ed.ac.uk/wadler/links/epsrc05/case.pdf.

[23] D. Florescu, C. Hillary, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, and A. Sundararajan. "The BEA Streaming XQuery Processor". VLDB *Journal,* Vol. 13, No. 3, pp. 294-315 (2004).

[24] W3C XML Query Working Group (M. Fernandez et al, editors). *XQuery 1.0 and XPath 2.0 Data Model (XDM)* (21 Nov 2006). See http://www.w3.org/TR/xpath-datamodel/.

[25] Java Community Process (D. Nuescheler, editor). JSR-000170 *Content Repository for Java Technology API* (17 June 2005). See http://jcp.org/aboutJava/communityprocess/final/jsr170/index.html.