XQSE: An XQuery Scripting Extension for the AquaLogic Data Services Platform

Vinayak Borkar, Michael Carey, Daniel Engovatov, Dmitry Lychagin, Till Westmann, Warren Wong

BEA Systems, Inc., 2315 North First Street, San Jose, CA 95131, U.S.A.

Contact: mcarey@bea.com

***Abstract*— The AquaLogic Data Services Platform (ALDSP) is a BEA middleware platform for creating services that access and manipulate information drawn from multiple heterogeneous sources of data. The integration logic for read services is specified declaratively using the XQuery language. ALDSP 3.0, available in December 2007, includes a new XQuery-based Scripting Extension – XQSE – that enables developers to write procedural as well as declarative logic without leaving the XQuery world. In this paper, we describe the XQSE extensions to XQuery and show how they help to support important new classes of data services in ALDSP 3.0.**

I. Introduction

In mid-2005, BEA introduced the AquaLogic Data Services Platform. The goal of ALDSP is to make it easy to design, develop, deploy, and maintain a data services layer in the service-oriented architecture (SOA) world. ALDSP provides a declarative software platform for building data-centric services that integrate and service-enable information drawn from multiple heterogeneous enterprise data sources [9]. The foundation of ALDSP is built on XML, XML Schema, and XQuery. Read services are specified declaratively using the XQuery language. Updates are supported by propagating data changes back to affected sources through the analysis of read services' XQuery definitions. When this analysis is not possible, or when some of the affected sources are non-SQL sources (e.g., information coming from a Web service call), ALDSP developers have had to – until now – write Java code to manually take over update processing.

ALDSP 3.0 is being released in December 2007. One of the key contributions of ALDSP 3.0 is that it significantly extends the update capabilities of ALDSP. One of the extensions is the introduction of a new XQuery-based language called XQSE (XQuery Scripting Extension), pronounced "excuse". XQSE adds a set of procedural constructs on top of standard XQuery to enable data service developers to write custom update-handling logic and other procedures without having to cross the chasm from the XQuery world to the Java world.

In this paper, we describe the XQSE extensions to XQuery. We also briefly review the initial XQSE design requirements, gleaned from talking to actual ALDSP developers, and we explain how the XQSE extensions to XQuery address those requirements. Section II provides an overview of ALDSP, both from a query perspective and an update perspective, setting the stage for the introduction of XQSE. Section III describes XQSE, covering the XQSE features in ALDSP 3.0 as well as a handful of additional XQSE constructs targeted for subsequent releases. Section IV briefly compares XQSE with other proposals that have added imperative constructs to the XQuery language; it also discusses the relationship of XQSE to XQuery-SX, the new W3C effort to procedurally extend XQuery 1.0. Section V concludes the paper.

II. The AquaLogic Data Services Platform

To frame our discussion of XQSE and its use in query and update service creation in ALDSP, it is important to first understand the ALDSP world model.

A. Modelling Data and Services

ALDSP targets the SOA world, so it is based on a service-oriented view of data. ALDSP models an enterprise (or a portion of interest of the enterprise) as a set of interrelated data services [10]. Each data service offers a set of service calls that an application can use to access instances of a particular coarse-grained business object type (e.g., customer, order, employee, or service case). ALDSP 3.0 supports two kinds of data services, entity data services and library data services.

An entity data service can be thought of as a service-enabled business object. It has a “shape” that characterizes the information content of its business object type. XML Schema is used to describe each data service's shape. A data service also has a set of methods, service calls that provide ways to read and manipulate instances of the data service’s underlying business objects. In ALDSP 3.0, these methods are classified as functions (read-only) or procedures (with side effects). Supported operation types for entity data services include read functions, which provide ways to fetch instances of the data service’s objects; write procedures, which modify (create, update, delete) instances; navigation functions, which enable traversal from one instance object from the data service (e.g., customer) to one or more instances from a related data service (e.g., order); and library functions and procedures, which are other supporting methods related to the data service. In contrast to an entity data service, a library data service contains only library functions and procedures. Each data service method is realized as an XQuery function (or as an XQSE function or procedure, as described herein) that can be called from a client program, called from ad hoc queries, and/or used when creating other, higher-level logical data services.

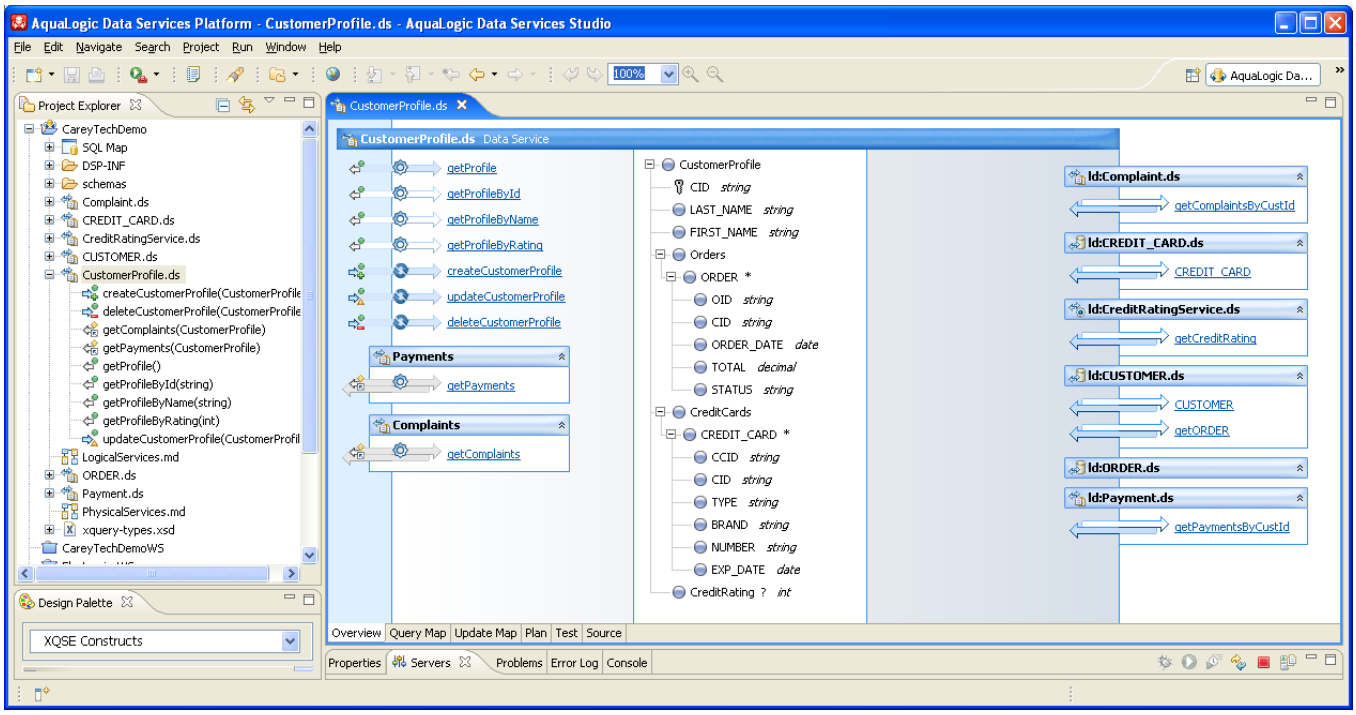

Figure 1: Customer profile data service, design view

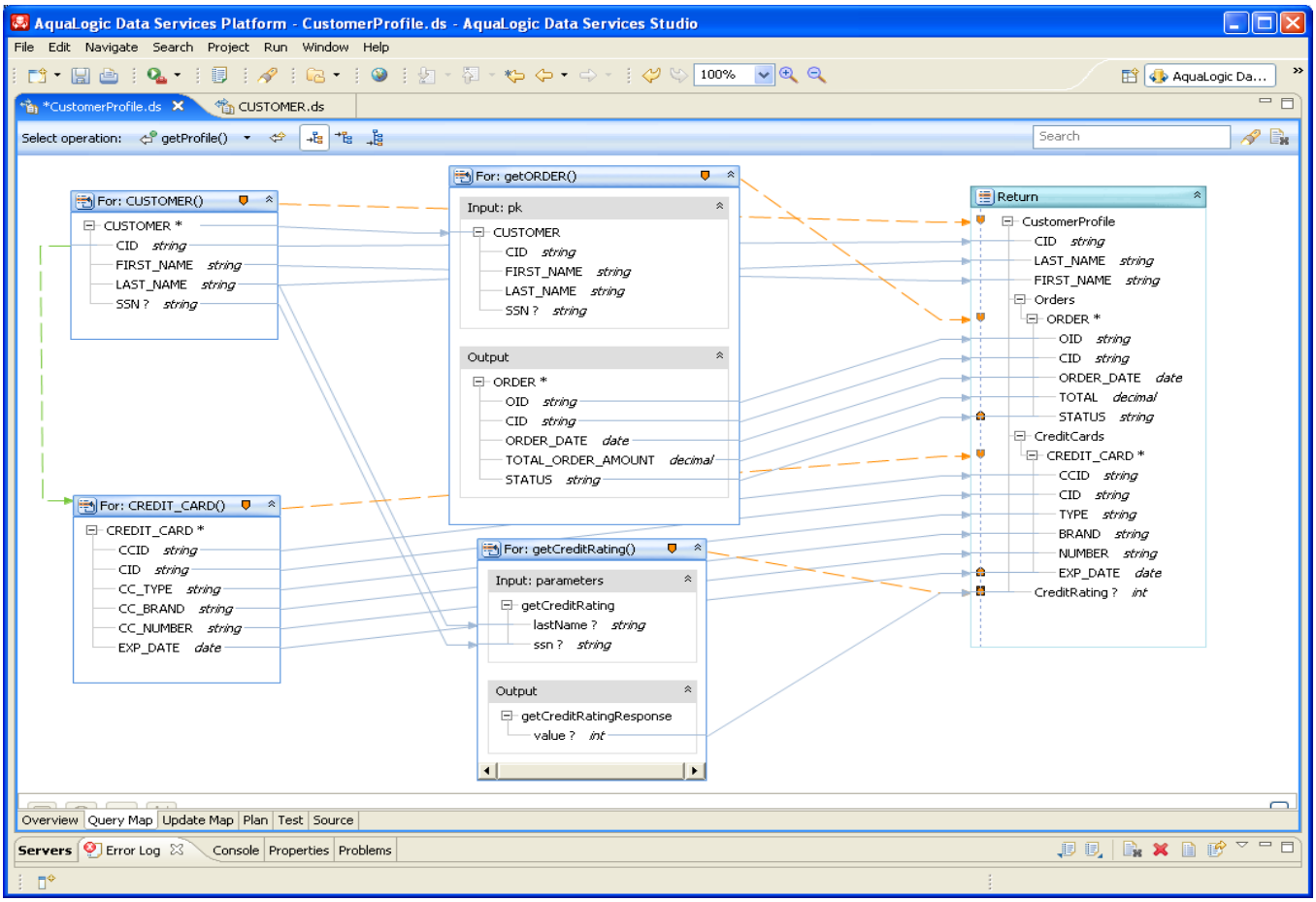

Figure 2: Customer profile data service, primary read method, graphical view

Figure 1 shows a screen capture of the ALDSP 3.0 design view of a simple entity data service. The center of the design view shows the shape of the data service, and the left side of the design view shows the various methods available to users of the data service. The right side of the design view summarizes the dependencies that this data service has on other data services that were used to create it.

When pointed at a data source by a data service developer, ALDSP first introspects the source's metadata (such as SQL metadata for a relational data source or WSDL files for a Web service). This introspection guides the automatic creation of one or more physical data services that make that source available for use in ALDSP. Introspecting a relational data source yields one entity data service (with one read method and three update methods, create, update, and delete) per table or view. The data service shapes in this case correspond to the natural "XML view" of a row of each table or view. In the presence of foreign key constraints, RDBMS introspection also produces navigation functions to encapsulate the join paths suggested by the constraints. Introspecting a Web service data source (based on WSDL) yields a library data service with multiple methods, one per Web service operation. The methods' input and output types correspond to the schema information found in the WSDL. Other functional data sources are modelled similarly. The result of data source introspection is a uniform, "everything is a data service" view of an enterprise's data sources, a view well-suited for further use in composing higher-level data services using XQuery.

B. Defining Services Using XQuery

The language used to specify the logic for most data service functions is XQuery. Figure 2 shows the graphical XQuery editor view for a key part of Figure 1's logical data service, showing how the main read method of the data service would look when ALDSP is integrating information from two relational databases and a Web service. Figure 3 shows the resulting function's XQuery source code. In this example, one database holds the CUSTOMER and ORDER tables, while the CREDIT_CARD table is in a different database. In addition to these relational sources, a document-style Web service provides a credit rating lookup capability that takes a customer's name and social security number and returns his or her credit rating.

The read method in Figure 3, getProfile(), takes no input arguments and returns the customer profiles for all customers listed in the CUSTOMER table in the first relational database. Note that the contents of the tables are accessed via XQuery function calls. For each customer, the ORDER table is accessed to create the nested <ORDERS> information; this access uses the navigation function that ALDSP automatically created for this purpose based on foreign key metadata. Also for each customer, the table CREDIT_CARD in the other database is accessed, and then the credit rating Web service is called. The result of this function is a series of XML elements, one per CUSTOMER, that integrate all of this information from the different data sources. Once the integration and correlation of data has been specified in this function, which serves as the primary (or "get all instances") read method for the data service, the remaining read methods for this data service become trivial to specify. This is also shown in Figure 3, where one of the other read methods is shown at the end.

C. Update Services in ALDSP

ALDSP utilizes Service Data Objects [11], a.k.a. SDO, to support updates as well as reads for data from data services. The ALDSP APIs allow a client application to invoke a data service, then operate on the results, and finally submit the modified data back to the data service from whence it came to persist the changes. Figure 4 illustrates how the ALDSP SDO-based Java client API works in such a use case, showing the relevant client code snippet in the middle. A typed SDO object is first obtained from the customer profile data service through a call to the ALDSP Java client API. The object is operated on by the client application, leading to changes in one or more of its underlying element values. The changed SDO object is then returned to ALDSP via the update method of the data service.

Figure 4 shows the data service data as it flows from and to the ALDSP server. In the update call, the new XML data is sent back along with a serialized change summary that identifies those portions of the data that have been changed and also records their previous values. The ALDSP server examines the change summary in order to determine how to propagate the changes back to the underlying sources. Similar to reads, unaffected data sources are not involved in an update, and unchanged portions of source data are not updated.

To automatically propagate client data changes to (just) the relevant backend data sources, ALDSP must identify where the changed data originated from. Basically, the data lineage must be determined. ALDSP computes the required lineage by analyzing a specially designated "primary" data service read function (by default the first read function, preferably the "get all" function if there is one). Primary key information, join predicates, and query result shapes are used together with the change list to determine which data from which data sources are actually affected by a given update.

We now sketch the ALDSP update decomposition process. An update operation enters ALDSP at runtime as a C/U/D (create, update, or delete) call on a data service, such as our example customer profile service, and is then decomposed into a set of lower-level updates to be propagated to the affected sources. In ALDSP 2.5, a data service developer had the option to associate a Java update override with a data service at any level. To do so, the data service developer specified a Java method to which control was to be handed over when updates occurred on that data service. The update override could either extend or replace the default update handling logic of ALDSP, providing a way to enforce business rules, insert computed values, or even totally alter the way the update was handled if so desired.

```
declare function ns1:getProfile() as element(ns1:CustomerProfile)* {
   for $CUSTOMER in cus:CUSTOMER()
   return <tns:CustomerProfile>
             <CID>{fn:data($CUSTOMER/CID)}</CID>
             <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>
             <FIRST_NAME>{fn:data($CUSTOMER/FIRST_NAME)}</FIRST_NAME>
             <Orders>{
                for $ORDER in cus:getORDER($CUSTOMER)
                return <ORDER>
                          <OID>{fn:data($ORDER/OID)}</OID>
                          <CID>{fn:data($ORDER/CID)}</CID>
                          <ORDER_DATE>{fn:data($ORDER/ORDER_DATE)}</ORDER_DATE>
                          <TOTAL>{fn:data($ORDER/TOTAL_ORDER_AMOUNT)}</TOTAL>
                          <STATUS>{fn:data($ORDER/STATUS)}</STATUS>
                       </ORDER>
             }</Orders>
             <CreditCards>{
                for $CREDIT_CARD in cre:CREDIT_CARD()
                where $CUSTOMER/CID eq $CREDIT_CARD/CID
                return <CREDIT_CARD>
                          <CCID>{fn:data($CREDIT_CARD/CCID)}</CCID>
                          <CID>{fn:data($CREDIT_CARD/CID)}</CID>
                          <TYPE>{fn:data($CREDIT_CARD/CC_TYPE)}</TYPE>
                          <BRAND>{fn:data($CREDIT_CARD/CC_BRAND)}</BRAND>
                          <NUMBER>{fn:data($CREDIT_CARD/CC_NUMBER)}</NUMBER>
                          <EXP_DATE>{fn:data($CREDIT_CARD/EXP_DATE)}</EXP_DATE>
                       </CREDIT_CARD>
             }</CreditCards>
             {  for $getCreditRatingResponse in cre3:getCreditRating(<cre2:getCreditRating>
                      <cre2:lastName?>{fn:data($CUSTOMER/LAST_NAME)}</cre2:lastName>
                      <cre2:ssn?>{fn:data($CUSTOMER/SSN)}</cre2:ssn>
                    </cre2:getCreditRating>)
                return <CreditRating?>{fn:data($getCreditRatingResponse/cre2:value)}</CreditRating>
             }
          </tns:CustomerProfile>
};

declare function ns1:getProfileById($cid as xs:string) as element(ns1:CustomerProfile)* {
   for $CustomerProfile in ns1:getProfile()
   where $cid eq $CustomerProfile/CID
   return $CustomerProfile
};
```

Figure 3: Customer profile data service, two read methods, XQuery source excerpt

When the update method of a data service is called to submit a changed SDO or a changed set of SDOs back to ALDSP, the atomic unit of update execution is the update call. (The same is true for create and delete calls.) In the event that all the affected sources are relational and can participate in two-phase commit (XA), the entire update operation will run as one atomic transaction across the affected sources. Since data is often read in one transaction, operated on remotely by a client, and re-submitted later with changes, ALDSP supports several optimistic concurrency options to enable a data service designer to choose how to have the system decide if a given change or set of changes can safely be applied. Supported choices include requiring one of the following to be enforced:

- All values that were *read* must still be the same (at update time) as their original (read time) values.
- All values that were *updated* must still be the same as their original values.
- A *chosen subset* of the values that were read (such as a timestamp or a version id) must still be the same their original values.

ALDSP uses this choice in the relational case to condition the SQL update statements that it generates (i.e., “sameness”

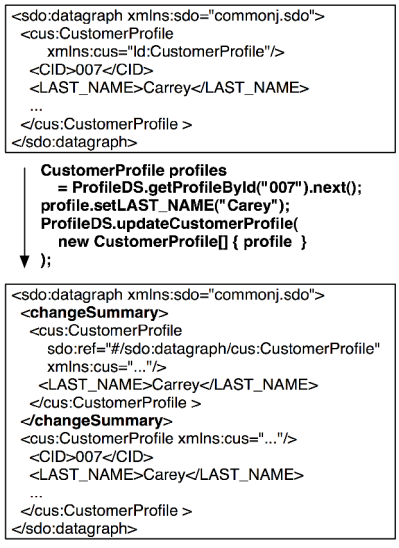

Figure 4: Disconnected updates via SDO programming model

is expressed as part of the where clause for update statements sent to the sources). When updates must be propagated to non-relational sources, or if more reliable update execution is required over non-XA sources, the data service developer had to resort to the use of Java update overrides in ALDSP 2.5.

III. XQSE: An XQuery Scripting Extension

Over the past two years we have watched ALDSP users having to move from the declarative world of XQuery, while integrating their data, across the "impedance mismatch" chasm into Java in order to handle non-relational updates or to replace or augment the automated update handling of ALDSP. Based on this, as well as on being in agreement with the goals of references [5]-[7], we initially decided that ALDSP 3.0 would procedurally extend XQuery directly along the lines of the XQueryP language [7]. As our work proceeded, however, and we thought more carefully about XQueryP's model, syntax, and semantics, we realized that its tight (dual mode) integration with XQuery would likely confuse ALDSP users. Our analysis led to several additional design requirements, and XQSE was born. Key requirements in addition to procedural logic support and the allowance of side effects included:

(1) XQSE should "loosely wrap" XQuery, extending it similarly to how stored procedure languages extend SQL. This is a familiar and comfortable approach for users with data backgrounds – it allows them to first learn XQuery and then apply their pre-existing XQuery mental model to learning the XQSE extensions.

(2) XQSE should "make sense" to users used to traditional languages such as Java, C++, and C#. Thus, XQueryP "features" like requiring all procedural looping constructs to return a value, e.g., the concatenation of values from all loop executions [7], are less desirable than clearly distinguishing between expressions and statements.

A. An Overview of XQSE

XQSE extends XQuery with a new construct, the statement. To model XQSE program evaluation, the XQuery processing model is extended accordingly, adding the concept of statement execution. Execution may involve looping, branching, changing existing instances of the XQuery Data Model (XDM), computing new XDM instances, assigning instances of XDM to variable references, causing other side effects (e.g., by invoking an externally defined action that does so), and so on. Statements are executed in a well-defined order, and the subsequent execution of another statement and/or the evaluation of a subsequent expression will observe the results of any side effects, variable bindings, and changes to the dynamic context from the statements that precede it.

Statements are composed of other statements and/or expressions, possibly including updating expressions. For compound statements, well-defined rules describe the order and the conditions of the contained statements' execution and expression evaluation. If an expression that is enclosed in a statement is an updating expression a la [3], the execution of that statement will include both the computation and the application of the expression's updates. Statements are not composable inside expressions.

XQSE introduces a new class of callable unit of execution in addition to the function – namely, the procedure. The body of a procedure is a statement, which of course may be a compound statement. A return statement is included in XQSE to direct execution back to the calling context. Procedure calls cannot be used in place of function calls in an XQuery expression unless the called procedure is annotated as having no side effects. (XQSE supports the notion of read-only procedures, called "XQSE functions" in ALDSP 3.0, to permit developers to define the procedural equivalent of an XQuery function when a desired computation can be more naturally expressed procedurally.)

In XQSE, both the syntax and the semantics of the W3C XQuery [2] and XQuery Update Facility [3] (XUF) expressions are undisturbed. The XQuery prolog is extended to include procedure declarations and to allow an XQSE statement to appear in place of a query body to provide a location for the main body of an XQSE program.

Since ALDSP 3.0 does not implement the W3C XUF extensions to XQuery yet, several of the features mentioned in the following description – notably the update statement – are not yet implemented. In ALDSP 3.0, a set of external XQSE procedures (create, update, and delete) are automatically provided instead as a callable means to modify relational source data. A future ALDSP release will add support for XUF-style updates (once XUF has been finalized by W3C).

B. Extensions to XQuery 1.0

We now describe the XQSE procedural extensions to the W3C XQuery language in detail.

1) Extensions to the processing model: The concept of statement execution is added. Statement execution consists of sequential atomic operations that include evaluation of an XQuery expression, making changes to instances of XDM by applying a pending update list, assigning variables, and executing user-defined or external procedures. An operation may have side effects that are visible to subsequent operations.

2) *Extensions to the Prolog:*

Prolog ::= ((DefaultNamespaceDecl | Setter | NamespaceDecl | Import) Separator)* ((VarDecl | FunctionDecl | ProcedureDecl | OptionDecl) Separator)*

ProcedureDecl ::= "declare" ("readonly")? "procedure" QName "(" ParamList? ")" ("as" SequenceType)? (Block | "external")

An XQSE prolog may include procedure declarations as well as the other components of an XQuery prolog. Similar to functions, a procedure is identified by its expanded QName together with the number of parameters in its declaration. Procedure names and their numbers of parameters must therefore be included in, and satisfy, the same uniqueness constraints as defined for XQuery functions. Procedures can be external or user defined. XQSE procedure declarations follow the same rules as XQuery function declarations regarding parameter names and types.

3) *Extensions to the Query Body:*

QUERYBODY ::= EXPR | BLOCK

In XQSE, the Query Body may be either an expression or a Block statement. When the query body is an expression, it is evaluated in the same way as currently defined in XQuery with Update Facility. A block in this context is the entry point into the XQSE world.

4) *Statements:*

STATEMENT ::= CONTROLSTATEMENT | UPDATESTATEMENT | PROCEDURECALL | BLOCK | PROCEDUREBLOCK

CONTROLSTATEMENT ::= SETSTATEMENT | WHILESTATEMENT | ITERATESTATEMENT | TRYSTATEMENT | IFSTATEMENT | RETURNSTATEMENT | CONTINUESTATEMENT | BREAKSTATEMENT

VALUESTATEMENT ::= NONUPDATINGEXPRSINGLE | PROCEDURECALL | PROCEDUREBLOCK

The following set of XQSE statements has been added to XQuery in ALDSP 3.0: Block, Block declaration, Assignment, Return, Value statement, Procedure call, While, Iterate, If and Try-Catch. Some additional statements that are planned as part of the XQSE roadmap, but not part of ALDSP 3.0, are: Update statement, Procedure block, Continue, and Break.

5) *Block and block variable declarations:*

BLOCK ::= "{" (BLOCKDECL ";")* ((SIMPLESTATEMENT ";") | BLOCKSTATEMENT (";")?)* "}"

SIMPLESTATEMENT ::= SETSTATEMENT | RETURNSTATEMENT | IFSTATEMENT | CONTINUESTATEMENT | BREAKSTATEMENT | UPDATESTATEMENT | PROCEDURECALL

BLOCKSTATEMENT ::= BLOCK | PROCEDUREBLOCK | WHILESTATEMENT | ITERATESTATEMENT | TRYSTATEMENT

BLOCKDECL ::= "DECLARE" "$" VARNAME TYPEDECLARATION? (":=" VALUESTATEMENT)? ("," "$" VARNAME TYPEDECLARATION? (":=" VALUESTATEMENT)?)*

A Block is an ordered sequence of block variable declarations and statements. (Its exact grammar was chosen to allow some trailing semicolons to be optional.) To execute a block statement, each block variable declaration (if any) is executed once in the order written. Each statement in the block is then executed once in the order written. If a Return statement is executed, further execution is interrupted, as described by the semantics of the Return statement. The block statement by itself does not have a value. If the block statement constitutes the Query Body, and no return statement is executed, then the result of the query is an empty sequence.

A block variable declaration has semantics similar to a let-clause variable declaration in XQuery. If a block variable declaration includes a value statement, it is called an initializing statement and serves to provide the variable's initial value. The scope of the variable is the remainder of the Block, not including its initializing statement. Block variables differ from let variables in that they can be assigned and reassigned as long as they are in scope. (See Assignment, below.) In contrast, let variables are read-only value bindings. If a declaration inside a block does not specify a type for its variable, the variable's implicit type is item()*.

When the initializing statement is present, it is executed once and the XDM value returned by it is assigned to the variable. The type of the assigned value must match the declared type of the variable according to the Sequence Type matching rules. If an initializing statement is not present, the variable is not bound to a value. Any reference to such a variable, other than on the left-hand-side of an assignment statement, is an error until it has been initially assigned to.

6) Assignment

SETSTATEMENT ::= "SET" "$" VARNAME ":=" VALUESTATEMENT

The assignment statement replaces a variable's value with the value returned by the value statement. Only variables declared by a Block variable declaration may be assigned to.

The value statement is executed once. The XDM instance returned by the value statement is assigned to the named variable. If the value statement raises an error, the variable is left in its previous state and the error is propagated. The typed value returned by the value statement must match the declared type of the variable according to Sequence Type matching rules; if not, an error is raised.

7) Return statement

RETURNSTATEMENT ::= "RETURN" "VALUE" VALUESTATEMENT

A return statement interrupts execution and passes a value – an XDM instance – back to the calling context. First, an XDM value is computed by executing the Value statement. This value is called the return value. If a return statement is executed within the body of a user-defined procedure, then further execution of the procedure is interrupted and the return value is returned as the result of the procedure call. If a return statement is executed during the evaluation of the Query Body, then all further execution is stopped and the return value is returned to the environment as the result of the query. If a return statement is executed within a Procedure Block statement, then further execution of the sequence of statements in the procedure block is interrupted and the return value is the return value of the procedure block.

(: EXAMPLE - THE TIME WORN "HELLO, WORLD" PROGRAM IN XQSE. :)
{ RETURN VALUE "HELLO, WORLD"; }

8) Value statement

VALUESTATEMENT ::= NONUPDATINGEXPRSINGLE | PROCEDURECALL | PROCEDUREBLOCK

A value statement computes and returns an XDM value, called a result. If the body of a Value statement is an XQuery expression returning an XDM value, then the expression must return an empty pending update list. It is evaluated according to the rules of XQuery (i.e., no side effects). The value is returned as the Value statement's result.

Procedure Call and Procedure Block are executed once according to the rules defined for them. Their return value is returned as the result of the value statement.

```
(: example :)
set $x := 10 + $y;
set $z := ns:myprocedure($y);
```

In this example, `10 + $y` and `ns:myprocedure($y)` are both Value statements.

9) Procedure declaration and call

PROCEDUREDECL ::= "DECLARE" ("READONLY")? "PROCEDURE" QNAME "(" PARAMLIST? ")" ("AS" SEQUENCETYPE)? (BLOCK | "EXTERNAL")

PROCEDURECALL ::= FUNCTIONCALL /*RESTRICTED TO PROCEDURES */

A procedure declaration is syntactically similar to an XQuery function declaration. It can optionally be flagged as readonly to indicate that it is essentially a function, except for having been written procedurally in XQSE. (In ALDSP 3.0, we provide an alternate syntax, "DECLARE XQSE FUNCTION", for users' further conceptual convenience.)

The body of a user-defined procedure is a Block statement, and the procedure will be executed by executing its body. If a return statement is executed, its return value will become the value of the procedure call. If no Return statement is executed when the last statement in the Block is reached, the return value will instead be an empty sequence. Within a procedure body, the typed value of the return must match the declared type of the procedure according to the XQuery Sequence Type matching rules. If the types do not match, an error is raised.

A procedure call is executed by first evaluating the parameters of the call in the order written. Since the values of the parameters are defined by XQuery expressions, just like functions in XQuery, their evaluation has no side effects.

10) While

WHILESTATEMENT ::= "WHILE" "(" NONUPDATINGEXPR ")" BLOCK

The While statement is used for conditional iteration and is executed as follows: The expression in the parenthesis (called the test expression) is evaluated. If its effective Boolean value is false, the body is not executed and execution of the While statement stops. If the effective Boolean value of the test expression is true, then the body of the While statement – a Block statement – is executed.

Execution of the body may cause side effects that would change the value of the test expression. The test expression is evaluated again each time through the loop, and this is repeated until the effective Boolean value of the test expression is eventually evaluated to be false or until a Break or Return statement is evaluated within the body.

The While statement does not return a value. If needed, an Assignment statement may be used to capture any results from the contained computations.

```
(: example :)
declare $y, $x := 3;
while ($x lt 100) {
  fn:trace($x);
  set  $y := ($y, $x);
  set $x := $x * 2;
}
```

11) Iterate

ITERATESTATEMENT ::= "ITERATE" "$" VARNAME POSITIONALVAR? "OVER" VALUESTATEMENT BLOCK

The Iterate statement is used for data-driven looping over a sequence, e.g., to loop over each of the results of an XQuery expression. The Iterate statement defines an iteration variable and an optional positional variable whose scopes are the body of the Iterate statement.

Execution of Iterate works as follows. First, the Value statement is executed once. It returns a sequence of items called a binding sequence. The iteration variable is assigned to the items in the binding sequence in the order of the sequence. The positional variable iterates over ordinal integers representing the position of the item in the binding sequence. After each assignment, the body of the Iterate statement is executed once. The execution of an Iterate statement may be interrupted by a Break statement or a Return statement. If any of the statements in the body change the binding sequence variable itself, the result of the subsequent execution is undefined.

12) If

IFSTATEMENT ::= "IF" "(" NONUPDATINGEXPR ")" "THEN" STATEMENT ("ELSE" STATEMENT)?

The If statement is used for conditional execution. The conditional expression is first evaluated. If its effective Boolean value is true, the statement inside the Then clause is executed. If its effective Boolean value is false and an Else clause is present, then the Statement inside the Else clause is executed.

13) Try-Catch

TRYSTATEMENT ::= "TRY" BLOCK CATCHCLAUSESTATEMENT+

CATCHCLAUSESTATEMENT ::= "CATCH" "(" NAMETEST ("INTO" VARNAMEEXPR (("," VARNAMEEXPR)? "," VARNAMEEXPR)?)? ")" BLOCK

The Try-Catch statement is used for error processing and is executed as follows: First, the body of the Try statement is executed. If any operation raises a dynamic error, e.g., by calling fn:error(), then Try execution is interrupted. Note that executing the Try statement may have caused permanent side effects before the error was raised. Such side effects are not "rolled back".

The NameTest expressions in the Try statement's Catch clauses are evaluated in the order written. If the name test matches the QName of the raised error, then the body of that Catch clause, and only that Catch clause, is executed once. Before Catch clause execution, up to three optional variables may be defined by including an Into clause within the Catch clause. If so, these variables will be assigned the QName identifying the error, its message, and any diagnostic items as defined by the fn:error() function of XQuery.

```
(: example :)
try {
  udp:dothis( );
  udp:dothat( );
  set $x := $y div 0;
  return value $x;
} catch (*:* into $e, $m) {
  fn:trace($e, $m);
  return value "Error";
}
```

C. Additional XQuery Extensions

We now turn to those XQSE features that are part of the design but that do not appear in the ALDSP 3.0 release.

14) Update statement

UPDATESTATEMENT ::= EXPRSINGLE /*MUST BE AN UPDATING EXPRESSION*/

An update statement consists of an updating expression. To execute an update statement, its updating expression is evaluated according to the rules defined by the XQuery Update Facility [3]. No side effects are permitted during the expression's evaluation. The resulting pending update list is then applied as described in the XQuery Update Facility specification. Execution of the update statement therefore constitutes a snapshot, and all applied changes are visible to subsequent statements and expressions.

15) Continue and Break

CONTINUESTATEMENT ::= "CONTINUE" " (" ")"

BREAKSTATEMENT ::= "BREAK" " (" ")"

The Continue and Break statements are included for convenience in further controlling the execution of the While and Iterate statements. If a Continue statement is executed, it interrupts execution of the body of the immediate While or Iterate statement and starts the next iteration. If a Break statement is executed, it stops the execution of the immediate While or Iterate statement.

16) Procedure block

PROCEDUREBLOCK ::= "PROCEDURE" BLOCK

The Procedure Block allows an "in-place" XQSE procedure definition to be used in place of a Value statement. It is included in the language for orthogonality and completeness. Its body – the Block statement – is executed once. If a return statement is encountered, its return value is the return value of the Procedure Block. If the last statement in the body is executed, and it is not a return statement, then the value of the Procedure Block is an empty sequence.

D. Sample XQSE Usage in ALDSP

As mentioned earlier, the main motivation for XQSE was to provide XQuery-centric support for a set of important use cases where customers of ALDSP, prior to the arrival of ALDSP 3.0, had been forced to switch to Java, using Java update overrides, in order to solve their problems. In this section we briefly walk through a series of use cases to show by example how XQSE addresses each of them. While doing so, we will present each use case in the form of a brief textual description followed by an XQSE procedure that addresses an example of such a use case. A full set of "How To" examples that includes these XQSE use cases and more can also be accessed via the BEA ALDSP 3.0 online documentation site [12] for readers who would like to see the full data service context for a given use case or who would like to download a trial version of ALDSP 3.0 and then try the XQSE extensions to XQuery out for themselves.

1) Use case 1: user-defined update

ALDSP 3.0 will automatically generate create, update, and delete methods for physical relational data services as well as logical data services whose read logic it can introspect and reverse-engineer. The default delete operation takes as input an XML data instance having the full data service shape. The first use case shows how to augment the generated methods by adding an additional XQSE procedure to the data service that deletes an employee object based solely on its employee ID. This example simply looks up the employee to be deleted and then calls the default delete method on the resulting object:

```
(: user-defined delete method :)
declare procedure tns:deleteByEmployeeID
                    ($id as xs:string?) as empty()
{
  declare $emp as element(empl:Employee)? :=
                    tns:getByEmployeeID($id);
  tns:delete($emp);
};
```

2) Use case 2: imperative computation

Some computations are easier for "data users" to express in an imperative manner. The second use case shows how to write a simple read-only procedure (an "XQSE function" in ALDSP 3.0 parlance) to compute the management chain given an employee ID. This procedure will then be callable as a data service function from either XQSE or XQuery since it is read-only. This example simply walks the management hierarchy from the given employee up via a while-loop, computing and then returning the sequence of employees encountered along the way. The top employee in the company has no manager, which is how the computation terminates:

```
(:  iterative computation of management chain :)
declare xqse function tns:managementChain
  ($id as xs:string?) as element(empl:Employee)*
{
  declare $curEmp as element(empl:Employee)? :=
                      tns:getByEmployeeID($id);
  declare $empChain as element(empl:Employee)* :=
                        $curEmp;
  declare $mgrId as xs:string? :=
                      fn:data($curEmp/ManagerID);
  while (fn:not(fn:empty($mgrId))){
    set $curEmp := tns:getByEmployeeID($mgrId);
    set $empChain := ($empChain, $curEmp);
    set $mgrId := fn:data($curEmp/ManagerID);
  }
  return value ($empChain);
};
```

3) Use case 3: transform and copy

The third use case illustrates data-driven iteration and how it can be used to perform a "lightweight ETL" operation. This example copies a sequence of data out of one data source, transforms it based on the formatting needs of a second source, and inserts the transformed data into the second source. The example uses an XQuery helper function to effect the data transformation and an XQSE procedure that uses an Iterate statement to access the data to be copied. Notice that the transformation requires an auxiliary data access call to find the manager name for each employee:

```
(:  data transformation function :)
declare function tns:transformToEMP2
    ($emp as element(empl:Employee)?
  as element(emp2:EMP2)?
{
  for $emp1 in $emp return <emp2:EMP2>
    <EmpId>{fn:data($emp1/EmployeeID)}</EmpId>
    <FirstName>{
      fn:tokenize(fn:data($emp1/Name),' ')[1]
    }</FirstName>
    <LastName>{
      fn:tokenize(fn:data($emp1/Name),' ')[2]
    }</LastName>
    <MgrName>{
      fn:data(ens1:getByEmployeeID(
        $emp1/ManagerID)/Name)
    }</MgrName>
    <Dept>{fn:data($emp1/DeptNo)}</Dept>
  </emp2:EMP2>
};

(:  etl lite procedure :)
declare procedure tns:copyAllToEMP2() as xs:integer
{
  declare $backupCnt as xs:integer := 0;
  declare $emp2 as element(emp2:EMP2)?;
  iterate $emp1 over ens1:getAll() {
    set $emp2 := tns:transformToEMP2($emp1);
    emp2:createEMP2($emp2);
    set $backupCnt := $backupCnt + 1;
  }
  return value ($backupCnt);
};
```

4) Use case 4: augmenting ALDSP C/U/D behavior

The final use case here is an example of how XQSE can be used to author create, update, and delete methods that replace (but internally utilize) the system-provided C/U/D methods of ALDSP. This use case is related to the preceding example, but rather than bulk-copying (i.e., batching) data from source 1 to source 2, XQSE is used here to write a create method that invokes create operations on both sources simultaneously for the objects being created. This method could then serve as the primary create method for a logical data service that "fronts" the two data sources where replication is of interest:

```
(:  replicating create method  :)
declare procedure tns:create
    ($newEmps as element(empl:Employee)*)
  as element(empl:ReplicatedEmployee_KEY)*
{
  iterate $newEmp over $newEmps {
    declare $newEmp2 as element(emp2:EMP2)? :=
                    bns:transformToEMP2($newEmp);
    try { tns:createEmployee($newEmp); }
    catch (* into $err, $msg) {
      fn:error(xs:QName("PRIMARY_CREATE_FAILURE"),
        fn:concat("Primary create failed due to: ",
                  $err, $msg));
      };
    try { emp2:createEMP2($newEmp2); }
    catch (* into $err, $msg) {
      fn:error(xs:QName("SECONDARY_CREATE_FAILURE"),
        fn:concat("Backup create failed due to: ",
                  $err, $msg));
    };
  }
};
```

IV. Related Language Efforts

The XQSE design is aimed at satisfying the published requirements for the scripting extension for XQuery [1], and XQSE has been submitted as input for consideration by the W3C XQuery-SX working group. Other notable proposals for XQuery procedural extensions have included XL [5], XQuery! [6], and XQueryP [7]. The XL language was designed to support XQuery-based Web services scripting. Some of its features have since been picked up in the development of the XQuery Update Facility. The XQuery! language design focused primarily on the transactional semantics of updates, and did not of itself include a number of useful procedural programming facilities. Last but not least, as mentioned earlier, we initially chose the XQueryP language as the intended basis for our ALDSP XQuery scripting extension. However, during the course of analysis and design, we identified aspects of XQueryP that concerned us, given the data and procedural programming backgrounds of our typical users. We therefore went in a different direction with XQSE, but it borrowed freely from the language constructs and syntax proposed for XQueryP. We therefore focus here on the key differences between XQSE and its predecessor XQueryP.

XQueryP introduced the concept of "sequential mode" into XQuery expression evaluation. When executed in sequential mode, XQuery expressions have a strict order of evaluation so that external function calls and/or the application of atomic updates happen in a defined order. In this mode, any side-effects become visible to subsequent evaluation steps. The same query syntax therefore has two different meanings, i.e., can produce different results, in XQueryP, depending upon the evaluation mode. This clearly has the potential to confuse users. To support its "dual-moded composability" approach, a number of the XQueryP constructs, including while loops and blocks, are defined in a manner that can be freely composed inside any XQuery expression which is being evaluated in sequential mode. In contrast, statements in XQSE are not composable with expressions in general. We feel that this provides users with a cleaner (and important) separation between declarative and procedural semantics. It also allowed us to easily preserve and apply existing query optimizations [8] within the declarative parts of an XQSE program. Also in support of composability, "procedural" constructs in XQueryP are all defined to return a value. Even a While loop returns a value in XQueryP – it returns the concatenation of the results from the repeated sequential evaluation of its body expression. In contrast, XQSE statements generally do not return values, which we again feel is more user-friendly, at least for "data users" and/or developers whose experience base stems from working with traditional 3GLs. Finally, for updates, the snapshot scopes in XQueryP can only be a single atomic update expression or an updating function call. In the XQSE design, an Update statement defines the scope of a snapshot, and it can be as small or as large as dictated by the application logic.

V. Conclusion

ALDSP is an XQuery-based integration platform for authoring and managing data services, services that integrate and serve up information drawn from multiple heterogeneous enterprise data sources. The operations to access, integrate, and transform data are defined declaratively as XQuery functions. To enable ALDSP data service developers to develop richer logic (particularly update logic, side-effecting operations, and other such functionality) without being forced to leave the world of XML and XQuery, the December 2007 release of ALDSP, ALDSP 3.0, has introduced XQSE, an XQuery Scripting Extension. In this paper we have provided a detailed description of the XQSE design, covering the ALDSP world that it is a part of, the key requirements that fed into its design, the syntax and semantics of each of the XQSE constructs, and its efficacy in simplifying various common ALDSP use cases. The resulting language design naturally embeds and extends XQuery much in the way that stored procedure languages in the relational world extended SQL. We have also touched on how the approach taken in XQSE compares with those proposed in the most widely known related XQuery extension languages.

References

[1] XQuery Scripting Extension 1.0 Requirements. World Wide Web Consortium. XQuery Scripting Extension Requirements. W3C Working Draft, 23 June 2007. See http://www.w3.org/TR/xquery-sx-10-requirements.

[2] XQuery 1.0. World Wide Web Consortium. XQuery 1.0: An XML Query Language. W3C Recommendation, 23 January 2007. See http://www.w3.org/TR/xquery//.

[3] XQuery Update Facility Requirements. World Wide Web Consortium. W3C Working Draft, 03 June 2005. See http://www.w3.org/TR/xquery-update-requirements.

[4] XQuery Update Facility 1.0. World Wide Web Consortium. W3C Working Draft, 28 August 2007. See http://www.w3.org/TR/xquery-update-10/.

[5] D. Florescu, A. Grünhagen, and D. Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. Proceedings of WWW2002, Honolulu, HI, May 2002.

[6] 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.

[7] 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.

[8] V. Borkar, M. Carey, D. Lychagin, T. Westmann, D. Engovatov, and N. Onose. "Query Processing in the AquaLogic Data Services Platform". Proceedings of VLDB '06, September 2006, Seoul, Korea.

[9] M. Carey et al, "Data Delivery in a Service-Oriented World: The BEA AquaLogic Data Services Platform", Proceedings of ACM SIGMOD 2006, June 2006, Chicago, IL.

[10] V Borkar, M. Carey, N. Mangtani, D. McKinney, R. Patel, and S. Thatte, "XML Data Services", Int'l. Journal of Web Services Research 1(3), 2006.

[11] K. Williams and B. Daniel, "An Introduction to Service Data Objects", Java Developer's Journal, October 2004.

[12] BEA AquaLogic Data Services Platform 3.0, BEA Systems, Inc., December 2007. See http://edocs.bea.com/aldsp/docs30/.

Appendix: EBNF for XQSE

Prolog ::= ((DefaultNamespaceDecl | Setter | NamespaceDecl | Import) Separator)* ((VarDecl | FunctionDecl | ProcedureDecl | OptionDecl) Separator)*

ProcedureDecl ::= "declare" ("readonly")? "procedure" QName "(" ParamList? ")" ("as" SequenceType)? (Block | "external")

QueryBody ::= Expr | Block

Statement ::= ControlStatement | UpdateStatement | ProcedureCall | Block | ProcedureBlock

ControlStatement ::= SetStatement | WhileStatement | IterateStatement | TryStatement | IfStatement | ReturnStatement | ContinueStatement | BreakStatement

ValueStatement ::= NonUpdatingExprSingle | ProcedureCall | ProcedureBlock

UpdateStatement ::= ExprSingle /*Must be an updating expression*/

ProcedureCall ::= FunctionCall /* Restricted to procedures */

ProcedureBlock ::= "procedure" Block

ReturnStatement ::= "return" "value" ValueStatement

SimpleStatement ::= SetStatement | ReturnStatement | IfStatement | ContinueStatement | BreakStatement | UpdateStatement | ProcedureCall

BlockStatement ::= Block | ProcedureBlock | WhileStatement | IterateStatement | TryStatement

Block ::= "{" (BlockDecl ";")* ((SimpleStatement ";") | BlockStatement (";")?)* "}"

BlockDecl ::= "declare" "$" VarName TypeDeclaration? (":=" ValueStatement)? ("," "$" VarName TypeDeclaration? (":=" ValueStatement)?)*

SetStatement ::= "set" "$" VarName ":=" ValueStatement

WhileStatement ::= "while" "(" NonUpdatingExpr ")" Block

IterateStatement ::= "iterate" "$" VarName PositionalVar? "over" ValueStatement Block

IfStatement ::= "if" "(" NonUpdatingExpr ")" "then" Statement ("else" Statement)?

TryStatement ::= "try" Block CatchClauseStatement+

CatchClauseStatement ::= "catch" "(" NameTest ("into" VarNameExpr (("," VarNameExpr)? "," VarNameExpr)?)? ")" Block

ContinueStatement ::= "continue" "(" ")"

BreakStatement ::= "break" "(" ")"

NonUpdatingExpr ::= Expr /* Must be non-updating */

NonUpdatingExprSingle ::= ExprSingle /* Must be non-updating */