DXL: Data eXchange Language

Roelof van Zwol V. Jeronimus M. Fokkinga Peter M.G. Apers

University of Twente,
Centre for Telematics and Information Technology,
Department of Computer Science
P.O.box 217, 7500 AE, the Netherlands
{zwol, jeronimu, fokkinga, apers}@cs.utwente.nl

Abstract

With large volumes of data being exchanged on the Internet, query languages are needed to bridge the gap between databases and the web. Furthermore, the differentiation in data types used by web-based applications is ever growing, despite all standardization efforts. The Data eXchange Language (DXL) provides an extensible base language designed to exchange data from heterogeneous sources into a single target.

One application of DXL, the focus in this article, is to retrieve data from databases, and yield the result in an XML document. However, the real application area of DXL is much broader since DXL provides a framework which allows data of a particular source to be queried and/or constructed by its original query language. This is achieved by DXL's extensibility mechanism which allows other query languages to be embedded into a DXL query.

The scope of this article is to compare DXL to other related query languages, discuss DXL's features and architecture, and present the base language definition of DXL. Furthermore we will discuss two extensions of DXL which allows us to query and construct databases and XML documents. Finally we will use these extensions in a newsgroup example, to illustrate DXL's main features, with respect to querying heterogeneous sources, and its recursive behavior.

1 Introduction

The need to exchange information on the Internet is growing, like the Internet itself. Due to the diversity in data formats one or more conversions are needed to exchange data between web-based applications, since a commonly accepted data exchange standard is lacking. The eXtensible Markup language (XML), a W3C[1] standard, is developed over the past few years to provide a universal format for structured documents and data on the web. A collection of related standards is defined around XML which (in general) aim to support the transformation of one XML-document into another.

When it comes to data exchange over web-based applications, these standards, like XQuery [CFR+01], XSL-T [WWWC99a], but also other query languages proposed by the various research groups involved, can be invoked to do the job. Provided of course that the applications are capable to handle their input and output in XML.

However, the majority of todays (web-based) applications are not using XML as their exchange format. Unfortunately, this complicates the exchange of data over the web. The Data eXchange Language (DXL) provides a framework for data exchange over (web-based) applications. More precise, DXL allows heterogeneous sources to be queried, and the results to be integrated into a single target. To achieve this, DXL uses an extensible mechanism which allows other query languages to be embedded into a single DXL query. A DXL extension is a plugin which allows DXL to query and/or construct a single data type.

[1] The World Wide Web Consortium, home-page: `http://www.w3.org/`

State of the art

The Data eXchange Language (DXL) described in this article, depends heavily on XML related technology. However, it cannot be compared with XML query languages, such as XQuery [CFR+01], Quilt [CRF00], XML-QL [DFF+99] and others [BC00, CCD+99, MFK01]. There are similarities, but most of them are more related to the use of XML, rather than to the functionality offered by those languages. The main reason for this is that data-type dependent query functionality is kept out of the DXL control language. The DXL control language itself only provides the functionality needed to connect to, and exchange data between, heterogeneous sources and targets.

More closely related is RXL [FST00, FMT01]. RXL stands for Relational to XML transformation Language, and is a declarative query language that aims at querying a relational DBMS, and viewing the results in XML. RXL forms the main source of inspiration for DXL. However, the scope of DXL is larger, since it does not aim at a specific data-type, like RXL does. Furthermore, RXL has a more limited power when it comes to constructing structures with an arbitrary level of nesting (recursion). This is caused by the explicit definition of the target structure in the RXL query, while DXL follows a more data-driven approach, where based on templates the target structure can be defined recursively. Also interesting is the reverse transformation of RXL, i.e. transformations of XML into a relational database. This is done in STORED [AFS99].

A second source of inspiration is XSL-T [WWWC99a], the W3C standard for XML Style-sheet Languages. XSL-T uses, like DXL, a template-based approach. But unlike DXL, XSL-T templates are called based on the input XML document. Again XSL-T can be seen as a subset of DXL, since it allows any XML document to be transformed into any data-type. The DXL extension for XML even borrows some of the instructions from XSL-T to construct new XML documents.

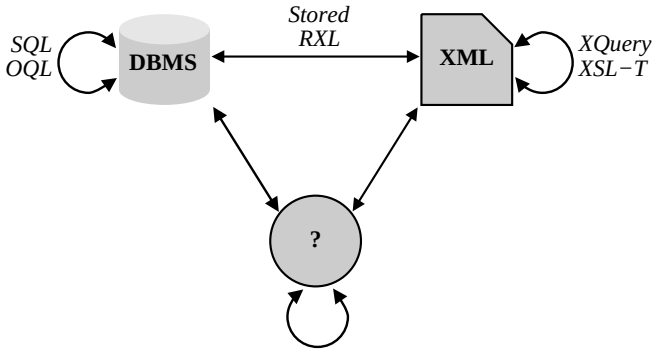

Figure 1: DXL transformations.

The arrows drawn in Figure 1 show possible transformations, supported by DXL, under the condition of course that a DXL extension for that particular data-type is available. For some transformations the related query languages are positioned around the arrows, indicating their relation to DXL, as discussed above.

Organization of this article

We will start the remainder of this article by discussing DXL's features and applications, DXL's architecture and a simple DXL example in Section 2. Section 3 provides insight into the control language. The extensibility of DXL is presented in Section 4. In that section, two extensions of DXL are discussed: the Postgres plug-in, and the XML plug-in. A case-study, based on Newsgroups, is presented in Section 5. It illustrates the power of DXL, when it comes to building recursive target structures, and querying heterogeneous sources. We will conclude this article in Section 6 with the conclusions.

2 Data eXchange Language

One of the biggest problems for general *data exchange* is the flexibility needed to deal with all kinds of different *data-types*. In fact, each data-type has its own specific characteristics. To exploit these characteristics in the best way possible, query languages are needed that solely target a specific data-type. When defining just one query language that targets heterogeneous sources (data-types), compromises will have to be made which will result in a less powerful query language. Therefore DXL does not incorporate all kinds of functionality that target a specific data-type directly.

The goal is to provide a flexible and extensible framework that allows other query languages to be embedded. Besides that, DXL needs to be capable of transporting any data from its sources into the target of a DXL query.

In Section 2.1 the features supported by the DXL framework, and some applications of DXL are discussed. A trivial example is presented in Section 2.2, to give a first impression of DXL. The system architecture that implements DXL is presented in Section 2.3.

2.1 Features and applications of DXL

DXL explores the following four main features:

- **Heterogeneity**. The main contribution of DXL is that it offers a data-type independent approach for data exchange over multiple sources. To achieve this, all data obtained from the sources is stored internally, in the form of a set of XML elements, as the parameters of a DXL query.

- **Extensibility**. To embed other query languages in DXL, extensibility is crucial. The DXL control language has two instructions (`query` and `construct`) for this purpose. These instructions must be defined for each data-type in a plug-in (extension). The `query` instruction connects to and retrieves the requested information from the source, and stores the intermediate result in parameters. The `construct` instruction retrieves information from DXL's parameters and uses it to construct the desired target data-type.

- **Templates**. The behaviour of a DXL query is specified in one or more templates. Each template defines a part of the query. The body of a template often contains a sequence of instructions that defines the template's behaviour. Within a template it is only possible to query one source. Thus if multiple sources need to be queried, there are at least as many templates.

- **Recursion**. DXL has a recursive behaviour, i.e. templates can be called recursively which allows source/target data-types to be queried/constructed recursively. As a result, arbitrarily deeply nested structures can be queried and constructed. In case of newsgroups, where messages can have any number of follow ups, such a property is essential. The recursive behaviour of DXL will beillustrated in the example of Section 5.

In practise, DXL queries have the intention to grow relatively large which will make it unlikely that inexperienced users will formulate queries by hand. However, due to its XML-based nature, it is fairly easy to generate DXL queries. The *Webspace Query Tool* [vZ02] uses DXL internally to construct complex queries over web-based document collections. It fetches meta-data from the webspace object server and integrates these results with XML fragments that are obtained from several XML documents. As a result, the generated DXL query constructs an XML document which yields the requested information. In another application, DXL is used in combination with the *Feature Grammar Engine* [WSK99, dVWAK00] developed at CWI for the automatic reconstruction of web-sites, and multimedia retrieval in combination with the *Webspace Method* [BvZW+01, vZA00].

2.2 'Hello World! example

Before presenting the architecture and language definition, the 'Hello world' example will be discussed to give a first impression of DXL. The code fragment of Figure 2.a shows a DXL query that consists of a single template. Line 2 of this query specifies that the plug-in defined in the driver 'dxl.client.xml' is used to create the target XML document 'result.xml'. To construct this document all child instructions embedded in the body of the template with name 'main' are executed sequentially. The precise syntaxes and semantics of DXL are explained in Sections 3 and 4.

The execution starts with the definition of a parameter with the name 'param1' and value 'Hello World!' in line 5. Next the construct instruction of lines 6 – 10 is called. The construct, which is implemented by the plug-in 'dxl.client.xml', allows the instruction element to be embedded in its body. The element defines the root element 'message' of the target document. Within the element instruction an attribute instruction is called, which defines the attribute with the name 'text' for the XML element 'message'. The value 'Hello World!' stored in the parameter $param1 is passed into the value of the attribute. The result of the query is a simple XML document, as shown in Figure 2.b.

```
01. <?xml version='1.0' encoding='UTF-8'?>
02. <dxl_query driver='dxl.client.xml' target='result.xml'>
03.
04.   <template name='main'>
05.     <param name='param1' select='Hello World!'/>
06.     <construct>
07.       <element name='message'>
08.         <attribute name='text' value='$param1'/>
09.       </element>
10.     </construct>
11.   </template>
12.
13. </dxl_query>
```

(a) DXL query

```
01. <?xml version='1.0' encoding='UTF-8'?>
02. <message text='Hello world!'/>
```

(b) DXL Target: result.xml

Figure 2: 'Hello World' example.

2.3 DXL's system architecture

DXL's implementation is based on the system architecture presented in Figure 3. The architecture consists of three components: a *kernel*, a *control language*, and one or more *plug-ins*. The current implementation is based on Java which allows DXL to be used on various platforms. The kernel contains four classes: *DxlEngine*, *DxlDriver*, *DxlInterface*, *DxlInstruction*. The process starts by initiating the *DxlEngine* with the *DxlDrivers*, which refer to the available plug-ins. Next a DXL query is validated by the *DxlEngine*, and a connection is made with the target of the query.

The *DxlEngine* is then ready to process the query. Each query should at least contain one template with the name 'main'. The *DxlEngine* will start executing the DXL instructions that are found in the body

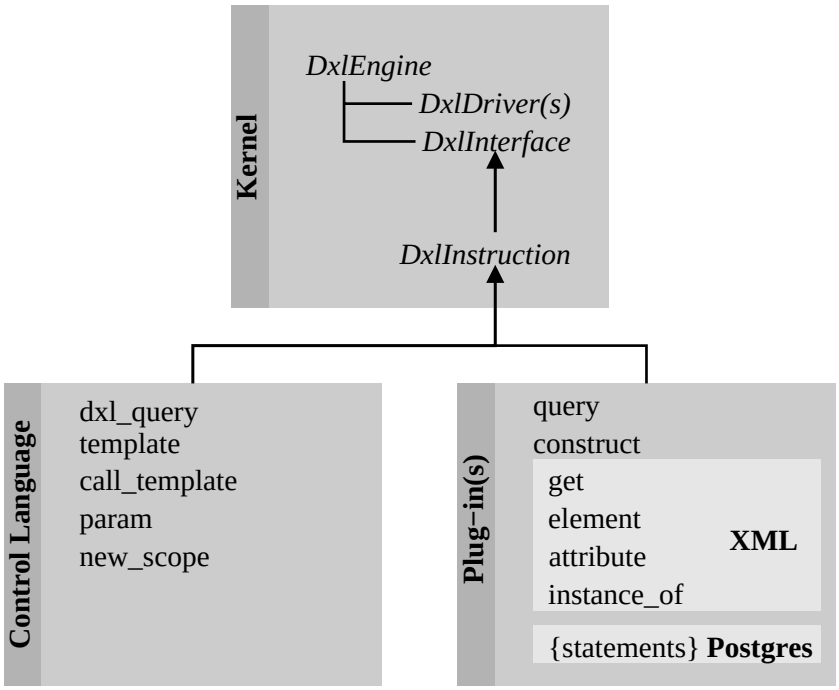

Figure 3: DXL system architecture

of that template. All instructions, both the control language instructions, and the instructions defined in the DXL plug-ins, obey the *DxlInterface*, and use the *DxlInstruction* as the abstract implementation of all instructions.

Each plug-in requires that at least the construct and query instructions are implemented. All instructions base their implementation on the pre-defined abstract *DxlInstruction*, and only have to implement their own *process()*-method, which specifies the behaviour of an instruction.

The *control language* component of Figure 3 contains the instructions that form the DXL framework. The *plug-in* component shows the instructions, that are defined for the two extensions discussed in Section 4: the XML plug-in and the Postgres plug-in.

3 Control language

The goal of this section is to present the syntaxes and informal semantics of the DXL control language. The grammar used to describe the control language is based on the following notational conventions:

Iteration is denoted by $*$.
Sequencing is denoted by j u x t a p o s i t i o n.
Optionality is denoted by underlining.
Literal text is denoted by sans serif font.
Non-terminals are denoted by identifiers in *italics*.
Meta-notation is denoted by ⟨Roman font with double angle brackets⟩.

The grammar of the control language uses the following non-terminals, which are explained during the discussion of the semantics of DXL:

tname ::= $([a-z] \mid [A-Z] \mid [0-9])*$

dname ::= *tname*(.*tname*)*

location ::= ⟨string (preferably a URI)⟩

pname ::= ⟨subset of the XPath standard⟩

expression ::= $*pname* | ⟨text, not beginning with $⟩

Below the first two rules of the grammar for the control language are given:

dxl_query ::= <dxl_query driver=*dname* target=*location*>
*template template**
</dxl_query>

template ::= <template name=*tname*> *instruction** </template>

A *dxl_query* is a query on one or more data sources, producing the answer in one target. The target of a *dxl_query*, together with the corresponding driver, are defined as attributes of the root tag of a *dxl_query*. A driver is of the form *dname* which forms a textual string separated by dots. Since the implementation is based on Java, the driver always refers to a specific Java package that implements the plug-in. The value of the optional target attribute points to a location, which can be interpreted by the plug-in. If no target is specified, the output will be send to standard out. It is preferred that a *location* is specified as a valid Uniform Resource Identifier (URI)[WWWC93].

The body of a *dxl_query* contains at least one *template*. Each template is uniquely identified by a name attribute of the form *tname* (a simple textual string). The semantics of DXL also prescribe that there should be a *template* with the name 'main'. The execution of a *dxl_query* will start with processing the *instruction*s that are found in the body of this *template* in a sequential order.

The grammar of an *instruction* is expressed by the following rule:

instruction ::= *param* | *call_template* | *new_scope* | *query construct*

Within the body of a template, the following *instruction*s can be found: *param*, *call_template*, *new_scope*, and *construct*, which can be directly preceded by a *query* instruction. The *query* and *construct* instructions provide the means to embed foreign languages in DXL. By one or more *construct*s the structure of the target data-type is defined. Each *query* instruction, however, uses its own driver and source attribute, to connect to and to query a particular source. Furthermore, only one *query* instruction is allowed in the body of a *template*.

Internally, a *query* instruction generates a *result set* which contains the results of the queries executed on the source of that *query*. Instead of calling the following *construct* from the body of the (parent) *template*, the *construct* is then called from within the *query* instruction, for each item in the result set.

Below, the 'general' grammar rules for *query* and *construct* are given:

query ::= <query driver=*dname* source=*location*>
⟨*this.driver* →`query`⟩
</query>

construct ::= <construct>
⟨*dxl_query.driver* →`construct`⟩
</construct>

The particular plug-in that should be used for the execution of a *query* instruction is determined at run-time by the driver attribute of the *query*. Also at run-time, the *dxl_query* tries to connect to that source, and starts executing the instructions that are embedded in the body of the *query*. Each plug-in defines what instructions are allowed in the body of a *query* or *construct* instruction. Section 4 describes the different implementation strategies that can be followed by a plug-in to address a particular data-type. At the end of that section, two plug-ins are described that follow different approaches.

Crucial for the success of DXL is its parameter mechanism. All data exchange from heterogeneous sources to the target of a *dxl_query* is carried out by the parameter mechanism. Each *dxl_query* starts with an empty *parameter stack*. Whenever a particular source is queried, the intermediate results from that source are pushed on top of the parameter stack. When executing the following *construct* instruction, the required information is taken from the parameter stack and transferred to the target.

The grammar of the control language is extended by the following rules, which describe the *instruction*s that can influence the state of the parameter stack:

param ::= <param name=*pname* select=*expression*/>

call_template::= <call_template select=*tname*/>

new_scope ::= <new_scope/>

With *param*, a new parameter with the name *pname* is pushed on top of the parameter stack. The value of this parameter is defined by the attribute select and is of the form *expression*. An *expression* can either be the reference to a parameter, using the notation $*pname* or a text fragment.

Each parameter in the stack is referred to by its name. Internally, the value of a parameter in DXL is represented by an XML *document fragment*. A document fragment has a hierarchical structure onto which (complex) structures of the source data-type that is queried, are mapped. Mappings from relational, object oriented or many other structures to a hierarchical structure have been described in literature [EN94] and should be used to transfer the data from the external source to the internal parameter stack of DXL.

Another advantage of using XML for the internal representation of data is that XPath [WWWC99b] expressions (*pname*) can be used to traverse through the document fragment of a parameter. With XPath expressions the relevant fragments of data can be fetched directly from the document fragment, and be transferred into the result of the *dxl_query*.

With the *call_template* other templates, embedded in the body of *dxl_query*, are called (recursively). For that purpose, the attribute select is used, referring to the unique name (*tname*) of a *template*.

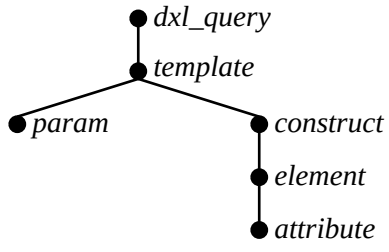

Figure 4: DXL execution tree for the 'Hello world' example.

While processing a *dxl_query*, a dynamic *execution tree* is built and executed, following a (left-to-right) depth-first approach. The level of nesting in the execution tree is equal to the number of nested DXL instructions. In Figure 4 the execution tree is given for the 'Hello world' example of Figure 2. The maximum level of nesting for that DXL query is five. When the execution tree is entering a deeper level,

a new *scope* is pushed on top of the parameter stack. As soon as all instructions are carried out for a particular branch of the execution tree, a higher level is addressed and the top-most scope is popped from parameter stack. Figure 5 shows the state of the parameter stack, while processing the *attribute* instruction. At that point, the (top-most) value of 'param1' is fetched from the parameter stack.

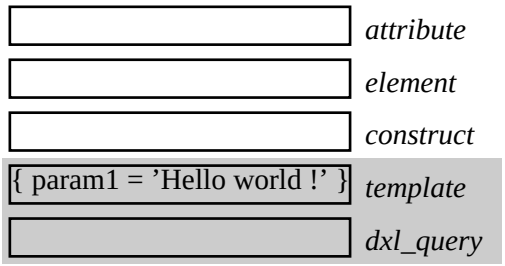

Figure 5: DXL parameter stack for the 'Hello World!' example

The last instruction defined in the DXL control language is *new_scope*. The effect of invoking this instruction is that the parameter stack will become temporarily inaccessible for all instructions, that belong to the branch of the execution tree underneath the instruction, from which the *new_scope* is called. Lets return to the 'Hello World!' example again. Inserting a *new_scope* in between lines 6 and 7 of the query (Figure 2), will cause the grey shaded part of the parameter stack as presented in Figure 5 to become inaccessible for all instructions underneath the *construct* instruction. As a result, the execution of the example will fail, because the *attribute* instruction can no longer access the parameter 'param1'.

4 Extending DXL

From the control language definition, it might be clear that DXL really is a framework for data exchange, since it contains no instructions that target a specific data-type directly. For that purpose DXL's extensibility mechanism is used. At run-time, when executing a DXL query, a connection is made to the source or target, using the driver that points to a specific *DXL plug-in*. I.e., such a driver string refers to the name of a Java package implementing the plug-in for a single data-type. Each plug-in should at least contain an implementation of the instructions (classes) query and construct, which are responsible for querying and constructing a certain data-type.

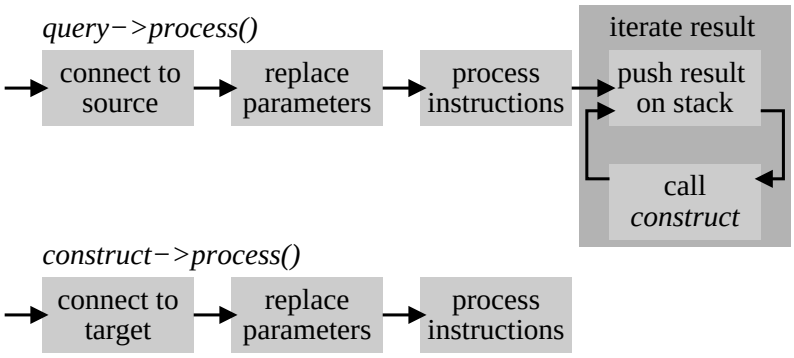

Figure 6: General procedure for creating DXL plug-ins.

In Figure 6 a general procedure for defining DXL plug-ins is presented. Each plug-in should implement at least two classes, to allow a specific data-type to be queried and constructed. The *process()*-method is reserved for this purpose, and is the only method that should be implemented by a class, defining

an instruction. For both the *query* and *construct* instructions a general procedure can be followed, as proposed in the figure.

Both procedures start by connecting to the source or target. Next, parameters are replaced, and the child-instructions, encapsulated in the body of the instruction, are executed. This ends the procedure for the *construct*. For the *query* instruction two extra steps are required. While iterating over the result set, intermediate results are stored on top of the parameter stack, and the related construct is called to process the results of the query.

Although each plug-in has its own characteristics, three categories are distinguished based on the syntax of a plug-in: (1) *XML-based* extensions, (2) *plain-text* extensions, and (3) *mixed plain-text / XML* extensions.

Defining a pure XML-based extension requires more effort to be put in the creation of the plug-in. Usually this means that more classes need to be implemented. The bright side of defining an XML-based extension is that instructions defined by the control language, or even by other already existing plug-ins, can be easily integrated.

On the other hand, a plain-text extension can be realised really quickly, but will not fully benefit from DXL's extensibility. The third category of extensions, consisting of mixed plain-text / XML extensions, often are a good compromise. They integrate the best of the before-mentioned categories.

In the remainder of this section two DXL plug-ins are presented: the Postgres plug-in, which is an example of a plain-text extension, and the XML plug-in, an example of an XML-based extension.

4.0.1 The Postgres plug-in

Although the DXL control language is specified using XML as its syntax, any DXL extension is free to use its own syntax. The Postgres plug-in embeds the Postgres statements directly as plain-text into the body of the query and construct instructions. As a result, a simple DXL extension is created.

query ::= <query driver='dxl.client.db.postgres' source=*location*> ⟪PostgreSQL statement⟫ </query>

construct ::= <construct> ⟪PostgreSQL statement⟫ </construct>

The driver of this plug-in is defined in the package dxl.client.db.postgres, and uses the JDBC[2] standard (URI) to connect to the Postgres database server.

The Postgres plug-in follows the steps of the procedure, as described in Figure 6. During the first step, the plug-in connects to the source or target database. Secondly, it replaces all parameters found in the PostgreSQL statements by the values, obtained from the parameter stack. I.e., to be able to distinguish the DXL parameters from the regular PostgreSQL statement, the parameters are of the form: ${*pname*}. The statements are executed on the database server, and the results obtained in a result set. During the iteration steps, a row is fetch from the result set, and stored as a parameter of the form: *tablename*@*attributename*. The related *construct* is called for each row of the result set.

4.0.2 The XML plug-in

Contrary to the Postgres plug-in, the XML plug-in follows an XML-based approach. Many XML query languages have been proposed, that can be used to realise the XML extension. The implementation of the XML plug-in is based on a mixture of the XPath and XSL-T standards [WWWC99b, WWWC99a].

The grammar of the *query*-side of the XML plug-in is based on the following rules:

[2] JDBC stands for Java Database Connectivity

xpath_expr ::= ⟪expression, according to the XPath standard⟫

query ::= <query driver='dxl.client.xml' source=*location*>
get∗
</query>

get ::= <get from=*xpath_expr* select=*xpath_expr*/>

The *query* instruction is defined in the package dxl.client.xml. The body of *query* contains a sequence of *get* instructions, which builds a result set containing the results of the (*get*-)queries on an XML document.

The *get* instruction uses the select attribute to fetch the requested data from the document. The optional from is used to jump to a particular node in the XML document, from which point the *select* will operate. Both attributes use XPath expressions to traverse through the XML document. The result of a *get* instruction is always a result set, containing zero or more XML document fragments. Each document fragment is stored in DXL, and is referred to by the name of the root element of the document fragment.

The *construct*-side is partially based on the grammar below:

construct ::= <construct> *XML_instr*∗ </construct>

XML_instr ::= *element* | *attribute* | *instance_of* | *call_template* | *param* | *new_scope*

The body of *construct*, as defined in *XML_instr* uses three XML specific instructions to create an XML document: *element*, *attribute*, and *instance_of*. Furthermore, it allows three instructions of the DXL control language to be called from the body of a *construct*: *call_template*, *param*, and *new_scope*. The result is a tight integration of the plug-in with the DXL framework which allows a more flexible definition of DXL queries.

Below, the grammar for the XML-specific instructions needed to construct an XML document is given:

element ::= <element name=*expression*> *XML_instr*∗
</element>

attribute ::= <attribute name=*expression* value=*expression*/>

instance_of ::= <instance_of select=*expression*/>

The *element* instruction is used to add an element to the result of the query. To be able to create the hierarchical structure of an XML document, a sequence of *XML_instr* can be called from the body of an *element* instruction. The name of the element is the value returned by the *expression*.

The *attribute* instruction is used to add an attribute definition to the current element. Both name and value are required attributes of this instruction, and its values are defined in an *expression*.

The last instruction is *instance_of*. This instruction is mostly used to add text fragments to the body of an XML element. The select evaluates an *expression*. The result can be a textual string, or even an entire XML document fragment, copied directly from the parameter stack into the result of the DXL query.

5 The 'Newsgroup' case study

The power of DXL will be illustrated by two example queries. Both queries are based on the 'Newsgroup' case. On a newsgroup, messages are posted which can be characterised by an author, a subject, and a message body. For each particular message one or more follow-ups can be posted. As a result a tree-like structure is created, having an arbitrary level of nesting. The first query shows how DXL's recursion mech-

anism is used to generate the tree structure of a newsgroup from a flat database table. The second query illustrates how heterogeneous sources are queried, and the results obtained from those sources combined in a single target.

5.1 Query 1: Recursion

The data of the first query is fetched from a Postgres database with the name 'newsgroups'. This database contains various newsgroups, one of which contains the messages posted on the newsgroup 'utwente.music'. These newsgroup tables all contain the attributes: *id*, *parent*, *sender*, *subject* and *message*. In Appendix A.1 the definitions of these database tables are given.

The goal of the first query is to reconstruct the tree structure of this newsgroup from the (flattened) relational table. This requires a recursive operation, because for each message, it will have to be determined, whether the newsgroup contains some follow-up messages. The DXL query that reconstructs the newsgroup tree in an XML document is given in Figures 7, 8, and 9.

Figure 7 shows the 'main' template which indicates the beginning of the query. Line 2 indicates that the result of the query is an XML document, and that the result is stored in the target 'result.xml'. The body of this template only consists of a construct instruction (lines 4–8), where the root element of the XML document is set, and a call is made to the template with name 'groups' (line 6).

```
01. <?xml version='1.0' encoding='UTF-8'?>
02. <dxl_query driver='dxl.client.xml' target='result.xml'>
03.   <template name='main'>
04.     <construct>
05.       <element name='newsgroups'>
06.        <call_template select='groups'/>
07.      </element>
08.    </construct>
09.   </template>
```

Figure 7: DXL Newsgroup example: 'main' template.

Figure 8 shows the 'groups' template. The body of this template consists of four instructions that have to be executed sequentially. Starting with the two param instructions of lines 12 and 13, a new parameter node 'newsgroup' is created, containing the attributes 'name' and 'table' with their values. An additional parameter (line 15) is needed for the initialisation of the instructions in the 'news' template. The parameter 'n@id' is set to -1, to find the root messages of the newsgroup.

At the end of this template, the construct instruction is evaluated. The instruction starts with the element declaration 'newsgroup', after which an attribute is added to this element (line 18). Next, A call_template is executed to call the *news* template (line 19).

The 'news' template, shown in Figure 9, is the most interesting template of the entire query, since its task is to obtain the data from the newsgroup table, and to perform the recursive operation that creates the tree structure.

The body of this template consists of two instructions. The first instruction is a query instruction (lines 24–27), directly followed by a construct instruction (lines 28–38). Starting with the query instruction, a connection is made to the database, and the statement in line 27 is executed, following the replacement of the parameters by their corresponding values. In the next step an iteration is performed over the result set, and for each result the construct instruction of lines 28–38 is executed.

This construct instruction starts with adding a 'message' element to the result of the query (lines 29–37), followed by some attribute declarations, defined in the body of the element instruction, see lines 30–32. Lines 33–35 contain an element instruction, which adds the XML element 'body' as a child element to

```
10.   <template name='groups'>
11.
12.     <param name='newsgroup@name' select='utwente.music'/>
13.     <param name='newsgroup@table' select='utwente_music'/>
14.
15.     <param name='n@id' select='-1'/>
16.     <construct>
17.       <element name='newsgroup'>
18.         <attribute name='name' value='$newsgroup@name'/>
19.         <call_template select='news'/>
20.       </element>
21.     </construct>
22.   </template>
```

Figure 8: DXL Newsgroup example: 'groups' template.

'message'. Within the XML 'body' element the value of '$n@message' is added, using the instance_of instruction on line 34. The parameter '$n@message' contains the text of the current message. Finally, a **recursive** call to the 'news' template is made by the call_template instruction on line 36. This template will be called, as long as there are follow-up messages on the current message.

5.2 Query2: Heterogenity

The second query elaborates on the first query. Instead of just reconstructing one newsgroup, the goal is to reconstruct several newsgroups. This illustrates DXL's power with respect to querying heterogeneous sources, and integrating the result into a single target. The target is again an XML document, with the name 'result.xml'. For this query, we can reuse the templates 'main' and 'news', shown in Figures 7, and 9, respectively. The 'groups' template needs to be altered, as is shown in Figure 10. Lines 11–14 show a query instruction, instead of the earlier parameter definitions of 'newsgroup'. The query connects to the XML document 'newsgroups.xml'[3] which contains information about newsgroups, i.e., for each newsgroup, its name and the corresponding table name.

Instead of just declaring the 'newsgroup' parameter, the get instruction fetches all *newsgroup* nodes under the root *newsgroups*, and iterates over the result set of this query, More precise, for each item (XML node) in the result set of the query, a 'newsgroup' parameter is stored in the DXL framework, and the construct of lines 17–22 is executed.

The result of the second query is shown in Appendix A.3. A screen shot is taken from Mozilla, which shows a visualisation of the XML document. For that purpose, an XSL-T stylesheet is used in combination with Cocoon[4].

6 Conclusions

The need to open database technology for communication over the Internet is growing. The lack of a commonly accepted standard for data exchange is complicating this job. Specialised query languages are introduced to exchange two or more data sources.

The intention of DXL is to provide a more general framework that is capable of querying heterogeneous sources and of combining the intermediate results, through the intervention of XML, into a single target. 'Unwilling' to make compromises all functionality that addresses a specific data-type is left out of the DXL

[3] The content of XML document 'newsgroups.xml' is shown in Appendix A.2

[4] `http://xml.apache.org/cocoon/`

```
23.   <template name='news'>
24.     <query driver='dxl.client.db.postgres'
25.            source='jdbc:postgresql://pub/newsgroups'>
26.       select * from ${newsgroup@table} n where n.parent='${n@id}';
27.     </query>
28.     <construct>
29.       <element name='message'>
30.         <attribute name='id' value='$n@id'/>
31.         <attribute name='subject' value='$n@subject'/>
32.         <attribute name='sender' value='$n@sender'/>
33.         <element name='body'>
34.           <instance_of select='$n@message'/>
35.         </element>
36.         <call_template select='news'/>
37.       </element>
38.     </construct>
39.   </template>
40. </dxl_query>
```

Figure 9: DXL Newsgroup example: 'news' template.

control language. Through the definition of DXL extensions, any query language can be embedded in a DXL query. This allows DXL to optimally exploit the characteristics of a specific data-type, while, if the functionality would have been offered by DXL directly, compromises would have to be made, reducing the power of the query language.

Two extensions of DXL have been described and illustrated in an example, to show the power of DXL in this field. Recently proposed query languages, such as XQuery, also claim to support this kind of functionality (directly). Further research is needed to investigate the similarities and differences in this field. Contrary to XQuery, DXL's template-based nature allows a recursive definition of the target structure which is especially useful when creating XML targets with a nested structure.

```
10.   <template name='groups'>
11.     <query driver='dxl.client.xml'
12.            source='newsgroups.xml'>
13.       <get from='newsgroups' select='newsgroup'/>
14.     </query>
15.     <param name='n@id' select='-1'/>
17.     <construct>
18.       <element name='newsgroup'>
19.         <attribute name='name' value='$newsgroup@name'/>
20.         <call_template select='news'/>
21.       </element>
22.     </construct>
23.   </template>
```

Figure 10: DXL Newsgroup example: revised 'newsgroups' template.

References

[AFS99] A.Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *proceedings of the ACM SIGMOD International Conference on Management of Data*, 1999.

[BC00] A. Bonifati and S. Ceri. Comparative analysis of five XML query languages. *SIGMOD record*, 29(1):68–79, 2000.

[BvZW+01] H.E. Blok, R. van Zwol, M. Windhouwer, M. Petkovic, P.M.G. Apers, M.L. Kersten, and W. Jonker. Flexible and scalable digital library search. In *proceedings of the twenty-seventh International Conference on Very large Data Bases (VLDB'01)*, Rome, Italy, September 2001.

[CCD+99] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a graphical language for querying and restructuring XML documents. In *proceedings of the International World Wide Web Conference (WWW*, pages 1171–1187, Canada, 1999.

[CFR+01] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: A query language for XML. Technical report, World Wide Web Consortium (W3C), `http://www.w3.org/TR/xquery`, Februari 2001.

[CRF00] D. Chamberlin, J. Robie, and D. Florescu. Quilt: an XML query language for heterogeneous data sources. *Lecture Notes in Computer Science*, December 2000.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. Querying XML data. *IEEE Data Engineering Bulletin*, 22(3):10–18, 1999. XML-QL.

[dVWAK00] A. P. de Vries, M. A. Windhouwer, P. M. G. Apers, and M. L. Kersten. Information access in multimedia databases based on feature models. *New Generation Computing*, 18(4):323–339, October 200.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems: Second Edition.* California: Addison-Wesley, 1994.

[FMT01] M. Fernandez, A. Morishima, and W.C. Tan. Publishing relational data in xml: the silkroute approach. *IEEE Data Engineering Bulletin*, 24(2):12–19, 2001.

[FST00] M. Fernandez, D. Suciu, and W.-C. Tan. SilkRoute: trading between relations and XML. In *proceeding of WWW9*, 2000.

[MFK01] I. Manolescu, D. Florescu, and D. Kossmann. Answering xml queries over heterogeneous data sources. In *proceedings on the International Conference on Very Large Data Bases (VLDB)*, Rome, Italy, September 2001.

[vZ02] R. van Zwol. *Modelling and searching web-based document collections.* PhD thesis, Centre for Telematics and Information technology, Enschede, the Netherland, April 2002.

[vZA00] R. van Zwol and P.M.G. Apers. The webspace method: On the integration of database technology with information retrieval. In *proceedings of Ninth International Conference on Information and Knowledge Management (CIKM'00)*, Washington DC., USA, November 2000.

[WSK99] M.A. Windhouwer, A.R. Schmidt, and M.L. Kersten. Acoi: A system for indexing multimedia objects. In *proceedings of International Workshop on Information Integration and Web-based Applications and Services*, Yogyakarta, Indonesia, November 1999.

[WWWC93] (W3C) World Wide Web Consortium. Uniform resource identifier (URI). http://www.w3.org/Adressing/, 1993.

[WWWC99a] (W3C) World Wide Web Consortium. The extensible stylesheet language (xsl). http://www.w3.org/Style/XSL/, November 1999. W3C Recommendation.

[WWWC99b] (W3C) World Wide Web Consortium. W3c - xpath. http://www.w3.org/TR/xpath, November 1999. W3C recommendation.

A 'Newsgroups' case-study

Abstract

This appendix contains the background material for the 'Newsgroup' case-study, which is used to illustrate DXL's capabilities, with respect to querying heterogeneous sources, and the recursive definition of a target data-type.

A.1 Newsgroup table definitions

The table definitions, presented in Figure 11, are used by the Newsgroups database.

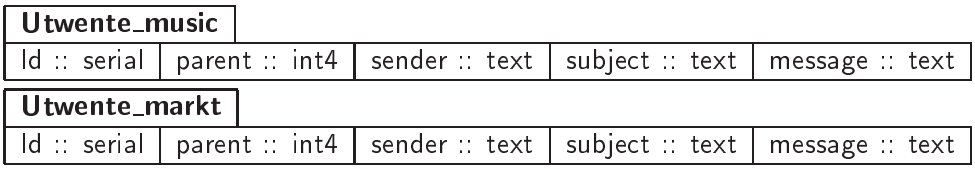

Figure 11: Table definitions for the *Newsgroups* database.

A.2 XML document 'newsgroup.xml'

In Figure 12 the content of the XML document 'newsgroups.xml' is shown. This document is used in the case study of Section 5.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<newsgroups>
  <newsgroup name="utwente.markt" table="utwente_markt"/>
  <newsgroup name="utwente.music" table="utwente_music"/>
</newsgroups>
```

Figure 12: XML document: *'newsgroup.xml'*.

A.3 Screen shot of 'newsgroup.xml'

Figure 13 shows a screen shot of the result of the DXL query discussed in Section 5.2. The visualisation is realised, using an XSL-T stylesheet, and Cocoon.

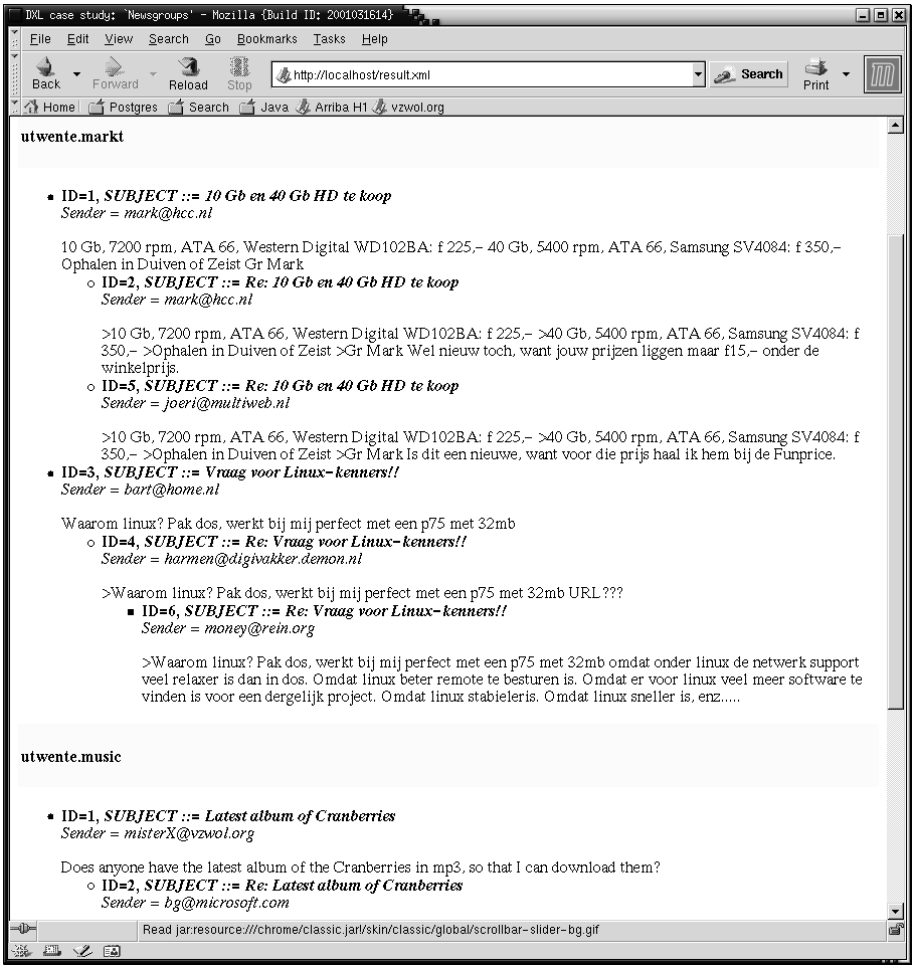

Figure 13: Screen shot containing result of Query 2.