Blaise to OLE-DB-Relational Data Migration? Yes!

T.R. Jellema. Senior Consultant
NAMES B.V.
Guido Gezellestraat2
2394TT Hazerswoude Rijndijk
The Netherlands
TRJELLEMA@NAMESBV.NL

Yes!!!

This paper has its roots in an innocent question posed during the International Blaise User Conference 2000 to the Blaise Development team just after having been told about the exciting new capabilities of Blaise with regards to the capability to gain access to OLE-DB data sources. The question was deceptively simple: Were we (Blaise Users) going to see the OLE-DB equivalent of the ASCII-Relational export. The question had been anticipated well ahead. The answer in short was 'NO!!!'.

This article deals with the question and as suggested above provides an answer in the affirmative. Generating a generic database structure off a Blaise data model can be done programmatically and is quite straightforward after a spending a reasonable amount of time in getting to know the Blaise API. The utility program that we developed performs two tasks: 1) to generate a database structure, and 2) to use the Blaise API (rather than BOI files) to move the data across from the data model into the OLE-DB database.

Why bother?

Blaise is good, very good, in data capture and editing. Many statistical offices are quite happy to use Blaise and Manipula to process the data u until the point where the data will be analyzed and tabulated using a statistical package. However more and more statistical offices have invested in the development of relational databases, and there is a need for transferring the Blaise data into these relational databases.

There is a perfectly workable solution for this problem. It requires you to first define the database structure that will receive the survey data. Then you will export the data from the Blaise database into ASCII tables that correspond with the structure of the relational database. Then batch uploading utilities such as Data Transfer Services (DTS) from Microsoft can be used to populate the relational database tables. The unfortunate part is that in case of complicated surveys, quite a lot of custom Manipula code needs to be written to generate the required ASCII tables.

The problem with this approach is of course that you will need to invest the time to define the database structure and subsequently you will need to write out the various Manipula scripts to write out the ASCII datafiles. It is time consuming to say the least. Also one might argue why it is necessary to use the ASCII format to generate an intermediate file – why not import directly from the Blaise database and -most of all- why not generate a database structure off the Blaise metadata? Certainly for those of us who have a passing knowledge of relational desktop databases such as Access, and who want a quick solution to our data-migration needs the solution presented above is cumbersome.

With Blaise III the problem had already been solved to some degree. It was (and is) possible to prepare a Blaise to ASCII relational export script using the Manipula setup wizard, and subsequently to use Cameleon scripts to generate setup files for the Oracle client server environment, or the Paradox (DOS version) desktop database. These setups could be used to generate the database tables and upload the ASCII relational data, and provided a workable solution. Unfortunately no Cameleon setups have since then been added to Blaise to support Windows databases. (although this is full well feasible). Therefore this solution was lost except to those few who were capable to develop their own Cameleon setups to do exactly what they wanted to achieve (such as generating a SQL script to define the metadata for a client server database).

The question that we pose in this article is whether the new capabilities on offer with the Blaise Component Pack (BCP 1.0) and Blaise 4.5 can be used to provide a significantly improved data migration tool.

Outline of a solution

What we would like to achieve is to be able to export a Blaise database to a relational database with the minimum amount of effort. This means that we will stick, for the moment, to the database structure implied by the ASCII relational data format.

The first task at hand is to programmatically define the database structure on the target database. Noting that we will present a generic solution for relational databases that are supported by OLE-DB (MS-Access, MS-SQL Server and Oracle), we would like to use the OLE-DB API to perform this task. The actual library used for this purpose is the ADOX library (Microsoft ADO Extension for DDL and security).

On the other side we will need to use the Blaise API to obtain all of the relevant information from the Blaise data model that we want to migrate to the database. We will need to make a correct translation for all of the various data types used in Blaise.

After defining the database structure, the next step that we would like to achieve is the actual migration of the data. Remember, we are interested in a one-stop solution. Here there are a number of choices. It is possible to migrate the data using the bulk import of ASCII relational data prepared by Manipula and the relational database utility for data import. We can also explore the use of BOI files (the Blaise to OLE-DB interface files) to import the data. Finally we can just use the Blaise API straight on to read Blaise data and to write out the relational tables.

In this paper we have chosen to explore the use of the Blaise API as a means not only to generate the database structures, but also as a means to populate the database tables. This is the closest we can get to a one-stop solution.

ASCII Relational Format

The ASCII relational format is a convenient and straightforward way of ‘unpacking’ a Blaise data model in a series of relational data tables. It was introduced in Blaise-III and is a logical development from the way storage was organized in older versions of Blaise. The basic organizing principle of the ASCII relational format is that each block type is stored in a separate file. Because there are potentially very many blocks in Blaise data models, the metadata definition supports the definition of *embedded* block types. These cause block type questions that have been embedded to be exported to the file representing

the enclosing block definition. If all block types are defined as embedded this implies that all the data is exported into a single file (ASCII export).

All questions that are defined as a blocktype are written out as separate records in the corresponding ASCII table. Each record written out corresponds to a particular *instance* of the block type. In each of the *enclosing* blocks the fields that correspond to a blocktype are written out as fields of type integer, and the *instance* reference is the value stored in these fields.

All of the ASCII tables produced by ASCII relational output have a unique identifier. The unique identifier is the primary key of the questionnaire, or a simple serial number if no primary key exists, and what is called an instance number, referring to the instance of the block type. If a data model has four fields that are defined as a block type, then the instance number refers to each of these fields. The instance number applies to all instances in the data model, and does not count local instances. In this way the unique identification of records in all data files produced by ASCII relational output is maintained in two fields.

The real advantage of the ASCII relational format is its closeness to the Blaise datamodel and the simplicity of the primary key data in the resulting tables. It is however a difficult format in which to define foreign keys and relationships between tables. Also it is often not practical to generate a separate table for each individual block and it is recommended to use the *embedded block* syntax to eliminate superfluous tables from the output.

Automated Data Definition and Data Migration.

The Blaise component Pack 1.0 offers programmatic access to Blaise data and metadata. With the component pack you will be able to access this information with any language that supports the Microsoft COM (Common Object Model) specification. This is actually quite a large collection of tools. (VB-Script, Visual Basic, Visual Basic for Applications, Delphi, C++, C# etcetera.). It will turn out that you will benefit from a programming environment that allows you to create your own object hierarchies and that allow you to use recursion. For our own implementation we have used Delphi. There are two reasons for this. Using Delphi we have been able to develop our own BCP aware components that can be used and re-used in any project. Also because of its support for pointer variables, it allowed us to define intermediate data structures that greatly facilitate the handling of the BCP data structures.

Exploring the Blaise Component Pack 1.0

The Blaise Component Pack consists of a number of libraries and controls. The Blaise API is the major part of the BCP. It exposes a large variety of objects that allow you to read almost any property or attribute that exists in metadata as well providing you with read and write access to Blaise databases.

For our purposes we need only tap a small amount of the capabilities of the Blaise API. We will be interested mainly in finding our way in the DatabaseManager object, the Database object, the Field object and the FieldDef object and the associated collections. When we were getting to use these objects we found the use of these objects not immediately intuitive. Therefore we will elaborate a little here.

Fields collections

The database object represents at the same time the datamodel (the Blaise metadata) and the database. In order to have a live database object you will have to use the Opendatabase method from the Database manager object.

From the database object we will need to access the various properties that allow us access to fields. There are in principle three distinct ways to do this

1. The Blaise datamodel has two properties (Fields and DefinedFields). By using either of these properties you can obtain a fields collection. (Basically a list of fields). The difference between the Fields property and the DefinedFields property is that the former contains all field *instances* in the datamodel and can be queried for data entered. The Definedfields property on the other hand yields a list of all defined fields in particular instances of block definitions and array questions and set questions that are *not instanced*. The fields and definedfields properties are indexed properties. This means that you will have to provide some index in order to obtain a Fields collection. For instance to obtain a collection of datafields you will need to provide a parameter blfkField:

   FieldColl := bl4database.Fields[blfkData];

2. The Blaise datamodel has a Field property. By using the Field("Fieldname") property of the Datamodel object you can obtain a reference to the field object of that name. This presupposes that we have a list of field names at our disposal, and we know which is the field that we want to access.

   aField := Bl4Database.Field['Ident.Person.Name']

3. Each Blaise Field has a Fields and DefinedFields property. With these properties you will get a collection of fields that are contained in a block, array or set question. You can use the Dictionaryasfield property of the datamodel to obtain a reference to a Blaise field representing the datamodel, or the topmost level in the fields hierarchy. You can use this field as a starting point for a *recursive* routine that will traverse all the hierarchical levels in the Blaise datamodel, and will expose all instanced and non-instanced fields in the datamodel using the Fields property that exists in each field object.

   aField := Bl4Database.Dictionaryasfield

Fields and Blocks

However we are not just interested in obtaining a simple list of fields, or even a hierarchical structure of fields. First and foremost we need to have a complete list of all the block definitions in the datamodel. Because there is no Blocks property in the Blaise Database object, we will need to find each and every field that is a block definition. You can do this by sing the Blaisefield.datatype property that should have the value represented by the blftBlock constant.

Unfortunately the Fields and DefinedFields properties from the Database object are not really useful fior this exercise. The fields collection obtained from the fields property totally skips such fields, it only contains *instanced* fields. The fields collection obtained from the Definedfields property of the data model appears to be more promising, as all the defined blocks are neatly identified, and all of the fields definitions are clearly available. Unfortunately the Definedfields property does not provide a list of all possible instances of members of array fields. It is just too much work.

Using Recursion to get where we want.

The third option provided by the Blaise-API is an alternative and in our opinion better way in which to access the various block definitions and block instances. This is by using recursion on the elements of the fields property collection of individual fields. For instance let us investigate the case of an array field. The Blaise-API contains a Field object for the entire array question. In order to access the array members all you need to do is to obtain the fields collection of this field object, and deal with the individual fields in the collection. You can verify how this works by using the field selector component in the API that shows the various hierarchical levels.

Now whenever we encounter a block definition, we record it, and add to it each instance of a field with that block type. Each of the references is given an instance number. Subsequently for each of the (non-embedded) block definitions we trace out the fields of that block and any embedded clocks contained in it. For any field that is defined as a non-embedded block the type is changed into an integer field, it will contain the instance reference number of the block field instead, allowing for the relational structure. For each block definition we build an array of instances and each instance of the clock contains an array of the instanced fields contained within. Each instance of a block corresponds to a complete data record in a relational table except for the key information.

In order to be able to easily manipulate the information we are collecting recursively, we are actually generating a data structure that identifies all the block definitions, and for each block definition, (1) a list of all the fields that are defined in the block, and (2) a list of all of the instances of that block. Then for each block instance we also compile a list of references (pointers) to the actual Blaise fields. This is shown in Figure 1 below. When the time comes to process write out the database definition we will only need to iterate the Block Definition list (A), and for each of the block definitions, the block field definition list (B) to define the associated database table. For the actual data migration, each element in the Block Definition list (A) will be associated with a database table, and for each element of the block instance list (C) a record will be written out to the appropriate database table with the corresponding values contained in the block field instances array (D).

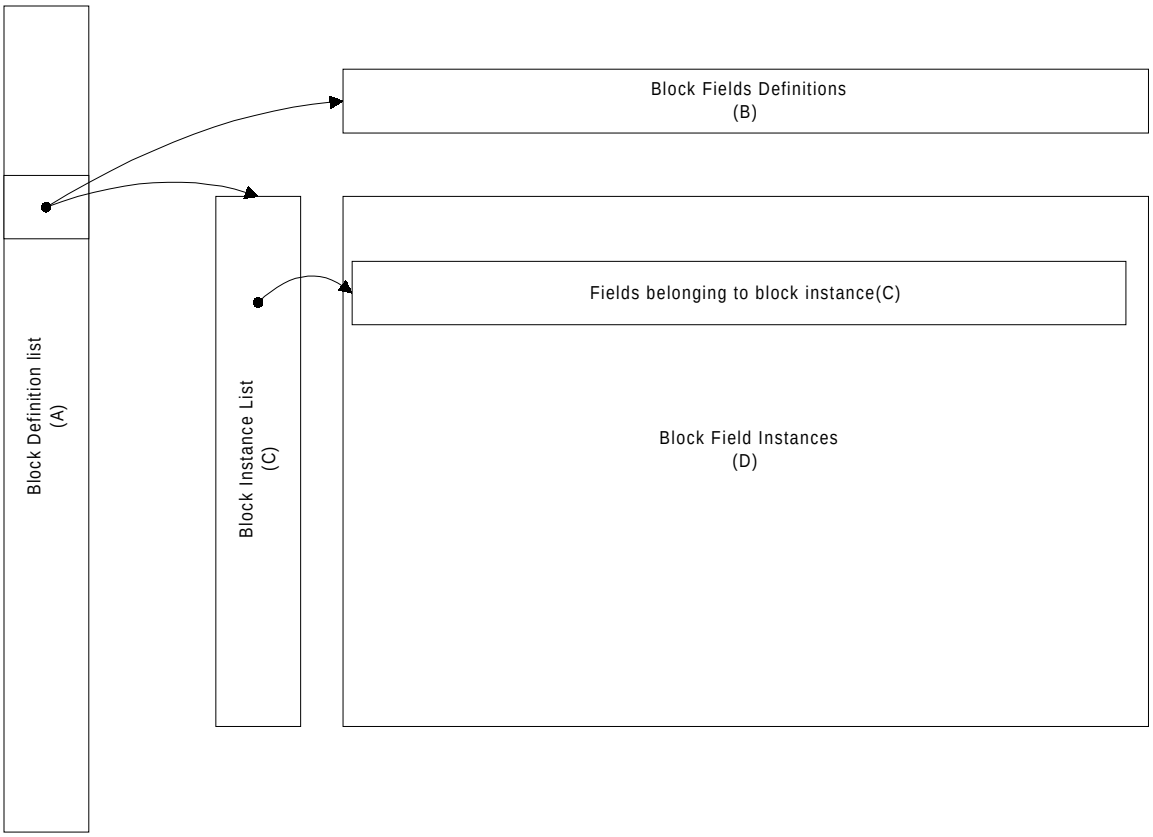

Figure 1 Intermediate Data Structure

The really nice thing about the ASCII relational format is that the key information is stored in two fields only. Each table has the fPrimary field, that is a collation of the primary key fields in the data model, or the internal form number of the questionnaire and each table has an instance number, allowing multiple instances of the same block occurring within the same questionnaire. This is true even for the data model table. When translating the data it is therefore not required to pay much attention to the construction of the primary key of detail tables. We can mimic this behavior in the Blaise API, as the database object provides a property that provides the text representation of the primary key value.

The drawback of the ASCII relational method is the existence of the instance fields in the data model record layout. For on, they will probably break certain database products that have an upper limit in the number of columns to a table. It is easy to define an array of 500 elements in Blaise. If these elements correspond to blocks, this means that the containing block or data model will have 500 instance fields. For a Database Administrator tables that contain large numbers of instance reference fields are quite ugly. It is quite hard to define relational integrity rules in such a structure.

Producing the Metadata.

Once we have obtained a list of all the different block definitions, and allowed for the existence of embedded blocks, it becomes possible to write out table definitions on the basis of the Blaise Metadata. The definition of the database structure in relational databases can be done by preparing a database script that contains the various statements containing the DDL (Data Definition Statements) for the relational database, or by using a special library, the ADOX library. (ADO Extensions for DDL and Security). The ADOX library provides a series of objects (catalog, table(s), field(s), index, key) that allow the definition

of all aspects of an OLE-DB database. Our task is merely applying the methods provided with these objects.

The most difficult part of this is finding the correct translation of Blaise datatypes into OLE-DB data types. Particularly difficult are date and time types. As you may know, the BOI file format does not support translation of Blaise data and time types into the OLE-DB date and time counterparts. Instead it translates dates and time into string information. This is because there are different OLE-DB data types for dates and different OLE-DB data types for time. For instance a DATE field in MS-Access has a OLE-DB data type adDate and a DATE field in Microsoft SQL server has a data type adDBDate.

The makers of Blaise have found the inconsistency in the treatment of date and time fields in OLE-DB so embarrassing that they actually do not support the translation of Blaise DATE and TIME types into anything else than text format.

On the other hand, it is quite cumbersome to first migrate data from Blaise containing dates that are transferred into a string format, and subsequently to have to convert these again in the appropriate types after migration of the data. It is perhaps more convenient to allow the user to choose which OLE DB Date and time type is appropriate. The other Blaise types are translated straightforwardly.

An intriguing option offered by the Blaise API is the inclusion of data in the OLE-DB tables that is actually only available at runtime, such as auxfields and parameters. This can be achieved by using the correct Blaise fieldkind selector set in defining the fields collections obtained from the bock instances. Optionally it is possible to write out such information as well.

Exporting the data.

After having defined the database structure in the OLE-DB database along the principles of the ASCII-relational output, it becomes necessary to actually export the data. Here again we have a choice as to the implementation.

We could use the ASCII file format as an intermediate file format, and use the bulk read options provided with the target database software or we could specify a BOI file for each table, and generate MANIPULA code for each of the tables. Alternatively we could attempt to write the information in the database out to the OLE-DB database directly using the Blaise-API functions. The latter option appears to be more elegant and is the one that we have implemented.

In our migration program we need to define an ADO recordset object for each block type definition that does not involve embedded blocks. These link into the various tables that were defined using ADOX. Because of the potentially large volume of changes that will be made to the data, it is decided to cache the data in memory (using ADO) and to write out the changes for every 100-200 Blaise data records.

Then the program iterates through all of the Blaise database records, and for each of the block types (that are not embedded) it writes a record to the corresponding database table for each instance.

The program: OLE-DB Relational Export.

Interface

The program has a quite simple user interface. It consists of a number of screens that can be accessed through a tabbed notebook. The first two screens are purely informational, these are the structure view and the defined fields view. The Block structures view actually shows the contents of the data structure that is diagrammatically presented in Figure 1. There is also an options page that allows you to set some preferences, such as choosing the correct Datefield data type.

In the structure view the program shows the hierarchical structure of the data model. This tree view shows virtually the same information as the Blaise field selector object included in the BCP 1.0. However because of problems with we experienced with the BlaiseFieldSelector we opted to develop our own component. The tree view shows the structure of an example data model with a large amount of block definitions.

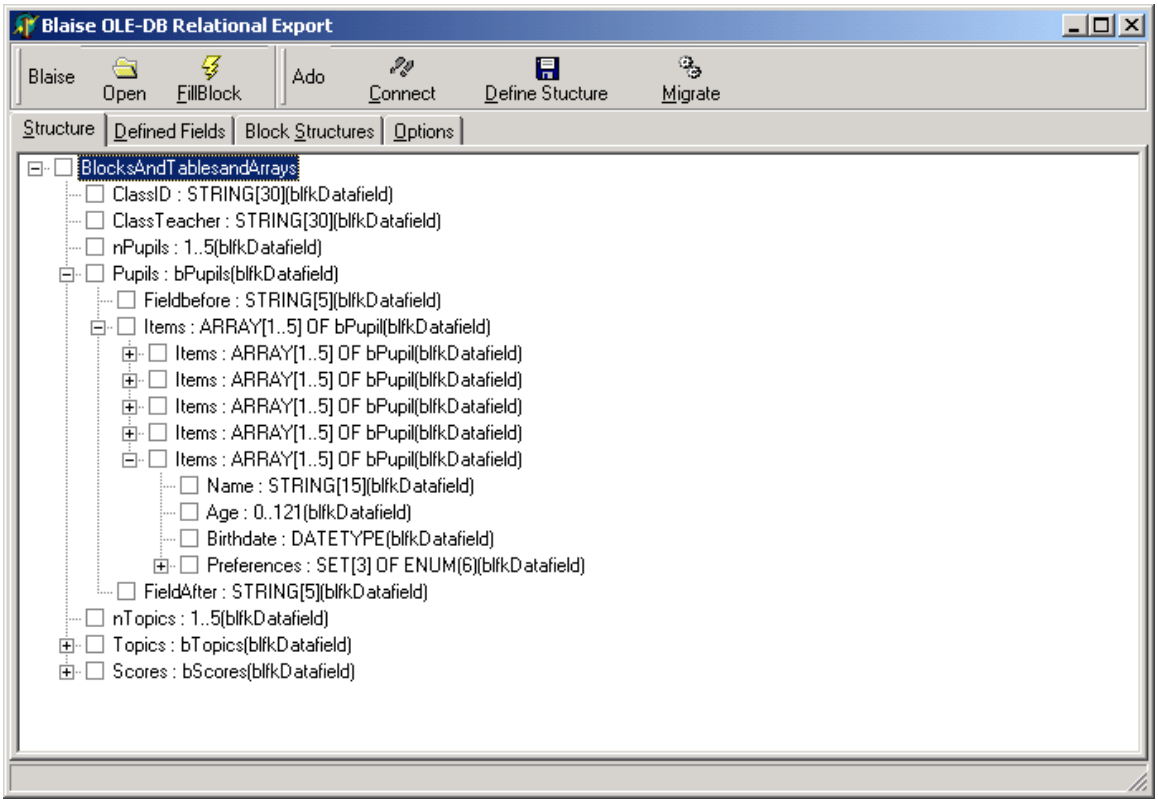

Figure 2 The Data model Structure View

In the Block Structures View you can see a tree representation of the various BLOCKS that have been defined, and the definition of the fields contained in each block, as well as the instance references for each block. For each of the blocks shown in the Block Structures view, a database table will be created. When migrating, for each instance encountered of such a block, a record is created in the database table.

Here you can see that the block bPupil contains information on name, age, birthdate and preferences. You can also see that the pupil block is instanced five times in the array Items.

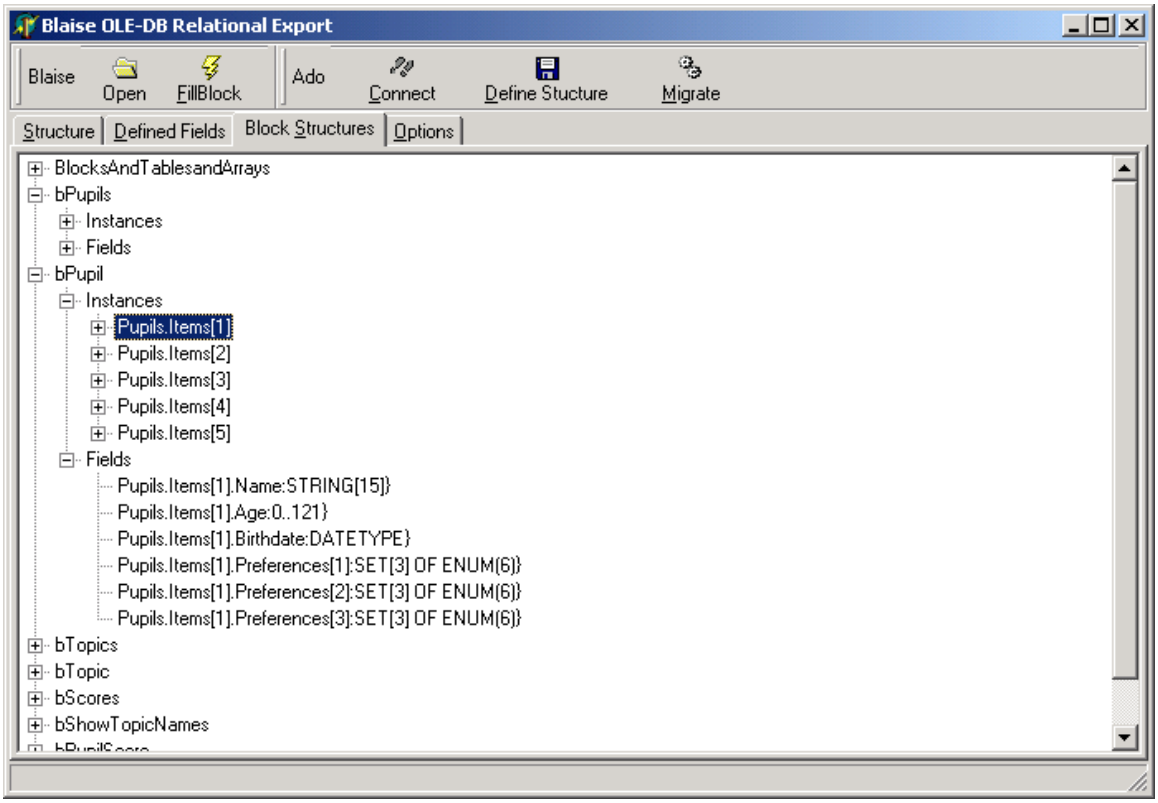

Figure 3 The Datamodel Block Structure View

The block structure view therefore allows you a complete overview of the relational structure that will be created.

To operate the program is quite simple, it is a matter of consecutively defining the following information:

1. Press the 'Open' button to open the Blaise database and data model.
2. Press the 'Fillblock' button to generate the block structures
3. Press the 'Connect' button to open an existing OLE-DB database
4. Press the 'Define structure' button to generate the database structure
5. Press the 'Migrate' button to transfer the data from the Blaise database into the relational database.

Data types and conversions

In the program most Blaise datatypes end up translated as you would expect it:

Table 1 Conversion of Datatypes

BLAISE Datatype	ADO (OLE-DB) datatype
INTEGER	AdInt
REAL	AdDouble
STRING	AdVarWChar
CLASSIFICATION	Advarwchar
ENUMERATION	AdInteger
DATETYPE	AdVarWChar
	AdDate
	AdDBDate
TIMETYPE	AdVarWChar
	AdDate
	AdDBTime
OPEN	Not supported
BLOCK	AdInteger

With regards to the translation of Date and Time fields you have a choice, you can specify this in the options page.

Performance

The program provides an easy and complete way of migrating data from *any* Blaise datamodel to *any* relational database that is supported by the OLE-DB standard. This gives a user a great amount of flexibility. The migration of the data is however prone to take a long time; the combined performance of the Blaise API and the ADO libraries is less than inspiring. It is probably faster to export data to ASCII and subsequently import it into the relational database than to perform the task directly using the API approach. This is due largely to the overhead incurred in adding data record by record into an OLE-DB table. As mentioned earlier, we found it quite necessary to cache large amounts of data in memory rather than adding data, record by record, to the relational tables.

BOI

One of the major new features in the BCP and Blaise 4.5 is the ability to pull in data from OLE-DB datasources into Blaise. Although it is possible, using a Manipula script, to pull data out of a datamodel into a relational table, it does not seem that BOI files are designed for this task. One indication of this is the absence of Blaise to OLE-DB and/or an ASCII to OLE-DB option in the Manipula Wizard. Another indication is that the Blaise OLE-DB Interface Wizard only allows you to define a BOI file based on exiting database tables; but does offer you to generate a Blaise datamodel that conforms to the database table layout. Although it is possible to use a BOI file that has been set up to allow relational data to be brought into Blaise to pull data from Blaise into a relational table this is cumbersome in the context of relational data. You will need to specify a BOI file for each target database table. Subsequently you will need to write a Manipula script that will parcel out all of the required data into the component tables of the relational database. Not a one stop solution!

Note: Model-language block tagging.

In one of the presentations of the previous IBUC Pierzchwala has suggested using a language definition in the data model (language MDL) to indicate which blocks should be included in separate relational tables. In the metadata definitions, each (non-embedded) block that has an MDL description is written out to the table corresponding to the Block description. The method is flexible, because (as does the EMBEDDED keyword) adding such description items will not break the data model and these descriptions may be provided at a later date without requiring migration of data to a new data model.

Pierzchwala does however not indicate in his article how he deals with identifying instances or how the primary and foreign keys are defined on the basis of this information. It is however possible to define database structures on the basis of model language block tagging once rules are defined about the construction of relationships between the various database tables. The advantage of this method can be that block instances of the same type are assigned different model languages, and therefore will be included in different database tables. For instance address information can be collected on the respondent as well as the place of work of the respondent. It may be that such information needs to be separated into two tables. Another case would occur if instances of different blocks would be assigned to the same MDL tag. This would mean that the table structure corresponding to the MDL tag would contain the fields from both blocks.

With a satisfactory solution to the definition of primary keys and foreign keys for the relational tables the model language block tagging technique can be implemented in an equally straightforward way as the ASCII relational format presented in this paper.

References

1. Blaise 4 Windows, Developers Guide
2. Blaise Component Pack 1.00, Beta Version, Helpfile
3. Programming Access 2000, Rick Dobson, Microsoft Press.