Some Techniques In Blaise Data Editing

Rob Groeneveld, Statistics Netherlands

Showing the number of errors in a form in the DEP

When editing a data file, it can be desirable to show the number of errors in a form on the Data Editing screen. As an example, we have this Blaise block:

```
BLOCK Blk_Person
FIELDS
  Name      "What is your name?":       STRING[20]
  Gender    "Are you male or female?": (Male, Female)
  Age       "What is your age?": 0..120
  Children  "How many children do you have?": 0..25
RULES
  Name
  Gender
  Age
  CHECK
  Age < 21 "Do not interview older people"
  IF (Gender = Female) AND (Age > 15) THEN
    Children
  ENDIF
ENDBLOCK { Blk_Person }
```

And a Blaise database:

Name	Gender	Age	Children
Kevin	Male	33	1
Anne	Female	34	
Nick	Male	44	2

This database contains inconsistencies: the males should not have children (a route error) and the ages of all three persons are not below 21 years, so they all have the hard error caused by the Age being over 21. We want to report these errors in the form of error counters on the screen. It is also an option to store these errors in the database for later retrieval. The checks on the errors in the record will be done outside the block, so the rules are checked wholly inside the block. We introduce four fields in the main datamodel:

```
  HardError "Number of hard errors": INTEGER[3]
  SoftError "Number of soft errors": INTEGER[3]
  RouteError "Number of route errors": INTEGER[3]
  NumberOfErrors "Total number of errors": INTEGER[3]
```

We could be tempted to try this in the rules:

```
  HardError := HardError + 1
```

To count the number of hard errors. However, this could result in an increase in the field HardError each time a field in the record is changed because the rules of the datamodel are executed every time.

It is better to use the function GETERRORINFO on the record being edited. This is a function which can be applied to a file identifier in Manipula. For example,

```
  Temp1.GETERRORINFO(1, 'KIND')
```

produces a string denoting the kind of the first error in the current record in the file Temp1. It can be a HARD, SOFT, ROUTE, SUPPRESSED or an IMPUTATION error. In order to use this function we must resort to a Manipula (or Maniplus) alien procedure. This is declared in the fields section in the main datamodel:

```
PROCEDURE NErrors
  PARAMETERS
    EXPORT pHardError
   ALIEN('TestErrors.msu', 'NErrors')
ENDPROCEDURE
```

And is invoked in the rules section like this:

```
NErrors(HardError)
HardError.SHOW
```

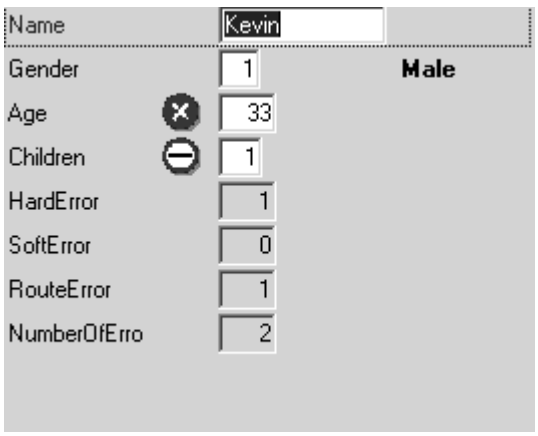
To show the field HardError as calculated in the procedure. We need a definition of the procedure in a Manipula setup which we will call TestErrors.man. This procedure has at its start

```
PROCESS TestErrors
USES
  ErrorModel 'TestErrors'
TEMPORARYFILE
  Temp1: ErrorModel
SETTINGS
  INTERCHANGE = SHARED
```

And defines the procedure NErrors for the parameter HardError as follows:

```
PROCEDURE NErrors
  PARAMETERS
    EXPORT HardError: INTEGER
AUXFIELDS
  ErrorIndex, TotalError: INTEGER
INSTRUCTIONS
  TotalError := Temp1.ERRORCOUNT
  FOR ErrorIndex := 1 TO TotalError DO
    IF Temp1.GETERRORINFO(ErrorIndex, 'KIND') = 'HARD' THEN
      HardError := HardError + 1
    ENDIF
  ENDDO
ENDPROCEDURE
```

The other error counters are calculated in the Manipula procedure in the same way. After implementing all this, the data editing screen for Kevin looks like this:



The numbers of hard errors, soft errors and the total number of errors appear on the screen.

Cross-record data editing in the DEP

Fields in another record can provide additional checks on fields in the record being edited in the DEP. Using a Manipula procedure, editing is possible with links to other records in the same or a different file, either a Blaise file or a BOI file. This increases the range of data editing problems to which the DEP can be applied.

For example, companies were asked about sales and purchases of various goods at various quantities and prices. There answers were not always consistent with one another: when the same transaction was involved, one company sometimes reported a different amount or price than another company. The data have been entered into a Blaise table and it is now up to the data editor to reconcile the various reports when they conflict with one another. The data editor can decide to trust one of the companies involved or use some other means to make the data consistent.

As an example of the power of Manipula procedures, we present here a table with data on buying and selling various drinks. Various people bought and sold drinks to one another and reported on them. However, the price the seller reports on the sale differs from the price the buyer reports on the purchase of the same commodity. The table contains on both prices and presents to the data editor the choice of adjusting the selling price to the purchase price or vice versa, or no to make no adjustment.

This is the table after the data entry phase:

Name	ArticleSold	SellingPrice	Buyer	ArticleBought	PurchasePrice	Seller
Anne	juice	0,90	Nick	milk	0,60	Sandra
Kevin				coffee	0,70	Laura
Laura	coffee	0,55	Kevin	tea	0,65	Nick
Nick	tea	0,80	Laura	juice	0,75	Anne
Sandra	milk	0,85	Anne			

The interpretation: for example, Anne sold juice to Nick at the price of 0,90, while Nick reported on the same sale a price of 0,75. Anne bought milk from Sandra at a price of 0,60, but Sandra reported selling milk to Anne at the price of 0,85. Now it is possible that Anne or Sandra reported the price correctly and the other one made a mistake. The data editor has to base her decision on some criterion, using outside knowledge, for instance, she knows that Anne reports more reliably than Nick, or a paper receipt is available on the sale of milk by Sandra which settles the matter.

The datamodel BuyAndSell.bla contains the definitions of the fields in the table above and a call to a Manipula procedure:

```
PROCEDURE AdjustOtherField
  PARAMETERS
    Dummy1: STRING
  ALIEN('BuyAndSell.msu', 'AdjustField')
ENDPROCEDURE
```

The dummy string parameter is necessary because a procedure must have at least one parameter.

In the RULES section this procedure is called:

```
AdjustOtherField('')
```

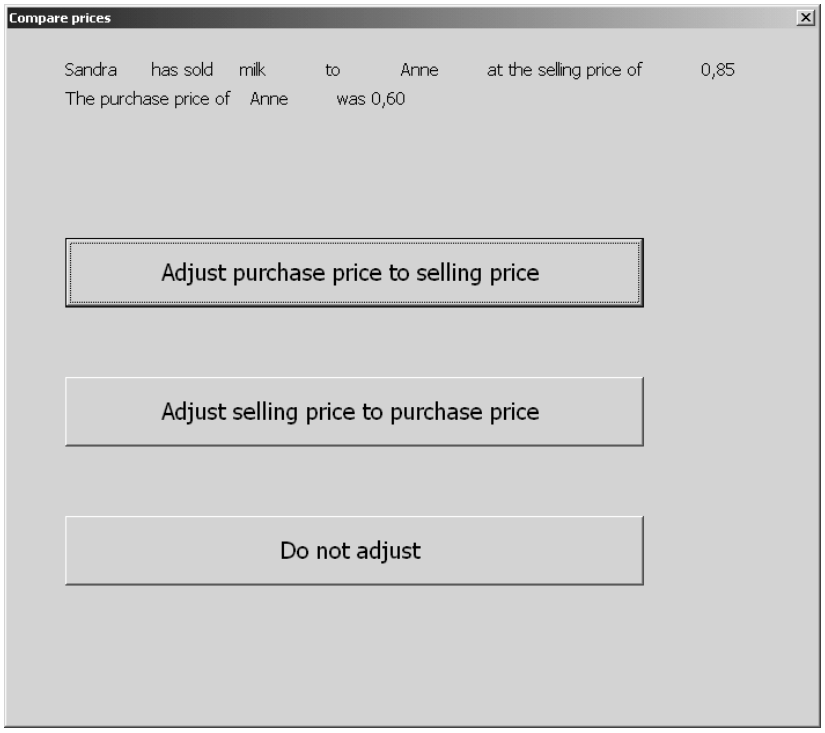
The Manipula setup BuyAndSell.man uses the same datamodel and the setting INTERCHANGE = SHARED:

```
PROCESS BuyAndSell
USES
  BuyAndSellModel 'BuyAndSell'
UPDATEFILE
  Upd1: BuyAndSellModel('BuyAndSell', BLAISE)
```

```
SETTINGS
  INTERCHANGE = SHARED
```

Note the use of the updatefile. The procedure AdjustField is also in this setup. It uses various other procedures and dialogs, among them the dialog Main, which asks the editor if she wants to compare prices or not, and if yes, reports the purchase and selling prices for the commodities a given person has sold or bought and presents three possible actions: to adjust the selling price to the purchase price, to adjust the selling price to the purchase price or not to adjust the prices.

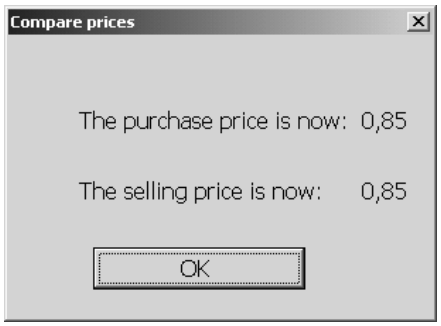
If the data editor wants to compare the prices the dialog Main is shown:



Behind the scenes, this involves a search and read of the record for the seller:

```
IF Upd1.SEARCH(Seller) THEN
        Upd1.READ
```

The various prices are assigned to auxiliary fields and if the first button is pressed in the dialog Main the purchase price is made equal to the selling price and the records written back to the updatefile. The result is reported in another dialog:



The files of these examples are available from the author.

Table-like presentation in the Data Entry Program

A table presentation is a handy tool to group related data in a form. The syntax for a table in a datamodel is the same as for a block, except for the word TABLE at the start of the block (and ENDBLOCK or ENDTABLE at the end of the block). With some effort, the same effect can be obtained by special values in the Mode Library. As an example, we have a datamodel about turnover, profit and loss in companies. Each record contains the data for turnover, profit and loss for one company, over several years in the past:

```
BLOCK Blk_FinancialData
FIELDS
  Year: 1995..2020
  Turnover, Profit, Loss: REAL[6, 2]
RULES
  Year
  Turnover
  Profit
  Loss
ENDBLOCK { Blk_FinancialData }
```

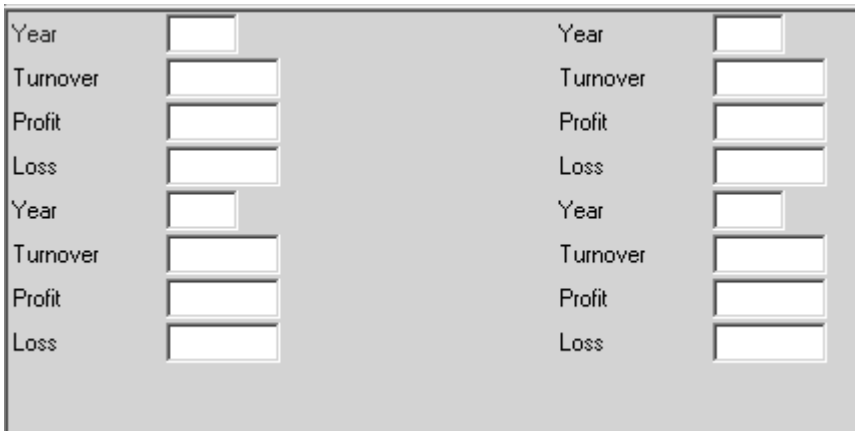
We use this block in another block:

```
BLOCK Blk_FinancialBlock
FIELDS
  SurveyYears: ARRAY[1..5] OF Blk_FinancialData
LOCALS
  I: INTEGER
RULES
  FOR I := 1 TO 5 DO
    SurveyYears[I]
  ENDDO
ENDBLOCK { FinancialBlock }
```

So there will be five sets of data, for five different years. The default grid is applied to the field, which is simply defined as

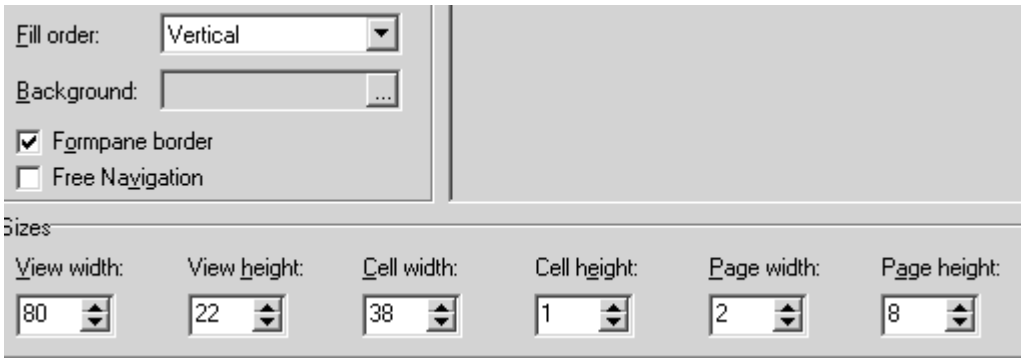
```
FIELDS
  FinancialField: Blk_FinancialBlock
```

With a standard mode library this will be shown as:

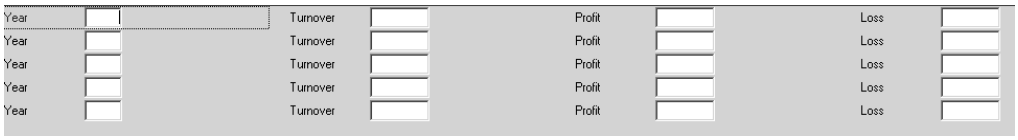


with the last block on another page.

The relevant part of the standard mode library is this:



The fill order in the default grid is vertical, the page width is 2 and the page height is 8. If, however, we change the fill order to horizontal, the page width to 4 and the page height to 5, we get a table-like presentation of the fields:



And if we change the fill order to vertical, the page width to 5 and the page height to 4, we get a kind of grouped presentation in which the years with their corresponding data are shown in vertical groups:

This paper was inspired by questions from the Statistics Netherlands Project “Joules In Motion” put to me by Nicolette de Bruijn and Tom Guldemond.