A New Tool for Visualizing Blaise Logic

Jason Ostergren, Rhonda Ash, The University of Michigan

Overview

The Health and Retirement Study (HRS) is a national longitudinal survey, administered biennially since 1992, on a variety of topics associated with aging and retirement. The HRS instrument is modified in the period between each wave both to improve existing sequences and to handle new circumstances, new research and new public policy, all while maintaining its longitudinal value. HRS has put considerable effort over the years into redesigning many sequences and sections scattered across the instrument. These redesigns require input from people at every level from the researchers to the programmers to the interviewers, with each group possessing different understandings of the workings of the instrument and what can or should change.

One section which has been the focus of multiple redesigns relates to employment and pensions. HRS produced the third major redesign of this sequence for its 2012 instrument. This occurred, somewhat unusually, in a series of face-to-face meetings between HRS programmers, testers, specification writers, and Co-Investigators. The meetings typically involved the use of a projector to demo relevant parts of the instrument and the review of printed specification documents. The redesign was more laborious than it needed to be in part because it was difficult for all parties to envision the effects of changes as they were being discussed.

Afterward, HRS began work on a tool to facilitate the redesign process in the future. The goal was to develop a visual representation of the instrument which could be useful in such a setting. The tool needed to allow easy-to-follow editing of the visualization which would help to preview the effect of a proposed change on flow and then could later be deciphered by programmers and turned into useable code. The result of this effort is a tool we refer to as “Visual Blaise,” which is the topic of this paper. Visual Blaise is a Windows desktop application that graphs the logic of a Blaise instrument one block at a time. It allows elements to be dragged and dropped, cut and pasted, and allows for the addition of new elements. Finally, it is capable of exporting a text file with the logic for each block written as a RULES section in correct Blaise syntax.

Problem Description

In the fall of 2011, HRS embarked on a months-long redesign of the portion of its instrument covering pensions. The pension section has a history of complex logic, spread out across multiple blocks and loops, interspersed with sections covering employment, job history, and other related topics. In fact, this section has seen rewrites and changes almost every wave since Blaise was adopted in 2002, but the changes for the 2012 wave were meant to be more fundamental. While the longitudinal value of the section obviously had to be preserved as much as possible, our Study Director wanted to completely change the structure of the data, consolidating all pension questions into one contiguous area of the instrument in a single array. In addition, he was willing to alter, drop or add content far more than would typically be allowed in a rewrite. Above all, the section needed to be made friendlier to respondents and interviewers, who had frequently gotten lost in prior designs, thus impeding the process of collecting high quality data.

The process of this major rewrite constitutes the backdrop for the development of the Visual Blaise tool discussed in this paper. One part of that process was simply deciding on and prototyping a Blaise program structure that would roughly match the known indispensable content and the desired data structure outcome. Another part was analyzing and merging similar content from separate pension sections into one. Yet another was designing a flow which was flexible enough to handle the fact that respondents are often unsure about key information regarding their pensions. However, the development staff could not make the decisions about what to cut, what to add, and how to handle the new logic independently. The development of the rewrite became an iterative process (unusual for HRS) centered on weekly meetings in which programmers, spec writers and researchers were present together. Typically, the programmers would demo the latest version of the section, explaining limitations, and showing what would happen to particular types of respondents. The researchers would discover problems with the design and suggest solutions, tinker with question text, codeframes and skips, and would also propose new sequences to handle certain kinds of respondents or pensions more optimally. Spec writers would attempt to record the ideas and fixes that were suggested and turn them into actionable changes for the next round.

As an example of this, the thorniest issues tended to revolve around the need to guide respondents into the correct sequences and to exclude them from other ones, all while providing escape hatches for respondents who get stuck in an inappropriate area. This is a uniquely difficult portion of the HRS interview precisely because respondents' understanding of their own pensions is often sketchy. This means that when a major flow-control question comes up, the design has to take into account that the respondent may answer incorrectly and only later discover their mistake due to nonsensical follow-ups. Getting the design right was thus a dance among the competing needs for logical clarity from the programmers to write reliable code, the need to allow respondents multiple points at which to correct their paths, and the need to avoid asking too many clarifying questions, among other things.

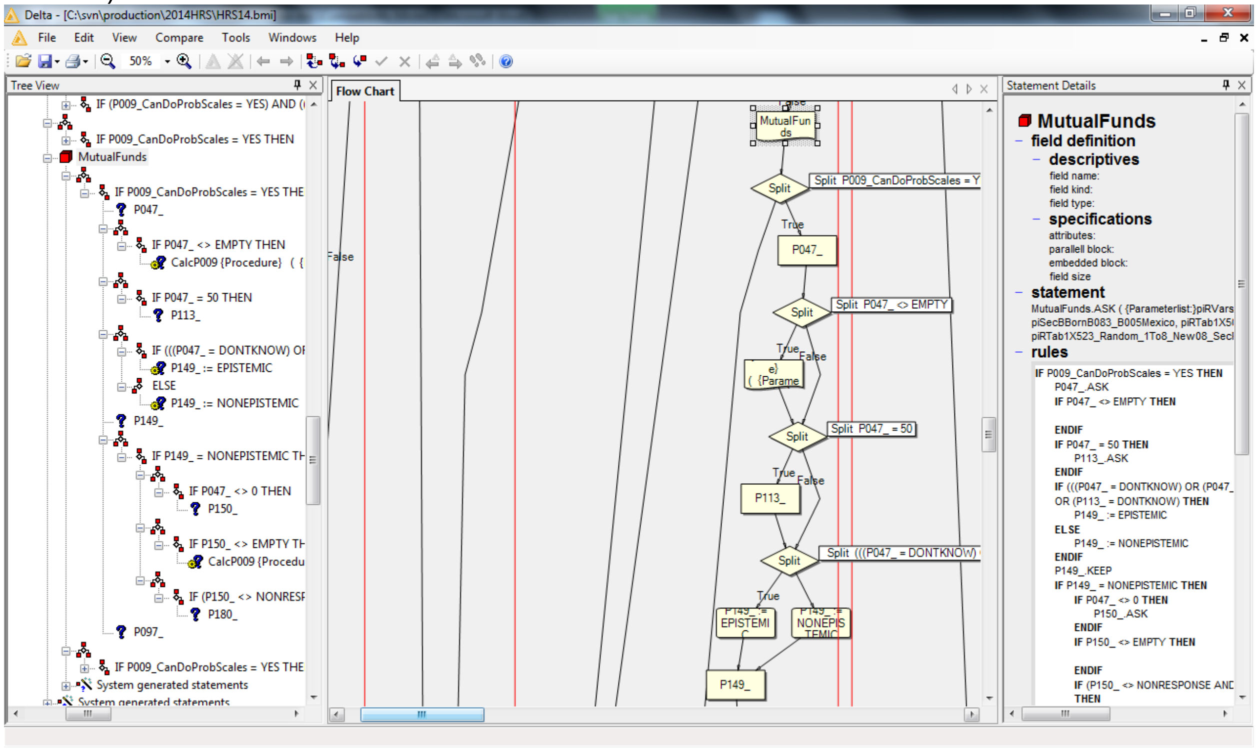
This unusual design process also highlighted a gap in our documentation abilities. While we could demo using the DEP and a projector and we could pass around questionnaire documentation, neither of these could really convey to everyone present how the sequences were bound together by the logic and what it would mean to move a sequence, for example, or add a new escape hatch question in the middle of a sequence. Existing flowchart tools like Delta have not gained favor at HRS, perhaps due to the unsuitability of the format for finding and understanding long complex sequences. We found ourselves making flowcharts by hand, which was time-consuming, or occasionally looking directly at code, which was unsuitable for some people present, or, most often, going with quick sketches on the whiteboard or nothing at all to look at when making decisions. This inability to discuss changes from a common understanding of the flow led to a lot of backtracking when discussions or demos in subsequent meetings revealed logical failures in the design.

After the end of the development period for the HRS 2012 instrument, permission was obtained to work on a prototype for a tool that could answer the need for some way of visualizing survey logic which would be accessible to people with different backgrounds. Additionally, we wanted to be able to manipulate the visualization, so as to make it possible to observe the effects of changes immediately, particularly when there were several pieces to handle at the same time. That prototype, which we call "Visual Blaise" (a temporary name that may stick due to inertia), will be described in detail below.

Solution Concept

There are a few stylistic and organizational things about the way the flow chart in Visual Blaise operates which are intended to be different from typical flow charts and other Blaise tools such as Delta. We do not have a deep understanding of the history or classification of flow charts, so there is no grand theory behind the style of the diagrams used in this tool, only our observations about what ways of thinking about flow seemed to speak best to HRS staff.

Figure 1: a portion of a block, containing the HRS questions “P047_” and “P113_” as shown in the Delta tool (note for comparison: the “P149_” variable shown at the bottom is actually a keep statement).



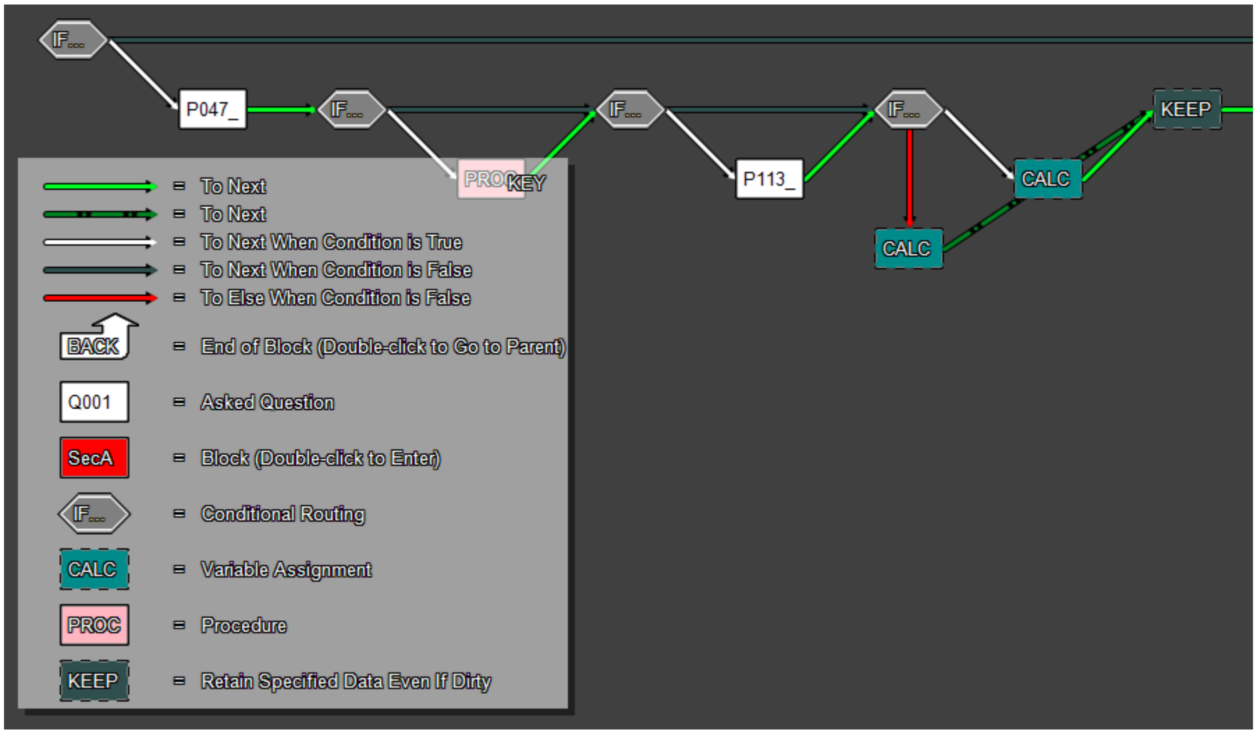
```
IF (P009_CanDoProbScales = YES) AND (
IF P009_CanDoProbScales = YES THEN
MutualFunds
  IF P009_CanDoProbScales = YES THE
    P047_
      IF P047_ <> EMPTY THEN
        CalcP009 {Procedure}  ( {
      IF P047_ = 50 THEN
        P113_
      IF (((P047_ = DONTKNOW) OF
        P149_ := EPISTEMIC
      ELSE
        P149_ := NONEPISTEMIC
    P149_
      IF P149_ = NONEPISTEMIC TH
          IF P047_ <> 0 THEN
            P150_
          IF P150_ <> EMPTY TH
            CalcP009 {Procedu
          IF (P150_ <> NONRESF
            P180_
    P097_
  IF P009_CanDoProbScales = YES THE
System generated statements
System generated statements
```

```
MutualFunds
- field definition
  - descriptives
      field name:
      field kind:
      field type:
  - specifications
      attributes:
      parallell block:
      embedded block:
      field size
- statement
  MutualFunds.ASK ( {Parameterlist:}piRVars
  piSecBBornB083_B005Mexico, piRTab1X5
  piRTab1X523_Random_1To8_New08_Sec
- rules
  IF P009_CanDoProbScales = YES THEN
     P047_.ASK
     IF P047_ <> EMPTY THEN

     ENDIF
     IF P047_ = 50 THEN
        P113_.ASK
     ENDIF
     IF (((P047_ = DONTKNOW) OR (P047_
     OR (P113_ = DONTKNOW) THEN
        P149_ := EPISTEMIC
     ELSE
        P149_ := NONEPISTEMIC
     ENDIF
     P149_.KEEP
     IF P149_ = NONEPISTEMIC THEN
        IF P047_ <> 0 THEN
           P150_.ASK
        ENDIF
        IF P150_ <> EMPTY THEN

        ENDIF
        IF (P150_ <> NONRESPONSE AND
        THEN
```

Figure 2: the same logic, as shown in the “Visual Blaise” tool.


IF...
P047_
IF...
PROC KEY
IF...
P113_
IF...
CALC
CALC
KEEP
= To Next
= To Next
= To Next When Condition is True
= To Next When Condition is False
= To Else When Condition is False
BACK = End of Block (Double-click to Go to Parent)
Q001 = Asked Question
SecA = Block (Double-click to Enter)
IF... = Conditional Routing
CALC = Variable Assignment
PROC = Procedure
KEEP = Retain Specified Data Even If Dirty

The first significant stylistic difference was to make the chart horizontal with the shallowest layer of logic always at the top. Typical flow charts (and Delta) begin at the top and flow downward, expanding to the left and right, with “True” arrows sometimes angling rightward and other times angling left. The reason for making the chart horizontal is mainly about the screen real estate available on widescreen monitors and desks with multiple monitors (which are common at HRS). That is, a typical monitor setup can display much more logic horizontally than vertically. Perhaps the opposite is true if a paper printout is the goal, but we were more interested in a group looking at a screen in a meeting room (and, in any case, paper would not allow dynamic changes, so it couldn’t be the main view). The vertical ordering of the flow chart according to logical depth matters, in part, because one question that seems to float to the top of design discussions at HRS regularly is which questions are asked of (almost) everyone. When splits only angle in one direction and are consistent about which direction “True” and “False” arrows go, it is possible to glance across the top and instantly survey what are the common questions.

Another ambition driving this project was to use colors and motion to improve understanding of the diagram. The arrows are color coded and dashed, if necessary, to help identify their role (bright white coloring for “True” arrows, red for “Else” arrows and gray/green for “False” arrows). It is possible to click and drag a statement icon around (on release, it snaps back into place) for purposes of clarifying its connections in particularly busy segments. The way Visual Blaise handles “Else” is significant also. While “True” arrows always point diagonally downward and to the right and “False” arrows always point horizontal or upwards and to the right, “Else” arrows are distinguished by pointing directly downwards. This helps to keep the diagram compact, but it also helps in situations where there is a series of “ElseIfs” by visually setting that construction apart from the normal flow.

The last significant difference in this visualization was to break it up by blocks. As was suggested above, part of the reason that few at HRS have been interested in Delta may be that the navigation of the instrument (to find the sequences of interest) is done via the statement tree control. The statement tree control is very difficult to navigate for HRS, because there are many layers of difficult to follow conditions above the main blocks, making it hard to unearth the major pieces of the block hierarchy. Of course there is a find feature, but it seems like the inability to just "see" the main blocks scares off many users.

What this means in Visual Blaise is that the visualization features the logic for only one block at a time, and a user must navigate to child blocks or parent blocks manually (via mouse, keys, or menu, as will be described later). At first this seems like a limitation, but since the logic of a block is largely independent of its parent or child blocks, it doesn't hinder understanding too badly. More importantly, it limits the "canvas" a user is looking at to a manageable amount, free from artifacts of parent block logic which tend to be decreasingly relevant the farther removed they are (HRS has a proliferation of testing conditions and gate conditions at the top level which clutter our variable universes and contribute to the aforementioned difficulties "seeing" main blocks in the statement tree control).

Object Structure

This paper will not go into great detail about how Visual Blaise is programmed, except for two topics that relate directly to handling Blaise logic: importing the metadata and the internal object structure. This is helpful for understanding how editing works in this tool, which is described in the next section.

The first topic, importing, has to do with the fact that Visual Blaise does not use the Blaise API at all. Instead, it makes use of the xml file produced by HRS' BlaiseRules tool described in the IBUC 2010 proceedings.[7] That tool makes use of the RulesNavigator in the Blaise API to extract all the logic and metadata from a specified compiled Blaise file (.bmi) and outputs it in an xml format, organized hierarchically by logic. Visual Blaise reads this xml file and converts it into an object structure in memory which it can manipulate in order to generate flow charts. It is important to note that the BlaiseRules tool is now incorporated into MQDS (as of April 2012), and the xml output discussed here is actually produced and saved as a byproduct of running MQDS, so that anyone who has installed MQDS can make use of it.

One other point to make about importing is that Visual Blaise has an option to import an abridged version of the instrument, which ignores things like keep statements and assignments, in order to focus attention on asked questions. It retains any conditions necessary to reach the asked questions, but it ignores any conditions which only lead to discarded statements. Visual Blaise permanently flags this version as abridged, and it is not possible to switch back to the full version. However, it seems quite useful for some people, since the reduced amount of logic makes it easier to understand.

While building this software, the notion was to develop (though what it was called wasn't investigated until it came time to write this paragraph) something like a "node graph architecture" for the core object structure and visualization. This would allow for combining the visual advantages of a flow chart with the drag and drop editing capabilities that might be found in a tree view. Additionally, it was imagined as being coded like a linked list, so as to allow easy insertion and removal of elements.

Under the hood, that meant that as the metadata was imported (as described above) it was converted into statement objects, which would become the nodes in the graph. Each Statement object contained a minimum of one and a maximum of two object references. Most statements have only a "next statement" reference, kind of like a linked list. Condition and loop (for..do) statements, however, have two. Typically this would be a "True" statement which represents where the program flows if the statement evaluates true and a "next statement" for when it evaluates false. However, where there is an else

[7] IBUC 2010 Conference Proceedings, pp. 46-53.

(including elseif) involved, there is no real “next statement” (actually it is present, but for cosmetic purposes) and instead there is a reference to an “Else” statement. With this set of nodes and links, the logic of any block can be described. After that, it is not terribly complicated to display them graphically.

As for the block hierarchy, as the metadata is imported, Visual Blaise keeps a running collection of these linked statement collections for each block. This master collection is keyed so that each set of statements can be matched with their corresponding parent block statement, so as to link the entire datamodel together.

Editing Visually

A big part of the goal of this tool is to allow editing of the logic within the flow chart. This would make it possible to experiment with changes in a way that is more accessible to non-programmers, with the idea being that the results could be saved for later use by the programmer who would turn them into Blaise code. To that end, the ability to drag and drop or cut and paste statements in the flow chart was incorporated.

The drag and drop capability is limited to a single statement. If a selected statement is dragged over another and dropped, Visual Blaise will insert the dragged statement before the statement it is dropped on. The cut and paste capability operates similarly, via right-click pop-up menu when a statement is selected. The “cut” statement is inserted before the statement where “paste” is selected. Cut and paste functionality has the added advantage of being able to move the statement to another block.

The cut and paste feature can also operate on multiple statements, which is useful for moving whole sequences. Any sequence which can be selected can be cut and pasted in the same fashion as a single statement. Selection of multiple statements is possible by selecting a starting (leftmost) statement and then holding down the Shift key and pressing arrow keys until the desired sequence is highlighted, similar to highlighting text via keyboard in a word processor. However, the way selection occurs is limited to prevent nonsensical segments from being operated upon. What this means is that when a user proceeds beyond a split (condition or loop) with the selection, the entirety of the loop or condition, including everything within it, is brought into the selection as a whole.   If that is more than the user intends to move, then the user would have to select the relevant subsidiary parts and move them one at a time. There were potentially other ways to handle this multi-select functionality, but we opted for the one which seemed most likely to prevent the user from unintentionally breaking the structure of nested conditions.

Finally, Visual Blaise allows other operations to be performed on selected or multi-selected statements such as deletions or inserts of new statements.

Output

Once changes to logic have been specified, they can be saved in Visual Blaise format and retrieved later. It is assumed that ultimately a programmer will look at the changes and implement them in the Blaise editor. There are a few ways that a programmer could approach this. One would be to work directly from the Visual Blaise flow chart on screen or by printing a copy of the relevant blocks, if appropriate. Another would be to work from a log of changes produced by Visual Blaise while edits are taking place. Finally, Visual Blaise allows any block (or all blocks) to be exported to a text file showing the rules section(s). The output is formatted so it can be pasted over the existing rules of a block and compiled in Blaise (assuming the Fields and other parts are already set), with end ifs and the like properly inserted.

However, comments and some other markup are not preserved, so a more likely use case is that a programmer would simply use this output as a specification, or else a compare tool to merge the changes.

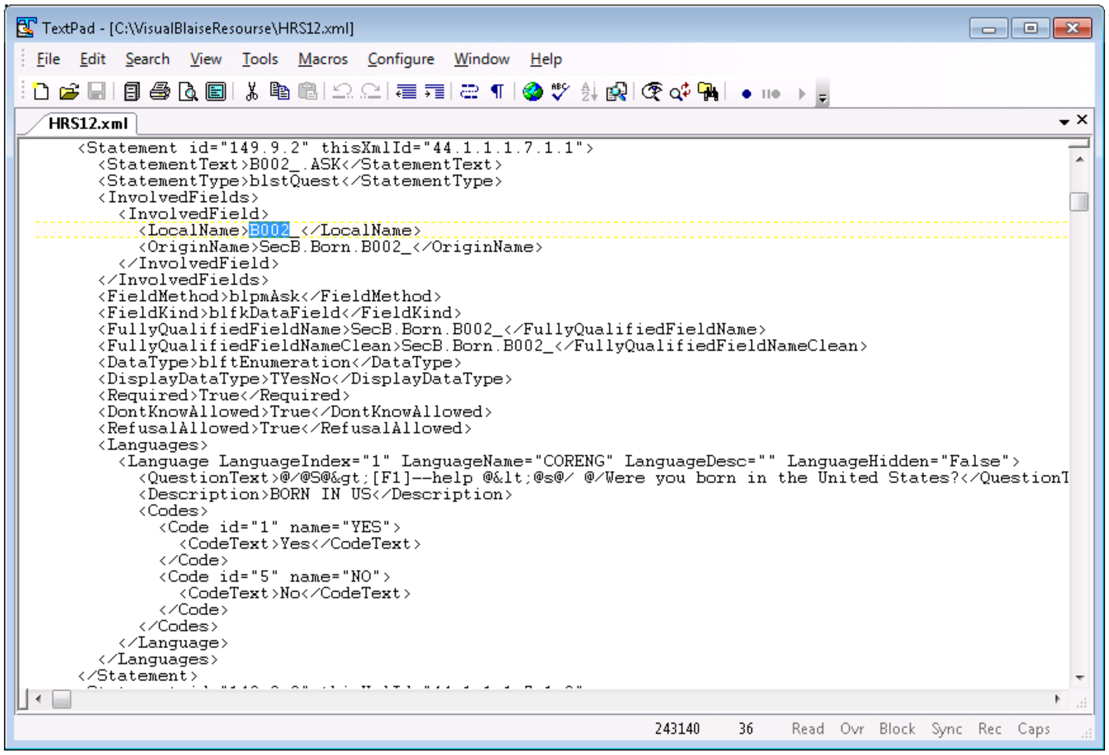
Operating Details of Visual Blaise

The remainder of this paper describes how Visual Blaise works from the user perspective, by illustrating some of the main functions.

Preparation Before Opening Visual Blaise

In order to run Visual Blaise, you will need the compiled files from your survey (.Bmi, .Bxi, etc). A separate program needs to be used to generate an XML file from those compiled files for use as input. As noted above, this can be done by the program called "Blaise Rules" included with MQDS, or by simply running MQDS itself. The XML that Blaise Rules generates is in the format that Visual Blaise uses to construct a flowchart.

Figure 3: an example of a question statement in XML output from Blaise Rules

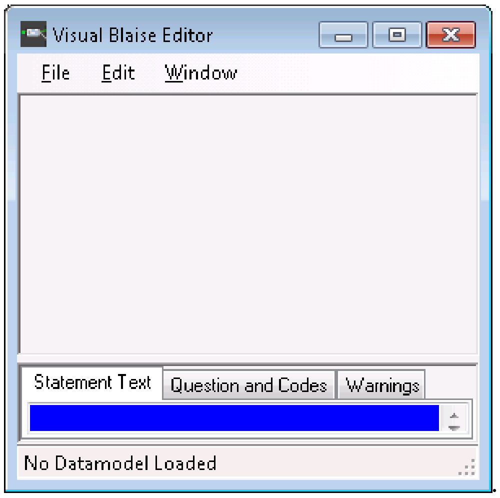

TextPad - [C:\VisualBlaiseResourse\HRS12.xml]
File Edit Search View Tools Macros Configure Window Help
HRS12.xml
243140 36 Read Ovr Block Sync Rec Caps

```
<Statement id="149.9.2" thisXmlId="44.1.1.1.7.1.1">
  <StatementText>B002_.ASK</StatementText>
  <StatementType>blstQuest</StatementType>
  <InvolvedFields>
    <InvolvedField>
      <LocalName>B002_</LocalName>
      <OriginName>SecB.Born.B002_</OriginName>
    </InvolvedField>
  </InvolvedFields>
  <FieldMethod>blpmAsk</FieldMethod>
  <FieldKind>blfkDataField</FieldKind>
  <FullyQualifiedFieldName>SecB.Born.B002_</FullyQualifiedFieldName>
  <FullyQualifiedFieldNameClean>SecB.Born.B002_</FullyQualifiedFieldNameClean>
  <DataType>blftEnumeration</DataType>
  <DisplayDataType>TYesNo</DisplayDataType>
  <Required>True</Required>
  <DontKnowAllowed>True</DontKnowAllowed>
  <RefusalAllowed>True</RefusalAllowed>
  <Languages>
    <Language LanguageIndex="1" LanguageName="CORENG" LanguageDesc="" LanguageHidden="False">
      <QuestionText>@/@S@&gt;[F1]--help @&lt;@s@/ @/Were you born in the United States?</QuestionT
      <Description>BORN IN US</Description>
      <Codes>
        <Code id="1" name="YES">
          <CodeText>Yes</CodeText>
        </Code>
        <Code id="5" name="NO">
          <CodeText>No</CodeText>
        </Code>
      </Codes>
    </Language>
  </Languages>
</Statement>
```

Visual Blaise Functionality

Once you have obtained an appropriate XML from your survey, run the Visual Blaise executable. You will see a standard Windows menu and several tabs that will be described below

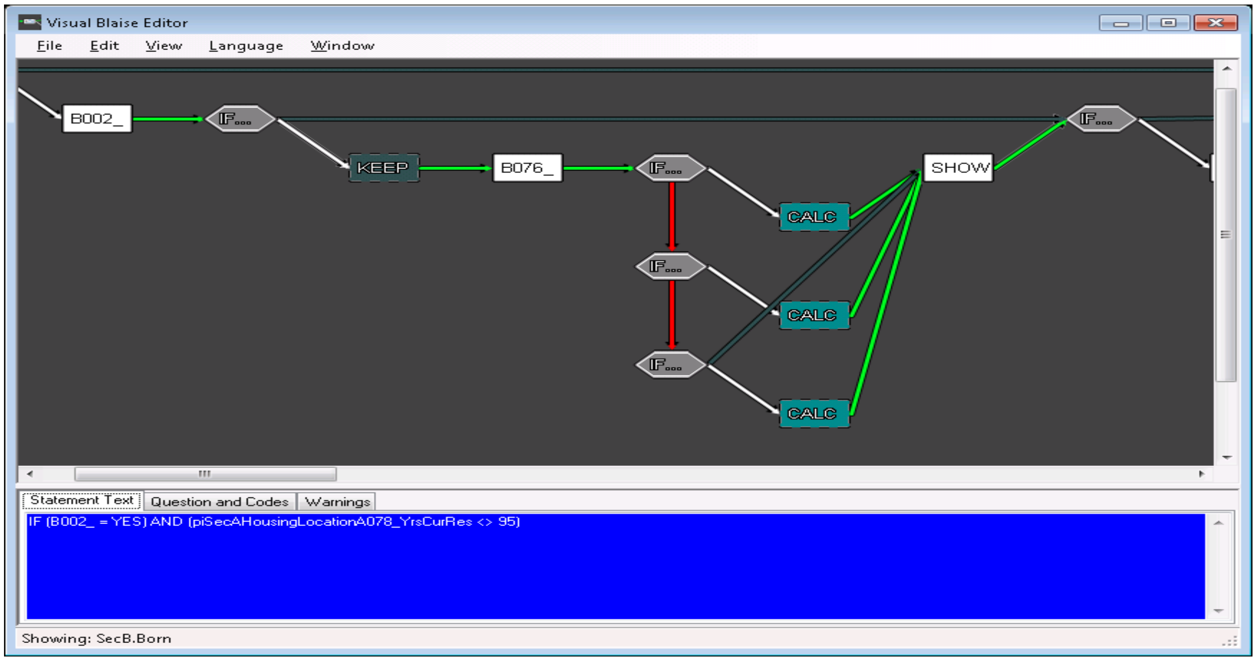
Figure 4: the starting screen of Visual Blaise



File Menu

There are several File->Open menu options to choose from. The **From XML** option will provide a full view of the medadata with KEEPs, SHOWs, CHECKs, SIGNALS, and expanded logic. This view is great for a technical review, but may be too much information for a casual observer who does not need to know all the internal Blaise details.

Figure 5: Visual Blaise showing all statements, including keeps, assignments and conditions that do not lead to asked questions

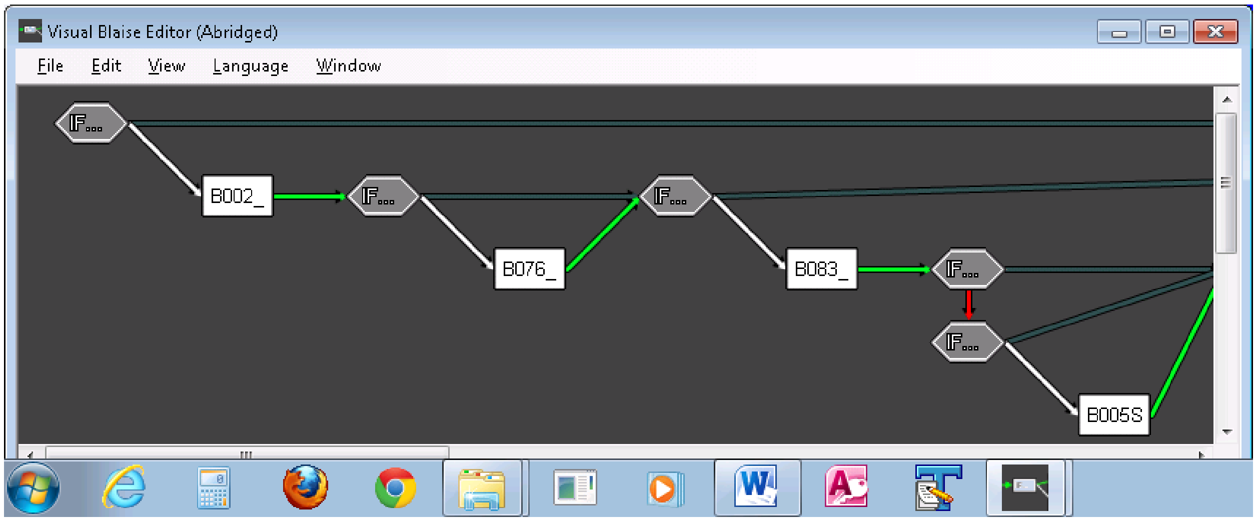


IF (B002_ = YES) AND (piSecAHousingLocationA078_YrsCurRes <> 95)

Showing: SecB.Born

The **From XML, Abridged** option will omit several background elements and show only the major flow information. This view may be easier to follow for the "not-so-techy" users.

Figure 6: Visual Blaise showing an abridged version of the same sequence



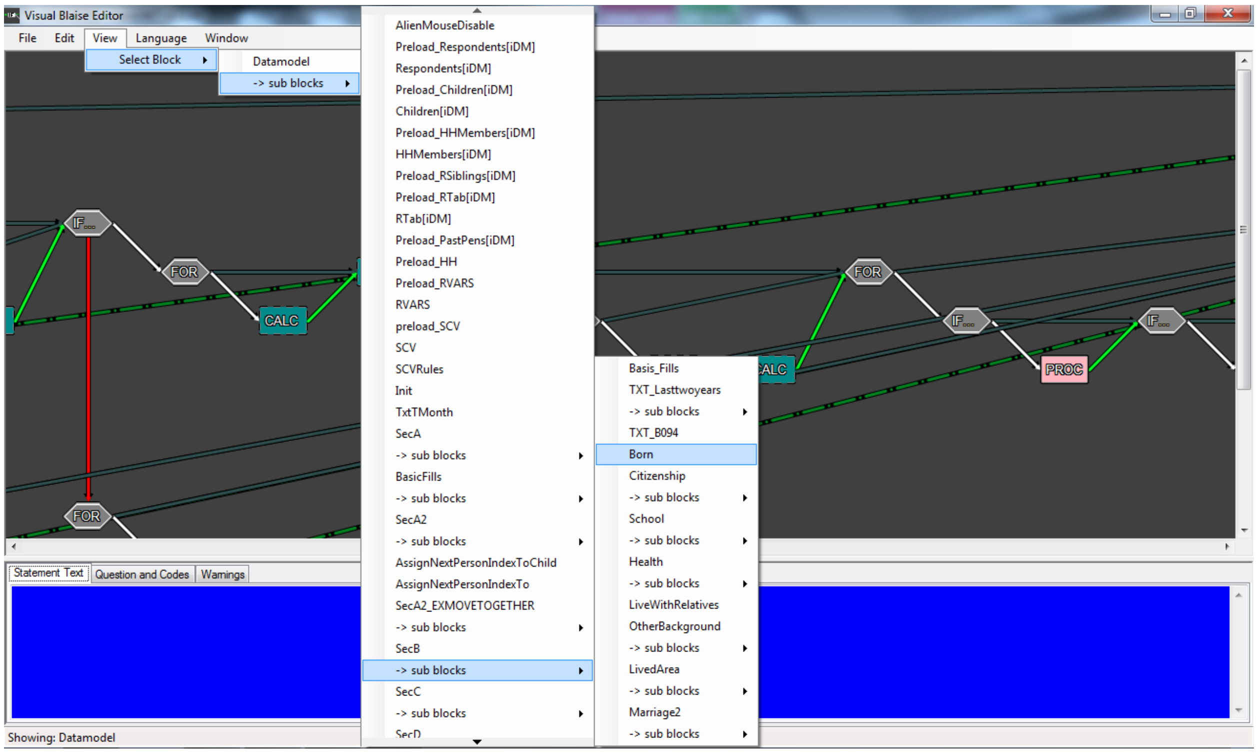
Edit Menu

If you are looking for a specific variable within the block in which you are currently located, you can use the menus Edit **Find** field functions, or [CRTL]+ F. Then, [F3] will allow continued searching for the next instance of that variable. The Edit menu allows **Undo** (up to 4 levels) or [CRTL]+ F.

View Menu – Select Block: Finding Where and Getting There

It can be frustrating to find a variable in a very large instrument. Visual Blaise provides the ability to move to a specific block without having to navigate through the editor, block by block. This feature is located under the View menu. A drop-down of blocks is shown. Sub-blocks are provided as off-shoot drop-downs. This makes it easy to see the structure of your survey and move to the block that you wish to review and alter.

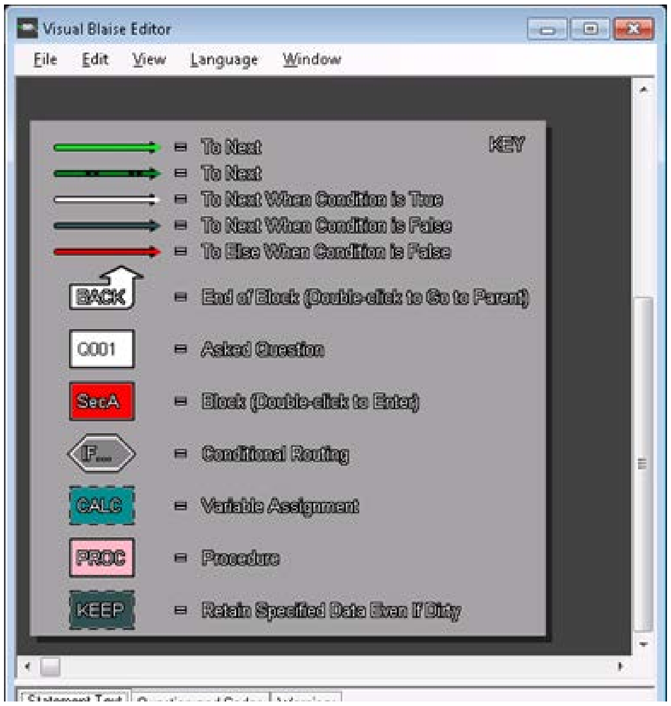
Figure 7: selecting a block via the “View” menu



Window Menu – Show Key

To help with the navigation and symbols there is a Key. This is toggled on and off in the Window menu option.

Figure 8: the symbol key



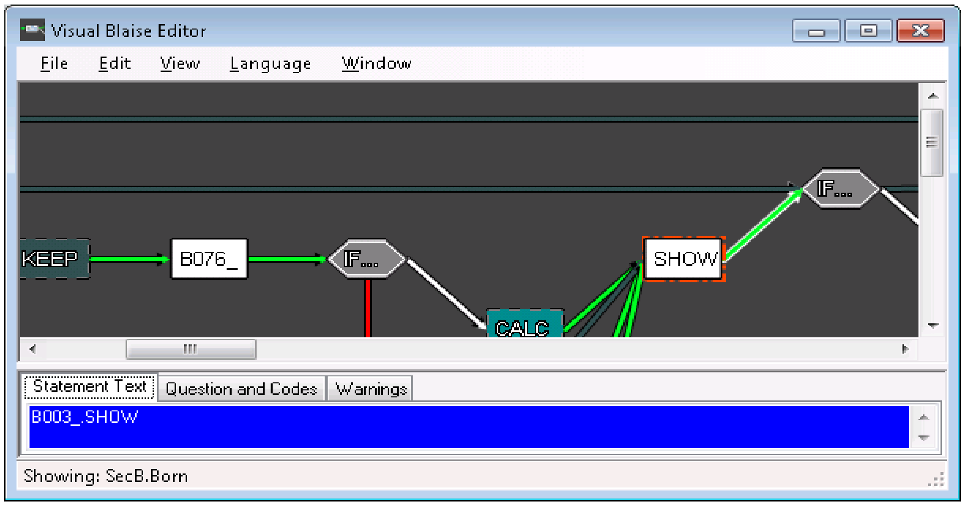
→	= To Next
→	= To Next
→	= To Next When Condition is True
→	= To Next When Condition is False
→	= To Else When Condition is False
BACK	= End of Block (Double-click to Go to Parent)
Q001	= Asked Question
SecA	= Block (Double-click to Enter)
IF...	= Conditional Routing
CALC	= Variable Assignment
PROC	= Procedure
KEEP	= Retain Specified Data Even If Dirty

Attribute Tabs

At the bottom of the screen there are three tabs: “Statement Text”, “Question and Codes” and “Warnings”. As you move around in the flow diagram you will see attributes of each element presented. The information displayed will depend on the type of element selected.

Statement Text

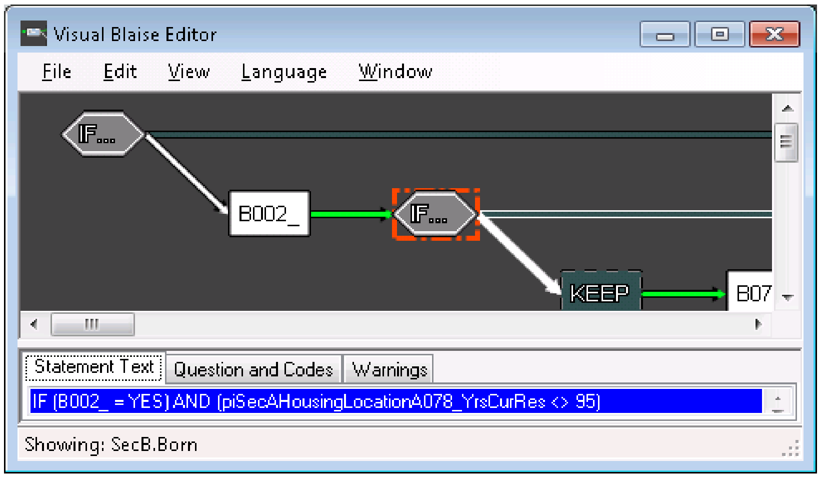
Figure 9: with a variable selected



B003_SHOW

Showing: SecB.Born

Figure 10: with conditional routing selected, the "Statement Text" tab shows the text of the condition



IF (B002_ = YES) AND (piSecAHousingLocationA078_YrsCurRes <> 95)

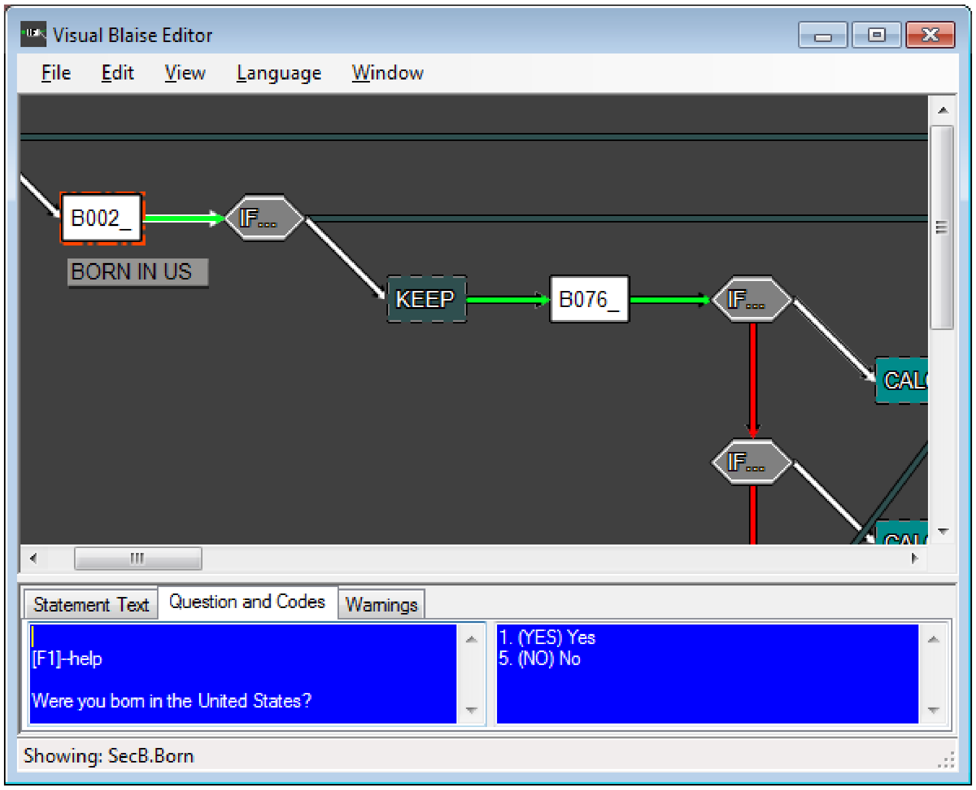
Showing: SecB.Born

- **Checks or Signals** are represented by two elements, one indicating a hard or soft check and the other with the content of the check in the "Statement Text" tab:
  (((B006_ >= 1880) AND (B006_ <= A501_CurDateYear)) OR B006_ = NONRESPONSE
- **Calculations** show the assignment in the "Statement Text" tab:
  B005S := 'Mexico - assigned'
- **Procedures or Blocks** display the call and the parameter list.
  TXT_B094 {Procedure} ({Parameter list:}piRespondents1X065ACouplenss, piInitA106_NumContactKids, FLB094)

Question and Codes Tab

The Question and Codes tab displays the variable text and code frame.

Figure 11: showing question text and codes of selected question (B002_)



[F1]-help

Were you born in the United States?

1. (YES) Yes
5. (NO) No

Showing: SecB.Born

Warnings Tab

The warning tab provides an alert to possible errors when changing the flow location of a variable. For example:
8:56 PM: Moved or cut B008_ used in (Check Statement) *NOT ((B006_ = EMPTY AND B007_ = EMPTY) AND B008_ = EMPTY) CORENG "You must answer one of these categories"*

Use Case

The following section will attempt to describe the design process and the uses of this tool in that process. We will walk through a few of the main commonly requested activities, including maneuvering through the flow chart, adding and deleting variables, manipulation of variables, and tracking connections.

Starting with the specifications from the last wave, the first step would be to review any reported problems during the data collection and determine the need to alter the specifications. The relevant Co-Investigators, by reviewing data and monitoring current affairs, will request changes to the specifications. Our Design Coordinators collect these requests and oversee the development process of the main survey instrument. This process starts with making decisions and developing specifications based on these change requests, on field feedback, on staff analysis of data, and on wait-list items that were not handled in the previous wave. Programmers will then add new content, cut unneeded fields, or change the sequence of the questions that are asked.

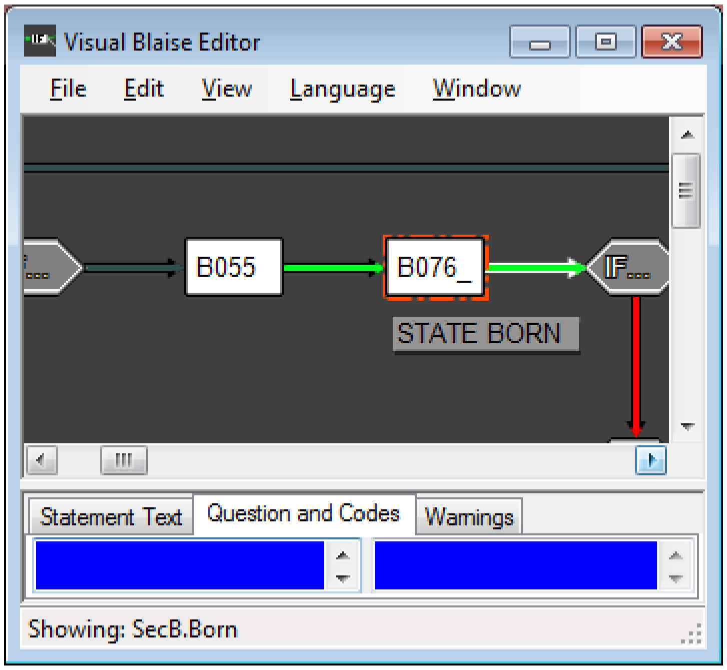
For more complicated sequences, a tool like Visual Blaise provides a visualization of the change and its impact on the block before actually making the change. Users are able to add, delete or change locations of a field and see what happens to the flow.

Getting Familiar with the Tool – Mouse and Keyboard Controls and Feedback

When the user clicks the mouse on a statement, it becomes selected and Visual Blaise offers some visual feedback:

- A flashing border is drawn around the selected statement.
- An asked question which is selected also displays its descriptor in a transparent gray box below the icon.
- A flashing highlight is drawn around the routing arrow extending from the statement, or from multiple arrows if the selected statement is a condition.

Figure 12: the highlighted border, routing arrow, and descriptor box of a selected question

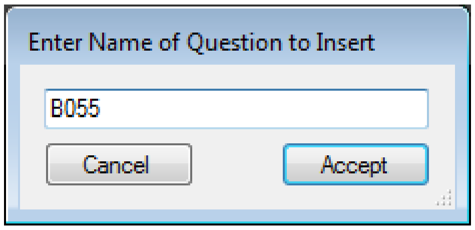


- Clicking and dragging a statement allows the user to see more clearly via the motion of the arrows where the connections lead when a segment is particularly busy.
- Pressing the right arrow key or the down arrow key moves the focus (and highlighting) from one selected statement to the next via the routing arrows. Pressing the left arrow key moves the focus to the previously selected statement (reversing the previous "history" of key presses, so it will not move further back than the selection starting point.
- Pressing the Enter key or double-clicking on a Block or Procedure statement will change the view to show the rules for that sub-Block or Procedure instead. Pressing the Backspace key or double-clicking on the "Back" return arrow icon at rightmost end of the rules will return the view to the parent Block or Procedure.
- Holding shift and pressing the right arrow key expands the selection to include multiple statements for the purpose of performing large moves or cuts.

Adding a Field

Via the View menu, Find function, or mouse and keyboard controls, the user can locate the part of an instrument where a change, such as adding a new question, needs to take place. Once at that location, right-clicking on a statement will cause a pop-up menu appear with options for inserting, deleting, or cutting statements. Insert will allow selection of what type of element you want add. Selecting a Question or Block type will prompt the user for the name of the new question or block.

Figure 13: pop-up prompt for field name


Enter Name of Question to Insert

B055

The new field will be inserted to the left of that element. (In the future you will be able to enter the question text and codes here, but for now only the Rules section is included in the output.)

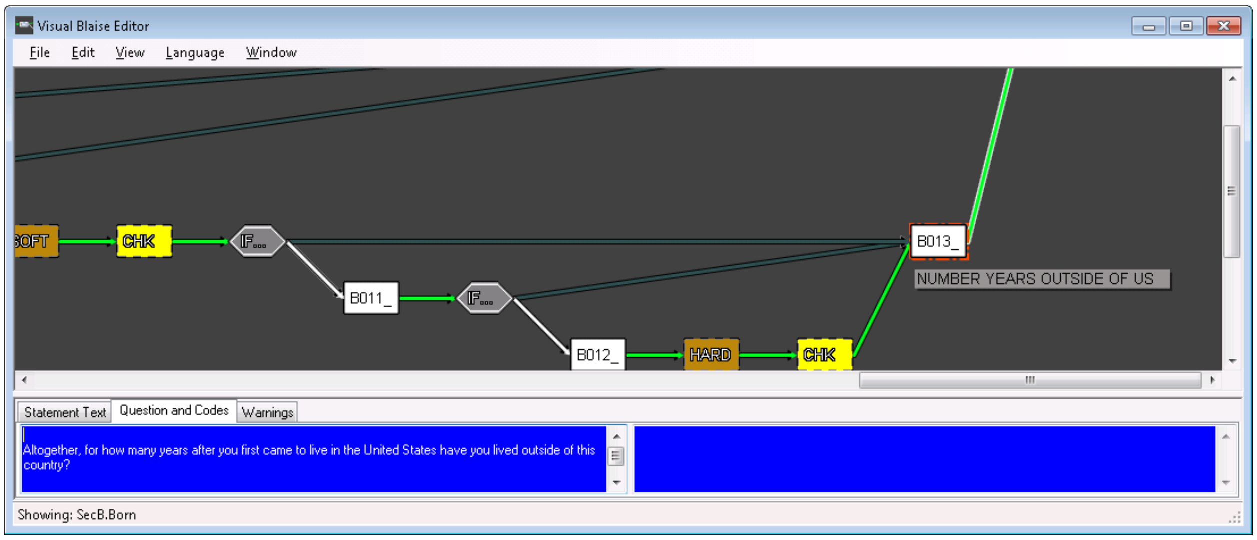
Deleting a Field

Similarly, a selected question (or other statement) can be deleted via the same right-click menu. The warning tab will display references to the deleted field that need to be deleted or fixed (however, this is not comprehensive; warnings are still a work in progress at this stage).

Moving Elements Around

The following example illustrates a field move request that utilizes Visual Blaise's element moving capabilities. Suppose that new fields were inserted last wave and, after reviewing the data, the investigators would now like to move fields around for better flow and coverage. They decide to move B013_ to before B011_, which can be accomplished in Visual Blaise by clicking and dragging B013_ above B011_ and dropping it there. Since there is no logic dependent on B013_ , the dragged statement will be inserted to the left of B011_ and no warnings will be generated. Notice that all of the flow arrows automatically realign as appropriate for this change.

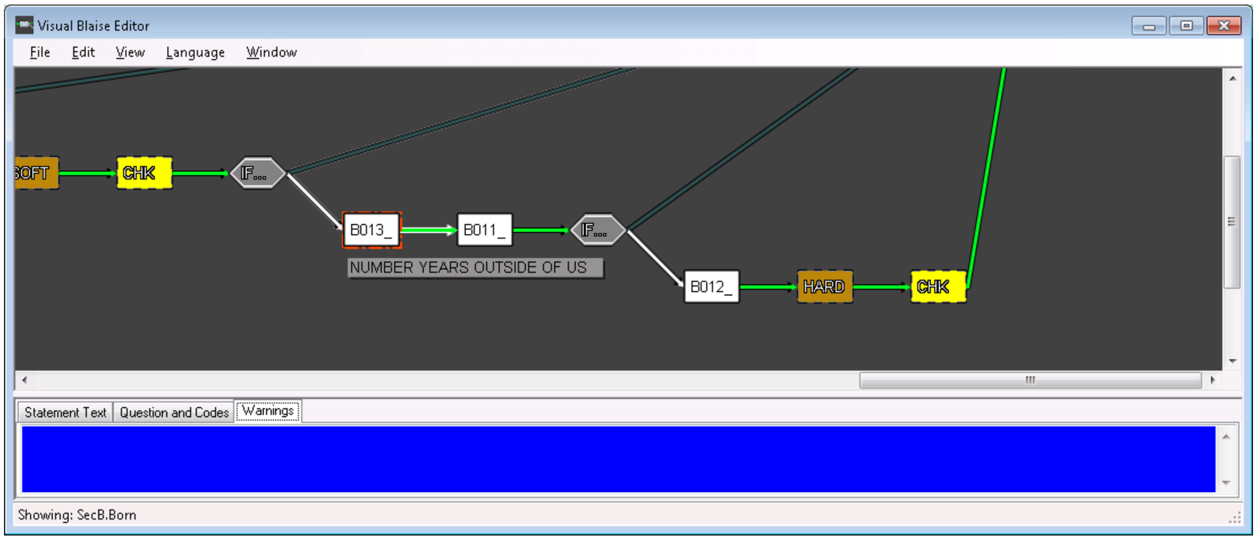
Figure 14: before moving question B013_ via drag and drop



Altogether, for how many years after you first came to live in the United States have you lived outside of this country?

Showing: SecB.Born

Figure 15: after moving question B013_ via drag and drop



Showing: SecB.Born

More complicated moves might produce warning messages. If there are references to the element being moved in other statements passed over by the move, each reference should be reviewed and altered, if needed. For example, if a field is moved to a location after a conditional statement that references it, that reference most likely would need to be deleted. The warning messages are meant to alert the user to such contingencies.

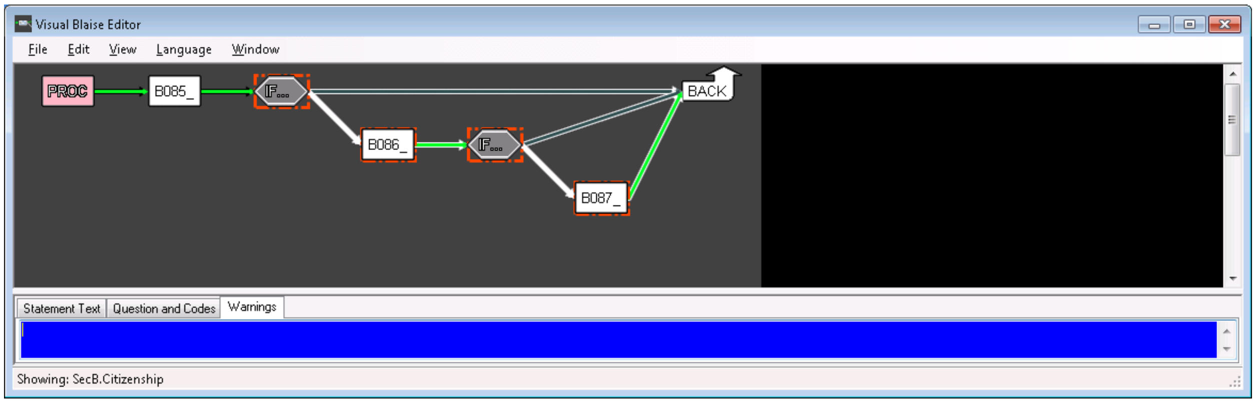
Example:

```
FieldA
if FieldA = yes then
   FieldB
endif
```

If Field A is moved to a location after Field B the conditional routing reference may need to be deleted or updated to something else.

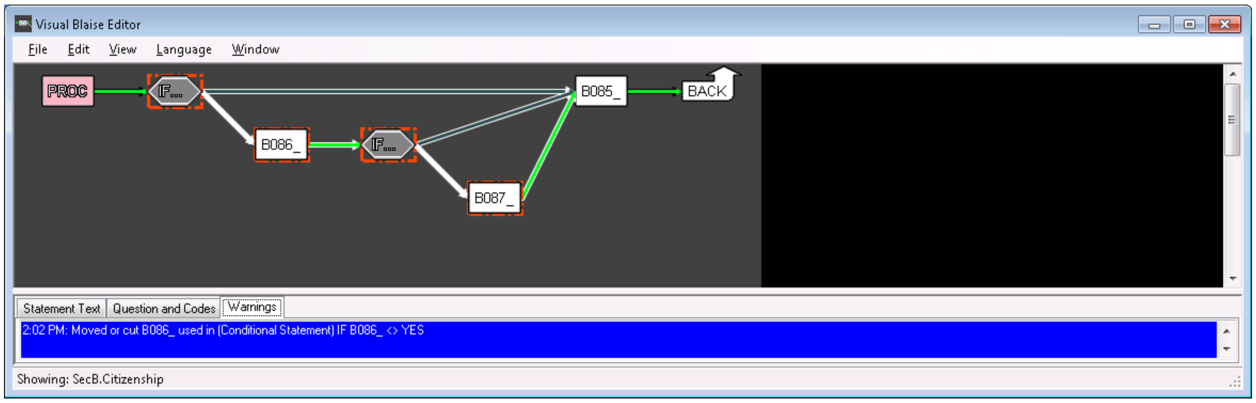
Another issue that arises during section redesign efforts is the need to move a large group of logic as a group. To select more than one element, first click on the statement you want to start with and then press shift-right arrow until all desired statements are selected.

Figure16: selecting multiple statements following the question B085_


Visual Blaise Editor
File Edit View Language Window
PROC
B085_
IF...
B086_
IF...
B087_
BACK
Statement Text | Question and Codes | Warnings
Showing: SecB.Citizenship

Then, to move the selected group, right click on the group and select "cut." Next, select the element before which the group should be inserted, right click and select paste.

Figure 17: multiple selected statements cut-and-pasted before B085_


Visual Blaise Editor
File Edit View Language Window
PROC
IF...
B086_
IF...
B087_
B085_
BACK
Statement Text | Question and Codes | Warnings
Showing: SecB.Citizenship

2:02 PM: Moved or cut B086_ used in (Conditional Statement) IF B086_ <> YES

In the above example, the Warnings tab reports that the far right IF condition in the moved group refers to the field that now comes after it. This flow will need to be fixed or deleted.

Producing and Reviewing Output

Now that the flow has been reviewed and updated, it is possible to output the changes into a Rules section readable by Blaise.

Selecting "Save" from the file menu offers two options: to save the flow to a Visual Blaise file or to save to a Blaise rules text file.

- Saving to a Visual Blaise file will save any changes to this point so you can retrieve and continue later.
- Saving to a Rules file will produce a text file with the logic that is represented in the Visual Blaise view. An option to save the Rules section of every block or only the currently viewed block is provided.

The output after the change from the above example would appear like this:

```
IF B085_ = YES THEN
   B086_.ASK
   IF B086_ <> YES THEN
      B087_.ASK
   ENDIF
ENDIF
B085_.ASK
```

Conclusion

We have only just finished this tool as a beta version and begun to use it in our design process. We are hopeful that this tool will allow investigators and coordinators to experiment with the design of a section and refine the flow to an optimal state more efficiently and with a better understanding of what they will be able to achieve by making each change. We believe this tool will also result in considerable time savings for the programmer, as the output can potentially be merged into the main instrument and, with a little “tweaking,” become the next wave’s rules.