Performance and Design

Jean-Pierre Kent, Statistics Netherlands

1. Introduction

The Data Entry Program of the Blaise III package has features specifically designed to ensure short response times even for very large and complex data models. This article describes the performance aspects of an interviewing program. It presents the main lines of the mechanisms designed to optimize response times and explains some of the basic rules that a data model designer should follow to allow these mechanisms to work efficiently.

Writing this article has contributed to identifying some problems of Blaise III 1.0 that had not been caught by beta testers or by production users. These problems have been solved in Blaise III 1.1. There is no guarantee that the facts presented in this article work as stated in any release of version 1.0.

2. Response times in interviews

For the Data Entry Program (DEP), response time is the time needed to react to each of the user's input or edit actions. The system may need this time to calculate a new route, impute fields or create new question and answer texts. In this paper I assume that the DEP is being used in interviewing mode (dynamic routing is turned on).

2.1 Data Consistency

The DEP of Blaise is built in such a way as to guarantee the consistency of the values whatever the interviewer does and in whatever order. Whenever values previously entered are modified under dynamic checking, the state of the whole form is recomputed to reflect the new situation.

2.2 Time in the data entry / editing process

The Blaise Data entry program is asynchronous. In simple words, this means that the time of occurrence of some of the actions defined in the RULES is unpredictable. For instance if you write

```
A := 5
B := A
```

then you know that B will always be 5, because B:=A will always be executed after A:=5. If however you write

```
A.ASK
B.ASK
```

there is no way to predict the order in which the values of A and B will be supplied or corrected. Take a look at the following example:

```
Age
IF Age < 15 THEN
   School
ELSE
   Job
ENDIF
```

School and Job could be simple yes/no questions, but they could also be blocks with lots of questions and sub-questions. We can imagine a situation in which the interviewer has to correct the value of Age after filling in the answers of some of the questions about School. This can cause some of the entered values to become irrelevant, and lead the system to backtrack into the other branch of the route.

2.3 What should be performed?

We want to be sure that whenever a value is changed, all the RULES that depend either directly or indirectly on this value are executed. It is, however, not a trivial task to determine the dependency relationships between values and rules. Due to the potential complexity of conditions and loops nested in one another, this information changes during the interview and cannot be computed at preparation time. Unless some sort of run-time mechanism takes care of tracking down what has to be done, the only way to ensure nothing relevant is skipped is to perform all the rules after every modification of the contents of the form!

This, in fact, is what the older versions of Blaise do: every time the cursor leaves a field during a CAPI-mode interview, the whole model is recomputed, and the interviewer cannot start editing another field before this work has been done.

2.4 Data model complexity

When survey designers had to take the limited memory of XT computers into account, they were forced to define small models whose rules could be executed very fast. As computer memories grew, designers tended to build larger and larger models, until they reached the internal limits of Blaise 2. It is in this context that speed became a relevant aspect for Blaise.

One usually assumes that performance is a hardware issue. A program that is slow on today's machines will be fast on tomorrow's, so speed is only a question of technology.

This assumption, however, does not always hold. Execution time depends not only on hardware, but also on the complexity of the task at hand. For a Blaise program, this is the complexity of the rules (in earlier versions: the combined complexity of the route and of the check). This, in turn, is a function of the size of the data model.

2.5 Exponential growth

If the complexity of the rules were a linear function of the size of the model, there would hardly be a problem: as memories become larger, processors also become faster, so performance of Blaise models could be expected to remain stable. The number of consistency rules, however, tends to grow faster than the number of fields.

Consistency rules are usually a combination of two or more fields; so one can theoretically write one consistency rule for every possible combination of two or more fields. Although such an extreme case will never be encountered in real-life data models, it remains true that the number of rules tends to grow exponentially as a function of the number of fields.

3. Lazy checking

One of the prerequisits of the specification of Blaise III being the possibility of using all available memory (up to 16mb), it was clear that something had to be done to prevent an explosion of processing times. The system had to be designed to reduce computing activity to a minimum. It had to be capable of detecting useless work and refusing to execute it. This is what we call lazy checking.

3.1 Level of processing

For many reasons the block was seen as the best unit of administration for lazy checking. By block in this article we mean a block field: this is a field (defined in a FIELDS or AUXFIELDS paragraph) whose type is a block.

In this section we will give an overview of the mechanisms that make lazy checking possible. This information is crucial for understanding how data model design can contribute to the performance of the Data Entry Program.

3.2 A block functions independently

While the rules of a data model are being performed, each instance of a block plays its part independently of other blocks. If you know about object oriented techniques, see a block as an object exchanging messages with other objects. If you don't, just see a data model as a play in which blocks are the actors. The actors talk to each other, ask questions, react to questions by supplying answers or taking actions. Note that the data model itself is a block. So whatever is said here of blocks is also true of the data model.

It is important not to see the rules of a block as a procedure being called by the rules of another block: it is the sole responsibility of the block itself to decide whether its rules should be performed, basing its decision on the information available to it.

3.3 On or off the route?

One of the important concepts for understanding the lazy checking mechanism is that of "being on the route". The conditional rules (rules specified within an IF ... ENDIF construction) allow the definition of multiple potential routes through the questionnaire, one of which will become active at interview time, when values can be computed for the conditions of the IF rules and for the bounds of the FOR rules. Any field whose ASK, SHOW or KEEP method is part of the active route is said to be on the route. This notion is very important in the present context, because a block only talks to those of its sub-blocks that are on the route. The others are systematically ignored.

Although routing is primarily an interviewing concept, with dynamic routing turned on, this mechanism is also active while in data editing mode, when dynamic routing is turned off.

3.4 Detecting changes

Although communication between blocks takes place mainly while the rules are being executed, blocks are also active in another phase, when values are entered or edited.

It is the responsibility of every block to keep track of any change taking place in one or more of its fields. Whenever such a change takes place, this activates a process in which the block marks itself as changed, and communicates this information to its owner. This will cause its owner to mark itself as changed and pass the message further up.

This goes on until the message reaches the data model. During an interview, the data model's owner is the DEP. The DEP will react to changes by asking the data model to perform its rules.

3.5 Selective execution of rules

The rules section is executed as one compound rule, computing the values for conditions and activating the correct branches of conditional checks and routes. This causes consistency rules to be evaluated and error messages to be generated if necessary. It is through the execution of its rules that a block decides which of its sub-blocks are on the route. During this process there is no difference between the ASK, SHOW and KEEP methods; all they do here is to say: "this field is on the route".

A block performing its rules will send every one of its sub-blocks that are on the route a message telling it that its turn has come. For clarity we will first concentrate on the simplest case, considering blocks with no parameters.

The receiving block can react to its owner's message in two different ways: it can either trigger its own rules or ignore the message. The rules will only be triggered if the block is marked as changed.

By ignoring its owner's message, a block not only ensures that its own rules are skipped: it also avoids any form of communication with its own sub-blocks, so that all blocks owned directly or indirectly by it will be left out entirely.

After executing its rules, the main block (the data model) unmarks itself as changed and tells all of its sub-blocks to do the same. This action will be repeated recursively by all the blocks that are marked as changed, allowing every block to ignore its owner's message next time the rule are executed.

A block can be marked as changed if the interviewer or respondent changes any of its field values. But this can also be the effect of an imputation carried out by another block within the same check of the data model. For example the statement

```
Household.HHSize := Household.HHSize + 1
```

will mark the block Household as changed.

3.6 Lazy checking in the example model

To illustrate the lazy checking mechanism, let us have a look at the following data model:

```
DATAMODEL Evaluation
   BLOCK BComment
      FIELDS SayIt: STRING[80]
      RULES SayIt.ASK
    ENDBLOCK

    BLOCK BReadability
      FIELDS Level: 1..10
      RULES Level.ASK
    ENDBLOCK
  FIELDS
    Readability: BReadability
    Comment: BComment
  RULES
    Comment.ASK
    Readability.ASK
  END
```

This trivial model has two blocks and a two-level hierarchy, which is sufficient to illustrate the issues at hand.

When a new record is created, all blocks are initially marked as being off the route. This changes as soon as the rules of the data model are executed. The first rule (Readability.ASK) does the following: it tells Readability that it is on the route, which causes it to mark itself as changed, because its route status has changed; then control is passed to Readability, which triggers its own rules. The effect of Readability's rules is to put the field Level on the route. Now Readability passes control back to Evaluation. The next rule in Evaluation is Comment.ASK. Comment and SayIt will go through the same process as Readability and Level.

What happens if the interviewer enters a value for Readability.Level? The Readability block will enter the new value for Level and mark this field as changed; then it will mark itself as changed and pass this information up to the Evaluation block. Evaluation will mark itself as changed and because this is the main level the DEP will detect this and will ask Evaluation to perform its rules. This time, however no change has taken place within the block Comment, so when its time has come Comment decides to skip its rules. In a similar way the rules of Readable will be skipped when the interviewer enters a value for Comment.SayIt.

3.7 Taking advantage of the hierarchic structure

If a data model has a hierarchic structure, which is usually the case for interviews designed with Blaise, the mechanism described above is very effective in reducing computations. A change will cause work to be done in just one branch of the tree: in the block in which a change has taken place, and in the owners of this block. The number of blocks triggering their rules is thus a function of the depth of the tree and not of the overall size of the data model. If the model is made of independent blocks, this should compensate for the potential combinatoric explosion of the number of checks.

4. Communication through parameters

The mechanism sketched above assumes that block ownership is the only form of dependency between blocks. In most models, however, it is necessary to establish dependencies of another nature.

4.1 Block dependencies

The syntax of Blaise allows the designer to make use of values residing anywhere in the data structure. Whenever this possibility is used in the rules of a given block, this block becomes dependent on all the blocks it draws upon.

If for instance we write the following rule in a block called BChild:

```
Age < Father.Age
```

then all instances of BChild become dependent on Father.

The syntax of Blaise also allows a block to compute a value for a field of another block. For example:

```
Father.Age := Age + Difference
```

Although the syntax allows information interchange between blocks from all parts of the model, the only lines of communication supported by the system are vertical. Now we will explain how information is

passed over ownership lines from the block in which it resides or is computed to the block in which it has to be used or stored.

4.2 The vertical communication channels

We have seen that whenever a block receives control from its owner, it will only trigger its rules if it has detected some change in the values that it manages. If, however, it accesses information from other blocks, this mechanism will not suffice. If any of the values it accesses has changed this can lead to a different result, so the rules have to be recomputed.

In order to be able to detect such a change, a block has to keep track of imported values over time. The fields managing values accessed from other blocks are called import parameters. The designer is free to define import parameters himself, and it is often useful to do so. However it is always possible to let the system generate those parameters internally.

The passing of import parameters plays an important part in inter-block communication. Before passing control to one of its sub-blocks, the owner passes it the values for all its import parameters. The receiving block compares each parameter with its previous value before storing it, and marks itself as changed if one of the values is different. This ensures that any change in one or more of the values accessed by a block will cause the rules of that block to be executed.

4.3 Example of a generated import parameter

Let us return to the model outlined under 3.6 and expand the rules of Bcomment with an unconditional check rule:

```
RULES {BComment}
  Sayit.ASK
  IF Sayit = "Good" THEN
     Readability.Level > 5
  ENDIF
ENDBLOCK
```

Now BComment has become dependent on a value that it does not own. This has consequences both for the preparation of the data model and for inter-block communication at run time. At preparation time, the system will create an import parameter for block BComment. At run time Evaluation will extract the value of Readability.Level and pass it to Comment every time the rule Comment.ASK is executed, causing Comment to mark itself as changed every time the value it receives has changed from the previous run.

The effect of adding that rule, in terms of execution complexity, is that while the rules of Readability are executed only when Readability.Level changes, those of Comment are triggered every time any of the two fields of the model changes.

4.4 Imputing an extraneous field

What happens if a block, instead of accessing an extraneous field, imputes it a value? The block will be expected to return the same value every time, whether it executes its rules or not. In order to be able to impute values without computing them, a block needs fields to keep track of all imputed values. These fields are called export parameters.

4.5 Example of a user-defined export parameter

Now, instead of using the value of Readable.Level in block BComment, let us admit we want to modify the value of an extraneous field passed through a parameter. To do this, we will add a PARAMETERS section in the block, and add a rule performing the imputation:

```
BLOCK BComment
    PARAMETERS
        EXPORT LevelParam: INTEGER
    FIELDS
        Sayit: STRING[80]
    RULES
        Sayit.ASK
        IF Sayit = "Good" THEN
            LevelParam := 8
        ENDIF
ENDBLOCK
```

Now we are left with the task of telling the main block Evaluation that Readability.Level is the destination field for the value computed by Comment for its parameter. So the rule Comment.ASK becomes:

```
Comment.ASK (Readability.Level)
```

Let us now see how inter-block communication takes place in this new situation. Whenever the interviewer enters a value for Comment.Sayit, Comment will be marked as changed and so will Evaluation. This will cause the rules of Evaluation and BComment to be triggered, computing a new value for Comment.LevelParam. This value will be passed back to Evaluate, which will store it in Readability.Level. If this value differs from the value previously stored in that field, Readability will mark itself as changed, which will cause it to perform its rules when its turn comes.

4.6 Timing aspects

Note that the mechanism outlined in the previous paragraph heavily relies on the order specified in the RULES of Evaluate. If we change this order:

```
Readability.ASK
Comment.ASK
```

then a change in Comment will never trigger the rules of Readability: by the time the imputation of Readability.Level takes place, Readability has already been called, at a time at which it was not marked as changed. When Readability is called, it skips its rules; it will not get another chance after the imputation takes place.

This feature is potentially dangerous: a block receives a value and is denied a chance to check it and compute the consequences. However, if you know that the imputed values are always correct, and that neither the routing of the block nor the values it manages depend on them, you can take advantage of this and put blocks on the route in an order that minimalizes checking activity due to imputations.

4.7 Import or export?

Although import and export parameters are both used by the system to keep track of values over time, their function is totally different. Imported values have an influence on the RULES, and a change in imported values contributes to triggering them. Setting and comparing the values of IMPORT PARAMETERS is the first step in the communication taking place between a block and its owner. EXPORT PARAMETERS have no such influence. They are used to enable a block to return the correct values, whether or not it decides to trigger its rules. Returning the values of EXPORT PARAMETERS is the last step in inter-block communication.

It is often necessary to access the value of a field before modifying it. In such a case we need a TRANSIT parameter. TRANSIT PARAMETERS perform both the functions of IMPORT and EXPORT PARAMETERS. This is why a TRANSIT parameter internally needs two fields: one to keep track of the imported value in order to detect changes, and one to keep hold of the exported value. TRANSIT PARAMETERS perform both functions of detecting changes in accessed values and returning imputed values. So they are more costly, both in terms of memory use and of execution time, than IMPORT and EXPORT PARAMETERS.

4.8 Example of a generated TRANSIT parameter

Let us modify again the definition of block BComment:

```
BLOCK BComment
    FIELDS Sayit: STRING [80]
    RULES
        SayIt.ASK
        IF Sayit = 'Good' AND Readability.Level < 5 THEN
            Readability.Level := 5
        ENDIF
ENDBLOCK
```

This will make sure that whenever SayIt has the value 'Good'. Level will be at least 5.

At preparation time the system will add a transit parameter to the data structure of BComment. At run time Evaluation will extract the value of Readability.Level and pass it to Comment, ask for the return value and store it back in Readability.level. Both Comment and Readability will compare the values supplied by their owner and mark themselves as changed if necessary.

4.9 Export parameters are always user-defined

Wherever the designer would define export parameters, the Blaise system will always create transit parameters. This is because it can never be sure that the value of the field is not used before it is computed. So it is always wise to define export parameters yourself: this will spare memory (the block will not need a field to keep track of the value of the parameter before it is called), communication time (the block will not receive the value of the parameter from its owner), and execution time (a change in the value of the parameter will not cause the block to trigger its rules).

4.10 When does a block need a parameter?

In the cases sketched above, Evaluation transfers the value of Readability.Level to Comment through Comment's import or transit parameter. But where does Evaluation get it from? Evaluation can access it directly, because it owns it. The rule is: a block has direct access to its own (aux)fields and those of all the blocks that it owns either directly or indirectly. It can access them both to read their values and to change them. The only thing it cannot do with fields of its sub-blocks is put them on the route. But whenever a block needs to access a field that it does not own, it has to count on its direct owner to pass it as a parameter.

5. The impact of arrays

Although an array can be seen as a list of fields of one given type, in which each element is treated separately, they often behave as a whole, and as such they can have a heavy influence on performance.

5.1 Arrays and routing

The elements of an array can be routed in two different ways: either separately, in direct route rules:

```
Person [1].ASK
Person [2].SHOW
Person [3].KEEP
```

or as a group, within a loop rule:

```
FOR i := 1 TO 3 DO
   Person [i].ASK
ENDDO
```

These two examples use constants, either for the indexes in the direct route, or for the bounds of the loop. This results in elements 1 to 3 of the array being routed unconditionally: whatever happens during the interview, all three elements are always on the route.

In real-world models, however, arrays are usually larger, and a limited number of elements are expected to be on the route for each interview. This can be expressed by using FIELDS or AUXFIELDS for the bounds of the loop:

```
FOR i := FirstPerson TO LastPerson DO
     Person [i].ASK
ENDDO
```

At preparation time the system will define the layout to display all Person fields having indexes ranging from the lowest possible value of FirstPerson to the highest possible value of LastPerson. At run time, only those comprized between the actual values of FirstPerson and LastPerson will be on the route.

5.2 Arrays and parameters

Parameters and arrays interact in two ways to affect performance: first, if a block has parameters and is used for the definition of an array, the parameters will be replicated for all elements of the array; second, accessing a field of a block that is part of an array can cause Blaise to generate too many parameters.

5.2a: Parameters in an array of blocks

The examples in the rest of this article will all be taken from the data model in the appendices. Let us take a look at the question text of the field Name in the block BPerson. It refers to the local variable i, which is part of the table THouseHold. This causes Blaise to generate a parameter for i in the block BPerson. This block, however, has 10 instances: Person[1] to Person[10]. So the value of i will influence the behaviour of all ten blocks of the Person array.

Although the value of i changes all the time, this change is not visible for the individual Person blocks: whenever i has, for instance, value 2, HouseHold will communicate with Person [2]. So i is always 2 for this block. The block, however, does have to receive this value, compare it with the old value, and store it, and this can take time if there are any INPUT and TRANSIT parameters, and if the array is large.

5.2b: accessing a field in an array of blocks

The other point is not trivial: if a block is part of an array, its fields can only be accessed through and indexed reference to the array. If the index is a constant, Blaise knows at preparation time which element to access, and it can generate one single parameter. If, however, a field or an expression is used for the index, neither Blaise, at preparation time, nor the owner of the accessing block, at interview time, can determine which field is relevant: this forces Blaise to generate as any parameter fields as there are elements in the array of blocks.

Take a look at the question texts in the block BEducation of the example in the appendix. They use ^HouseHold.Person[i].Name to display the name of the person being interviewed. When Blaise is preparing this data model, there is not enough information to ensure that the owner of instances of BEducation can make the correct choice. The only solution to this dilemma is to generate parameter entries for all the names in the Person array. So BEducation, as well as BActivity, ends up with 10 Name parameters.

The effects of 5.2a and 5.2b can be compounded, if a block that is part of an array accesses fields from an array of locks. In the example, the series of 10 Name parameters is present 10 times in Education and 10 times in Activity. The owner of these two arrays, Info, does not own those Name fields, so it also has to receive their values from its owner, the main block Survey. This gives us a total of 10 * (10 + 10 + 1) = 210 parameters for passing the Name of the respondent to the question texts.

If you want to see how many parameters are generated in a given data model, you can ask the Structure Viewer to show them. First you need to modify the structure viewer options through the menu item OptionsöStructure Viewer: then go to the Structure Pane and activate the entry for Internal Parameters. This will ensure that the viewer will integrate this type of information in the Structure Pane. The tree structure in Appendix 2 shows the data structure of the example in Appendix 1, including internal parameters.

6. How design affects performance

The preceding sections have presented the lazy checking mechanism, outlining its interaction with the three main features affecting its effectivity: routing, the passing of parameters, and arrays. We will now build upon this information to show what has to be done to reduce computing activity to a minimum.

6.1 Interaction of the three features

The mechanisms of routing and parametrization have been designed in such a way as to ensure optimalization of lazy checking. The interaction of these two features do not lead to problems if you take a few elementary facts into account. We will first concentrate on these simple rules, before examining the impact of arrays.

6.2 Routing and parameters

There are two basic rules: try to minimize block dependencies as much as you can, and build truly hierarchical models.

6.2a: Keeping blocks as independent as possible

Whenever a block can compute a value by itself, it should not count on other blocks to provide it. It is, for instance, not a good idea to store the value of STARTTIME in a field of the datamodel, and access this field from all other blocks. The execution of the function wherever it is needed is more efficient than its value being passed through parameters. Although this value never changes, and will never be the cause of unnecessary execution of the rules, it has to be compared before the system can decide that it has not changed.

It once occurred that a developer submitted a model with very poor performance, asking us what to do about it. We found out that the question text of every single field was defined as "^QText", and RULES sections imputed the value of QText for every field on the route. QText was a local variable of the main block. No wonder this was slow: every block was receiving the question text of the latest routed field, and returning the question text of its own last routed field. The blocks were wasting time exchanging irrelevant information!

Before accessing a value in another block, always ask yourself if it can be computed locally. This should be possible if the value is not dependent on information stored in other blocks.

Block dependencies are often due to a computation being performed at a level that is too low. Let us return to the example given under 4.8: a value for Readability.Level is computed from within the block Comment. This causes the generation of a TRANSIT parameter. This, in fact, is not necessary: the computation can take place in the common owner of the two blocks involved. What we must do is restore the original rules of BComment, and place the computation in the RULES of the main block, which then become:

```
RULES { Evaluation }
  Comment.ASK
  IF Comment.SayIt = Good AND Readability.Level < 5 THEN
    Readability.Level := 5
  ENDIF
  Readability.ASK
END
```

Now take a look at the Survey example in the Appendix. The RULES THouseHold have two checks to verify that no more than one person is declared as Head or Spouse. This is the right place to perform such checks. If we had placed them within the block BPerson, the incrementation of the variables HeadCount and SpouseCount would have caused the creation of 2 * 10 = 20 TRANSIT PARAMETERS to access them.

6.2b: building truly hierarchical models

Always try to find a good balance between the width and the depth of the hierarchical tree. If you have 1000 instances of block, it is not a good idea to declare an array indexed from 1 to 1000 like in:

```
BlockInstance: ARRAY [1..1000] OF BlockDef,
```

because the owner will be communicating with all 1000 instances, which is not efficient. It is better to nest arrays in one another, like so:

```
BLOCK BLevel1
    BLOCK BLevel2
        BLOCK BlockDef
        .
        .
        ENDBLOCK
    FIELDS
        Instance: ARRAY [1..10] OF BlockDef
    ENDBLOCK
FIELDS
    Instance: ARRAY [1..10] of BLevel2
ENDBLOCK

FIELDS
    BlockInstance: ARRAY [1..10] of BLevel1
```

Now each block owns only 10 blocks. In order to reach an instance of BlockDef, communication has to take place at three levels, so in total 30 blocks will receive a message, instead of 1000.

6.3 Routing and arrays

Arrays are usually defined for the storage of information about a variable number of instances. Very few records will need them all. To make sure only relevant instances are on the route, you need to be careful when choosing the type of the fields with which the bounds are defined.

The following example will illustrate the importance of this:

```
FIELDS
    Count "Number of persons to interview": 1..3
    Person: ARRAY [1..10] of BPerson
LOCALS
    i: INTEGER
RULES
    Count
    FOR i := 1 TO Count DO
        Person [i]
    ENDDO
ENDBLOCK
```

This construction will cause instances 1 to 3 to be displayed, because 3 is the maximum possible value of Count. It will also cause instances 1 to 3 to be routed only if their respective indexes are smaller or equal to the run-time value of Count.

But what happens to instances 4 to 10? For reasons that will not be discussed here, Blaise generates an unconditional KEEP rule for every field not explicitely routed. So instances 4 to 10 are always on the route. This was probably not the effect aimed at.

This can be corrected either by defining Count in the range 1..10, or by defining the index range of the array as 1..3.

Blaise wil not force you to harmonize the ranges of the loop bounds with those of the array index: you might want to process different chunks of the same array under different conditions and in different loops. So use of the correct ranges is left to the responsibility of the designer.

6.4 Extracting parameters from arrays

Let us try to minimize the number of parameters generated for HouseHold.Person[i].Name in the question texts of the BActivity and BEducation blocks of the model given in the appendix (see why under 5.2b).

Every instance of BActivity and BEducation will need just one instance of Name, and the designer can specify which. So what we must do is define a parameter in both blocks, and use it in the question texts:

```
BLOCK BEducation
    PARAMETERS
        Name: STRING
.
.
BLOCK BActivity
    PARAMETERS
        Name: STRING
    FIELDS
        Level "What is the highest education followed by ^Name?": EduLevel
```

Now we need to modify the RULES of BInfo, to pass a value for the defined parameters:

```
RULES
   FOR i := 1 TO HouseHold.HHSize do
     Education [i] (HouseHold.Person[i].Name)
     Activity [i] (HouseHold.Person[i].Name)
   ENDDO
```

Note that STRING is not a very satisfactory type definition for the parameter: each instance of the parameter will be 255 characters wide by default. To avoid this, the parameters have to be of the same type as the fields accessed. This means that the type definition has to be supplied at a level accessible both by the block owning the field referred to and by the block owning the parameter. In the example, the common owner of HouseHold.Person [i].Name and of the parameters defined for the names is the main block Survey. So to enable the definition of explicite parameters, you need to define their type in Survey. The definition could take the following form:

```
TYPE NameStr = STRING [22]
```

Now NameStr can be used for the type definition of both the Name field in BPerson and the Name parameter in BEducation and BActivity.

Explicite parametrization of the BInfo block is not necessary: it will have to access all the Name fields anyway in order to supply every instance of its sub-blocks with the value of the corresponding Name field. So we might as well let Blaise do the job.

The resulting number of Name parameters has now fallen from 210 to 30: 10 in BInfo, 1 in every instance of BEducation, and 1 in every instance of BActivity.

7. Appendices

7.1 Appendix 1: the Blaise III example's source

This example was built with pedagogical aims. The THouseHold table is meant as an example of how to design an efficient model. The BInfo block is just the opposite: it gives an example of what not to do. The necessary corrections are supplied in paragraph 6.4.

```
DATAMODEL Survey

   TABLE THouseHold
       BLOCK BPerson
           FIELDS
               Name "What is the name of person ^i?": STRING [22]
               Birth "When was ^Name born?": DATETYPE
               Married "Is ^Name married?": (Yes, No)
               Relationship "What is ^Name's relationship to the head of the
                            household?": (
                                Head   "^Name is head of the household",
                                Spouse "^Name is the head's spouse",
                                Parent "^Name is one of the head's parents",
                                Child  "^Name is one of the head's children")
            RULES
                Name Birth Married Relationship
                IF AGE (Birth) < 16 THEN
                  Married = No "^Name is too young to be married!"
                  Relationship = Child "^Name is too young to be anything but
                                        a child!"
                ENDIF
       ENDBLOCK {BPerson}
       LOCALS
           i: Integer
           HeadCount, SpouseCount: Integer
       FIELDS
           HHSize: 1..10
           Person: ARRAY [1..10] OF BPerson
       RULES
           HHSize
           HeadCount := 0
           SpouseCount := 0
           FOR i := 1 to HHSize DO
             Person [i]
             IF Person [i].Relationship = Head THEN
               HeadCount := HeadCount + 1
               HeadCount = 1 "There can be only one head of the household!"
             ELSEIF Person [i].Relationship = Spouse THEN
               SpouseCount := SpouseCount + 1
               SpouseCount = 1 "The head can have only one spouse!"
             ENDIF
           ENDDO
   ENDTABLE {THouseHold}

   BLOCK BInfo
       BLOCK BEducation
           TYPE
               EduLevel = (None, PrimarySchool, HighSchool, University)
           FIELDS
               Level "What is the highest education followed by ^Household.
                      Person [i].Name?": EduLevel
               Grade "What is the highest grade obtained by ^Household.Person [
                      i].Name?": EduLevel
           RULES
               Level Grade
               Grade <= Level "^Household.Person[i].Name cannot have a
                               ^Grade grade without having followed that
                               level of education!"
       ENDBLOCK {BEducation}
       BLOCK BActivity
```

```
            FIELDS
                Occupation "What is ^Household.Person[i].Name's main
                           occupation?": (School, Work, Unemployed, Retired)
                Income "What is ^Household.Person[i].Name's income?": 0..1000000
            RULES
                Occupation
                IF Occupation = School THEN
                  AGE (Household.Person[i].Birth) <= 18 "^Household . Person [i
                           ].Name is too old to be going to school"
                ELSE
                  Income
        ENDBLOCK {BActivity}

        LOCALS
            i: Integer

        FIELDS
            Education: ARRAY [1..10] OF BEducation
            Activity : ARRAY [1..10] OF BActivity
        RULES
            FOR i := 1 TO HouseHold.HHSize do
              Education [i]
              Activity [i]
            ENDDO
        ENDIF
    ENDBLOCK {BInfo}

 FIELDS
    HouseHold: THouseHold
    Info     : BInfo
 RULES
    Household.ASK
    Info.ASK
 END
```

7.2 Structure of the "Survey" example

```
BSurvey
🖑🕐HouseHold:  THouseHold
¤ 🖑🖘HHSize: 1..10
¤ 🕘🕐Person[1..10]: BPerson
¤  🖑🖘<i>: INTEGER
¤  🖑🖘Name: String[20]
¤  🖑🖘Birth: (Date)
¤  🖑🖘Married: Enum(2)
¤  🕘🖘Relationship: Enum(4)
🕘🖘Binfo: Binfo
  🖑🖘<HHSize>: 1..10
  🖑🖘<Name>[=10]: String[22]
  🖑🖘<Birth>[=10]: (Date)
  🖑🖘BEducation[1..10]: BEducation
  ¤ 🖑🖘<Name>[=10]: String[22]
  ¤ 🖑🖘<i>: INTEGER
  ¤ 🖑🖘Level: EduLevel
```

⌑ 🕓↳Grade: Edulevel

🕓↳BActivity[1..10]: BActivity

  🖇↳<Name>[=10]: String[22]

  🖇↳<Birth>[=10]: (Date)

  🖇↳<i>: INTEGER

  🖇↳Occupation: Enum(4)

  🕓↳Income: 0..1000000