A programming language for implementing integrations

Background

The beginning of an XML document may look like:

```
<?xml version=”1.0”?>
<Invoice>
  <Header>
    <InvoiceDate>2003-11-24</InvoiceDate>
```

The most natural way of assigning a date to the element `<InvoiceDate>` is probably:

```
Invoice.Header.InvoiceDate.pcdata = ”2012-10-24”;
```

EDI-C is perhaps one of the very few programming languages which address data in an XML document in this way. Components of EDI messages (EDIFACT, X12, TRADACOM), JSON messages and structured ASCII files are addressed using the same method. EDI-C is more than a language to process XML, EDI, JSON and other types of transaction files. The language also contains everything that an integration developer needs: SQL, COM, over 400 practical built-in functions and lots more.

The language is called EDI-C but why the letter C? The intent has not been to invent a new programming language with a homemade syntax but instead to use the perhaps most widespread language syntax style – the one that is used in C, C++, Java and C#.

A basic idea when designing EDI-C has been: avoid inventing something if there are constructs and methods which can be ”borrowed” from well-known programming languages. We have chosen to pick suitable parts from C, C++, Java, C#, Visual Basic and even from Cobol and PL/1. We have tried to keep away from the very verbose in these languages. Our goal have been: “make it as simple as possible”. EDI-C is aimed at those who know C, C++, C#, Java – but also for those who only have come in contact with those languages briefly. Many integration consultants may not have the time to learn yet another programming language – they are probably occupied with other things.

For an ambitious designer of programming languages this may seem to be an act of cowardice. We have not tried to invent new syntax constructs, just to simplify them for those who use the programming language for integration purposes. Our reward is the “ahah!” moment many programmers experience when they come in contact with EDI-C for the first time. Someone has said “The ingenious part of EDI-C is that it is not too ingenious”.

EDI-C is an interpreted language

An EDI-C source program code is compiled and a binary ”executable” file is created. This file is read by the interpreter (the virtual machine) which executes the statements. The method is nearly the same used when executing Java programs.

Basic syntax

EDI-C supports all the usual statements used in C, C++, C# and Java.

Assignment statement

```
a = b + c / d;
```

If statement

```
if (e == ”123”)
  f = g();
else
  ++h;
```

For statement

```
for (i = 1; i < 100; ++i)
  {
  if (j(i) > 100)
    continue;
  k();
  }
```

While statement

```
while (l < m)
  if (n(l + m) == 1)
    break;
```

Do-while statement

```
do
  o += abc ();
while (o < 100);
```

A more powerful switch statement

```
switch p
  {
  case “ABC” :
    q = 2;
```

```
  break;
// Note the expression
case (123 + i) :
  q = 3;
  break;
// Note the function call
case (xyz ()) :
  q = 4;
  break;
default :
  break;
}
```

A simple for-each statement

```
integer iArray [4]= {3, 5, 8, 13};
foreach (integer iValue in iArray)
  {
  println (iValue);
  }
```

The goto statement does not exist in the language - someone (E.W.Dijkstra), is supposed to considered it to be harmful.

The basic data types are integer and string. Automatic type conversion is done for these. It is quite correct to write:

```
integer i = 4711;
string s = ”10000”;
s = s + i; // s now contains ”14711”
i = s + 1; // i now contains 14712
```

String concatenation is done using & as an operator.

```
string s = ”12345”;
s = s & ”ABC”; // s is now ”12345ABC”
```

Functions can be declared in any order. No function prototypes are needed. Below is an example of a function:

```
// A function that adds or subtracts some days
// from a date specified as year and day number
string AddDate (string sDate, integer iDays)
  {
  // Convert date to format: YYYYMMDD
  sDate = dateformat (sDate, ”YYYYMMDD”, ”YYDDD”);

  // Add/subtract some days
  sDate = addate (sDate, iDays);

  // Return the result in the format: YYDDD
  return dateformat (sDate, “YYDDD”, “YYYYMMDD”);
  }
```

Complex data types in EDI-C – a background

One of the basic needs of system integration is to convert data from one format to another. A system can export transactions in a specific format. These transactions probably cannot directly be imported into another system – the receiving system requires its ”own” import format.

This means that a mechanism is needed which transforms or, to use another word, converts transactions to a suitable import format. This has been known for many years in the EDI community. In these cases, the conversion is made in two steps:

My system	Convert transactions of ”my” system into an agreed EDI format (i.e. EDIFACT) and then send the EDI file to the receiving system.
Receiver’s system	Convert the received EDI file to a format that can be processed by the receiving system.

Programs that converts data to and from EDI formats are called syntax-converters or EDI-translators. These programs are often parameter driven and also contain a simpler interpreted programming language. By using program code and special functions you can, for example, convert an incoming date:

```
2003-11-24
```

to another date format

```
20031124
```

which suits my ERP system.

In a syntax-converter, the format of the transactions files are often shown graphically. The structuring information are perhaps stored in a SQL data base or in an XML schema. The data structures are then accessible through function calls in the programming language.

There is however a drawback by using such a method. The programming language and the file structures really do not “speak the same language”. The structures and the program code are in two different environments and the developer has to connect them via function calls.

What is then so special with EDI-C ?

The transaction file structures and their components are an integrated part of the programming language - they are data types defined in the language.

What does this mean ?

Each component in a file structure is directly addressable. Below we show the starting example again:

```
Invoice.Header.InvoiceDate.pcdata = ”2012-10-24”;
```

Or if the element names does not conform to valid C names:

```
Invoice.”Inv-Header”.”Inv.Date”.pcdata = ”2012-10-24”;
```

The addressing notation above means that name checking can be done at compile time . This also means better safety and higher speed at execution time.

Below we show how EDIFACT components are addressed in EDI-C:

```
if (icINVOIC.INVOIC.DTM[i].C507.2005 == “137”)
  invoiceDate = icINVOIC.INVOIC.DTM[i].C507.2380;
```

Note that no function calls are needed when accessing the data elements.

Data types for XML

The data types for XML are `xmldoc`, `xmlelem` and `xmlpcdata`. The two data types `xmldoc` and `xmlelem` can also contain definitions for attributes.

```
xmlpcdata pcdata;

xmlelem OrderHead
  attribute (name=”Buyer”)
  attribute (name=”Seller”)
  {
  pcdata;
  };

xmlelem Article
  attribute (name=”ArticleNumber”)
  {
  pcdata;
  };

xmldoc Order
  {
  OrderHead;
```

```
  Article [1000];
  };
```

XML groups (`choice, all, group, sequence`) are declared as elements with special attributes. The predefined attributes `version, space, lang, encoding` and `standalone` are addressable.

Pcdata and attributes in elements contain data. Normally, this data is of the `string` type and its length could be from 0 to some upper limit. In XSD schemas is it possible to restrict the data which is allowed for specific elements and attributes. An EDI-C variable of the type `xmlfacets` defines these restriction values.

```
xmlfacets facetsArticle type="integer" maxLength="12" minLength="10";

xmlelem Article
  attribute (name="ArticleNumber" facets="facetsArticle")
  {
  pcdata;
  };
```

The developer can access all elements and attributes in a complete XML document.

```
Order.OrderHead.pcdata = ”123456”;
Order.OrderHead.#Buyer = ”ACME INC”;

// As long as there are article data
for (i = 1; xmldata (Order.Article[i]); ++i)
  println (Order.Article[i].pcdata);
```

Adressing XML components can be done by using XML pointers. The statements above can be written:

```
// As long as there are article data
for (xmlptr pArtikel = Order.Artikel[1]; xmldata (pArtikel); ++pArtikel)
  println (pArtikel.pcdata);
```

Reading, parsing and validation of a complete XML document is done by a call to a built-in function:

```
xmlread (Order, ”testfil.xml”);
```

Since the XML structure is described in the program source code and then compiled, the reading, parsing and validation, at execution time, can be done at high speed. This method make it possible to read and parse an XML document at a speed of 10MB per second.

A built-in function is used for writing an XML document.

```
xmlwrite (Order, ”testfil2.xml”);
```

Data types for JSON

The structure of a JSON document is described by using the XML variables in a special way.

Let us assume that we have a JSON document like below:

```
{
  "colorsArray":
    [
      {
      "colorName":"red",
      "price":"123.56"
       },
       {
       "colorName":"green",
       "price":2345
       },
       {
       "colorName":"black",
       "price":3456
       }
     ]
 }
```

This JSON document can be declared using the `xmlpcdata, xmlelem` and `xmldoc` variables like below:

```
xmlpcdata pcdata;

xmldoc JSON
  {
  xmlelem colorsArray
    {
    xmlelem Array type="choicegroup"
      {
      xmlelem Object [99] type="sequencegroup"
        {
        xmlelem colorName
          {
          pcdata;
          };
        xmlelem price
          {
          pcdata;
          };
        };
      };
    };
  };
```

Note in the example above that the JSON pair names are declared as XML element variables and the pair value is declared as pcdata. A JSON array should be declared as an XML CHOICE group and a JSON object as an XML SEQUENCE group.

Data types for EDIFACT

Data types for EDIFACT are: `element, composite, segment, segmentgrp, message` and `interchange`. They are declared like:

```
element 0051 a..2;
element 0052 an..3;
.
.

composite S009
  {
  0065;
  0052;
  .
  .
  };

segment UNH
  {
  0062;
  S009;
  .
  .
  };

message ORDERS
  {
  UNH;
  BGM;
  DTM[35];
  LIN [10000]
    {
    QTY;
    .
    .
    };
  };

interchange icORDERS
  {
  UNB;
  ORDERS [999];
  UNZ;
  };
```

All components can be addressed directly:

```
icORDERS.ORDERS.UNH.S009.0065 = ”ORDERS”;
```

The addressing of components in an EDIFACT tree can also be done by using EDIFACT pointers. See example below:

```
// As long as there are messages in interchange
for (ediptr pMsg = icORDERS.ORDERS; edidata (pMsg); ++pMsg)
  // Do something for each message
```

Reading, parsing and validation of an EDFACT interchange/file is done by calling a built-in function:

```
ediread (icOrders, ”testfil.edi”);
```

writing is done using another built-in function:

```
ediwrite (icORDERS, ”testfil2.edi”);
```

Data types for flat files

By flat files we mean structured ASCII files. In EDI-C we also use this file type to declare files of related types: the American EDI standard X12, TRADACOM, Excel binary files and the SIE format. The data types are: flatfile, flatrecgrp and flatrec.

```
flatrec OrderHead recid (”001”)
  {
  Buyer an..35;
  OrderNumber n..10;
  };
flatrec Article recid (”002”)
  {
  ArticleNumber n..15;
  Description an..50;
  };

flatfile Order variable style (Undefined) recordsep (”\r\n”)
  {
  OrderHead;
  Article [10000];
  };
```

Note the attribute `style` in the `flatfile` declaration. This attribute specifies the type of file and how it should be read and written. An example is: X12. At the moment there are about 10 different file styles. New file styles can easily be added.

The developer can address any field in any record in a flat file – see example below:

```
Order.OrderHead.OrderNumber = 123456;
Order.OrderHead.Buyer = ”ABC AB”;

for (i = 1; flatdata (Order.Article[i]); ++i)
  println (Order.Article[i].ArticleNumber);
```

Addressing components in the flat file tree can also be done by using pointers. The above statements can then be written as:

```
// Loop as long as there are articles
for (flatptr p = Order.Article[1]; flatdata (p); ++p)
```

```
println (p.ArticleNumber);
```

Reading, parsing and validation is done by calling a built-in function:

```
flatread (Order, ”testfile.inh”);
```

Writing a flat file is done with another built-in function:

```
flatwrite (Order, ”testfile2.inh”);
```

Full support for SQL is included

The programming language includes embedded SQL. The interpreter maps the SQL statements to suitable ODBC API functions. This means that all data base systems which have ODBC drivers can be accessed from EDI-C.

```
EXEC SQL SELECT orderno, seller, buyer
  INTO :icORDERS.ORDERS.BGM.1004, :icORDERS.UNB.S002.007,
    :icORDERS.UNB.S002.004
  FROM orderhead
  WHERE orderno = :MyOrderNumber;
if (SQLSTATE != ”00000”)
  // Not found
```

All SQL commands can be used including calls to stored procedures.

```
EXEC SQL CALL StoredProcAbc (:Arg1, :Arg2);
while (SQLSTATE == “00000”)
  EXEC SQL GET StoredProcAbc INTO :RecSetCol1, :RecSetCol2, :RecSetCol3;
```

Accessing COM objects

Microsoft COM object are supported. Using them resembles COM handling in Visual Basic. Type conversions are done automatically. Arguments of the type “User defined” are permitted.

A COM object is declared like:

```
object myObj;
```

and is created like:

```
myObj = createobject (”someCOMmodule.exe”;
```

Al the objects methods and properties are now accessible from the EDI-C program:

```
myObj.myMethod (123, ”ABC”);

myVar1 = myObj.myProp1;

myObj.myProp2 = myVar2;

++myObj.myProp3;

myObj.myProp4 &= ”ABC”;
```

Built-in functions

At the time of writing, there exist over 400 built-in functions. Built-in functions are constantly added, mostly at customers request.

We have tried to make the functions as ”high level” as possible. In the example below, lower case letters in an ASCII file are converted to upper case letters and then all characters are translated to EBCDIC.

```
string s = readfile (”t.txt”)
s = toupper (s);
s = convertmem (s, 1, 2)
writefile (“t.txt”, s);
```

The statements above could have be written like:

```
writefile (“t.txt”, convertmem (toupper (readfile (”t.txt”)), 1, 2));
```

For the complex data types (XML, EDIFACT, structured ASCII files) there are many powerful functions.

There are about 40 different built-in string functions. Among them are: `substr, trim, replace, search, regexp, translate`.

Decimal arithmetic and editing can be done by special functions. Number with up to 25 integers and 10 decimals can be handled.

Among ASCII file functions are: `open, read, readfile, readnl, write, writefile, writeln, close, tell, seek, unread`.

Some file and directory are available like: `chdir, direxist, dirname, fileappend, filecopy, findnext, findnextdir`.

Other programs (.EXE, .DLL) can be called, both synchronously and asynchronously. Other EDI-C programs be called – even single functions in these programs.

Dates and times can be manipulated in various ways. There are also functions for logging, tracing, zip-file handling.

There are about 20 functions for "token" processing.

A short description of how EDI-C is implemented

This section describes briefly the implementation of the EDI-C compiler and interpreter. First some words about the compiler and then some information about the interpreter.

The programming language contains embedded SQL with totally different syntax and operators than used in the C syntax. Partly because of this, we use the recursive descent parsing method for analyzing source programs . This method also gives more freedom at semantic analysis as well as good error handling.

The main part of the compiler is a C++ class using a number of "help-classes" like: lexical analysis, symbol table, statement tree manager, error handling and some more. There are classes for the complex data types XML, EDIFACT and flat files. A special class handles COM objects and their variables. Since the compiler is a class of its own has it been easy to build it into the Development System DS. It is also encapsulated in a COM object for special purposes.

The output from a successful compilation is a byte-code file containing among other things: the symbol table and a statement tree. The statement tree is optimized for fast execution in the interpreter. The compiler output file also contains prototype trees for all complex data types which have been used in the EDI-C source code.

The interpreter is implemented as a C++ class using a number of sub-classes. As for the compiler, it has been easy to encapsulate the interpreter into Inobiz DS but also in Inobiz IS and Inobiz RT. The interpreter exists also as a .NET component and is used in this form as an OEM-product by other EDI and integration systems.

The interpreter is optimized for high speed execution. Direct addressing of variables are implemented as the VARIANT method in COM.

The functions for reading and writing XML documents, EDI files and transaction files are implemented using the DOM method. There are however variants of these functions which use a hybrid of the DOM and SAX methods. This means that very large files, up to 10GB, can be read and written.

Both the compiler and the interpreter are implemented in ”low level” C++ which has resulted in compactness and high execution speed. The size of the compiler is 400 KB and the interpreter size is 1.2 MB. Note that all built-in functions and file parsers (XML, EDIFACT, X12, JSON, TRADACOM, CSV, EXCEL, flat files) and all SQL processing are included.