Helping non-Blaise Programmers to Specify a Blaise Instrument

Mark Pierzchala, Westat, US
Graham Farrant, National Centre for Social Research, London, UK

I. Overview

Some computer-literate researchers now specify their questionnaires directly in the Blaise language, while others who rely on programmers to create Blaise instruments have become familiar with many of the conventions and special features of the system. However, there are many researchers, survey sponsors, and subject matter experts who know very little of how Blaise works, but who still need to specify their questionnaire for a Blaise programmer.

There is the potential for a great deal of misunderstanding and wasted effort if specifications of questionnaire requirements given to a programmer do not reflect the way that Blaise works. Conversely, there can be great gain if specification writers are aware of some of the more powerful features and development methodologies available for Computer-assisted interviewing (CAI) questionnaires in Blaise. These aspects are covered in Section II, *What the Interviewer Sees*, and Section III, *Source Code*. This paper then presents two complementary kinds of specification. **Formal specification** is concerned with codifying the research in terms of concept, question definition, routing, checks, computations, valid values and so on. Formal specification is treated at length in Section IV, *Drafting the Blaise Questionnaire*. **Interface specification** focuses on presentation and usability and aims to help the interviewer to understand the questionnaire. In particular, it tries to help to understand how to handle unanticipated events such as ad-hoc navigation. This is treated in Section V, *Specifying the Blaise Interface*. We consider testing to be an integral part of specification, thus a brief treatment is given in Section VI, *Testing the Questionnaire Program*. Section VII, *Special Topics*, gives an overview of some special topics. Section VIII, *Alternative Approaches*, notes other ways to go about producing questionnaires. Selected references appear in Section IX. Appendix A provides detailed information on navigation and ways to make it more powerful. Appendix B shows a detailed sample specification. Appendix C covers some important details about block specification. Appendix D gives information about data readout possibilities.

This paper constitutes a simple guide to Blaise conventions for non-programmers, to enable them to specify a questionnaire that is rooted in Blaise efficiently, and which a programmer can turn into source code quickly. The Guide is based on documents in use at the National Centre for Social Research in the UK and at Westat in the US. It aims to convey the key conventions and features of Blaise rapidly, without requiring users to specify precise syntax.

1. The Intended Audience for this Guide

This Guide is intended for readers who possess little or no familiarity with Blaise or computer programming. Those who are familiar with Blaise may still find it useful. The reader is likely to be a researcher or academic who needs to draft a Blaise questionnaire, or a survey sponsor who needs to understand and comment on a draft. It may also help survey users to interpret Blaise source code for an existing questionnaire.

The goal of this Guide is not to turn the reader into a Blaise programmer. Rather, the aim is :

- To present some of what is possible with Blaise,
- To introduce simple Blaise conventions, and
- To explain how to achieve *transparency*, so that the specification is clear to everyone involved - researchers, programmers, and sponsors.

This last aim is important. Blaise is a programming language, and a great deal of programming time and effort can be wasted in trying to translate a poorly specified draft questionnaire.

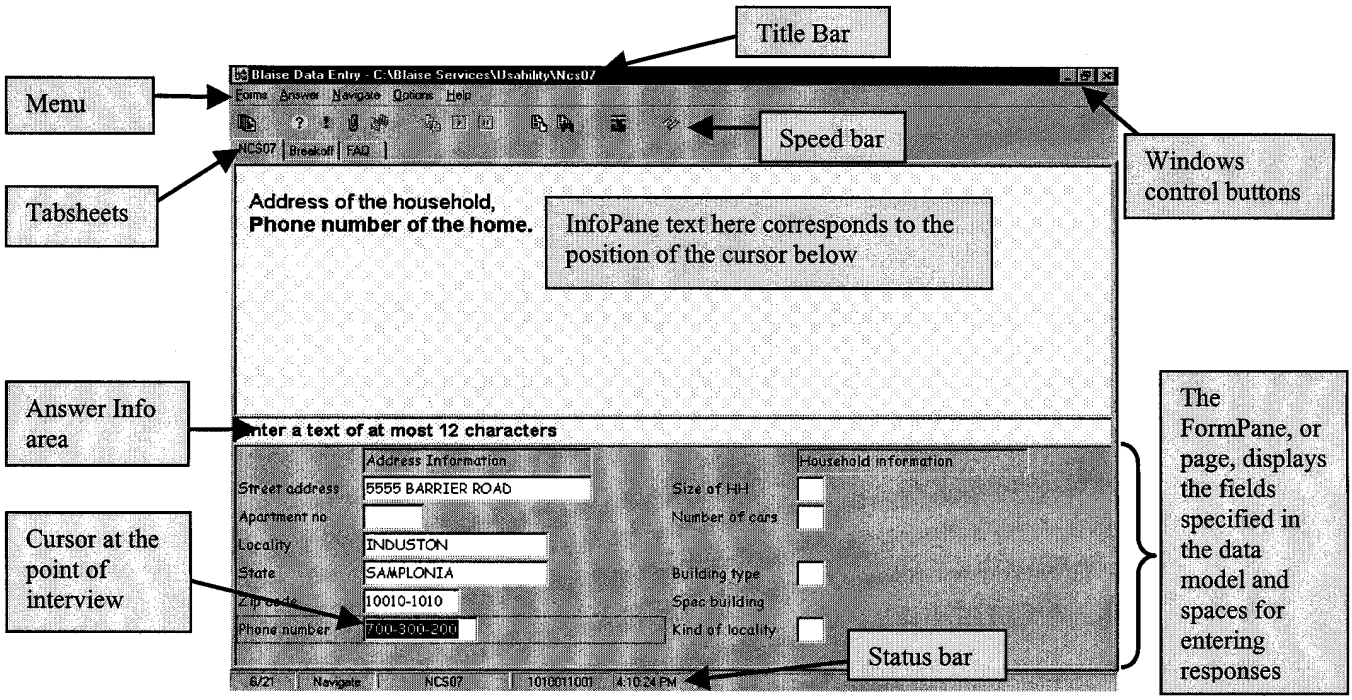
II. What the Interviewer Sees

Before specifying in Blaise, it is helpful to know how the interviewing interface and other aspects of Blaise work, and how they differ from paper or other systems.

1. Split-Screen Interviewing Paradigm and Enhancements

The interviewer collects data with the Data Entry Program (DEP). The Blaise **Data Entry Program** features a distinctive split-screen display shown in Figure 1. The **screen** in Blaise refers to the entire area of the Blaise window, from the title bar on the top and extending to the status bar on the bottom. Thus the concept of a screen in Blaise, a many-question presentation, is much broader than in some other systems. There are also several analogies between a Blaise screen and a page in a paper questionnaire. This aspect is most powerful if explicitly recognized and specified.

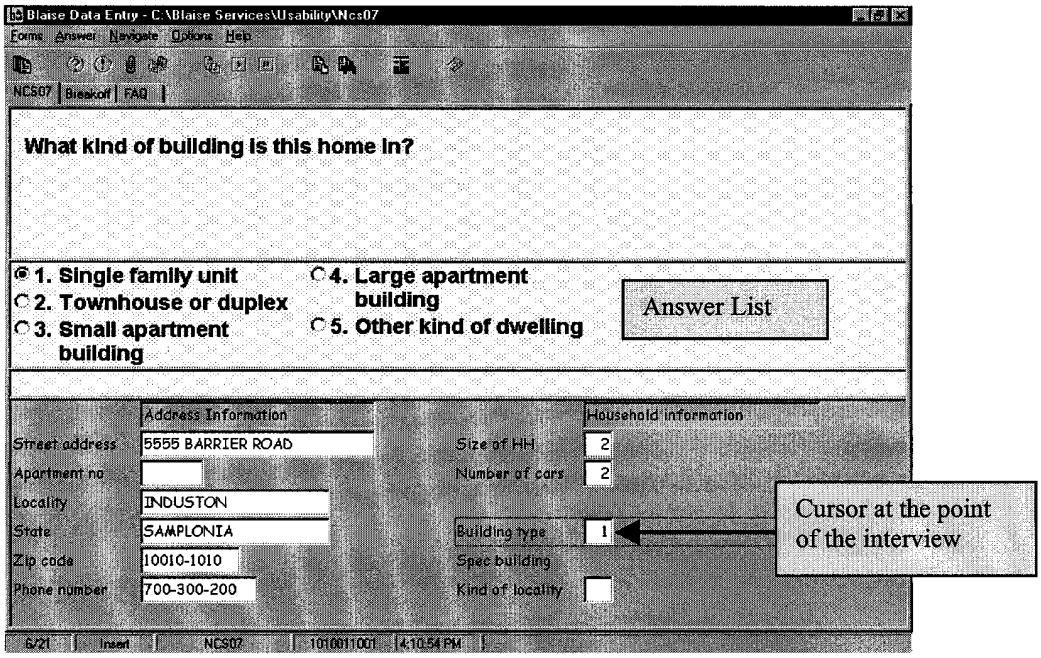
Figure 1: Blaise Screen



The upper part of the screen is called the **InfoPane**. It contains question text and other information meant for the interviewer. The lower part of the screen is the **page** or **FormPane**. It contains data entry cells and the cursor moves from one data entry cell to another. Question text displayed in the InfoPane corresponds to the position of the cursor in the page. The term **page** is used in this paper for the bottom part of the screen because a page on the Blaise screen corresponds to a page in a paper questionnaire. The term is intuitive to the interviewer, and the Page Up and Page Down keys move backwards and forwards, one Blaise page at a time.

Figure 1 shows the Blaise screen with the point of the interview at the *Phone number* field. In Figure 2 the interview has proceeded to the *Building type* field and the question text has changed accordingly. Note that Building type is an **enumerated field**. Figure 2 below displays the answer list. The answer list is associated only with **enumerated** (pre-code) and set (code-all-that-apply) questions. The answer list displays radio buttons for enumerated questions and check boxes for set questions.

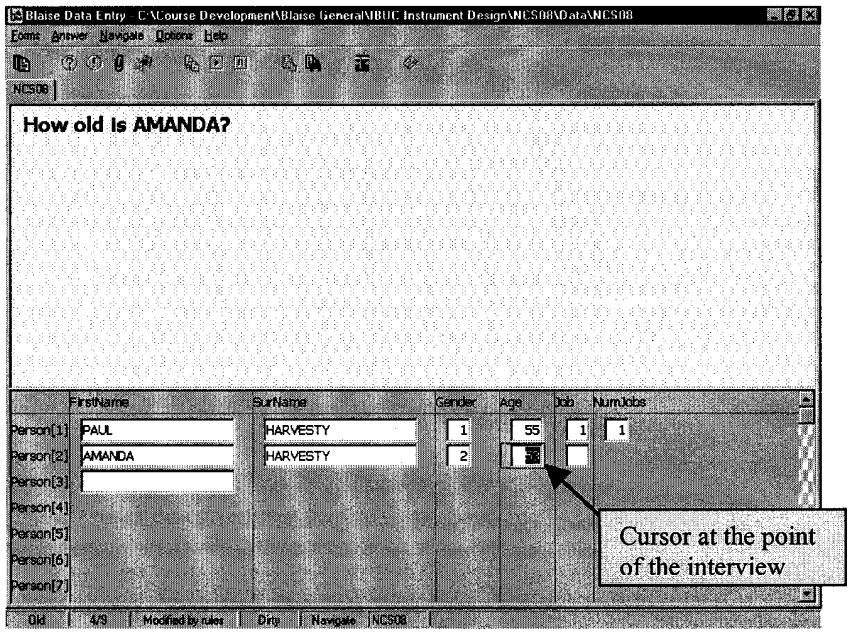
Figure 2: Showing the Answer List Area for an Enumerated (Pre-Code) Question.



The interview proceeds down the first column then the second column, taking into account skips programmed in the rules.

You can use data entry tables in the Data Entry Program as illustrated in figure 3. The table in Blaise can be very flexible in its operation. For example, movement can be strictly controlled (the cursor forced to the right to complete a row), or movement within a table can be free using arrow keys or pointing device. A table can be bigger than the physical screen, horizontally and/or vertically. In the example below, if enough rows are filled in, the table will scroll down.

Figure 3. A Table in Blaise



2. Generated Screens

Blaise screens are generated during preparation (compilation) of the instrument, based on program instructions in the application source code and on general presentation information stored in configuration files. The Blaise split-screen presentation has been accurately described as a many-question-at-a-time system, a page-based system, and as a cursor-based system. The split screen uses limited space efficiently and displays values of several or many fields at one time. This provides interviewers with an overview and allows them to learn the routing based on respondents' answers. They quickly become familiar with the instrument and how questions relate to one another. This makes it easy to navigate and to correct answers.

3. How the Interviewer Works

When asking questions, the interviewer reads the question from the InfoPane, enters the answer in the page, and presses the Enter key. The cursor moves to the next data entry cell in the page and the question text in the InfoPane changes accordingly. Since the values of many questions are presented in the page at one time, the interviewer can verify that the correct response has been typed. The normal forward movement of the interview, including skips, is controlled by the formal *flow* or *routing* specification programmed in the instrument (see Section III for a more detailed discussion of this topic).

Navigation

An often forgotten aspect of specification is navigation. This is important because interviews do not always proceed in the way that we plan. A respondent may wish to change a previous answer and the interviewer will need to back up to make the change. In Blaise, it is very easy to navigate in an ad-hoc manner and to correct answers, even when it is necessary to navigate over many pages. Thirteen (or more) methods of navigation are covered in Appendix A. It is useful to think of navigation in terms of short-range, mid-range, instrument-wide movement, and non-linear motions. There are many ways to enhance the understandability and usability of the Blaise instrument with these kinds of movements in mind. These enhancements are described in detail in appendix A. The basic point to remember is that the basic Blaise paradigm solves many problems of overview and navigation, and that you have options that can enhance the paradigm.

If the result of moving backwards and changing a response is to change the flow of the questionnaire, the new route is immediately implemented. By pressing the End key, the interviewer will be taken to the new route if necessary to fill in the missing details.

III. Source Code

The previous section depicted the Blaise questionnaire as it appears to the interviewer. Underlying every Blaise questionnaire is a set of instructions called **source code**, written in a programming language. The source code for a survey is, in effect, the new Computer Assisted Interviewing (CAI) equivalent of the old paper questionnaire. Source code is **compiled (prepared***)* by the Blaise system into a **data model**. The data model (or **instrument**) is used by the Blaise **Data Entry Program** to run the questionnaire on the interviewers' computers.

The key document in creating a Blaise questionnaire is the **source code**. When we are drafting Blaise questionnaires, we are really drafting source code using the Blaise language. This can be done in any word processor or text editor. A lot of source code in Blaise is not difficult to write or to understand. The questions and answers are described in the source code in much the same way that they will appear on the screen. But source code has other special features and conventions that make writing it quite different from writing a paper questionnaire. This section is merely an introduction to what source code looks like so that you better understand how your specifications relate to the eventual source code. See Section IV for detailed guidelines.

1. Fields and Rules

Blaise **fields** define the instrument's data definition, while **rules** determine the flow of the questionnaire, edit checks for a field or between fields, and computations (math or text manipulations). The following example shows a simple Blaise data model.

```
DATAMODEL FieldsAndRules "Fields and Rules"

FIELDS
  Name "What is your name?" : STRING[20]
  Age "How old are you?" : 0..120, RF, DK
  Job "Are you employed?" : (YES, NO)

RULES
  Name
  Age
  IF Age >= 16 THEN
    Job
    SIGNAL
    IF Job = Yes THEN
      Age <= 85
      "It is unusual for people to work above the age of 85,
        have I mistyped something?"
    ENDIF
  ENDIF
ENDMODEL
```

When the programmer prepares the source code above, the electronic instrument is created. (To prepare a data model, simply press the F9 key in the Blaise Control Centre.) For this simple example, one screen is generated on which all three questions appear. The appearance of the Blaise page and question text is determined by configuration settings that have already been determined, either by default, or by defining your own appearance beforehand. The Blaise database is also generated. This data model contains places for three data items. The Rules section illustrates conditional routing (*Job* is asked only if *Age* is greater than or equal to 16). A soft check is invoked if the person has a job and is over 85 years old.

2. BLOCK and Structure Point of View

Blocks hold related groups of fields and their rules. A block can represent a section or sub-section of a questionnaire. Following is a simple illustration.

```
DATAMODEL BlockDemo

LOCALS
  I : INTEGER

FIELDS
  Job "Are you employed?" : (YES, NO)
  NumJobs "How many jobs do you have?" : 1..10

BLOCK BJobDetail
  FIELDS
    Employer "What is the name of the employer?" : STRING[30]
    KindJob "What kind of job do you have?"
    : (worker, supervisor, manager, director, other)
    JobOS "Please specify the kind of job you have." : OPEN
  RULES
    Employer
    KindJob
    IF KindJob = Other THEN
      JobOS
    ENDIF
ENDBLOCK

FIELDS
  JobDetail : ARRAY [1..10] OF BJobDetail
RULES
  Job
  NumJobs
  IF Job = Yes THEN
    FOR I := 1 TO NumJobs DO
      JobDetail[I]
    ENDDO
  ENDIF
ENDMODEL
```

Blocks can be defined once and reused several or many times, as demonstrated above. Here, the arrayed block *JobDetail* is called once for every job. For example, if a respondent has two jobs, the block is called twice. The reusability of blocks is an extremely powerful feature. If they are used appropriately, blocks can greatly reduce the programming and maintenance burden of questionnaires and reduce their internal complexity. This is because many questionnaires ask the same or similar group of questions about different topics. Appendix C illustrates two ways of customizing details within the same block structure to different subjects. That is, giving them different question text, edit limits, and so on, without having to reprogram the whole thing two or more times. It also illustrates how to separate within-block concepts from extra-block concepts, which is very cost effective for some surveys.

3. Types, Procedures, and Other Blaise Generalizations

In addition to reusable block definitions as described above, **type sections** (or libraries) and **procedures** offer a way to encode frequently used constructions in the Blaise source code. A **type** is a response definition. For example. many questions allow *Yes* and *No* as valid responses. In this situation, you can define a type definition, *TYesNo*, and use it in multiple places. Or, your questionnaire might use an agreement scale many times (*agree strongly*, *agree*, *neither agree or disagree*, *disagree*, *disagree strongly*) and you can define a type called *TAgreeScale5*. It is possible and advantageous to define each of these responses once and use the definition throughout the questionnaire. Use of pre-defined types reduces programming time and eases maintenance. For example, if your questionnaire all of a sudden had to be put in a second language, the use of types allows you to specify the alternate response text in one place, instead of in many places.

A **procedure** provides a way to define complex computations in a reusable segment of code. This is equivalent to defining a user function. For example, while Blaise has a RANDOM function, it does not offer a way to choose m unique elements out of n possible elements. The programming for this is quite complex, but once done in a procedure, this code can be reused in an instrument or between instruments.

4. Languages, Spoken and Unspoken

Blaise has long had a language capability that allows interviewers to switch between spoken languages. The LANGUAGES declaration has also become a place for declaring unspoken languages with other uses, as shown in the following example.

```
LANGUAGES =
  ENG "English",          {spoken}
  FRA "French",           {spoken}
  HLP "Help"              {unspoken}
  MML "Multimedia",       {unspoken}
  MDL "Metadatalanguage"  {unspoken}
```

In this example, two spoken and three unspoken languages are declared. Each has a 3-character identifier and a description between quotes. The identifiers such as ENG or HLP have no meaning to Blaise, except that these identifiers may be used later in the source code. In the developer's environment, it is possible to state which languages are spoken and which are not. Spoken languages are available to interviewers, while they never know about the unspoken ones. A function key can be used to toggle between spoken languages during the interview.

The unspoken languages are used for other reasons, including multimedia questions (sound, images, video), Microsoft® WinHelp links for question-by-question help, or as a repository for additional field- or block-level metadata. See section 5 below for details.

Blaise knows which language is in use. You can state IF conditions based on a language. For example, a text fill might be computed one way for English and another way for French.

5. The Many Uses of a Field Definition

The **field** is the basic unit of data definition in Blaise. There are six elements of field definition, as shown in the following example.

```
FIELDS
  FieldName  (FieldTag)
             "FieldText" / "FieldDescription"
             : FieldValue, FieldAttributes
```

Section IV, below, details how to explain field specifications to the Blaise programmer. Here we describe what can be done with the six elements of fields with respect to screen display and downstream metadata description, taking into consideration the various needs of users and data export and analysis. The point of this discourse is to allow you to specify what you want without confusing the programming or specification.

Following are descriptions of Blaise field elements:

- The *FieldName* is used in the rules of a Blaise program to describe routing, edit checks, and computations. It can be displayed on the **page** of the Data Entry Program (DEP), and in the edit jump dialog. It can be used as a downstream metadata identifier for the question through a Cameleon setup. Cameleon setups provided with the Blaise system use the *FieldName* as the default downstream identifier. *FieldName* is unilingual. It is required.
- The *FieldTag* may be displayed in the **page** of the DEP (and in the edit jump dialog) if the *FieldName* is also displayed. In the DEP, it can be used to jump to a field through the jump dialog. It can be used as a downstream metadata identifier for the question through a Cameleon setup. It is not used in the Rules section. *FieldTag* is unilingual. It is optional.
- The *FieldText* is almost always displayed in the InfoPane and is used as the question text. Short field text can be displayed in the Blaise Page though this is a specialty use. Cameleon can access the value of the *FieldText*. It is not used in the Rules section. *FieldText* is multi-lingual and can take fills. It is optional.
- The *FieldDescription* may be displayed in the Blaise page as an alternative to the *FieldName*. It can also be displayed in the edit jump dialog. This is a new feature in Blaise for Windows which started in 1999. Cameleon can access the value of the *FieldDescription*. It is not used in the Rules section. *FieldDescription* is multi-lingual but cannot take fills. It is optional.
- The *FieldValue* defines valid values and is represented in the Blaise page by a data entry cell. *FieldValue* can be accessed by Cameleon. *FieldValue* is displayed in the edit jump dialog. It is required.
- The *FieldAttributes* include whether *Don't Know* (DK), *Refusal* (RF), or *EMPTY* are allowed. *FieldAttributes* can be accessed by Cameleon. They are required though if you do not state anything the defaults are NODK, NORF, and NOEMPTY.

Possible Uses of Field Elements and A Suggestion for a Field Name Convention

There can be several ways a field is known to various users of the system. For the interviewer, a readable identifier of 1, 2, or 3 words helps immensely in understanding the instrument and in navigation. The specifier, project staff, and data review personnel may be more comfortable with a question number. Data analysts probably prefer something between a question number and a description, but they would like to have the description too. In fact, their downstream package, for example SPSS, may require variable names of eight characters or less but allow a label of up to forty characters. In a two-language setting, where interviewers are not bilingual, it may be required that the interface be either totally in one language or totally in another language, including all screen display elements. There are yet other required identifiers that may not be obvious, including a link to WinHelp for some or all questions, and perhaps some entity identifier for a downstream relational database system. And as discussed in Appendix C, it is desirable to not attach block-level information to a field-level name.

Given these considerations, we present a field naming convention that allows you to reconcile all these requirements including taking advantage of new features such as descriptions in the page and use of WinHelp for question-by-question help.

An example for elementary fields (not block fields) is shown below:

```
FIELDS
  FieldName  (FieldTag)
             ENG "English Field Text"
             FRA "French Field Text"
             HLP "WinHelp link"
           / ENG "English Field Description"
             FRA "French Field Description"
           : FieldValue, FieldAttributes
```

Where:

- *FieldName* is used in the Rules (you cannot get away from this) and it is the default downstream metadata identifier. It is *not* used in the Blaise page or in the edit jump box. The interviewer never sees it in the screen configuration advocated in this paper.
- *FieldTag* is used for question numbers if they are specified. It is also used for selected section jump points. For example, a household enumeration table may be given a tag of **hh**. Then the interviewer can jump to **hh** to return to that section. For example: **A** to jump to section A, **B** to jump to section B, etc.
- *FieldText* is used as the question text in the InfoPane for spoken languages such as English or French.
- *FieldText* for the HLP language is used as the WinHelp link. If there is no WinHelp for a field, then this entry is left blank.
- *FieldDescription* is used as the interviewer identifier and additionally as a label in a downstream system for all spoken languages. When the interviewer switches languages, everything on the screen, including the visible field identifiers, switch to the next language too. If there is an edit, then the fields are identified by the readable *FieldDescription* in the appropriate spoken language.

The following example shows block fields:

```
FIELDS
  FieldName
    ENG "English banner" {at the top of every field in the block.}
    FRA "French banner"
    MDL "Entity"         {if used in your survey}
    : BlockDefinition
```

The question text defined in a block field definition can be used as a banner, appearing at the top of every InfoPane for every field in the block. The MDL language was mentioned above as a possible repository of additional metadata. It is possible, for example, to use an MDL *FieldText* or *FieldDescription* as an alternative downstream metadata identifier. This is an advanced topic and requires someone that knows how to program Cameleon (or for someone to share a Cameleon script).

6. In Blaise, a GoTo is a No-No

Many paper questionnaires, and many other interviewing systems, use the GoTo statement to direct interview flow. Blaise does not. It implements skips solely through IF conditions. In other words, you state the condition under which a field is on the route. This is known as using **gates** and is also known as stating the **universe** of the field. It is felt by many that by eliminating the use of GoTo (in any system) the source code is better structured, easier to read, and more easily maintained.

If your routing specification is in GoTo format, someone will have to translate it into the converse convention of IF conditions. This can be surprisingly difficult, enormously time consuming, and prone to error. It should not be left to the programmer to do this. The specifier, or someone else, should be the one to state flow in terms of IF conditions. This should be recognized as a separate task. You can produce flow charts to aid the translation, and you can provide block-level scenarios to test the routing.

7. Edits are Stated in a Positive Sense (Usually)

For historic reasons, edits in Blaise are usually programmed in a 'positive' sense. That is, the edit should be stated in terms of what should be correct, not what is incorrect, as shown in the following example:

```
IF Job = Yes THEN
  Age >= 14 "Respondent is too young to have a job."
ENDIF
```

The following example is also valid, but many long-term Blaise programmers may not know this.

```
IF (Job = Yes) AND (Age < 14) THEN
  ERROR "Respondent is too young to have a job."
ENDIF
```

The point is to make sure the specifier and the programmer have this straight between them.

8. Data Export, Data Structures, and Metadata Description

Blaise supports three kinds of data export. ASCII export produces one flat rectangular file as output. ASCIIRELATIONAL export produces one data file per unembedded block, thus resulting in several or many output files. The third kind of data export is a custom export where a Manipula program exports data according to (potentially many complex) instructions. This custom Manipula program can either be hand programmed or Cameleon can generate it based on Blaise metadata contained in the field declarations. There are a few issues to keep in mind in any of these options. Appendix D discusses these three options at length. Suffice to say for now that data export is something one should keep in mind at the time of specification.

9. New Possibilities in the Windows Version of Blaise

The first Windows versions of Blaise were designed to be upwardly compatible with Blaise III, the last DOS version. It is possible to invoke a Blaise III instrument in Windows, and execute a perfectly acceptable Windows instrument. However, Blaise offers more features as time goes on. An incomplete list includes font size and font style possibilities; native multi-media features such as audio, graphics, and video; tab sheets and better labeling for parallel blocks; a much more configurable interface and a tool (mode library editor) with which to do that configuration; a configurable user menu; WinHelp for question-by-question help; an audit trail; enhanced CATI management features; use of field descriptions in the page (instead of field names); ability to tie function keys to parallel blocks, DLLs, or executable programs; enhanced use of DLLs in the rules; identifiable checks and signals; ASCII external files and external files in memory, and edit masks for formatted data entry (e.g., dashes in a phone number). An important new capability, that of Open Blaise Architecture, will be released after the writing of this paper in Blaise version 4.5. This release will allow tighter integration with other Windows systems and open up many new possibilities that are beyond the scope of this paper.

10. Adapting an Instrument

It is easy to change a Blaise data model. Probably more than in any other system, you can easily add or delete questions or whole blocks, change the flow of interview, establish new edit limits, and so on. There is a drawback, however. Even minor changes to the data model can result in a change in the Blaise data file definition. This can be unexpected. For example, if you add an edit, or a response to a pre-coded question, then the new data file can be incompatible with the old data file.

It is not difficult to write a Manipula setup that can translate the data from one version of a data model to another. However, there is an operational problem. You might have different versions of Blaise data sets lying around, and this can be difficult to manage. For example, if one interviewer does not receive an update to a data model, then when she sends her data in, you may try to process it with the wrong Blaise data model and the process can end abnormally.

From the standpoint of specification, let it suffice to say that it is always desirable to send a completely correct instrument to the field. This is helped by clear specification and by knowledge of the process that produces a specification. In those situations where a data model must be changed in the field (with resulting data definition changes) the organization executing the field work must have procedures in place that can handle this transition in data. Such a data definition change need not be traumatic if it is properly anticipated.

11. Capacity and Performance

It has been proven in a few Westat data models that exceedingly large data models, with hundreds of thousands of defined fields, edits, and computations, can execute quickly even on relatively low-end computers (see for example, Frey, 2000). It may take an expert to design such a data model, but it can be done. In both Westat data models, time of instrument administration is not large. That is because they are well-specified in a research sense. There are a large number of potential questions, but in any given interview, only a small proportion are actually asked.

Just because you can specify such large data models does not mean that you should. You have to consider respondent burden as measured by the numbers of questions that are asked. It is possible to specify a large data model, have it run well, and yet be a burden to the respondent. This is not a Blaise issue as much as it is a research methodology issue.

IV. Drafting a Blaise Questionnaire

This section covers the formal specification of the research. An example specification in Appendix B illustrates the guidelines offered here.

1. The Early Draft

Drafting a Blaise questionnaire usually has two stages:

- An early draft that follows only the most basic Blaise principles but doesn't include any frills or detail;
- A later, more developed, draft for the programmers that is more tightly specified, and which can quickly be turned into useable source code.

At the earliest stage of questionnaire development, while the draft is going back and forth between researcher and sponsor, there are good reasons for *not* writing very much that is Blaise-specific. It is inefficient to get into Blaise-related detail while the questions themselves are still subject to amendment or deletion. And sponsors and others who may be completely unfamiliar with Blaise conventions will find it difficult to read.

2. Designing the Questionnaire Structure

The considerations for a CAI questionnaire are very similar to those for a paper one. A questionnaire will usually divide naturally into modules according to the question theme or topic. Within topics, the questions may break into smaller sets according to narrower topics or sub-themes. These modules can often be programmed and tested independently in Blaise, as **mini-data models**. This is an efficient use of time, since each module can be specified, worked on and tested separately.

The general Blaise term for any such group of questions - broad or narrow - is a **block**. So you can have broad blocks and smaller blocks within them, For example, a block of questions on work, and a block inside on travel to work. When specifying blocks, don't re-invent the wheel. Programmers in survey organizations will have blocks of standard or near-standard Blaise code that can be re-used on new projects. This is particularly so for:

- Respondent classification section
- Admin block and outcome codes
- Household grids for adults and children

Talk to the programmers before starting to write a specification. They may have some off-the-shelf source code that can be tailored to your needs. It is also useful before getting started to review the questionnaire draft for the following:

- Common structures in the questionnaires that may form the basis for re-usable blocks.
- Common answer categories that may become Blaise types.
- Determine, even if in general terms, what kind of data readout will be necessary.

High-Level Descriptive Specification

When laying out the questionnaire, it is useful, and at times crucial, to describe how blocks relate to one another. For simple questionnaires, this might be as easy as stating that Section A comes before Section B and so on. For more complex data models, such as a hierarchical questionnaire with several respondents, you should explicitly state how the interview is to flow.

For example, should all one person's questions be asked before another, or should they be asked concurrently? Should you be allowed to leave one person before they are finished and proceed to a second? How should job questions be linked to a row in a household membership table? What happens if a member of the household is deleted but there are blocks of questions that relate to that member, how should they be treated? An example of such descriptive text is given in Section 2 of Appendix B, In the Enumeration Table, Across Member Rows.

3. Basic Specification Conventions and Getting Started

Section III above described basic Blaise terms such as fields and rules, and how they combine to make up a data model. In Blaise source code, fields and rules are separated. For most people this is not a natural way to draft a questionnaire. For all but the simplest of data models, drafting is more straightforward when the questions and their routing conditions are stated side by side. We suggest a compromise, using the 'natural' method but without making too much additional work when the draft is turned into source code.

The specifier and the programmer should agree on basic conventions for the specification. An example of how this is done is given in the first part of Appendix B.

4. Questions (FIELDS) Within a Block

The metadata and display aspects of fields were discussed in section III, above. It is important to make explicit choices about how the field definition elements will work for your survey, including screen display and metadata requirements. Here we discuss their actual specification.

Question Texts

Question text (Blaise FieldText) is placed inside double quotes, and followed by a colon:

```
"When did you take out that pension?" :
```

Every question must be assigned a unique name (Blaise Fieldname), alpha or alpha-numeric:

```
Pen6q
"When did you take out that pension?":
```

Specify show cards and interviewer instructions inside the quotes (and in capitals):

```
Pen6q
"SHOW CARD A  When did you take out that pension?
INTERVIEWER: IF IN DOUBT, REFER TO CALENDAR" :
```

Question Names

There are no hard and fast rules for naming questions. Some people have systems, such as, '*all questions in section A start with the letter A*' (See Appendix C for a discussion that discourages putting block-level meta-data as part of the elementary field definition when a block might be reused for a different, but related, topic).

A 'fast index' method can be used to create names with a dual reference: First, an alphabetic reference to a *module* of questions, and then to an *index number* within the module. So questions in a module about Education could be named

```
    Edu1, Edu2, Edu3 {and so on}
```

This is a lot easier and faster than inventing, for each individual question, a distinct name that attempts to summarize the subject matter.

One drawback with fast indexing is seen when questions are deleted or moved later on, thus breaking the number sequence; or when extra questions are added, in which case you'll need to insert an additional sequence letter (e.g. Edu5**a**, Edu5**b**). Another drawback is that the name is not descriptive of the individual question, making it less useful for the interviewer and the data analyst. If there is time to write meaningful names, this is the best option.

Many 'downstream' systems - SPSS being one example - allow a maximum of only eight characters for variable names, so it is easiest to restrict the name to this length. However, there are other ways to add descriptive information. One is via the *field description*, which is specified following the question text:

```
Pen6q
"When did you take out that pension?" / "self pension" :
```

The programmer can arrange for the field description to be displayed on screen instead of the *name*, which makes navigation clearer for interviewers. It can also be used as a descriptive *label* for downstream systems such as SPSS, rather than the default option of using the first forty characters of the question text as a label, which may not express the core intent of the question.

Another option for keeping track of questions is to use the *field tag*, which is specified following the question *name*:

```
Selfpen (q6)
"When did you take out that pension?" :
```

Here a *tag* is used to show the sequence number of the question within the module. Since the tag is not referred to in the routing, it is easier to keep up to date as drafts are amended. Tags can also be used to jump to a field or a section.

You should at least *give a meaningful name to key questions* that are referenced frequently in the questionnaire. Because the question *name* is referenced in the rules, the source code itself will be more readable if key question names are simple and self-explanatory (*Age*, *Sex*, *Tenure*, *EmpStat*, *MarStat*). A routing instruction like `IF Age > 50` is clearer than `IF HGrid4` > 50.

Blaise is case-insensitive, so *MarStat*, *MARSTAT* and *marstat* are all read as the same question. The preference is for logical mixed case (*MarStat*, *EmpStat*).

If you use the automatic Blaise facility to set up an SPSS data set, whatever Field Names you assign your questions in Blaise will be carried over as the SPSS variable names (including your case conventions).

5. Precoded Answers

Put each pre-code in double quotes, separated by an external comma, and put the full list of answer codes in parentheses (brackets). For example:

```
Pen14
"Who contributes to this pension: you, or your employer, or both of
you?" :
("respondent only",
"employer only",
"both")
```

In the final source code, each answer category also must be given a name with a maximum of eight characters). For example:

```
(resp   "respondent only",
 emplyr "employer only" ,
 both   "both")
```

The answer name is essential to the final Blaise specification of the *rules*, where names are referred to directly (*e*.g., *If Pen14 = resp*). However, in drafting the rules you can get by with just referring to the answer *text* (*If Pen14 = 'respondent only'*). On balance it is probably more efficient to begin with answer names, rather than to add them later.

The interviewer's screen will display the answer code text, unless no text was specified, in which case the name is shown. Where there is a single spoken language, it is unnecessary to write both name and text if the name itself is entirely self-explanatory. For example:

```
Nwarea2

"Is there a Neighbourhood Watch Scheme in this area?" :
(yes, no)
```

It would have been unnecessary to write:

```
"Is there a Neighbourhood Watch Scheme in this area?" :
(yes  "yes",
 no   "no")
```

since the screen appearance is identical in both cases. This changes for a bilingual instrument. In that case, you would have the following:

```
(yes  "yes", "oui",
 no   "no",  "non")
```

It also pays to pay attention to your text conventions. If the responses *yes* and *no* are **not** to be read to the respondent, then use the following:

```
(yes  "YES", "OUI",
 no   "NO",  "NON"
```

or, in a single language setting:

```
(YES, NO)
```

For the same field you can provide labels for some codes, but not others:

```
MarStat  "What is your marital status?"  :
(single     "single, never married",
 married,
 divorced,
 widowed)
```

6. Re-using the Same Answer Codes

In Blaise, a list of answer categories that is repeated frequently for different questions need only be specified once. They are declared as a TYPE and given a name. When the list is next required, only the TYPE name is specified; this will have the effect of calling up the full list. For example, the list below might be used at many different questions:

```
(agstr    "agree strongly",
 agree    "agree"
 neither  "neither agree nor disagree",
 disagree "disagree"
 disagstr "disagree strongly"), DONTKNOW
```

This can be specified just once as a TYPE, for instance

```
{TYPE: 'AGREEDIS':  answer codes "AGREE-DISAGREE"
(agstr    "agree strongly",
 agree    "agree"
 neither  "neither agree nor disagree",
 disagree "disagree"
 disagstr "disagree strongly"), DONTKNOW }
```

and then called up for different questions as required:

```
PolPow  "Do you agree or disagree that the police have too much
power these days?" :  {TYPE: AGREEDIS}

UniPow "And how about the trade unions - do you agree or disagree
that they have too much power nowadays?" : {TYPE: AGREEDIS}
```

Discuss this option with your programmer, who might have an off the shelf TYPE you can use.

7. Multi-Codes

Questions that allow more than one answer to be chosen are specified as SET questions, as shown in the following example:

```
"SHOW CARD B
What were your reasons for taking out this pension?   Please look

SET [2] OF
(morinc "To have more income in retirement",

state   "The state pension may not exist by the time I retire",
oth     "Some other reason")
```

Here, **SET [2] OF** means a maximum of two of the answers can be selected. Alternatively, omit the number (**SET OF**) to allow all codes to be selected.

8. Numeric Answers

For a numeric answer, specify the range of possible entries, separated by 2 dots (..). For example:

```
Age "What was your age last birthday?" :  16..120

PenNum "How many pensions do you have altogether?" : 1..5

PStart "Which year did you take out that pension?" : 1930..1995

Pamt "How much money did you contribute last time?" : 0.00..9997.00
```

Interviewers will not be able to key numbers outside the range you specify. Note that in fourth example above two decimal places are defined.

9. Text Answers

For some questions the interviewer can key the respondent's answer verbatim. Blaise features two ways to record text, OPEN and STRING.

Where the answers are to be assigned codes in a separate operation, it is advantageous to specify the question as OPEN in Blaise. All the answers to a particular OPEN question can be exported and viewed together, and coded in one operation, the resulting codes later being merged back in to the data set. The Blaise Data Entry Program also has an explicit review dialog of all OPEN fields. There is no limit to the number of characters that can be keyed in response to an OPEN question.

If you want the responses to a text question to be stored and exported as part of the data record, you should specify the question as STRING instead of OPEN. On screen, the STRING answer is entered by the interviewer in a very similar manner to an OPEN answer. You also need to specify the maximum number of characters that can be entered, *e.g.*,. STRING [80]. Use of string is indicated for short answer fields such as *Name*, *JobTitle*, *StreetAddress*, and so on.

A question often has an *other answer* category. If you want the interviewer to record that other answer, a separate question is needed, for example:

```
Pen27q
"SHOW CARD B  What were your reasons for taking out this pension?
Please look at this card and choose all the answers that apply." :
("To have more income in retirement",
 "Because it gives me life insurance cover",
 "The state pension may not exist by the time I retire",
 "Because most people at work are in the scheme",
 "People at work told me it was a good scheme",
 "some other reason (WRITE IN)" )

OthReas
"INTERVIEWER: ENTER OTHER REASON" : OPEN
```

10. Date and Time Answers

If you want the interviewer to enter a calendar date – For example, *21 9 2000*, specify the question as DATETYPE:

```
Pen6q
"SHOW CARD A
When did you take out that pension?
INTERVIEWER: IF IN DOUBT, REFER TO CALENDAR" : DATETYPE
```

If you want the interviewer to type in a time – For example, *7:00 am*, specify the question as TIMETYPE:

```
Pen7q
"What time do you get up in the morning?" : TIMETYPE
```

See the discussion below on Windows Configuration for details on how the interviewer can enter the date and time.

11. Wording Substitutions (Text Fills)

Sometimes you want to vary the question text according to circumstances. The reason might be to save the interviewer the effort of choosing among wording options (*e.g., he/she; did you/do you; first/next; this/that)* , or to make sure they read a special insert (e.g., *Thinking of just your main job, how much did you earn? How old is your daughter Sally?)*

One of the strengths of Blaise is that it can set up complex substitutions (called *text fills*), that interviewers find helpful. *However, they are time-consuming to set up and test.* The effort saved for the interviewers is time added to the tasks of researchers and programmers. It is easy to get carried away by text fills, but often the time-efficient practice simply is to state the options in the wording - e.g., *"When did (he/she) leave that job?"; "How long have you lived in this (house/flat)?"* - and let the interviewer choose, as they would do with a paper questionnaire.

Where you plan to use a fill, in the early draft, it is quickest just to put the optional wording in parentheses; adding the 'hat' symbol (^), which Blaise uses to denote a fill:

```
"(^Thinking just of your main job) How many hours a week do you
usually work?" : 1..120

"What was the purpose of the (^first/second) visit?" : STRING [80]

"Why did your child (^NAME) visit the dentist?" :
```

12. Routing (RULES)

Questionnaires are divided into *blocks* of questions, usually according to topic. You need to think of every individual question as being located somewhere within a block. It is common to have broad blocks (such as whole sections of the questionnaire) and smaller blocks (short sub-sets of questions) within these large blocks.

There are rules (routing instructions) for blocks. Within blocks, there are rules for individual questions. In the final source code, the rules are stated separately from the description of fields (questions and blocks). However, a more natural way to draft a questionnaire is to specify, in one place, the field *and* the condition under which it comes on the route.

Our recommendation is to write the draft in this natural way, stating the rules next to the fields. However the rules specification should be stated within curled brackets { }. In Blaise, any text placed inside curled brackets is a *comment* and is ignored by the program. Later, the draft rules can be extracted (or copied) to form the separate *rules paragraph* required by Blaise.

Remember that Blaise does not use GoTo. The specifier should translate from any GoTo specification to a gate or universe specification solely in terms of IF conditions.

Routing to a Block

In the draft, use a plain-English description of the rules, but also refer to the specific fields that determine the rules. For example:

```
{now, a short block of pensions questions:  if under retirement age
(men, Age<65, women, Age<60)}
```

Routing from a Precoded Answer

It can help if the rules are in bold text. First, specify the routing condition in plain English. Then, if possible, add the precise condition, referring to questions and answer codes by name. For example:

```
 {IF respondent has a personal pension: Pen5 = haspp }
 Pen6q
 "When did you take out that pension? " :  DATETYPE
```

Note that either condition stated separately would be less than optimal. *If respondent has a personal pension* is not precise enough; *Pen5 = haspp* is not clear enough.

Routing through a Sequence

Don't repeat routing unnecessarily. If the routing to the field has already been stated, for example, there is no need to re-state it above each following question until the condition changes.

Routing from a Numeric Answer

If one particular numeric answer to a question determines the routing condition, state that number (e.g., *If Visits = 0*). If the condition applies for a range of numeric answers, use the Blaise term IN:

```
Hea20
"How many times did you visit your doctor last month? " (0..5)

{IF HEA20 IN 1..5}
Hea21
"Did you receive a prescription on (this visit/any of these
visits)? "
```

In this example, the question Hea21 will enter the route only if the answer to Hea20 is a positive number in the range 1 to 5. The intention is to skip Hea21 if the answer to Hea20 is *none*.

13. Rules for Repeated Questions

With Blaise it is only necessary to describe a set of questions once. You then use the rules to tell Blaise either to re-use a block of questions; or to place questions in a table.

Reusable Blocks

Specifying a block of questions as re-useable is typically done when the information will be collected in different situations – for example a series of questions that is asked several times in reference to different people (e.g., job description questions, asked first about the respondent, then about their spouse). You should specify this type of repeated sequence as a block, and state the conditions under which the block should be repeated, as shown in the following example.

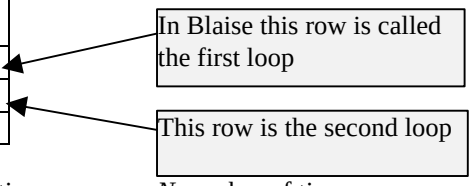
```
{new block: job details.  Ask for the respondent, then repeat for
their spouse / partner, if married or living as married (MarStat =
married or livemar}
```

Rules for Tables

A sequence of questions that would appear as a grid in a paper questionnaire would usually be defined as a **table**. For example, every adult in the household may be asked, in turn, their age, sex, marital status, and when they left education. In a paper questionnaire this appears as shown in Table 1:

Table 1

	Age	Sex	Marital status	Age left school
Adult 1				
Adult 2				
Etc..to *N*				



In Blaise the row cases (e.g., adults) are said to "loop" through the question sequence, *N* number of times.

The exact specification a CAI table can be complex, and is best left to the specialist programmer. When you have a set of questions that will form a table, use the terms TABLE and FOR to define what you want, followed by a single specification (one loop) of those questions and their routing.

```
{TABLE:  FOR each adult aged 16+, ask Age, Sex, MarStat, TEA }
```

The maximum number of rows in the table is determined by the answer to a preceding question. For example, the following code followed by *one* specification of the content and routing of those 3 questions.

,

```
PenNum
"How many pensions do you have altogether?" : 1..5

{TABLE: FOR each pension at PenNum, ask Pen6q, Pen7q, Pen8q:}
```

….

Here, the TABLE (of Pen6q, Pen7q and Pen8q) automatically opens a maximum of five times - or fewer, depending on the actual answer given at PenNum.

See Appendix B for an example of how to specify a table.

14. DON'T KNOWS, REFUSALS, and EMPTY

Blaise sets aside special keystrokes for the interviewer to record a *Don't Know (DK)* or *Refusal (RF)* for a particular question. (The default options are Ctrl+K and Ctrl+R, but these can be remapped to other keys or function keys.) Using these keys means that DK (don't know) or RF (refusal) will be recorded as a legitimate answer. This means you have to consider these possible answers when defining the rules.

Whether to Allow DK or RF

When you begin, you have a choice of two baseline settings. You can set your questionnaire to always allow DK (don't know) and/or RF (refusal) at any question; or you can set it to *always disallow* (i.e., forbid) them. From either baseline, you can then exempt particular questions. The *always allowed* setting is probably more common. The setting you choose depends on the nature of your questions. That is, you must consider. whether a DK and/or refusal is going to be, more often than not, a likely or legitimate response.

Under the *always allowed* setting, you use the Blaise terms , NODK and/or NORF to forbid these responses for a question. This mean that the questionnaire will not accept the keystrokes for *don't know* and *refuse* for that question. That is, these answers are declared out of range. You might want to do this at questions which are critically important for routing, and which are not thought to be sensitive or difficult (e.g., housing tenure).

This example forbids a *don't know* at the question *JobStat:*

```
EmpStat "Are you an employee or self-employed?" :
(emp   "employee",
 self "self-employed"), NODK
```

The following example forbids a refusal:

```
EmpStat "Are you an employee or self-employed?" :
(emp   "employee",
 self "self-employed"), NORF
```

This example forbids both:

```
EmpStat "Are you an employee or self-employed?" :
(emp   "employee",
 self "self-employed"), NODK, NORF
```

EMPTY

An EMPTY attribute allows the interviewer to move past a field without entering data. While not very common, they do have uses. For example, an apartment number field may be left empty if it does not apply to a household.

Routing Conditions from *Don't Know* and *Refusal*

If DK and Refuse are permitted answers, you need to consider carefully what the appropriate forward routing will be when those answers are given. It is easy, but dangerous, to overlook this. For example:

```
Hea20
"How many times did you visit the doctor last week?" (0..5)

{IF HEA20 IN 1..5}
Hea21
"Did you receive a prescription on (this visit/any of these
visits)?"
```

A *don't know* or *refusal* in an IF condition is evaluated as a 0. Thus, they will not be led to Hea21. To ensure they are included, the correct rules specification is:

```
{IF Hea20 > 0 or DK or RF}
```

If you want, for example, a follow-up question asked only for the *don't know* responses (but not *refusals*), use the term IN:

```
{IF Hea20 IN 1..5
note: including DK, not Ref}
Hea21
"Did you receive a prescription on (this visit/any of these
visits)?"
```

It is a common mistake to route don't know or refusal responses to an inappropriate follow up question:

```
{ask all}
Hunt8
"Do you agree or disagree with the statement: 'foxhunting is
cruel'?":
(agree     "agree",
 neither   "neither agree nor disagree",
 disagree  "disagree")
```

If you wanted to ask a follow-up question of those respondents who agreed or disagreed, you might think you could specify the routing as follows:

```
{IF Hunt8 < > neither}
Hunt9
"How strongly do you feel about that?" :
```

(< > means *is not equal to*). However, if you did that, people answering DK and refusal at Hunt8 would be asked the follow-up question. This is probably not what you intended. So be sure that the specification is absolutely clear:

```
{IF Hunt8 = agree or disagree ONLY; Skip if DK/Refusal}
Hunt9
"How strongly do you feel about that?" :
```

When you are testing your questionnaire, it is useful to enter DK, then Refusal, at every question, to check that people are not being sent on to inappropriate questions from these answers.

With some questions it may be preferable to include DK and refusal options in the list of answer codes directly available to the interviewer, as shown in the following example:

```
Vote
"Which party did you vote for in 1987?" :
(NoVote     "Didn't vote",
Cons        "Conservative",
Lab         "Labour",
Lib         "Liberal",
Other       "Other party")
DKnow       "Don't know",
Ref         "Refused to say"), NODK, NORF
```

If so, disallow the use of <Ctrl+K> and <Ctrl+R> for this question (e.g., by inserting NODK and NORF). Otherwise you may end up with two don't know codes and/or two refusal codes.

15. CHECKS

Sometimes you want the interviewer to be alerted to an answer that is factually inconsistent with a previous answer, or that appears very unlikely, or is simply not possible.

Checks are triggered by a response or combination of responses. A small window appears on the screen, relaying your message to the interviewer.

Probably the main function of checks is to catch interviewer mistakes in the-keying of answer codes, and in particular, of numeric answers (it is easy to hold down a key too long). Aside from this, checks can allow us to query an answer directly with the respondent, bringing some (or all) of the editing of questionnaires forward into the interview. This saves time later on for editors and researchers. It should improve data quality, as we are checking answers with the respondents themselves.

Do not add a check just because you can. In any questionnaire you could include literally hundreds of checks. Avoid the temptation to put in a check just because it is possible to do so, for the following reasons:

- A triggered check brings the interview to a temporary halt, while the interviewer reads your message and acts on it. Checks will inevitably slow down the interview.
- Checks can be very time-consuming to program and test.
- A check, even if not triggered, adds complexity to the program.

Where and How to Specify a Check

In the final source code, checks are included in the rules paragraph, separately from the fields. However, we are recommending that in drafting a questionnaire, the rules should be stated alongside the fields (questions). Therefore, write the check immediately after the question where you want it to be triggered. Later, it must be copied or moved into the rules paragraph.

Checks are like questions in that they appear only if certain conditions are satisfied. So, in plain English and with specific references, write an IF… condition under which the check will be triggered, followed by the wording of the check. Enclose all this in curly brackets. (See Section III for a note on the sense of an edit.)

Writing Check Messages

If an interviewer has simply made a keying error, the check will alert him to the error, which he can amend and move on without needing to consult the respondent. If they have the response given by the respondent, they need to let the respondent know there is a problem with the information.

There are three possible strategies for your message:

- Write the entire message to be read aloud.
- Write a message to the interviewers, which they then relay to the respondent.
- Implement a mixture of these.

The read aloud approach should be used whenever possible, as this makes the interviewer's task easier. There are no precise rules about what to say. Check texts should follow the standard questionnaire convention that lower case signifies material to be read aloud, while UPPER CASE is used for messages to the interviewer, such as instructions. Some examples of check messages are shown in the following example:

```
"The computer's asking me if I've put that in correctly - can I
just check, your weekly rent is {^RENT7}, is that right?  IF YES,
SUPPRESS WARNING AND CONTINUE"

"That's over 75 hours a week, is that correct?"
```

In a delicate or awkward situation, however, it may be preferable to tell the *interviewer* what the problem is, and let them decide how to handle it. The whole message should therefore be in UPPER case:

```
"INTERVIEWER: IT'S USUALLY NOT POSSIBLE TO WORK 30+ HOURS A WEEK
AND GET INCOME SUPPORT. TACTFULLY ENQUIRE IF BOTH ANSWERS ARE
CORRECT. IS IT 30+ HOURS?  IS IT I.S. OR SOME OTHER STATE BENEFIT?
[14]"
```

It can be useful for testing purposes to give each check message a unique reference number – see [14] in the above example.

Be succinct. Writing a check message should be like sending a telegram when you have a lot to say, but very little money:

- Start with a quick summary of the nub of the problem.
- Use simple words and short sentences.
- Re-draft check messages to cut out unnecessary material.

Hard Checks

Checks are either *soft*, allowing interviewers to suppress the warning and move on to the next question, or *hard*, requiring a change to be made.

Use hard checks sparingly. Very few responses are impossible. The respondent can be mistaken about an answer, but may nevertheless be quite convinced it is correct. A hard check forces them (or the interviewer) to change it. The issue is not "could the respondent *be in* a situation that would trigger the check?", but "could the respondent ever *think* they are in that situation?" If the answer is "yes", put in a *soft* check.

Use hard checks to pick up on impossible combinations of responses. A common example is where an interviewer makes an error-keying from a multiple choice list, entering *none of these* **plus** another answer code. You should routinely insert a hard check after such a question:

```
Ben1
"SHOW CARD P  Are you currently receiving any of the benefits on
this card?" :
SET OF
(is   "Income Support",
 jsa  "Jobseeker's Allowance"
 ssp  "Statutory Sick Pay"
 smp  "Statutory Maternity Pay"
 none "none of these")

{HARD CHECK:
 IF Ben1='none of these' AND any other answer is selected:
"'NONE OF THESE' IS AN EXCLUSIVE CODE"}
```

For contradictory answers, specify all question names and codes:

```
{HARD CHECK:
IF Ben1= smp and sex = male: "MEN CANNOT GET MATERNITY PAY"}
```

Note that there is no need to write a check to catch extreme values that fall outside the range already specified for a numeric answer. For example, if the range for Age is 0..120, it is pointless to add a check stating that Age must be greater than 0 and less than 120.

Soft Checks (Signals or Wwarnings)

Soft checks can be suppressed by the interviewer. Use soft checks to pick up implausible or unlikely responses. Soft checks on amounts are important, because keying errors commonly occur with these values. A soft check is shown in the following example.

```
Cig7  "How many cigarettes a day do you usually smoke?" : 1..100

{SOFT CHECK:  IF more than 80 cigarettes (Cig7>80):

"The computer's asking me if I've put that in correctly - can I
just check, you said {CIG7} cigarettes a day, is that right?
INTERVIEWER: IF YES, SUPPRESS WARNING & CONTINUE" }
```

Checks INVOLVING Other Answers

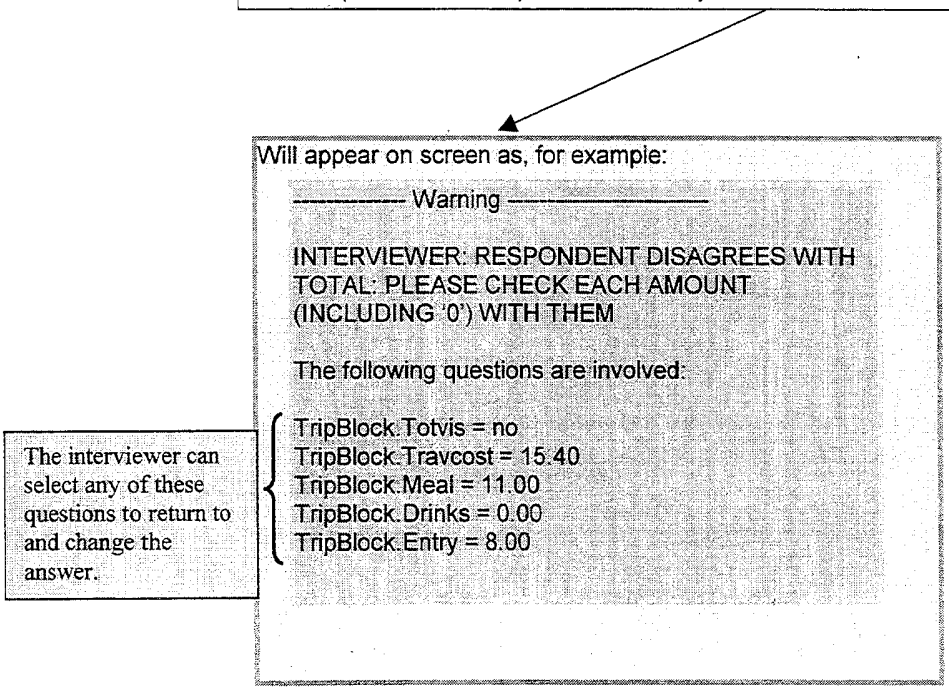
When specifying a check, you must state which answer(s) at which question(s) will trigger the check. In addition, you can also get the check to display other earlier responses to the interviewer, thus providing them with additional information, and allowing them to go back and amend any one of those earlier questions.

To do this, use the Blaise term INVOLVING to refer to responses that do not themselves trigger the check, but that you want to be displayed. Example: A check that arises if the respondent disagrees with a displayed computation of the "total money they spent on a trip," that calculated by adding together earlier answers about how much was spent on each component (travel, meals, drink, entry fees):

```
Totvis  "So that comes to a total of (TOTSUM) spent on that trip,
is that correct?" : (yes, no)

{SOFT CHECK, IF Totvis = no:  INVOLVING Travcost, Meals, Drinks

" INTERVIEWER: RESPONDENT DISAGREES WITH TOTAL: PLEASE CHECK EACH
AMOUNT (INCLUDING '0') WITH THEM"  }
```



16. Computations

You can get Blaise to do behind-the-scenes math and text manipulations. This can be useful when you want to confirm with the respondent a figure they have supplied. The specification for this situation is very similar to a fill.

Use COMPUTE, and give a name to the computed variable, to tell the programmer your requirements:

```
{ask all}
Drink2 "In the last 7 days, how many times have you visited a pub?"
: 0..25

Drink3  "And how many times have you visited a wine bar in the last
7 days?" : 0..25

{IF Drink2>0 and Drink3>0:
COMPUTE ^NUMBER:= visits to pub, + visits to wine bar:
and ask Drink4}
Drink4  "That's a total of ^NUMBER separate visits to pubs and wine
bars in the last 7 days, is that correct?" : (yes, no)
```

Derived Variables

If you know which variables you will be deriving for data analysis, and if you have time, you can make room for the derived variables in the questionnaire. The advantage is that the variables - with the correct texts and code frame - will then be in the correct place for the analysis. (Derived variables created by SPSS are placed at the end of the data set). Also, it will enable analysis to begin more rapidly.

Obviously, you do not want the derived variables to 'come on route' during the interview, so ask the programmer to keep them off the route (using the KEEP command)

17. Page Appearance Specification

In the columnar page format you can specify where to place a label, start a new column (NEWCOLUMN), skip a place in the column (DUMMY) or start a new page (NEWPAGE). The following shows how to state this specification.

{NEWCOLUMN}
{Blaise page label}
"Household introduction"

{DUMMY before Building type}

The effect of these instructions is shown in Figure 2 in the right-hand column. There you can see that column headed by the label "Household information" and that there is a space in the column before *Building type*.

V. Specifying the Blaise Interface

There are several aspects of the interface that could or should be the subject of specification. Some of these aspects can be declared across surveys, including: Windows configuration settings for each computer, major components of the Blaise screen (as given in Figures 1 and 2), design of the Blaise page, including navigational issues and groups of items, and InfoPane elements. The organization doing the fieldwork may already have defined Windows display guidelines and their interviewers may be used to them. If there are special needs, probably the easiest, least expensive, and surest way to arrive at necessary special displays is to define special requirements as a deviation from a known working standard.

1. Windows Configuration

Only a few important Windows configuration settings can be addressed here. These include how date and time are entered, screen resolution, color sets, and whether the bottom task bar should be visible or hidden. It helps the organization hold down costs if it can establish certain Windows configuration choices as standards.

Date and Time Regional Settings

Blaise inherits the format and manner of entry of date and time fields in the instrument from Windows. Thus, an American interviewer can enter date in Month/Day/Year format while his British counterpart would enter in the Day/Month/Year format, using exactly the same instrument. Regardless of how dates are entered, they are stored internally in Blaise in the Year/Month/Day format.

Screen Resolution

Screen resolution refers to the number of picture elements (pixels) that are used to display information on the computer screen. Currently there are two major resolutions to consider. These are 800 x 600 and 1024 x 768. You should know your target resolution ahead of time, including whether you have to satisfy both (this can happen in a multi-site study). An instrument that fits just right on an 800 x 600 screen will take up only a large fraction of a screen using the higher resolution and all the elements on the screen will be much smaller. Fortunately, the Blaise configuration settings are held in external configuration files. So, using a mode library file suited to 800 x 600 resolution as a preparation/compilation standard, and adapting its settings to 1024 x 768 implemented in a runtime override, is one way of proceeding with the same instrument.

Color Sets

Not all computers have the same allowed colors. You should make sure you know the color set of the target computers. Low-end CATI workstations or laptops may not have all the color definitions of your powerful desktop machine. Make sure that any color choices you make will work well on the target machines.

Font Size

When deciding a font size standard, keep in mind that the physical size of a character on a screen depends on four factors.

- Font size.
- Font style, especially proportional vs. non-proportional fonts.
- Screen resolution.
- Physical computer monitor size.

Readability of characters on the screen also depends on font style. Microsoft®'s Tahoma font is a good choice for clarity.

Windows Task Bar

The Windows Task Bar, usually found on the bottom of the computer screen, can be set so that it is normally hidden until the mouse pointer is moved over that location. This gives the total screen to Blaise, removes a distraction, and allows more vertical space for the InfoPane and Blaise page to share.

2. Major Components of the Blaise Screen

The ten major components of a Blaise screen are illustrated in Figures 1 and 2. Of the major components, only four are required by the Blaise system. These are the Windows control buttons, the title bar, the menu, and the FormPane (page). The InfoPane is almost always used for question text and the answer list is almost always used for response choices. Another often-used component is the status bar. The other components are discretionary, and their use, or non-use, can be defined as part of the general specifications. For some long question texts, or questions with a large answer list, there can be tension between the need for space in the InfoPane and the desire to increase data density in the page. For this reason, the speed bar, the tabsheets, and the answer info area are sometimes eliminated to provide more vertical space for the InfoPane and page to share.

Sub-Components

Most of the major components have sub-components or other aspects that can be included, excluded, or configured. An annotated listing of all these possibilities with recommendations would take about 30 pages of space using small font size, and thus is the topic of another paper. To give one example, it is possible to include or exclude provided Blaise menu choices, or to add your own menu choices. For now, the best ways to learn about configuration possibilities for each major component is to study the Blaise Developer's Guide and to explore the dialogs of the Mode Library Editor and the Menu Editor.

3. Design and Specification of the Page

The design and specification of the page (the Blaise FormPane) is often neglected but is just as important as the design of the InfoPane. In some surveys, it is more important. The Blaise page is analogous to a page in a paper questionnaire. It organizes and displays related data elements together. There are several features you can specify in the page that enhance the presentation for the interviewer. The page enhancements illustrated in Figure 1 include specification of readable field

descriptions to identify each field, labels that group related questions, use of two columns in the page with eight data entry cells in each column, and size 10 font (for an 800 x 600 display). An important new feature of Blaise for Windows is the use of the field description in the page as opposed to the more traditional field name. The field description gives more flexibility, since you can include spaces in the description. It also can be multilingual. You can also use the field description in the edit jump dialog.

Thirteen Methods of Navigation

The term **navigation** refers to movement through the questionnaire. The most common method of navigation in Blaise is standard normal movement from question-to-question throughout the interview. This normal movement is governed by rules in the data model, and can accommodate complex skip patterns. However, an interview can be interrupted in many unanticipated ways, including when an edit is invoked, the respondent changes his mind, or there is a break-off. When this happens, the interviewer often has to navigate to other parts of the questionnaire. There are twelve other ways in which an interviewer can navigate through a Blaise instrument in order to accommodate these situations. These are described in Appendix A.

On those occasions when it is necessary to engage in ad-hoc navigation, well-labeled and organized pages with high data density enable the interviewer to go where needed in order to make corrections. In general, high data density is desirable. By increasing data density, an instrument can be presented in fewer pages. This allows the interviewer to form an effective cognitive map of the entire questionnaire. The best way to handle ad-hoc navigation is to give the interviewer as much information as needed to go where necessary. With high data density and readable field descriptions in a page, the arrow keys are very useful for navigating to another field within that page (short-range navigation). The Page Up and Page Down keys are very useful for mid-range navigation between pages, especially with high data density accompanied by frequent and descriptive labels in each page as illustrated in Figures 1, 2 and 3.

Specification of Groups of Questions (Item Types)

Given the page-based nature of the Blaise display, virtually every field (question) in Blaise can be considered to be part of an overall group of questions (also known as an *item type*). For example, in Figure 1, the individual fields *Street address*, *Apartment no*, *Locality*, *State*, *Zip code*, and *Phone number* can be considered to be part of an overall address information group or type. Similarly, the fields *Building type*, *Spec building*, and *Kind of locality* can be thought of as part of an overall building information group or type. Other kinds of multi-question types are optimally handled the same way in Blaise; that is by including all related fields in the same page, giving them appropriate names, and by providing effective labels for all question groups. Some of these other possibilities include quantity/unit combinations and series of similar questions.

Font Size for the Blaise Page

The size of Blaise page elements, such as size of the field description text, or the data entry cell, is based on a specified font size. The smaller the size of the font, the higher data density, you can have. It is quite workable to specify a slightly smaller font size for the page than for the InfoPane. In the examples used in this paper, the page font size is 10.

Field Panes and Columns

A **field pane** is an area where the field description, data entry cell, and other related elements reside within the Blaise page. You can choose from several different elements to make up a field pane including space for an enumerated label and a remark indicator. A column in the page is made up of field panes stacked up on one another. Two columns of field panes is normal in Blaise, though it is possible to have only one column (do not do this). In one major project, interviewers chose to have three columns of field panes because their complex survey requires a great deal of ad hoc navigation and the greater data density facilitates that.

4. Design and Specification of the InfoPane

General layout standards of the InfoPane should be specified. These standards concern the question text display and related components. There are options for font types, font sizes, and whether context information should be displayed. The idea is to provide the interviewer with necessary information without making the InfoPane too cluttered or difficult to understand. The question text should stand out from auxiliary text. The text that is displayed includes not only the question itself and context header, but also instructions to the interviewer, lists of includes and excludes, indication that help is available, or tabular display of previously collected data. It is possible to define a hidden frame that helps separate the question text from the upper and left margins of the InfoPane.

Font Size and Font Style Choice

A good font size for question text (for 800 x 600 resolution) is 11, assuming decent monitor size. You can get more information in the same InfoPane than with size 12 font, without compromising on visibility. You should consider bolded text

to be the standard for question text, as this can help in outdoor situations where there is bright sunshine. A good font style choice is Tahoma, a special font designed by Microsoft® for readable display.

VI. Testing the Questionnaire Program

In theory, it should be possible to understand every aspect of how the questionnaire works on screen – its routing, fills, tables, appearance, etc. – from a thorough scrutiny of the source code. But in reality, this is not a feasible option, and you need to run the executable program and create questionnaires to check that everything is works as it should.

1. Office Testing

The most common method in the early stages, once the programmer has written the source code for the first time, is office testing, where the researcher(s) work through the questionnaire, creating different types of household and respondent scenarios in order to put the program through its paces. The researcher keeps written notes (this is best done concurrently with testing, using Microsoft® Word) and the programmer uses these to make further changes.

It can be useful, even crucial, to ask the sponsor to also provide scenarios for testing, as they may have a good idea of the types of respondents / households are of particular interest for the research study.

Testing can proceed in stages by use of mini-data models for development.

2. Interviewer Testing in the Office

Once the questionnaire is functioning as a complete instrument, it can be invaluable to bring one or two experienced interviewers into the office for a day or two of further testing. One option is for the interviewers to work on their own, going through the questionnaires using different respondent/household scenarios provided by the researcher, and keeping note of problems. (See Newman and Stegehuis, (2000) for discussion of automated error reporting and error tracking.)

A better method is to have them conduct mock interviews with the researcher as interviewee. These will be stop-start affairs, as problems are discovered and noted, and solutions are discussed.

A further improvement is for the researcher to act purely as an observer of the interview, with another researcher or other member of staff acting as the interviewee, giving answers according to household / respondent scenarios sketched by the researcher, and improvising thereafter. Being simply an observer and listening in on the interview can reveal new flaws and ambiguities in questions, and problems with question sequences, that have not been seen before. (Many researchers have had the experience of only realizing a problem exists when hearing a question spoken aloud by an interviewer during the pilot briefing.)

The fresh perspective offered by interviewer involvement will reveal new problems and issues not found in earlier testing. The researcher must keep note of all problems, and (either then or later) write a specification for the programmer detailing what changes are needed.

It might also be possible at this stage to bring some 'real' respondents in to the office and test the questionnaire on them, with the researcher observing and making notes. However, this tends not to happen, in part because of the practical problems of recruitment at short notice, but also because of the stop-start nature of the process.

3. Piloting

The field pilot is the traditional method for testing the survey process, including the questionnaire itself. It is particularly valuable in bringing in 'real' respondents, who are in situations not always envisaged by researchers during testing.

Interviewers should be encouraged to make notes of problems during their interviews. This might be done using the Blaise Notepad facility, or on paper. Keeping comprehensive notes in this way is not easy, and the interviewer should allow five to ten minutes to go over their notes after each interview, fleshing them out as necessary.

Researchers and sponsors should accompany interviewers for at least one day of work, if possible, and keep notes of any problems that occur.

The pilot comes towards the end of questionnaire development and is generally seen as the last chance to fine-tune the questionnaire before launch. Therefore pre-pilot testing must be as thorough as possible.

4. Amending the Program and Checking Changes

Testing and amending the questionnaire is usually very time-consuming. The reason is that it is not just a matter of trying to follow every possible course through the questions and correcting errors; it also involves sorting out problems and issues that were unforeseen when writing the original questionnaire draft, as well as making numerous small improvements.

Work Side by Side as Much as Possible; It Saves Time.

Note that this method would not be appropriate during the first stage of program writing, when the programmer has to build the source code from scratch, based on the paper draft. Here it would be a waste of resources for the researcher to be sitting alongside. Also, where a subsequent change involves major rebuilding work on a complex questionnaire structure, such as an event history or household grid, the programmer will probably prefer to do this work alone in the first instance.

However, much time can be saved if programmers and researchers work side by side when making changes to the source code, so that fixing the code is combined with testing the results. An efficient model is for the researcher to keep notes of problems found during testing, and to sit alongside the programmer while the source code is amended to fix the problems. Then, after each batch of changes, the programmer compiles a new version of the program and the researcher and programmer together run through it to test that the changes have been successful. If not, the source code is amended on the spot and the program recompiled and tested again.

This may seem *less* efficient than a division of labor into respective skill areas. But in fact it works much better than the alternative model, which is for the programmer to work alone, using the researcher's written notes to make changes to the source code, then supply executable programs for the researcher to test (alone). The disadvantage here is the strong likelihood of faulty communication and misunderstandings. Some common examples of this:

- The researcher's notes are ambiguous; the programmer interprets them in a different way than was intended. To discover and correct this requires a round of testing, a further written request, another amendment to the source code, an executable file to be supplied, and a further round of testing.
- Writing in haste, the researcher makes a minor error in specifying a change request. The mistake is realized when testing the new executable program, requiring a further written request, and so on (see above process).
- The programmer makes a simple mistake. This is picked up by the researcher in testing, who then writes another change request to correct it. (See above process).
- A researcher, having only an imperfect understanding of how Blaise works, spends some time writing an elaborate request for a change that is not technically possible or is better done in other ways.
- A researcher unwittingly requests a 'small' change that has significant knock-on effects elsewhere in the program and requires a lot of work to implement. If they had been aware of this, the request would have been modified or dropped.

VII. Special Topics

Instrument development in Blaise can be made relatively easy even for complex questionnaires, if you approach it correctly. This too-short section describes some approaches for instrument development.

1. Development of a Complex Data model Using a Simultaneous Bottom-Up and Top-Down Approach

Large or complex instruments can be developed efficiently using bottom-up and top-down approaches simultaneously. The bottom-up part of development refers to the use of mini-data models to develop sections of the questionnaire. The top-down part of development means programming at the instrument (or integration) level right from the start, using stubs to represent the various sections until they are complete. This dual approach allows work to proceed on easier block-level tasks by junior programmers, while a more senior programmer takes care of overall instrument integration (where the complexity often lies) at the same time. It allows for quicker development, since it is much faster to prepare a smaller data model than a larger one. This allows for more and quicker iterations in a given amount of time.

Mini-Data Models

Since Blaise III (the last DOS version of Blaise), where constructs known as **parameters** were introduced, it has been possible to develop blocks for complex data models by splitting the data model into mini-data models. Often this is done for section-level blocks or other major sections of the instrument. These blocks may themselves have sub-blocks that are not constructed in a mini-data model.

A mini-data model is one that contains the block to be developed and just the fields necessary to give the block needed information. Values from fields outside the block are assigned through parameters to the block under development. After the block is developed, it is inserted into the main data model. The only elements that are necessary to change are the parameter references.

Integrated Instrument

At the same time developers are working with mini-data models, an integration programmer can be working with the main instrument. This can be put together early in the project development with stubs for the blocks that are being developed through the mini-data models. As the blocks are finished, they are inserted into the main data model.

2. Prototyping Challenging Aspects of an Instrument

New types of surveys that are brought into Blaise for the first time quite often have special requirements. Some examples include the need for extreme ad-hoc navigation, many-to-many data relationships, complex data links between parts of the instrument, or accommodating unusual displays and navigation within a table. A good way to come up with solutions for challenging instrumentation problems is through prototyping. It is useful to separate these more challenging issues from the more common programming tasks and assign the prototyping tasks to a two-person team consisting of a senior programmer and a subject-matter expert. Their solutions can be shared with and implemented by more junior programmers. The mini-data model approach to instrument development lends itself nicely to this kind of prototyping.

3. Two or More Spoken Languages

Blaise has the natural capability to handle two or more spoken languages such as English and French. The following illustrations demonstrate this.

The *Phone number* field definition from Figure 1 is shown in English and French in the following example:

```
PhoneNum "Phone number of the home."
           "Le numéro de téléphone de la maison."
         / "Phone number"  "Numéro de téléphone" :
            STRING[12], EMPTY, DK, RF
```

The field definition has two entries for the question text and two entries for the description. All other components of the field definition are declared once.

Languages are set up in the instrument by declaring languages that are going to be used, as shown in the following example:

```
LANGUAGES =
  ENG "English",
  FRA "Français"
```

The interviewer switches languages either through a menu or through a function key. If the field description is used on-screen in the page, the visible field identifier switches languages too, as does the description used in the edit jump boxes.

Specification and Implementation of a Second Language

The translation and execution of a second spoken language in a Blaise data model can be complex. In addition to the field elements above, which are easily handled, there are texts for edit statements (in the rules), computations for text fills, and text for response choices. All of these text locations must be found and translated. Testing of a second language can be very tedious and perhaps done by only one or a few people fluent in the second language. When implementing a second language, you must leave enough time to adequately program and test the translations.

VIII. Alternative Approaches

While this paper assumes that there are at lease two parties involved in producing the instrument, a specifier and a Blaise programmer, there are alternative approaches. These are summarized.

1. Subject-Matter Experts Program the Blaise Source Code

This model has been used successfully in some organizations around the world (e.g., Manners, 1998). The block-level source code is often quite easy to program and can be done by a subject matter specialist with a few hours of training. A harder task in

complex data models is to assemble the blocks together so that they work with each other well. A senior and experienced Blaise programmer can do high-level integration, and from this, guidelines can be given to the block-level programmer about how the block will integrate with the overall instrument (see Pierzchala and Manners, 1997).

An advantage to this approach is that the overall specification and development time is reduced, because the original specification is done directly in Blaise, because there is no communication process between specifier and developer, and because the person programming the block can immediately prepare it and test it and make repairs.

It is essential in this kind of implementation that the organization arrive at agreed-upon instrument development methodology, programming standards, interface standards, and have standard high-level code for constructs such as kinds of tables into which the subject matter specialist can insert blocks.

2. Iterating to a Solution

In this model, a project specifier is teamed with an experienced and senior Blaise programmer who also has extensive survey experience. While there may be detailed field specifications and survey protocols, the way in which the instrument is put together is arrived at through iteration. In short, the specifier may verbally, or through brief written descriptions, tell the Blaise programmer what they are trying to accomplish. After discussion, the programmer develops appropriate prototypes (at the block level, at the integration level, or both), and comes back. Together they discuss necessary alterations. This process repeats until an agreed-upon instrument is produced.

With the right people, this method can work extremely well to produce high quality instruments at low cost. It can avoid the rigidity that can accompany programming to a specification. Especially important are the qualifications of the Blaise programmer. That programmer must understand how Blaise works best for the interviewers and other users, how it can fit into an existing survey infrastructure, and how solutions in similar surveys have worked out. Above all, this person must be expertly knowledgeable with the internal workings of Blaise and how to program robustly. Otherwise, you might get a mess.

3. The Paper Questionnaire as a Specification

If there is a paper questionnaire that is to be implemented in Blaise, the paper questionnaire should be a very good, if incomplete, Blaise specification. A paper questionnaire is typically an understandable document with well-labeled sections and sub-sections. Often there is a survey operations manual and an interviewer manual that can provide additional information. A paper questionnaire can provide a good overview of how the whole instrument works together, and can be helpful in identifying identical or similar data collection structures and question types. It is also helpful to talk with experienced survey operations people and interviewers to discuss how they go about collecting and processing the data. This overview can be very helpful in the overall design of the Blaise instrument.

Although the paper questionnaire can be a good starting point, is also an incomplete specification. It usually has question numbers and text, but it does not include SPSS or SAS variable names, field descriptions, valid values, or an indication whether a field can be EMPTY or can allow don't know or refusal. You can hand-annotate a paper questionnaire, noting Blaise blocks, question types, and similar concerns directly on the paper, or mark a reference on the paper that points to additional information in another document. This supplemental document can also provide information about text fills, edit checks, and other information necessary for an electronic instrument.

It is also extremely instructive to work with experienced interviewers, watch them conduct real or mock interviews with the paper instrument, and generally include them in the development and specification process.

4. Code Generation

It is possible to write a metadata collection instrument in Blaise or in a database system where a project person can state specifications. An auxiliary program, such as Manipula for Blaise or Visual Basic on another database system, can produce Blaise source code especially at the block level. Blaise fields are easiest to specify, while the within-block rules can be more difficult to handle.

The advantage of such an approach is that it eliminates much of the mundane specification and programming work associated with producing an instrument. There can be tradeoffs, however. For example, a programmer may make changes in the questionnaire source code after it is generated. If this source code were not used to repopulate the specification database, then the source code and the metadata database would be out of sync. It is not difficult to produce a parser to read field definitions back into such a database. It is much more difficult to parse the rules to repopulate the metadata database (since Blaise is a powerful fourth-generation language, it is possible to program rules in a way that metadata database does not anticipate, also the rules can be extremely complex). It is possible to produce a metadata database system that generates fields in a source code file that is separate from the rules, thus separating the issues associated with field metadata from the rules metadata.

The point of view of the metadata database and generation is extremely important. Many such systems have an item-based point of view that does not fully recognize block structure and type possibilities in Blaise. A better point of view for such a metadata system is a structure-based point of view that formally acknowledges the blocking structure of a Blaise instrument and can drastically reduce specification and development work. The National Agricultural Statistics Service in the U.S. uses such a point of view in a Blaise metadata database and generates over 40 complete questionnaires every quarter (Pierzchala, 1993, Schou and Pierzchala, 1993) in a highly specialized context.

The block-based point of view is illustrated in Appendix C, in the discussion of blocks and field names. In that example, if a field name were *MaizePAcres* this would be a specification with an item-based point of view. If the block were named *Maize* (or *Soybeans)* and the elementary field name *PAcres*, this is a block-based point of view, and this is what you should do.

IX. References

Farrant, G. and Thompson, K. (1998). How to write a Blaise-friendly questionnaire. Internal working paper, National Centre for Social Research, London.

Frey, R (2000). Developing a Blaise Instrument for the Spanish Bladder Cancer Survey. Proceedings of the Sixth International Blaise Users Meeting, Kinsale, Ireland, CSO Ireland.

Manners, T. (1998). Using Blaise in a Survey Organization Where the Researchers Write the Blaise Datamdoels. Proceedings of the Fifth International Blaise Users Conference, Statistics Norway, p. 125-138.

Newman, S. and Stegehuis, P (2000). Configuration Management and Advanced Testing Methods for Large, Complex Blaise Instruments. Proceedings of the Sixth International Blaise Users Meeting, Kinsale, Ireland, CSO Ireland.

Pierzchala, M. and Manners, T. (1997). Producing CAI Instruments for a Program of Surveys. In Computer Assisted Survey Information Collection, Wiley Series in Probability and Statistics, Couper et. al. (Eds).

Pierzchala, M. (1992). Generating Multiple Versions of Questionnaires. In Essays on Blaise. Proceedings of the First International Blaise Users Meeting, Voorburg, The Netherlands, CBS, pp. 131-145.

Schou, R. and Pierzchala, M. (1993). Standard Multi-Survey Shells in NASS. In Essays on Blaise 1993. Proceedings of the Second International Blaise Users Conference, London: OPCS, pp. 133-142.

Appendix A: Navigation in Blaise

Navigation is powerful in Blaise, and the layout of the page can enhance or detract from this power. This appendix lists the major navigation methods in Blaise, how they are used, and how they can be enhanced.

Navigation facility	Use	Programmer Enhancement
Normal forward movement	Forward movement during interviewing is governed by RULES specifications in the instrument.	High data density in the page allows the interviewer to understand the flow and content of the instrument.
Edit jump list	Lists fields in the instrument that the interviewer can jump to in order to fix edit failures.	Use readable field names or field descriptions to represent fields in the edit jump list.
Paging with <Page Up> and <Page Down> keys	Medium range navigation through sections of an instrument. This is a major method of navigation. <Page Up> goes back one Blaise page (one FormPane). <Page Down> goes forward one Blaise page. You cannot page down past fields that have not yet been reached in the interview.	Enhanced with forms based approach: • Dense data display (important), and • Labels at the start of each column or set of fields. • NEWCOLUMN or NEWPAGE at the start of a section.
Arrow keys	Short range navigation within a page (FormPane). Arrow keys move one cell at a time	Enhanced with dense data display, column labels, and readable field names or descriptions.
Home key	Used to jump to the first field in the instrument.	Users often use this if they want to review the data in a form. Subsequent navigation is facilitated with paging down through well designed Blaise pages with high data density.
End key	Interviewing: Jumps to the next appropriate question, taking into account changes of route based on changed data. Editing: Jumps to the last question in the instrument.	In interviewing, <End> skips past questions with EMPTY attribute. Thus do not give an EMPTY attribute to questions you require. The End key is one of the most important navigation keys in Blaise.
Parallel block navigation: • Parallel block lister. • Parallel blocks on tab sheets. • Function keys to access selected parallel blocks. • Function key to return to main interview.	Breaks the linear progression of the interview, including appointments; non-response; concurrent household interviewing; or where sections are completed by different individuals.	Use descriptions for parallel blocks (define these in **Project>Data model Properties**.) Use conditions to determine when each parallel block is available. Tabsheets for parallel blocks are available with Blaise Version 4.1, as is the ability to assign a parallel block to a function key.
Jump box	Jumps to any field with a field tag. It is very effective for jumping to sections where a section label is supplied.	Use short field tags at the start of a section. For example, for *Section A*, use a field tag of (*a*). The use of dense FormPanes enhances the section review once the user has jumped to that section. This is especially used with paging and arrowing, as described above.
Edit lister for multiple edits in a form.	Allows an on-line review, where there are multiple edit failures in a form. You can jump to any field listed in the edit lister.	Use of meaningful name or description for the field name.

Navigation facility	Use	Programmer Enhancement
Remarks lister	Reviews all remarks made in the instrument. Can jump to the field associated with any remark.	Use of meaningful name or description for the field name.
Open field lister	Reviews all open fields in the instrument. Can jump to the field associated with an open.	Use of meaningful name or description for the field name.
Mouse click	Move to any eligible field or parallel block tab sheet.	Provide a mouse or other pointing device. Also, high data density in the page.

Appendix B: Example Specification, Household Enumeration Grid

This appendix illustrates a specification for a household enumeration table for a household survey. It gives the conventions used, states what is to come before the table, how the table is to operate, what each row of the table is to contain, and what follows the table.

1. Specification Conventions

Languages

The primary spoken language is English. A second spoken language is French. Translation from English to French will be done after the questionnaire is programmed. For now, an indication to include French question text and French description text is included in the field specifications below.

For some questions, help text is available through WinHelp. This is indicated in a field specification with the HLP language indicator. The text specified for HLP is a link to the WinHelp file. The client will provide the WinHelp file and will use the links specified below.

Pre-defined Types

Some question response definitions are used commonly throughout the instrument. The following are used in the field specification section. French translations will be provided later.

TYesNo = (YES, NO)
TContinue = (Continue "[INT] PRESS 1 TO CONTINUE")
TGender = (MALE, FEMALE)

Blaise Page Labels

For pages of columnar display, Blaise page labels are specified. An example:

{Blaise page label}
"Household introduction"

Interviewer Instructions

Interviewer instructions are marked by all uppercase text and are preceded by an indicator as demonstrated.

[INT] ENTER 999 TO INDICATE THAT THERE ARE NO MORE MEMBERS.

Fills

The following fills are used. The ^ indicates that a text fill is specified in the question text.

Fill	Options / examples	How
^your_SP	[your] [John's]	When talking directly to the subject, use **your**. Otherwise use the first name of the subject and add **'s**.
^you_SP	[you] [John]	When talking directly to the subject, use **you**. Otherwise use the first name of the subject.
^Do_Does	[Do] [Does]	At the start of a sentence, when talking directly to the subject, use **Do**. Otherwise use **Does**.
^do_does_ {note the trailing _}	[do] [does]	Not at the start of a sentence, when talking directly to the subject, use **do**. Otherwise use **does**.
^Is_Are	[is] [are]	If talking to the subject, or about more than one person, use **are** otherwise use **is**.

"What is ^your_SP job title?

Text Enhancements

Text enhancements are given by the @A (etc.) in the question text. You can have 26 defined enhancements. These can be used for bolding, underscore, font style, font size, tab stops, and so on. The following Text enhancements are used. The @A (etc.) is defined for types of text. What @A (etc.) represent on the computer screen is determined by a configuration file.

Text enhancement symbol	Applied to	Appearance in this survey
Default (i.e., no enhancement)	All question text not given any other text enhancement.	Tahoma, Normal, 11 point, black
@B	Emphasized text	Tahoma, Bold, 11 point, black
@I	Interviewers instructions	Tahoma, Normal, 10 point, blue
@/	Line feed in the info pane	

The specification question text for the following:

"Have you remembered to include @Ball babies and small children@B?"

becomes

"Have you remembered to include **all babies and small children**?"

on the screen.

Sample Field Specification

```
{IF Job = Yes}
NumJobs
"How many jobs ^do_does_ ^you_Sp have?"
FRA ""
HLP "NumJobsDef"
/
"Number of jobs"
FRA ""
: 0..5
```

Entities

Database entities are given at the beginning of each specification section below. The following database entities are valid for this instrument.

Entity indicator	Entity meaning
HH	Household relational database table
PER	Person relational database table
EMP	Employer relational database table
LOC	Location relational database table
DAY	Day relational database table
TRIP	Trip relational database table

Spacing indicators

The following are spacing indicators of various types in the Blaise page.

Spacing indicator	Meaning
DUMMY	Space in the column in the Blaise page
NEWCOLUMN	Puts next field at the top of the next column of the Blaise page. If already in the last column of the page, it will put the column at the start of the next Blaise page.
NEWPAGE	Puts the next field at the start of the next Blaise page.

DUMMY, NEWCOLUMN, and NEWPAGE are given just before the field they apply to in braces.

```
{NEWCOLUMN}
{Blaise page label}
"Household introduction"
```

Flow Specification

Flow or routing instructions are given just before or just after a field in braces.

```
{IF FName <> EMPTY and FName <> 999}

{ENDIF}
```

Hard and Soft Edit Specification

Hard and soft edit instructions are given just before or just after a field in braces.

```
{HARD CHECK:
NumJobs = 0 implies Job = Yes.
"[E HH01] I recorded that you have a job and that the job number is 0. Which is correct?"
FRA"[E HH01]"}
```

Computation Specification

Computation specifications are given just before or after the field, in braces.

```
{COMPUTE number of household members from the entries in the enumeration table.}
```

2. Household Enumeration Section

In this section we describe how to specify a table in Blaise. The example, a household enumeration grid, is something that is commonly used in household surveys. There are many ways this can be done. The screen shot in Figure 3 at the start of this paper is from the table produced from the following specification.

General

The sampling algorithm, and by extension the validity of the eventual statistical analyses, depends upon a complete and accurate listing of household members. Household membership depends on criteria that have evolved over several decades based on experience in this nation and others. These criteria are not necessarily known by the respondent. Current

methodological practice indicates that the best way to proceed is to have the respondent give information about each person he/she believes is a member of the household, and after that list is complete, ask follow-up questions. This helps to make sure that the list is complete by the criteria of the survey. If someone is missing, then the interviewer is to return to the household enumeration grid and add the missing person or persons. Only when all criteria are covered, is the interviewer allowed to signal that the enumeration is complete. At that time the sampling algorithm is executed.

Experience shows that the best way to collect household data is within a grid display with freedom of movement. Each member is listed in a row of the grid. The interviewer can see all information about all members of the household.

Note that all information collected in the enumeration table is necessary for the proper working of the sampling algorithm that takes into account Age, Gender, and number of jobs. Therefore these data do not allow DK or RF. If the respondent does not know the information, or refuses to give proper information, then the interviewer is to break off the interview appropriately (using the break off block) without antagonizing the respondent. At a later time a re-visit will be attempted.

The respondent is allowed to give a fictitious first name and to refuse the last name. However, if there are two identical first names, such as 'John', then use the last name field to make them distinct. For example, John Jones Sr. and John Jones Jr. or alternatively, John 1 and John 2.

Before the Enumeration Table

Entity is HH

Introduce the household enumeration section to the respondent.

```
{NEWCOLUMN}
{Blaise page label}
"Household introduction"

HHIntro
"I am now going to ask you for information on all the members of the household.
@/
For each member I will ask name, age, gender and whether the person has a job, and if so, how many.
@/
This will ensure that the survey is representative of the whole community."
FRA ""
/
"Household Intro"
FRA ""
: TContinue
```

Maintain a count of household members from the enumeration table. Do not ask for this number. Do not limit the enumeration of members based on the number in this field.

```
{COMPUTE number of household members from the entries in the enumeration table.}
HHNumber
"Count of household members."
FRA ""
/
"Number in household"
FRA ""
: 0..25
```

In the Enumeration Table, Across Member Rows

Entity is HH.

Allow up to 25 people in the household.

The person speaking is to be the first row in the household enumeration table.

There must be at least one person in the household. If there are no valid members in the household, the interviewer should exit the form through the break off parallel block.

The respondent/interviewer can give the information about household members in any order. For example, data entry can be in horizontal order for one person at a time, or it can be vertical across people. In the latter, the respondent may wish to give all first names first, surnames for all members, and so on, in that order.

To leave the household enumeration table, an indication must be given to the computer that it is finished. This is done with an entry of 999.

Since data entry order for this table is free, it is necessary to check that all required cells are filled in. Therefore as the interviewer leaves the table, check to make sure there aren't any holes in it. If there is a hole to fill in, the computer should give a way to arrive back directly at the cell in the table, with an appropriate error message to the interviewer.

Make sure that first name and surname, taken together, uniquely identify individuals. For example, John Jones Sr. and John Jones Jr. It is helpful for the proper conduct of the interview that enough information is gathered about the names in order to make it clear to the respondent who is being talked about.

It is possible that people are listed that possibly should not be listed. By deleting the first name entry, the information in the row shall be deleted and all subsequent rows will move up one row. If the person is later determined to really be a member of the household, then data entry for that person will be redone (we don't want the respondent to think we're keeping data that are apparently deleted).

In the Enumeration Table, Within a Person Row

Entity is PER.

The following specifies the composition of, and the execution of the data collected for a person.

```
{Note, answer required, even if 999 to indicate that enumeration is finished.}
FirstName
"What is ^your_SP's first name?
@I[INT] ENTER 999 TO INDICATE THAT THERE ARE NO MORE MEMBERS."
FRA ""
/ "First name"
FRA ""
: STRING[20]

{IF FName <> EMPTY and FName <> 999}
SurName "What is your_^SP's surname (last name)?
FRA ""
/
"SurName"
FRA ""
: STRING[20], RF

Gender
"@IASK ONLY IF NOT OBVIOUS
What is ^your_SP's gender (sex)? @I "
FRA ""
/
"Gender"
FRA ""
: TGender
```

```
Age
"How old ^is_are ^you_SP?"
FRA ""
/
"Age"
FRA ""
: 0..120

Job
"^Do_Does ^you_SP have a job?"
FRA ""
HLP "JobDef"
/
"Have Job"
FRA ""
TYesNo

{IF Job = Yes}
NumJobs
"How many jobs ^do_does_ ^you_Sp have?"
FRA ""
HLP "NumJobsDef"
/
"Number of jobs"
FRA ""
: 0..5

{HARD CHECK:
NumJobs = 0 implies Job = Yes.
"[E HH01] I recorded that you have a job and that the number of jobs is 0. Which is correct?"
FRA"[E HH01]"

{ENDIF}
```

After the Enumeration Table

Entity is HH.

The following questions will be asked to ensure that no one has been missed. If any of the following are answered No, then move the interviewer back to the table, at the next appropriate row, in order to fill in the missing person(s).

```
{Blaise page label}
"Household verification"

RemChild
"Have you remembered to include @Ball babies and small children@B?"
FRA ""
/
"Remember children"
FRA ""
: TYesNo

RemBrdr
"Have you remembered to include @Bany boarders@B? "
FRA ""
/
"Remember boarders"
FRA ""
: TYesNo
```

RemPrnt
"Have you remembered to include @Bparents who may be living with you@B?"
FRA ""
/
"Remember parent"
: TYesNo

RemRel
"Have you remembered to include @Brelatives who are staying here@B?"
FRA ""
/
"Remember relatives"
FRA ""
: TYesNo

RemFrnd
"Have you remembered to include @Bfriends who are staying here@B?"
FRA ""
/
"Remember friends"
FRA ""
: TYesNo

When all questions above are answered Yes, then invoke the sample algorithm.

Once the sample algorithm is invoked, close off access to the household enumeration table and following questions. Leave the household introduction question available to the interviewer (it’s a landmark to them).

Appendix C: Modifying a Block for Related but Differing Topics

This appendix illustrates two ways of using the same block definition, but adapting it for different topics. The two methods are compared and some guidelines are given on when to use which. Other methods of efficiently specifying related blocks are also given.

The most important point to remember about blocks and structure point-of-view is that it pays to separate within-block concepts from between-block differences. For example, if you specify crop fields as *MaizePAcres*, *SoybeansPAcres*, and so on, you redefine (and cause the programmer to reprogram) the same structure (but with different field names) several or many times. This latter practice, by itself, can increase the programming and maintenance burden by an order of magnitude or more in many instruments. In specification, it is crucial to recognize the parts of the code that can be generalized.

1. First Method, Standard Block with Variable Text and Edit Limits

To give an example from agriculture, one group of questions concerns crops. This example has questions *PAcres* (planted acres), *HAcres* (harvested acres), and *Yield*. The field *Production* is derived from *HAcres* and *Yield*. Its value will be shown on the screen for the interviewer's benefit, but it cannot be changed there. It could be hidden entirely.

Example

```
BLOCK BCrop
  PARAMETERS
    IMPORT
      UpperLimit : INTEGER
      Crop : STRING
  AUXFIELDS
    Label : STRING[20], EMPTY
  FIELDS
    PAcres "How many acres of ^Crop did you plant?"
                / "Planted acres" : 0..100000, RF
    HAcres "How many acres of ^Crop did you harvest?"
                / "Harvested acres": 0..100000, RF, DK
    Yield "What was your yield for ^Crop?"
               / "Crop yield" : 0..1000, DK, RF
    Production : 0..100000000, EMPTY
  RULES                          {Within block}
    Label := Crop
    Label.SHOW
    PAcres
    IF PAcres > 0 THEN
      HAcres
      CHECK
      HAcres <= PAcres
     "Harvested acres must be less than or equal to planted acres."
      IF HAcres > 0 THEN
        Yield
        IF (Yield > 0) AND (Yield = RESPONSE) THEN
          SIGNAL
          Yield < UpperLimit
          Production := Yield * HAcres
          Production.SHOW
        ENDIF
      ENDIF
    ENDIF
ENDBLOCK
```

Notice how this structure is the same for different crops; for example, maize and soybeans. This can be programmed in Blaise through block reuse by giving the block two (or more) different names as shown below.

Example

```
AUXFIELDS
  UpperLimit : 1..100000
  Crop : STRING[30]

FIELDS                    {Define block Fields in terms of one
  Maize : BCrop            block definition}
  Soybeans : BCrop

RULES
  NEWCOLUMN
  UpperLimit := 130
  Crop := 'Maize'
  Maize(UpperLimit, Crop) {Call the Maize instance of the crop
                           block and pass in a specific edit limit
                           and a custom text element.}

  NEWCOLUMN
  UpperLimit := 55
  Crop := 'Soybeans'
  Soybeans(UpperLimit, Crop)  {Do the same thing for soybeans.}
```

While the concepts of planted acres, harvested acres, yield, and so on are the same for both maize and soybeans, there are differences in the question text, value definitions, edit limits, and downstream variable names. In the example, the parameters *Crop* and *UpperLimit* customize the same block structure for the different crops. The soft edit using a parameter *UpperLimit* is defined in a general way in the block but its value is changed before each instance of the block is called in the rules.

Two instances of the block were declared using block field names, one for maize and one for soybeans. Thus at the block level, it is possible to state additional metadata that can be used in conjunction with field-level metadata within the block. For example, Cameleon can be programmed to concatenate *Maize* and *PAcres* into *MaizePAcres* and also *SoyBeans* and *PAcres* into *SoyBeansPAcres* to uniquely identify field elements to downstream processes without destroying the reusability of a block definition. This is another advanced use of Cameleon.

The block can be arrayed. In the block below there are 10 instances of the same block or structure.

Example

```
FIELDS
  Crop : ARRAY [1..10] OF BCrop
```

2. Second Method, Different Fields Sections, Same Rules Sections

Sometimes the method considered above is not powerful enough. This can be the case where, for reasons of metadata export, you need to have more specific field-level metadata available. On the other hand, you do not wish to reprogram the rules for different metadata identifiers. This section describes an alternative that allows you to be more specific with specification at the field level, while allowing you to program the rules only once. Consider how the example of Blaise could have been implemented using the Blaise INCLUDE statement.

```
BLOCK BCrop
  PARAMETERS
    IMPORT
      UpperLimit : INTEGER
    IMPORT
      Crop : STRING
  AUXFIELDS
    Label : STRING[20], EMPTY
INCLUDE "BCrop.Fld"            {Include the fields section.}
INCLUDE "BCrop.Rls             {Include the rules section.}
ENDBLOCK
```

The example below shows how the fields section can change, but the rules section (the hard part) can stay the same.

```
BLOCK BMaize
  PARAMETERS
    IMPORT
      UpperLimit : INTEGER
    IMPORT
      Crop : STRING
  AUXFIELDS
    Label : STRING[20], EMPTY
INCLUDE "BMaize.Fld"           {Include the fields section.}
INCLUDE "BCrop.Rls             {Include the rules section.}
ENDBLOCK
```

and

```
BLOCK BSoyBean
  PARAMETERS
    IMPORT
      UpperLimit : INTEGER
    IMPORT
      Crop : STRING
  AUXFIELDS
    Label : STRING[20], EMPTY
INCLUDE "BSoybean.Fld"           {Include the fields section.}
INCLUDE "BCrop.Rls               {Include the rules section.}
ENDBLOCK
```

Where BMaize.fld is the following:

```
FIELDS
  PAcres (101) "How many acres of maize did you plant?"
              / "Maize planted acres" : 0..100000, RF
  HAcres (102) "How many acres of maize did you harvest?"
             / "Maize harvested acres": 0..100000, RF, DK
  Yield (103) "What was your yield for maize?"
             / "Maize crop yield" : 0..300, DK, RF
  Production : 0..100000000, EMPTY
```

And where BSoybean.fld is the following:

```
FIELDS
  PAcres (201) "How many acres of soybeans did you plant?"
              / "Soybean planted acres" : 0..10000, RF
  HAcres (202) "How many acres of soybeans did you harvest?"
             / "Soybean harvested acres": 0..10000, RF, DK
  Yield (203) "What was your yield for soybeans?"
             / "Soybean crop yield" : 0..100, DK, RF
  Production : 0..100000000, EMPTY
```

Note that in both *BMaize.fld* and *BSoybean.fld* the *FieldName*s are the same but that the *FieldTag*, *FieldText*, *FieldDescription*, and *FieldValue* have all changed. For both field sections, the rules structure, as held in BCrop.Rls, is the same.

```
RULES                              {Within block}
  Label := Crop
  Label.SHOW
  PAcres
  IF PAcres > 0 THEN
    HAcres
    CHECK
    HAcres <= PAcres
   "Harvested acres must be less than or equal to planted acres."
    IF HAcres > 0 THEN
      Yield
      IF (Yield > 0) AND (Yield = RESPONSE) THEN
        SIGNAL
        Yield < UpperLimit
        Production := Yield * HAcres
        Production.SHOW
      ENDIF
    ENDIF
  ENDIF
```

Methods 1 and 2 are two different ways of accomplishing almost the same thing. In the first, the block is more generalized but there is less flexibility in handling of field-level metadata. In the second, the fields section is different between the two crops, but the rules section is the same. One point is that there is a big payoff if you can specify in generalized terms to reduce the amount of programming and maintenance. A second point is that there are many different ways you can be creative in Blaise in order to cut down on the amount of specification, programming, and maintenance.

3. Specifying Variations on a Theme

Methods 1 and 2 above show different ways of adapting the same block structure to different situations. However, there can be blocks with similar structure. It is possible to specify one block as a variation on an already-defined block. Consider the block definition of BCrop above. This has fields *PAcres*, *HAcres*, *Yield*, and *Production*. A similar but larger block would have the fields *PAcres*, *HAcres*, *Yield*, *Unit*, and *Production*, where *Unit* is needed for some crops because there are alternate units of production (e.g., for Maize, tons versus bushels). Instead of specifying the whole blasted thing from scratch a second time, the specification for the second block can be based on the first "insert *Unit* between *Yield* and *Production*" and then give additional rules specifications that determine how to calculate the value of *Production*, based on yield and the unit given.

4. Super Blocks

For the two kinds of crops mentioned in 3 above, the only difference was that a field *Unit* was specified for one of the block types. An alternative approach is to define one block type for all crops, but where the *Unit* field is not needed, just route around it, with appropriate specification for computation of *Production*. This 'super block' will be more complex internally than either alternative mentioned in 3 above. On the other hand, there is now less code to maintain. This method should not be used very often.

5. Specifying in Terms of a Prototype

When there is a new and difficult data collection challenge, the best approach may be to construct several prototypes to see what the possibilities are. This is an iterative process, but when you arrive at a solution, one method of specification is to say, "program the table for topic X just like prototype 5". This will give your documentation people conniption fits, but they can document from the prototype the same time the programmer is implementing your survey in Blaise.

6. Specifying Commonly Used Structures

It is possible that a measurement block can be used several or many times in a survey. For example, Frey (2000) discusses the construction of 6 'time-spent' blocks that between them are used several thousand times in a data model. Each TimeSpent block contains several fields that measure how long a person was doing something. If you have such a situation, it may be possible to specify in the following way:

```
Exhaust "Were you exposed to exhaust fumes on your job?" : TYesNo
{IF Yes}
{Topic = Exhaust, Phrase1 = 'exhaust fumes'}
{Call TimeSpent1}
```

This way you can avoid having to repeat the specification of many fields and can be as efficient in specification as the programmer can be in the source code. ***Caution***, this is such a powerful method of specification that it is easy for the specifier to define a much too large questionnaire without realizing it.

Appendix D: Data Export Possibilities

This appendix discusses the three options for data export, and advantages and disadvantages of each. It also mentions Open Blaise Architecture that will give another option or options soon.

1. ASCII Export

ASCII export is suitable for small or medium sized data files of less than 32,000 characters in width. It produces one rectangular flat file. The Manipula program that does the export is just a few lines of code that can be constructed in a few minutes with a Manipula wizard in the developer's environment. The ASCII file description can be easily generated with standard Cameleon setups for SAS or SPSS from the Blaise field-level metadata. On the other hand, this simplest and easiest export method will not work for large data files. If there are arrayed fields (or arrayed blocks of fields), then Cameleon must make the field names unique in the output description. It does this by truncating the field name to 5, 6, or 7 characters and adds a number to the end (such as *Phone1*, *Phone2*). If the Blaise data model is large, or if its structure is not suited for flat file export, then you'll need either to ASCIIRELATIONAL output or a custom output. If you use ASCII export for a highly structured data model, you'll likely get a sparsely filled in data set. (Note, starting with version 4.5 of Blaise, there will not be any limit to the ASCII record readout length. Version 4.5 has not been released at the time of writing this paper.)

A partial example of a downstream metadata description in SPSS for a small data model is given below.

```
TITLE       'NCS07'.
FILE HANDLE  Ncs07
             /NAME='Ncs07.ASC'
             /LRECL=14380.

DATA LIST    FILE = Ncs07 /
             Region              1 -     2
             Stratum             3 -     6
             SampleNu            7 -    10

. . .

             PostCode          596 -   605   (A)
             PhoneNum          606 -   617   (A)
             KindHome          618 -   618

VAR LABELS
             Region          'Region'
             Stratum         'Stratum'
             SampleNu        'Sample number'

. . .
             PostCode        'Zip code'
             PhoneNum        'Phone number'
             KindHome        'Building type'
. . .
VALUE LABELS
             KindHome        1 'Single family unit'
                             2 'Townhouse or duplex'
                             3 'Small apartment building'
                             4 'Large apartment building'
                             5 'Other kind of dwelling'/
```

2. ASCIIRELATIONAL Export

The Blaise ASCIIRELATIONAL export generates several or many ASCII files according to the pattern of *embedded* and *unembedded* blocks in the data model. The EMBEDDED keyword, when used in a block declaration, means that Blaise will store that block's data with those of its parent block. In ASCIIRELATIONAL export, every unembedded block's data will be exported in a separate ASCII data file. The Manipula program that does this is simple; the only difference between this setup and the one for ASCII export above is that the word ASCIIRELATIONAL is substituted for ASCII.

Generating metadata descriptions for each ASCIIRELATIONAL output file is more complex and more time consuming. The idea is that a sub-data model has to be constructed for each unembedded block. Cameleon can in fact produce these sub-data model definitions. Recent improvements to this process has automated this step (before, there was still considerable handwork that had to be done in order to prepare (that is, compile) each sub-data model). Once each sub-data model is prepared/compiled, then an SPSS or SAS Cameleon script has to be run on each. When that is done, then each output file will have a metadata description. As mentioned, this whole process can now be automated, but for a very large data model, the automated process can take well over an hour on a fast computer.

There are a few issues with ASCIIRELATIONAL export. There is a wide class of data models where ASCIIRELATIONAL export more or less approximates a true relational representation of the data in a relational database sense, and thus this method of export is suitable. However this is not always the case. In a relational database, there are *entities* and there is usually one database table per entity. In a transportation survey, entities may include *Household*, *Person*, *Work Place*, *Location*, *Day*, and *Trip*.

Let us consider the *Person* entity. It is possible, that for considerations of methodology and operability during data collection, that person-level data are collected in several blocks, each of which is unembedded (which is proper). In this example, you can have several export files that logically belong to one entity. In ASCIIRELATIONAL export for this data model, you have several different export files for this entity. A conclusion that you should draw is that the database structure that is suitable for data collection does not always have the same structure of the desired study database. When this is true, then the data model specification and programming should anticipate this difference.

Another issue with ASCIIRELATIONAL export is the way that linking is done internally in the Blaise proprietary database. Blaise uses downward pointers to link blocks, that is, the parent block points downwards to child blocks. This is not the way that relational databases do their linking. So there has to be a translation step if the data are to be imported into a true relational database. On the other hand, the method of Blaise linking may not make a difference if other downstream packages are used. One way to handle linking when exporting data to a relational structure, is to explicitly declare links between blocks within the data model, and use the data model's rules to assign appropriate (upward pointing) links to these fields within blocks. This is true for hierarchical as well as many-to-many data relationships.

3. Custom Export

A custom-built Manipula setup can be used when the structure of the Blaise data model does not correspond well to that of the output data files. This must be done on a block-by-block, or worse, on a field-by-field basis. For a data model of any size, the instructions to handle this translation can number in the hundreds or thousands of lines. Such a program can be developed by hand. This is the hard, prone to error, and is difficult to maintain. For example, if the data model definition changes, then someone must adapt the hand-programmed export setup as well.

A better way to produce the required custom export programs in Manipula is to let Cameleon generate them, based on metadata specification in the Blaise data model. This is what the National Agricultural Statistics Service does. As noted in section VII, 4 above, this organization generates over 40 CATI and Interactive Editing Blaise instruments every quarter based on a metadata specification. In addition to the generated instruments, the surrounding Manipula infrastructure for data import and export is also generated, using Cameleon, based on the generated data models. This is a high-level advanced technique, however it is worth considering where ASCIIRELATIONAL export does not suffice. See Pierzchala (1992) for details.

In order to enable this specialized technique, you have to consider how the field-level metadata should be stated, and someone has to write a generalized Cameleon setup that can interpret the Blaise metadata, in turn producing the required custom Manipula programs. If this is executed correctly, there are huge benefits. First, no one has to program huge Manipula setups. Second, if the data model changes, you just re-execute the Cameleon scripts to re-generate the Manipula programs and you're off and running. This is an example of letting the Blaise metadata drive the downstream processing. It is an enormously robust method of operating.

4. ASCII representation of Don't Know and Refusal

Where DK or refusal is a permitted response, Blaise extends the permitted range of answers to allow for them. So if you have five substantive answer categories (or a numeric range 1..5), Blaise will reserve code **8** for refusal and **9** for DK. If you have specified 9 as the top limit for the substantive answers, Blaise will reserve **98** and **99**. If you use, say, **1..120**, it will reserve **998** and **999**. So the DK and Refusal codes are always the final two digits of whatever range you specify. When exporting data, the Cameleon generated data descriptions always take into account extended field length due to DK and RF specification. For numeric questions, a common practice is to reduce the top number by 3 in order to avoid extended field lengths on output. For example, if 10,000 is an upper limit, then for all practical purposes, you can specify 9,997 as the upper limit. If it is possible that 9,997 will be exceeded, then it is possible that 10,000 would be too, and that the original specification is too low.

5. Open Blaise Architecture

A new capability called Open Blaise Architecture (OBA) will be released in version 4.5 sometime after the writing of this paper. OBA will have a new option or options for data export. You should check on these options when that version is released.