

## **MANOEUVRING MULTI-DIMENSIONAL LANGUAGE DATA: A CODE SHUTTLE WITH XML AND DATABASE-OPERATION FUNCTIONS<sup>1</sup>**

J.H. KROEZE, T.J.D. BOTHMA, M.C. MATTHEE

### **ABSTRACT**

The article focuses on the conversion of linguistic data between an XML data cube and a three-dimensional array structure in Visual Basic 6 in order to eventually facilitate data access and manipulation. After a short consideration of the structures of the VB6 and XML databanks, conversion between the two is discussed (“round-tripping”), as well as essential database functions (create, read, update and delete) that may be performed on the linguistic data cube. The suggested software tool, therefore, acts as a “code shuttle” (a programme that moves data to and from two different storage structures) with added elementary database functionalities. Linguistic data from the Hebrew text of Genesis 1:1-2:3 is used to demonstrate the data operations.

### **INTRODUCTION**

In the electronic processing of language, one can concentrate either on the digital simulation of human understanding and language production, or on the most appropriate way to store and use existing knowledge. Both are valid and important. This endeavour falls in the second category, assuming that it is important to capture the results of linguistic analyses in well-designed, exploitable, electronic databases. XML, for example, can be used to mark up free text, to create a well-structured textual database. Since the data is separated from the manipulation and display thereof, the same data can be used for various purposes, and programmes or queries can be created to suit the researcher’s individual needs. This, however, necessitates the conversion of the data stored in XML format into a data structure, such as a three-dimensional

---

<sup>1</sup> This article is a revised and extended version of a short paper, “Round-tripping Biblical Hebrew linguistic data”, read at the IRMA 2007 conference, Vancouver, British Columbia, Canada, May 19-23, 2007. It is an edited version of a chapter in an unpublished doctoral thesis (Kroeze 2008:123-145).

array, which can then be processed efficiently by a computer programme.

This article focuses on the conversion of linguistic data of Genesis 1:1-2:3 between an XML data cube and a three-dimensional array structure in Visual Basic 6 in order to eventually facilitate data access and manipulation. After a short consideration of the structures of the VB6 and XML databanks, conversion between the two is discussed (“round-tripping”), as well as essential database functions (create, read, update and delete) that may be performed on the clause cube. The algorithms together build a shuttling tool that moves linguistic data between different storage structures. Some value is added to the transportation function of the tool by adding elementary database functionalities. Linguistic data from the Hebrew text of Genesis 1:1-2:3 is used to demonstrate the data operations.

Due to the interdisciplinary nature of this study, the target audience of the article is threefold. Humanities scholars, who may find the sections of programming code too technical, may focus on the explanatory discussions in order to gain a better understanding of the typical algorithmic processes that lie behind the software that they often use to study biblical texts, especially since a Biblical Hebrew text is used for experimentation. Computing scholars, who may find the programming sections relatively simple, could again be interested to see how database technology may be applied to linguistic knowledge. Readers who are skilled in both disciplines may find the manual-like discussions of the code useful in order to apply or adapt the logic for their own purposes.

The XML document containing the text and mark-up of Genesis 1:1-2:3 may be regarded as a “*native XML database*” (i.e., “a database designed especially for storing XML”), while the VB6 programme may be regarded as a “*content management system*” (i.e., “an application designed to manage documents and built on top of a native XML database”) (Bourret 2003). The native XML database stores the XML content, which consists of the original text (a phonetic version of the Hebrew text of Gen 1:1-2:3) with all the added XML tags and mark-up (indicating syntactic and semantic functions, etc.). Although it is a very basic system, it does fulfil the basic requirements to

qualify as a native XML database (cf. Vakali et al. 2005:65, 67): the hierarchically-structured XML document serves “as the fundamental unit of logical storage”, the schema serves as the “logical model for the XML document itself”, and the XML file saved on the permanent storage device uses a sequential, text-oriented file structure as “underlying physical storage model”.<sup>2</sup> The hierarchical structure of the XML database is demonstrated by the extract shown in Figure 1. The content management system is a database management system that operates on the data to allow editing and various views according to possible user needs.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Genesis1v1-2v3>
  <clause>
    <clauseno>Gen01v01a</clauseno>
    <headers>
      <header>Level</header>
      <header>Phrase1</header>
      <header>Phrase2</header>
      <header>Phrase3</header>
      <header>Phrase4</header>
      <header>Phrase5</header>
    </headers>
    <level1>
      <leveldesc>Phon:</leveldesc>
      <phrase1>brešit</phrase1>
      <phrase2>bara</phrase2>
      <phrase3>elohim</phrase3>
      <phrase4>et hašamayim ve'et ha'arets</phrase4>
      <phrase5>-</phrase5>
    </level1>
    <level2>
      <leveldesc>Translation:</leveldesc>
      <phrase1>in the beginning</phrase1>
      <phrase2>he created</phrase2>
      <phrase3>God</phrase3>
      <phrase4>the heaven and the earth</phrase4>
      <phrase5>-</phrase5>
```

<sup>2</sup> According to Smiljanić et al. (2002:17), however, a native XML database is **not** required to have the third property: it can be built on various types of databases or proprietary storage formats.

```

</level2>
<level3>
  <leveldesc>Phrase type:</leveldesc>
  <phrase1>PP</phrase1>
  <phrase2>VP</phrase2>
  <phrase3>NP</phrase3>
  <phrase4>NP</phrase4>
  <phrase5>-</phrase5>
</level3>
<level4>
  <leveldesc>SynF:</leveldesc>
  <phrase1>Adjunct</phrase1>
  <phrase2>Main verb</phrase2>
  <phrase3>Subject</phrase3>
  <phrase4>Object</phrase4>
  <phrase5>-</phrase5>
</level4>
<level5>
  <leveldesc>SemF:</leveldesc>
  <phrase1>Time</phrase1>
  <phrase2>Action</phrase2>
  <phrase3>Agent</phrase3>
  <phrase4>Product</phrase4>
  <phrase5>-</phrase5>
</level5>
</clause> ...
</Genesis1v1-2v3>

```

**Figure 1.** An extract of the Genesis 1:1-2:3 XML clause cube, which is representative of the hierarchy and structure of the file.

The platform independence of XML documents allows the marked-up text to be transported to other programmes “capable of making sense of the tags embedded within it” (cf. Burnard 2004). For this project, Visual Basic 6 (VB6) was chosen for this role because XML is essentially a hierarchical system that fits the three-dimensional array data structure facilitated by VB6 perfectly. VB6 was chosen above Visual Basic.Net because it is easier to make an executable file for dissemination in the older version. The structural programming also makes it easier to follow the logic of the algorithms. It would, however, be relatively easy to transform the programme(s) into a Visual Basic.Net format

since Visual Studio provides migration facilities. This would enable the use of pre-programmed classes, for example to extend, delete or edit the data in it. In this article, however, these CRUD (create, read, update and delete) functions had to be coded manually, since the size of arrays are static and do not allow automatic insertion and deletion of records (Petroustos & Hough 1999:219).

When converted into VB6 the databank module consists of a three-dimensional data structure. A multidimensional array is very suitable for a limited data set, such as the data in this project, due to its built-in indexing. Multidimensional online analytical products (MOLAP) “typically run faster than other approaches, primarily because it’s possible to index directly into the data cube’s structure to collect subsets of data” (Kay 2004). The VB6 programme discussed in this article and the following chapter may be regarded as a simple MOLAP tool.

The three-dimensional array in VB6 contains the records of the 108 clauses found in Genesis 1:1-2:3. Each clause has five or less phrases. Each phrase has five levels of analysis. One level of analysis is added to record the verse number as primary key for reference and searching purposes (this will leave five unused data fields per clause, which may later be used for additional metadata). An array of 200 x 5 x 6 is used to implement this data structure. Although a size of 108 in the first dimension would be sufficient to hold all 108 clauses in the clause cube, it was enlarged to 200 to allow room for appending more clauses’ analyses, as discussed in section 3. If the array were populated manually with data, the first clause could be coded as shown in Figure 2. The same underlying structure is used in this article to convert the data captured in an XML document into a VB6 array.

```
Option Explicit
Public Clause(1 To 200 1 To 5, 1 To 6) As String
Sub Main()
Clause(1, 1, 1) = "Gen01v01a"
Clause(1, 1, 2) = "bre$it"
Clause(1, 1, 3) = "in the beginning"
Clause(1, 1, 4) = "PP"
Clause(1, 1, 5) = "Adjunct"
Clause(1, 1, 6) = "Time"
Clause(1, 2, 1) = "-"
```

```

Clause(1, 2, 2) = "bara"
Clause(1, 2, 3) = "he created"
Clause(1, 2, 4) = "VP"
Clause(1, 2, 5) = "Main verb"
Clause(1, 2, 6) = "Action"
Clause(1, 3, 1) = "-"
Clause(1, 3, 2) = "elohim"
Clause(1, 3, 3) = "God"
Clause(1, 3, 4) = "NP"
Clause(1, 3, 5) = "Subject"
Clause(1, 3, 6) = "Agent"
Clause(1, 4, 1) = "-"
Clause(1, 4, 2) = "et ha$amayim ve'et ha'arets"
Clause(1, 4, 3) = "the heaven and the earth"
Clause(1, 4, 4) = "NP"
Clause(1, 4, 5) = "Object"
Clause(1, 4, 6) = "Product"
...
End Sub

```

**Figure 2.** VB6 code that could be used to create a three-dimensional array and populate one clause element with several layers of linguistic data.

## SHUTTLING LINGUISTIC DATA BETWEEN VB6 AND XML

One of the advantages of an XML database is the separation of the data and the manipulation thereof. The same data can thus be used for various purposes, and programmes or queries can be created to suit the researcher's individual needs. An XML document in itself is not very accessible for direct human inspection. Although it may be read in a simple word processor such as Notepad, the abundant use of tags poses an obstacle for human conception. One needs other software to process the data in such a repository efficiently, a tool to "bridge the gap between having a collection of structured documents and having a functional digital library" (Kumar et al. 2005:118).<sup>3</sup> The VB6 programme discussed in this article may be regarded as such a bridging tool. Another example is Petersen's (2004) MQL query language that enables complex

<sup>3</sup> <teiPublisher> is an open-source tool that aims to provide a customisable repository facilitating the dissemination of XML marked-up texts (see Kumar et al. 2005).

searches for patterns in annotated linguistic corpora such as the database of the Hebrew Bible developed by the Werkgroep Informatica (WI) at the Free University of Amsterdam.<sup>4</sup> However, according to Bourret (2003) “most native XML databases can only return the data as XML”.

Another benefit of XML is that it provides an independent public standard and cross-platform compatibility (Sasaki 2004:19). Since XML provides a platform-independent organisation of data, conversion is often necessary to make the data accessible for algorithms that implement efficient retrieval and human-friendly interfaces (cf. Ramsay 2004). The conversion of data encoded in XML is often necessary to satisfy very specific needs identified by researchers. For example, if different linguistic layers are annotated in separate, but related, XML databanks, it is necessary to programmatically merge these data sets into Prolog facts in order to associate them in a single database (Witt 2005:68, 71). Conversion into a standardised format enables researchers to compare various annotated layers in order to discover relations that exist between them (cf. Bayerl et al. 2003:165, 169).

If the XML data should be represented in a different, more human-readable, format, it should first be parsed by an application. In this experiment the data should be represented in an interlinear format which is more human-friendly to read. This necessitates the VB6 programme to read the data into an array in order to be printed as a series of interlinear tables on the screen. By removing the XML tags the primary textual data is restored and the layers of analysis become much more comprehensible.

The next sections of the article describe the conversion of linguistic data of Genesis 1:1-2:3 between such an XML data cube and a three-dimensional array structure in Visual Basic 6 in order to facilitate data access and manipulation. The conversion from and to XML format is called round-tripping. Round-tripping is the circular process of storing XML data in a database and recreating the document from the database, a process which often results in a different document (Bourret 2003). In this experiment round-tripping refers to the

---

<sup>4</sup> Also, compare the description of XML-QL as a relational complete query language in Deutsch et al. (1999).

process of converting a Genesis 1:1-2:3 XML document to a three-dimensional array structure in VB6 and saving it again in XML format. If no changes are done while the data reside in the array the second XML document should be an exact copy of the first (*ideal* round tripping – Smiljanić et al. 2002:16). However, the array phase could facilitate updates to be made, which should be reflected in the resulting target XML document after conversion. These CRUD facilities will be discussed towards the end of the article. Together these algorithms constitute a ‘code shuttle’ with added value in terms of basic database functionality.

### **From XML to VB6**

All data in an XML document is text (Bourret 2003). The mark-up itself is also text only: “markup consists of character strings carrying information about other character strings” (Huitfeldt 2004). For a linguistic database this poses, of course, no problem since it also contains text data only. Therefore, in VB6, all the variables of the three-dimensional array used in this study are also of type *string* only. The limitation of arrays that all the elements should be of the same type (string, integer, Boolean, etc.), therefore, poses no problem. To strip the XML code from its tags a lot of string processing is done (cf. Petroustos 1999:784-795).

An efficient way to prepare the Genesis 1:1-2:3 data for ideal round-tripping would be to ensure that empty elements (for example, where a clause has less than five noun phrases) are represented by a dash (-). The loop that reads the clause cube elements into the three-dimensional array can then simply assume that the next line in the XML document will be the next element in the data structure. Not all phrases have syntactic or semantic functions and these missing elements may also be rendered by a dash. This simple implementation is used in this experiment because it also ensures that after ideal round-tripping the XML document is an exact copy of the original document. However, it is possible, in order to reduce file size and to save memory space, to represent null values by simply omitting these elements in the document. The conversion programme will then have to evaluate the content of each line, using a selection structure



(such as an if-statement) in order to ensure the correct placement in the array. This procedure causes another form of, and probably more, overhead. An XML schema can only check the validity of data recorded in the XML file. Since absent elements are valid, another mechanism is needed to ensure correct conversion of such elements from the XML file into the three-dimensional array. On the VB6 side, empty elements could also be represented by zero-length string values in the array variables. To avoid problems during advanced array processing due to the null values the whole array should first be populated with dashes (as symbol of an empty element) which are then partly overwritten when the data is read in from the XML document. This ensures that all empty elements (or yet unused spaces in the array reserved for new clauses to be appended) contain dashes.

Before the data is converted an algorithm is used to count the number of clauses appearing in the XML file, and the result is stored in variables called *countclauses* and *maxArray*. The last-mentioned variable is used to limit processing in the rest of the VB6 programme to real data only (ignoring empty clause elements), and, therefore, its value should be adjusted when clauses are added or deleted during the array phase.

An extract of the code for this part of the programme is shown in Figure 3. It is assumed that all variables have been declared.

```
'Read XML file from disk into array
Public Sub Command1_Click()

'Initialise all array elements with empty element symbols
For iniArr1 = 1 To 200
  For iniArr2 = 1 To 5
    For iniArr3 = 1 To 6
      Clause(iniArr1, iniArr2, iniArr3) = "-"
    Next
  Next
Next

'Count number of clauses in the XML cube:
arrayMax = 0 'Reset total number of clauses in array
```



```
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
```

```
Wend
```

```
MsgBox ("There are ` & countclauses & ` clauses in the XML cube")
```

```
arrayMax = countclauses
```

```
Close #filenum1
```

---

```
'Populate array with data from XML file:
```

```
Open "Gen1_InputV15_RT1.xml" For Input As #filenum1
```

```
Line Input #filenum1, tempLine
```

```
Line Input #filenum1, tempLine
```

```
Line Input #filenum1, tempLine
```

```
For count1 = 1 To arrayMax
```

```
Line Input #filenum1, tempLine
```

```
Line Input #filenum1, tempLine
```

```
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,  
XMLstringLength, tempLine)
```

```
Clause(count1, 1, 1) = Mid(tempLine, XMLstringBeginPos,  
XMLstringLength)
```

```
Line Input #filenum1, tempLine
```

```
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 1, 2) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 2, 2) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 3, 2) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 4, 2) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 5, 2) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
```

```
Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 1, 3) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 2, 3) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 3, 3) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 4, 3) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 5, 3) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 1, 4) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
```

```
XMLstringLength, tempLine)
Clause(count1, 2, 4) = Mid(tempLine, XMLstringBeginPos,
XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
XMLstringLength, tempLine)
Clause(count1, 3, 4) = Mid(tempLine, XMLstringBeginPos,
XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
XMLstringLength, tempLine)
Clause(count1, 4, 4) = Mid(tempLine, XMLstringBeginPos,
XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
XMLstringLength, tempLine)
Clause(count1, 5, 4) = Mid(tempLine, XMLstringBeginPos,
XMLstringLength)

Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
XMLstringLength, tempLine)
Clause(count1, 1, 5) = Mid(tempLine, XMLstringBeginPos,
XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
XMLstringLength, tempLine)
Clause(count1, 2, 5) = Mid(tempLine, XMLstringBeginPos,
XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
XMLstringLength, tempLine)
Clause(count1, 3, 5) = Mid(tempLine, XMLstringBeginPos,
XMLstringLength)
```

```
Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 4, 5) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 5, 5) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 1, 6) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 2, 6) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 3, 6) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
               XMLstringLength, tempLine)
Clause(count1, 4, 6) = Mid(tempLine, XMLstringBeginPos,
                           XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos,
```

```

        XMLstringLength, tempLine)
    Clause(count1, 5, 6) = Mid(tempLine, XMLstringBeginPos,
        XMLstringLength)
    Line Input #filenum1, tempLine
    Line Input #filenum1, tempLine
Next
Close #filenum1
arrayflag = True
MsgBox ("XML cube Gen1_InputV15_RT1.xml converted to
array in RAM")
count1 = 1
Call ShowArray
End Sub

' Function used to strip XML tags before inserting data
into array
Public Sub DecodeXML(XMLstringBeginPos2 As Integer,
    XMLstringEndPos2 As Integer, XMLstringLength2 As
    Integer, templine2 As String)
XMLstringBeginPos2 = InStr(templine2, ">") + 1
XMLstringEndPos2 = InStrRev(templine2, "<")
XMLstringLength2 = XMLstringEndPos2 - XMLstringBeginPos2
End Sub

```

**Figure 3.** VB6 code used to convert linguistic data from XML format into a three-dimensional array.

Although validation of the XML file is usually done by means of a schema, it may again be done during the array state. In a subroutine the tags could first be stripped and selected data tested against a standardised list of valid entries (for example, syntactic and semantic functions). If the data does not conform to these values, an error message should be shown. The user should use this functionality to correct the data before any further processing takes place. In this experiment it is assumed that the validation of phrase types and syntactic and semantic functions has already been done. Only an example of validation code for some syntactic functions is shown in Figure 4.

```

Private Sub cmdCleanData_Click() 'Clean data - syntactic
functions
Dim count11, count12, count13 As Integer
Dim arrsyn1(108, 5, 6) As String

```



```

For count11 = 1 To 108 'Copy array for validation purposes
  For count12 = 1 To 5
    For count13 = 1 To 6
      arrsyn1(count11, count12, count13) = Clause(count11,
count12, count13)
    Next
  Next
Next
For count11 = 1 To 108 'Check syntactic functions
  For count12 = 1 To 5
    For count13 = 5 To 5 'Check only syntactic function
dimension
  'Extend this scheme to include all possible syntactic
functions
  'Here limited to those functions occurring in Gen 1:1-2:3
  If arrsyn1(count11, count12, 5) <> "Main verb" And _
arrsyn1(count11, count12, 5) <> "Copulative verb" And _
arrsyn1(count11, count12, 5) <> "Subject" And _
arrsyn1(count11, count12, 5) <> "Object" And _
arrsyn1(count11, count12, 5) <> "Object clause" And _
arrsyn1(count11, count12, 5) <> "Object cluster" And _
arrsyn1(count11, count12, 5) <> "IndObj" And _
arrsyn1(count11, count12, 5) <> "Complement" And _
arrsyn1(count11, count12, 5) <> "Copula-predicate" And
_
arrsyn1(count11, count12, 5) <> "Adjunct" And _
arrsyn1(count11, count12, 5) <> "Disjunct" And _
arrsyn1(count11, count12, 5) <> "Attribute" And _
arrsyn1(count11, count12, 5) <> "Conj" And _
arrsyn1(count11, count12, 5) <> "-" Then
  'User must clean data if following message is shown:
  MsgBox ("Synf " & arrsyn1(count11, count12, 5) & " in
vs " & _
arrsyn1(count11, 1, 1) & " is invalid")
  End If
Next
Next
Next
End Sub

```

**Figure 4.** Example of VB6 code that could be used to validate syntactic function elements during the array state.

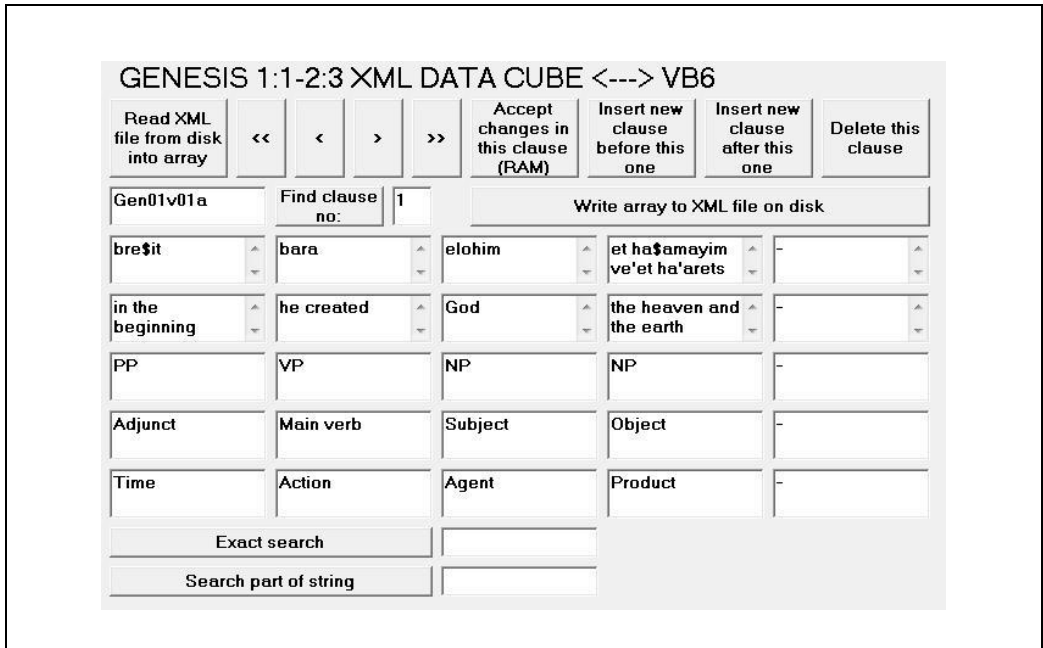
To show the contents of the array, the elements of each clause are displayed in a series of textboxes and labels, simulating an interlinear rendering. The code used is shown in Figure 5.

```
Public Sub ShowArray() 'Procedure used to display current
    clause on interface
    txtFind.Text = count1 'count1 is the array index of each
        clause
    txtC111.Text = Clause(count1, 1, 1)
    txtC112.Text = Clause(count1, 1, 2)
    txtC122.Text = Clause(count1, 2, 2)
    txtC132.Text = Clause(count1, 3, 2)
    txtC142.Text = Clause(count1, 4, 2)
    txtC152.Text = Clause(count1, 5, 2)
    txtC113.Text = Clause(count1, 1, 3)
    txtC123.Text = Clause(count1, 2, 3)
    txtC133.Text = Clause(count1, 3, 3)
    txtC143.Text = Clause(count1, 4, 3)
    txtC153.Text = Clause(count1, 5, 3)
    txtC114.Text = Clause(count1, 1, 4)
    txtC124.Text = Clause(count1, 2, 4)
    txtC134.Text = Clause(count1, 3, 4)
    txtC144.Text = Clause(count1, 4, 4)
    txtC154.Text = Clause(count1, 5, 4)
    txtC115.Text = Clause(count1, 1, 5)
    txtC125.Text = Clause(count1, 2, 5)
    txtC135.Text = Clause(count1, 3, 5)
    txtC145.Text = Clause(count1, 4, 5)
    txtC155.Text = Clause(count1, 5, 5)
    txtC116.Text = Clause(count1, 1, 6)
    txtC126.Text = Clause(count1, 2, 6)
    txtC136.Text = Clause(count1, 3, 6)
    txtC146.Text = Clause(count1, 4, 6)
    txtC156.Text = Clause(count1, 5, 6)
End Sub
```

**Figure 5.** VB6 code used to display one clause's linguistic analysis in a series of textboxes and labels on the interface.

When the user presses the “Read XML file from disk into array” button, the conversion is done and the first clause's data is displayed on the interface (see Figure 6). The array content is displayed, clause by clause, in a series of text

boxes and labels to simulate an interlinear rendering by paging through the records in the array.



**Figure 6.** The end-result after converting data from the XML clause cube into a three-dimensional array in VB6.

The code in Figure 7 is used to scroll through the data. The user may use the next (>) and previous (<) buttons to view the data clause by clause, or they may go directly to the first (<<) or last element (>>). The programme also enables rolling over from the last element to the first and *vice versa*.

```
Private Sub btnFirst_Click(Index As Integer) 'Move to the
    first element in the array
    count1 = 1
    Call ShowArray

End Sub

Private Sub btnPrev_Click(Index As Integer) 'Move to the
    previous element in the array
    If count1 = 1 Then
```

```
    count1 = (arrayMax + 1)
End If
count1 = count1 - 1
Call ShowArray

End Sub

Private Sub btnNext_Click(Index As Integer) 'Move to the
    next element in the array
If count1 = arrayMax Then
    count1 = 0
End If
count1 = count1 + 1
Call ShowArray

End Sub

Private Sub btnLast_Click(Index As Integer) 'Move to the
    last element in the array
count1 = arrayMax
Call ShowArray

End Sub
```

**Figure 7.** VB6 code used to scroll through the clause cube data.

The code in Figure 8 is used to display a required clause, the array index of which is shown in the clause number textbox.

```
Private Sub btnFind_Click() 'Show clause of clause index
    shown in textbox "Find clause no"

count1 = txtFind.Text
If count1 > arrayMax Or count1 < 1 Then
    MsgBox ("Invalid clause no")
Exit Sub
Else
    Call ShowArray
End If

End Sub
```

**Figure 8.** VB6 code used to display a required clause using its array index.

The code in Figure 9 is used to facilitate exact searches; for example, the user may enter “Gen01v07a” and click on the “Exact search” button to move directly to the eighteenth clause. If a parameter is used that appears more than once all clauses containing the parameter are shown one-by-one, paused by a message box. The parameter must match the searched item exactly, but it is not case sensitive.

```
Private Sub btnSearch_Click() 'Exact search

count1 = 1
flagSrch = 0
If txtSearch.Text = "" Then
    Exit Sub
End If

For countSrchl = 1 To arrayMax
    For countSrch2 = 1 To 5
        For countSrch3 = 1 To 6
            If StrComp(Clause(countSrchl, countSrch2, countSrch3),
                txtSearch.Text, 1) = 0 Then
                flagSrch = 1
                count1 = countSrchl
                Call ShowArray
                MsgBox ("Click OK to search next")
            End If
        Next
    Next
Next

If flagSrch = 0 Then
    MsgBox ("Not found")
Else
    MsgBox ("End of data cube reached")
End If

End Sub
```

**Figure 9.** VB6 code used to perform exact searches.

Users who are not acquainted with the sets of phrase types, semantic and syntactic functions used, may need a facility to do “fuzzy” searches. The “Search part of string” button enables one to type any part of a string to be searched within the elements; for example, one may enter “Ben” to find instances of the semantic function of Beneficiary; however, in addition to the required clauses, other clauses containing the Hebrew word *ben* in the phonological rendering will also be shown. The code in Figure 10 is used to do searches on parts of strings in the clause cube.

```
Private Sub cmdSearchPart_Click() 'Fuzzy search
count1 = 1
flagSrch = 0
If txtSearchPart.Text = "" Then
    Exit Sub
End If
For countSrch4 = 1 To arrayMax
    For countSrch5 = 1 To 5
        For countSrch6 = 1 To 6
            pos = InStr(1, Clause(countSrch4, countSrch5,
                countSrch6), txtSearchPart.Text, 1)
            If pos > 0 Then
                flagSrch = 1
                count1 = countSrch4
                Call ShowArray
                MsgBox ("Click OK to search next")
            End If
        Next
    Next
Next
If flagSrch = 0 Then
    MsgBox ("Not found")
Else
    MsgBox ("End of data cube reached")
End If
End Sub
```

**Figure 10.** VB6 code used to perform searches on parts of strings.

After discussing conversion from the array in VB6 back to an XML file in the following paragraph, more CRUD functionalities will be discussed. These procedures should also take place while the cube resides in the computer's RAM (random access memory) during its array phase.

### From VB6 to XML

The conversion of the content of the three-dimensional array in VB6 into the XML clause cube is more or less the reversal of the above process. Assuming that no updates have been done, it is of course not necessary to do validation again, but string processing will again be used to convert the variables to lines of text wrapped in applicable XML tags. The structure of the XML schema must strictly be adhered to in order to create a file that can again be read into VB6 using the same algorithm. In order to keep the original data intact the current date and time may be added to the name of the output file so that a different XML file is created each time when the button "Write array to XML file on disk" is pressed. If one wants to accept and store edited data permanently, the output file should have the same name as the input file. In the empirical experiment of this article, a copy of the XML clause cube is used for this purpose. Figure 11 shows the code that is used to write the linguistic data from the array into the XML clause cube on disk.

```
Public Sub Command3_Click() 'Write array to XML file on
disk

If arrayflag = False Then
    MsgBox ("Array is empty - not saved")
    Exit Sub
End If

filenum2 = FreeFile

'Create unique output file name (optional):
'outputname = "Gen1V15_Output_" & Format(Now,
"yyyymmddhhmmss") & ".xml"

outputname = "Gen1_InputV15_RT1.xml"
```

```

Open outputname For Output As #filenum2
Print #filenum2, "<?xml version='1.0' encoding='UTF-8'
standalone='yes' ?>"
Print #filenum2, "<?xml-stylesheet type='text/css'
href='Gen1XMLdb03c.css'?>"
Print #filenum2, "<Genesis1v1-2v3>"

For count1 = 1 To arrayMax
Print #filenum2, " <clause>"
Print #filenum2, " <clauseno>" & Clause(count1, 1, 1) &
"</clauseno>"
Print #filenum2, " <headers>"
Print #filenum2, " <header>Level</header>"
Print #filenum2, " <header>Phrase1</header>"
Print #filenum2, " <header>Phrase2</header>"
Print #filenum2, " <header>Phrase3</header>"
Print #filenum2, " <header>Phrase4</header>"
Print #filenum2, " <header>Phrase5</header>"
Print #filenum2, " </headers>"
Print #filenum2, " <level1>"
Print #filenum2, " <leveldesc>Phon:</leveldesc>"
Print #filenum2, " <phrase1>" & Clause(count1, 1, 2)
& "</phrase1>"
Print #filenum2, " <phrase2>" & Clause(count1, 2, 2)
& "</phrase2>"
Print #filenum2, " <phrase3>" & Clause(count1, 3, 2)
& "</phrase3>"
Print #filenum2, " <phrase4>" & Clause(count1, 4, 2)
& "</phrase4>"
Print #filenum2, " <phrase5>" & Clause(count1, 5, 2)
& "</phrase5>"
Print #filenum2, " </level1>"
Print #filenum2, " <level2>"
Print #filenum2, "
<leveldesc>Translation:</leveldesc>"
Print #filenum2, " <phrase1>" & Clause(count1, 1, 3)
& "</phrase1>"
Print #filenum2, " <phrase2>" & Clause(count1, 2, 3)
& "</phrase2>"
Print #filenum2, " <phrase3>" & Clause(count1, 3, 3)
& "</phrase3>"
Print #filenum2, " <phrase4>" & Clause(count1, 4, 3)

```



```

    & "</phrase4>"
Print #filenum2, "    <phrase5>" & Clause(count1, 5, 3)
    & "</phrase5>"
Print #filenum2, "    </level2>"
Print #filenum2, "    <level3>"
Print #filenum2, "    <leveldesc>Phrase
type:</leveldesc>"
Print #filenum2, "    <phrase1>" & Clause(count1, 1, 4)
    & "</phrase1>"
Print #filenum2, "    <phrase2>" & Clause(count1, 2, 4)
    & "</phrase2>"
Print #filenum2, "    <phrase3>" & Clause(count1, 3, 4)
    & "</phrase3>"
Print #filenum2, "    <phrase4>" & Clause(count1, 4, 4)
    & "</phrase4>"
Print #filenum2, "    <phrase5>" & Clause(count1, 5, 4)
    & "</phrase5>"
Print #filenum2, "    </level3>"
Print #filenum2, "    <level4>"
Print #filenum2, "    <leveldesc>SynF:</leveldesc>"
Print #filenum2, "    <phrase1>" & Clause(count1, 1, 5)
    & "</phrase1>"
Print #filenum2, "    <phrase2>" & Clause(count1, 2, 5)
    & "</phrase2>"
Print #filenum2, "    <phrase3>" & Clause(count1, 3, 5)
    & "</phrase3>"
Print #filenum2, "    <phrase4>" & Clause(count1, 4, 5)
    & "</phrase4>"
Print #filenum2, "    <phrase5>" & Clause(count1, 5, 5)
    & "</phrase5>"
Print #filenum2, "    </level4>"
Print #filenum2, "    <level5>"
Print #filenum2, "    <leveldesc>SemF:</leveldesc>"
Print #filenum2, "    <phrase1>" & Clause(count1, 1, 6)
    & "</phrase1>"
Print #filenum2, "    <phrase2>" & Clause(count1, 2, 6)
    & "</phrase2>"
Print #filenum2, "    <phrase3>" & Clause(count1, 3, 6)
    & "</phrase3>"
Print #filenum2, "    <phrase4>" & Clause(count1, 4, 6)
    & "</phrase4>"
Print #filenum2, "    <phrase5>" & Clause(count1, 5, 6)
    & "</phrase5>"

```

```
Print #filenum2, "    </level5>"
Print #filenum2, "  </clause>"
Next

Print #filenum2, "</Genesis1v1-2v3>"
Close #filenum2
MsgBox ("Array converted to XML and saved as " &
outputname)

End Sub
```

**Figure 11.** VB6 code used to save clause cube data from the three-dimensional array into permanent XML-formatted storage.

The first version of the XML file for this study was, actually, created in a similar way. The original data was written as code in a module of a VB6 programme that creates and populates a three-dimensional array with the clause cube data. Empty elements were not marked by a dash or other symbol, implying that the array contained null values in those variables. To fill up the array with symbols representing empty values, a for-loop was used, first of all, to populate the whole array with dashes, after which parts of the array were overwritten by those elements that do exist. The array was then converted into an XML file using the same set of code as the lines discussed above.

## **EDITING THE DATA IN THE CLAUSE CUBE**

Reading the data requires a procedure that displays the clause cube data as a set of two-dimensional tables (see above). For read-only purposes, this functionality, combined with the search functions discussed above, should be sufficient. However, it is very likely that some users would need the opportunity to add more clauses to the clause cube, to edit existing data, or even to delete records. Array-like functionalities make these types of operations relatively easy, especially if predefined functions exist which may be called, such as those available in collections or array lists. However, in this experiment, original code was written to facilitate full CRUD since arrays' sizes in VB6 are static and

cannot grow or shrink automatically.<sup>5</sup> This approach may be regarded as an example of “creative programming techniques” that may be used to overcome the limitations of simple arrays (cf. Petroustos & Hough 1999:219).<sup>6</sup>

In order to add clauses to the database, the size of the primary dimension of the cube had to be set large enough to reserve room for a number of extra clauses. This was done by declaring the size of the three-dimensional array as 200 (although there are actually only 108 in the databank, thus reserving space for 92 more clauses).

New records may be inserted either before or after the current clause. When the user identifies the location after which another clause should be inserted (see the “Insert new clause after this one” button on Figure 6), all the clause elements following this location in the computer's memory should be moved one place up to free the current set of variables for a new clause's data to be recorded. If the new clause must be inserted before the current one (see the “Insert new clause before this one” button on Figure 6), the current clause must also be moved one position up in the array.

The code used to create space for a new clause record preceding the one currently displayed on the interface is shown in Figure 12.

```
Private Sub Command4_Click() 'Insert new clause before
    current one into array

    arrayMax = arrayMax + 1

    For countAddRec = (arrayMax - 1) To count1 Step -1
```

<sup>5</sup> “In most programming languages, conventional arrays have a fixed size – they cannot grow or shrink dynamically to conform to an application’s execution-time memory requirements” (Deitel & Deitel 2006:1321). Dynamic arrays in VB6 is not an option since only the last dimension may be changed without losing existing data. Even if the Preserve keyword is used to maintain a multidimensional array's contents when it is resized (using the ReDim statement), only the last dimension may be changed (Petroustos 1999:769).

<sup>6</sup> Grow and shrink functionalities are facilitated by means of array lists in Visual Basic.Net, a fully object-oriented language (Deitel & Deitel 2006:1321; MacDonald 2006:207; Foxall 2006:77). Visual Basic 6 offers the use of collections instead (Crawford 1999:219-224).

```
countAddRec2 = countAddRec + 1
For count4 = 1 To 5
For count5 = 1 To 6
    Clause(countAddRec2, count4, count5) =
        Clause(countAddRec, count4, count5)
Next
Next
Next

For count4 = 1 To 5 'Clear new element in array
    For count5 = 1 To 6
        Clause(count1, count4, count5) = "-"
    Next
Next

Call ShowArray

End Sub
```

**Figure 12.** The VB6 code used to make space for a new clause record to precede the current one.

Figure 13 shows the code used to insert a new, empty clause record set following the current one.

```
Private Sub Command7_Click() 'Insert new clause after
current one in array

arrayMax = arrayMax + 1
count1 = count1 + 1

For countAddRec = (arrayMax - 1) To (count1) Step -1
    countAddRec2 = countAddRec + 1
    For count4 = 1 To 5
        For count5 = 1 To 6
            Clause(countAddRec2, count4, count5) =
Clause(countAddRec, count4, count5)
        Next
    Next
Next
```

```
For count4 = 1 To 5 'Clear new element in array
  For count5 = 1 To 6
    Clause(count1, count4, count5) = "-"
  Next
Next

Call ShowArray

End Sub
```

**Figure 13.** The VB6 code used to make space for a new clause record to follow the current one.

After the existing records have been moved up the array, old, redundant data in the freed space is overwritten with dashes and the new, empty record is shown on the screen. The user may now enter the new clause data here. After the user has typed the new information on the usual interface, he/she may press a button (“Accept changes in this clause (RAM)”) to save the new data into the array in the RAM. If he/she is satisfied that all the information is correct, he/she should press another button (“Write array to XML file on disk”, see Figure 6 above) to save the information to the target XML-file.

Tagging mistakes may be corrected by directly changing the information shown on the display and by saving the updates both to the RAM for immediate use and to the XML file for permanent storage. The same code is used to save new or updated data to the RAM (see Figure 14).

```
Private Sub Command2_Click() 'Accept changes in this
clause (RAM)
  Clause(count1, 1, 1) = txtC111.Text
  Clause(count1, 1, 2) = txtC112.Text
  Clause(count1, 2, 2) = txtC122.Text
  Clause(count1, 3, 2) = txtC132.Text
  Clause(count1, 4, 2) = txtC142.Text
  Clause(count1, 5, 2) = txtC152.Text
  Clause(count1, 1, 3) = txtC113.Text
  Clause(count1, 2, 3) = txtC123.Text
  Clause(count1, 3, 3) = txtC133.Text
  Clause(count1, 4, 3) = txtC143.Text
  Clause(count1, 5, 3) = txtC153.Text
```

```
Clause(count1, 1, 4) = txtC114.Text
Clause(count1, 2, 4) = txtC124.Text
Clause(count1, 3, 4) = txtC134.Text
Clause(count1, 4, 4) = txtC144.Text
Clause(count1, 5, 4) = txtC154.Text
Clause(count1, 1, 5) = txtC115.Text
Clause(count1, 2, 5) = txtC125.Text
Clause(count1, 3, 5) = txtC135.Text
Clause(count1, 4, 5) = txtC145.Text
Clause(count1, 5, 5) = txtC155.Text
Clause(count1, 1, 6) = txtC116.Text
Clause(count1, 2, 6) = txtC126.Text
Clause(count1, 3, 6) = txtC136.Text
Clause(count1, 4, 6) = txtC146.Text
Clause(count1, 5, 6) = txtC156.Text
End Sub
```

**Figure 14.** The VB6 code used to save new or edited clause data to the RAM.

If the user wants to delete a whole clause's data, he/she should be able to press a button to activate a procedure that removes the data of the clause currently shown on the display (see the "Delete this clause" button on Figure 6 above). The related set of clause variables is removed by moving all the following clauses' data one position down in the primary dimension, and by clearing the last element's data that is now duplicated in the second-last position (see Figure 15). These changes should also be saved to the RAM and the target XML file.

```
Private Sub Command5_Click() 'Delete this clause

For countDelRec = count1 To (arrayMax - 1)
    countDelRec2 = countDelRec + 1
    For count2 = 1 To 5
        For count3 = 1 To 6
            Clause(countDelRec, count2, count3) =
                Clause(countDelRec2, count2, count3)
        Next
    Next
Next
```

```
For count2 = 1 To 5 'Clear last element in array
  For count3 = 1 To 6
    Clause(arrayMax, count2, count3) = "-"
  Next
Next

If count1 = arrayMax Then
  count1 = count1 - 1
End If

arrayMax = arrayMax - 1

Call ShowArray

End Sub
```

**Figure 15.** The VB6 code used to delete a clause record.

The procedure used to save all current data in the array from the RAM to the target XML file for permanent storage and recovery has already been discussed in 2.2 (see Figure 11).

Adding these CRUD functions to the programme significantly enhances its functionalities by facilitating basic database procedures to create, delete and maintain data. Even though it could not be called “ideal round-tripping” anymore (since the contents of the source and target XML files differ), in practice, this scenario is preferable for an environment where the database is populated, corrected and expanded. In a situation where end-users should not be able to change the data, these functionalities should, of course, not be offered. The CRUD functionality currently is the only way to extend the database, manually and clause-by-clause, to larger parts of the Hebrew Bible. A more elegant solution would be to import existing data, but this need creates new challenges that fall outside the scope of this article.

## CONCLUSION

After an overview of the essential concepts of building a clause cube, either by using a three-dimensional array in VB6, or a hierarchically structured XML file, a method was proposed to convert the linguistic data between these two formats. “Ideal” round-tripping was implemented by means of string processing to either strip or wrap the primary data in XML tags. This enabled the transformation of the data from permanent storage into a temporary three-dimensional array in the computer's RAM, and *vice versa*. Round-tripping enables one to overcome the limitations of both the array and XML versions. Using only an array does not allow permanent storage, while viewing the data simply by using an XML style sheet does not allow advanced processing. By using both versions as different phases the data is stored elegantly and permanently as an XML file, while some of the functionalities of array processing, like searching and scrolling through the multidimensional clause representations, are facilitated by the consolidated system, which constitutes a code shuttle that carries the data safely between the two storage structures, ensuring that the essential content of the cube stays intact. Various viewing and searching functions have also been discussed.

In addition, some value was added to the mere transportation function of the code shuttle by adding elementary database functionalities. Basic create, update and delete functionalities were added to enable users to populate and edit the clause cube while it is in the array state and to save these updates both to the RAM and on permanent storage in XML format. One may conclude that these technologies are suitable for the efficient storage, transfer and processing of linguistic data. Since all essential database functionalities are now possible, the created software may be regarded as a linguistic information system. By conceptualising and testing the code shuttle, the authors hope to have made a small contribution to the interdisciplinary field of Linguistics and Information Systems.



## BIBLIOGRAPHY

- Bayerl, P S, Goecke, D, Lüngen, H & Witt, A 2003. Methods for the semantic analysis of document markup, in *Proceedings of the ACM-Symposium on Document Engineering (DocEng)*, Grenoble, France, pp. 161-170.
- Bourret, R 2003. *XML and databases*. Available: <http://www.rpbourret.com/xml/XMLAndDatabases.htm> [20 October 2003].
- Burnard, C 2004. A gentle introduction to XML. *Essays in humanities computing*. Available: <http://www.digitalhumanities.org/Essays/> [23 November 2005].
- Crawford, J (ed.) 1999. *Visual Basic 6 complete*. San Francisco, CA: Sybex.
- Deitel, H M & Deitel, P J 2006. *Visual Basic® 2005: how to program*. 3rd edition. Upper Saddle River, NJ: Pearson.
- Deutsch, A, Fernandez, M, Florescu, D, Levy, A & Suciu, D 1999. A query language for XML, *Computer Networks* 31:1155-1169.
- Dipper, S, Götz, M & Stede, M 2005. *Heterogeneity in focus: Creating and using linguistic databases*. Interdisciplinary Studies on Information Structure (ISIS) 2, Working Papers of the SFB 632. Potsdam: Universitätsverlag.
- Foxall, J 2006. *SAMS Teach yourself Visual Basic 2005 in 24 hours, complete starter kit*. Indianapolis, Indiana: SAMS.
- Huitfeldt, C 2004. Scholarly text processing and future mark-up systems. *Essays in humanities computing*. Available: <http://www.digitalhumanities.org/Essays/> [23 November 2005].
- Kay, R 2004. Data cubes, *Computer World* 38/13:32. Available: <http://web25.epnet.com.innopac.up.ac.za> [22 June 2004].
- Kroeze, J H 2008. *Developing an XML-based, exploitable linguistic database of the Hebrew text of Gen. 1:1-2:3*. Unpublished PhD (IT) thesis. University of Pretoria.  
Available: <http://upetd.up.ac.za/thesis/available/etd-07282008-121520/>
- Kumar, A, Schreiber, S, Arneil, S, Holmes, M, Bia, A & Walsh, J 2005. <teiPublisher>: a repository management system for TEI documents, *Literary and Linguistic Computing* 20/1:117-132.
- MacDonald, M 2006. *The book of Visual Basic 2005: .Net insight for classic VB developers*. San Francisco, FL: No Starch.
- Petersen, U 2004. Emdros: a text database engine for analysed or annotated text. Paper read at 20<sup>th</sup> *International Conference on Computational Linguistics*, Geneva, 2004. Available: <http://emdros.org/petersen-emdros-COLING-2004.pdf> [15 October 2004].
- Petroustos, E 1999. The complete Visual Basic 6 language reference, in Crawford, J 1999:744-904.
- Petroustos, E & Hough, K 1999. Object programming with Visual Basic, adapted from *Visual Basic 6 Developer's Handbook*, in Crawford, J 1999:169-224.
- Ramsay, S 2004. Databases, in Schreiber, S, Siemens & Unsworth 2004, Chapter 15.
- Schreiber, S, Siemens, R & Unsworth, J (eds.) 2004. *A companion to digital humanities*. Oxford: Blackwell. Available: <http://www.digitalhumanities.org/companion/> [15 October 2011].

- Sasaki, T 2004. Building an annotated corpus and a lexical database of Modern Hebrew in XML, *Kyoto University Linguistic Research* 23:17-45.
- Smiljanić, M, Blanken, H, Van Keulen, M & Jonker, W 2002. *Distributed XML database systems*. Available: <http://www.purl.org/utwente/38064> [27 July 2005].
- Vakali, A, Catania, B & Maddalena, A 2005. XML data stores: emerging practices, *IEEE Internet Computing* 9/2:62-69.
- Witt, A 2005. Multiple hierarchies: new aspects of an old solution, in Dipper, Götze & Stede 2005:55-85.

J.H. Kroeze  
School of Computing  
University of South Africa  
P.O. Box 392  
UNISA 0003  
kroezjh@unisa.ac.za

T.J.D Bothma  
Department of Information Science  
University of Pretoria  
Private Bag X20  
Hatfield, Pretoria 0002  
E-mail: theo.bothma@up.ac.za

M.C. Matthee  
Department of Informatics  
University of Pretoria  
Private Bag X20  
Hatfield, Pretoria 0002  
E-mail: machdel.matthee@up.ac.za