Methods of Integrating External Software into Blaise Surveys

Lilia Filippenko, Joseph Nofziger, Mai Nguyen & Roger Osborn (RTI International, USA)

1. Introduction

For surveys that require calls to external applications, Blaise provides support for various implementations. At RTI International we have studies that take the following approaches:

- Use Manipula setups to manage switching between a Blaise instrument and calls to external applications
- Use "Action" from a Blaise instrument and a BOI file to pass data between the Blaise database and MS Access databases
- Use an alien router from a Blaise instrument to call external applications and to collect data from them

Each method of integration has advantages and disadvantages. Although the alien routers may provide the most flexible solution, their development and testing could take more time than the first two approaches. In this paper, we will discuss the pros and cons of each implementation. We will also present our experience with several alien router implementations written in Visual Basic.Net and C# to meet the following aspects of integration:

- Seamless User Interface
- Ease of Programming (Blaise and Windows)

Finally, we present a case study of using an alien router to implement complex randomization of questions.

2. Invocation from Manipula setup

2.1. Case Management System Overview

At RTI International a Visual Basic Case Management System (CMS) program is used on laptops for many CAPI interviews. Along with providing user functions, the CMS manages invocation of the CAPI instrument. Various software packages or languages are used to develop instruments, including Blaise, ASP, CASES, and others. The type of software package is stored in configuration files and is used by the CMS to create and invoke an appropriate driver object to manage work with individual package.

Methods implemented by the driver and applied for every case on the laptop include:

- Import (load a case into the laptop database)
- Update events and status codes to the instrument database
- Conduct interview
- Export (transmit a case from the laptop)

For Blaise instruments, the driver invokes a single main Manipula setup, passing parameters to execute the requested method. The name of the data model and database is passed as a parameter. Our Manipula-driven architecture is illustrated in

Figure 1. The case ID and a study identifier are passed as parameters. The "Edit" function is used to run the Data Entry Program (DEP) for a case. This Manipula setup was developed in general way so that it can be prepared independently of the Blaise instrument with which it is used, and without changes to the CMS. This approach gives us the flexibility to make adjustments for different studies as needed. An example of this is the ability to call an external application while conducting the interview without returning to the CMS.

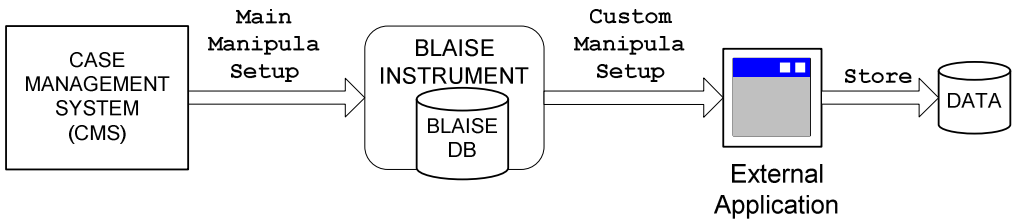

Figure 1 Manipula Setup Architecture

2.2. Example of Invoking External Applications from the Manipula Setup

RTI International conducted a study that required calls to three assessments in random order from the middle of the Blaise interview. Our approach was to associate a unique code ("sum_evt" in the snippets) with the completion of each assessment. The code can be used to decide which assessment to execute next. We added program code to the main Manipula setup to call a custom Manipula setup which randomizes the order of external applications, invokes them, and updates the code in the Blaise database.

First we define parameters to start DEP:

```
   'INVOKE':  { PARAMETER(1) - Case ID }
   strWorkF := PARAMETER(2)
   strInstName := PARAMETER(3)
   strProjectID := PARAMETER(5)
   strCaseId := ' /K' + PARAMETER(4)
```

Then we create a command string and start DEP for the Blaise instrument:

```
   strRun := strWorkF + '\' + strInstName + ' /G /X /C' +
     strInstName + '.diw @' + strInstName + '.bcf ' +
strCaseId
   Reslt := EDIT(strRun)
```

Next, we call the custom Manipula setup after the first part of the interview is completed:

```
   strTemp := GETVALUE(strInstName, strInstName,
      PARAMETER(4), 'main_case.sum_evt')

   IF (strTemp > '301') AND (strTemp < '306') THEN
      strRunTemp:='BlzUpdateINHL /W' + strWorkF + ' /P' +
         PARAMETER(4)+ ';Assessmt;' + strTemp + ' /Q'
      Reslt:= CALL(strRunTemp)
           {……}
   ENDIF
```

Finally, we check the code again in the Blaise database to determine if all external assessments were finished successfully, and start DEP again:

```
REPEAT
   strTemp := GETVALUE(strInstName, strInstName,
PARAMETER(4),
      'main_case.sum_evt')
   IF (strTemp >= '306') THEN
     strRun := strWorkF + '\' + strInstName + ' /G /X /C' +
     strInstName + '.diw @' + strInstName + '.bcf ' +
strCaseId
     Reslt := EDIT(strRun)
   ELSE
     { Assessments not finished }
     { Don't need to restart the instrument }
     EXITREPEAT
   ENDIF
UNTIL 1=1
```

3. Use of Action and BOI Files

The above method of invocation works well when Blaise and assessment data are stored separately. This is acceptable when data are to be analyzed separately. However, analysis can be facilitated by integrating the assessment data with the Blaise data at an earlier stage. Furthermore, data management is simpler when data are integrated into a single entity at the time of collection. With recent versions of Blaise including tools allowing us to pass data between Blaise databases and relational databases like MS Access and SQL Server, we have begun to make frequent use of BOI files to store all data in one place - the Blaise database - at the time of collection.

We chose to control invocation using an action because of its simplicity. Creating an action for a user defined type to start an executable application is quicker and less error prone than programming an alien router. This architecture is shown in Figure 2.

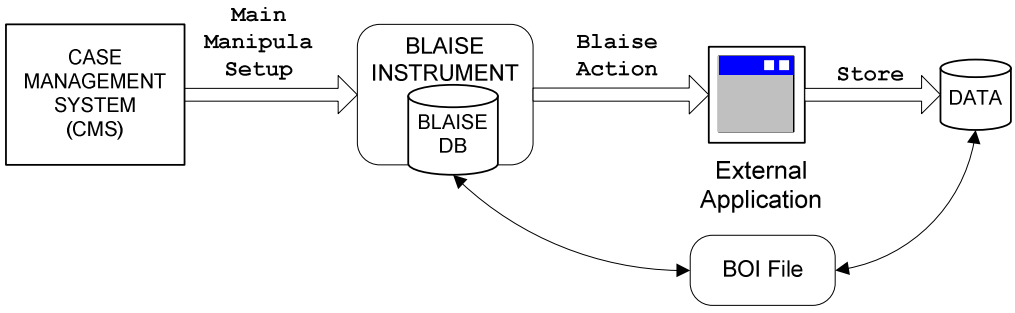

Figure 2 Action/BOI Architecture

3.1. Invoking the Assessment Using an Action

First we create a user defined type with the action “Start Executable”. We can then use this type to declare a field that is used as the entry point to invoke the assessment..

In case of a breakoff and subsequent re-entry into the instrument, we take steps to ensure that the incomplete assessments are invoked, and that already completed assessments are not improperly redone. A completion indicator field is used as a gate to determine whether to enter the assessment or to keep the existing data.

3.2. Data Transfer from MS Access to Blaise Database During Interview

Ideally, upon completion of an assessment, control passes back to Blaise. At that time the external BOI file is used to pull data from the assessment's Access database back to the Blaise database. This is accomplished by using the SEARCH method, with the case ID as a parameter, to locate the record in the MS Access database that is used by the external assessment to store the data. If the SEARCH is successful, then the READ method is used to assign values from the MS Access database to Blaise fields. The completion indicator field is set to value that will later be used to determine the state of the assessment.

3.3. Data Transfer Before Exporting Case from Laptop

A problem arises if the interviewer skips the question where the assessment would ordinarily be launched. We program a SIGNAL to remind the interviewer to conduct the assessment, but the SIGNAL may be suppressed. The interviewer may later realize the mistake and return to the appropriate question to launch the assessment. However, after proceeding past the action question the first time, Blaise would have tried to import the data using the .BOI file, even though the Access database would not contain any data for the case. The problem lies in the fact that the BOI file data import only occurs the first time. Despite the interviewer having now corrected the mistake and completed the assessment, the SEARCH method would not find data for the specified case.

A second problem situation occurs if the interviewer completes the assessment, but then, whether as a result of a mistake or system error, fails to return as planned to the Blaise instrument. If the system was shut down before Blaise regained control then there would be no opportunity to read data through the .BOI file. If the case happens to be exported at this point, for example due to transfer to another laptop, then without backup measures the assessment data would remain on the laptop where it was captured.

To be certain that data collected in the assessment will be in the Blaise database whether or not control is properly returned to Blaise, we created an external program that reads the assessment data and pushes it into the Blaise database immediately before transmitting the data back to our servers. A call to this program was added to the main Manipula setup under the "Export" option. It filters for only those cases for we expect to have data from the assessment in the Blaise database.

```
IF (Uppercase(strInstName)='CHILD') AND
   (strProjectID='9231')) THEN
   strTemp := GETVALUE(strInstName, strInstName,
      PARAMETER(4), 'WJComplete')
   IF strTemp <> '1' THEN
      strRun := strWorkF  + '\WJUpdateCase.exe' + ' ' +
         PARAMETER(4)
      Reslt := run(strRun, WAIT, HIDE)
   ENDIF
ENDIF
```

4. Alien Routers

Blaise alien routers enable external applications to be called from Blaise. Although Blaise expects an alien router to be a COM component, alien routers can be created in .NET utilizing COM interoperability. Using the Blaise Component Pack (BCP) alien routers can read and write variables in a Blaise database. This approach allows us to maintain all collected data in one place and therefore to simplify data management and later analysis.

One of our recent Blaise studies included seven alien routers and associated external applications. We used external applications for address verification, address mapping, random number generation, Woodcock-Johnson cognitive assessments, Event History Calendar (EHC) from the University of Michigan, and other purposes.

The primary goal was to provide a richer user data collection experience that minimized disruptions to the interview flow. Another goal was to simplify how the alien routers were used in Blaise code. In addition, we wanted to introduce some code reusability and common designs within the routers and applications. These goals were accomplished by incorporating the following:

- Set external application window to "Always on Top"
- Consistent look-and-feel amongst Blaise UI and external applications
- Use of complex data types in Blaise code
- Common architecture and code in routers

4.1. General Overview of Alien Router Architecture

Before examining the specifics of our implementation, a general overview of the alien router architecture is in order. The basic alien router architecture is shown in Figure 3. A Blaise program instantiates an alien router variable and calls it using the Blaise Router method. This loads a COM-based alien Router DLL and runs the method specified in the alien Router call. The router can use the Blaise Component Pack (BCP) API to read or write variables in the Blaise database. The Alien Router can also launch an external application using any of a number of mechanisms. One mechanism is to use the Start method of the .NET System.Diagnostics.Process class.

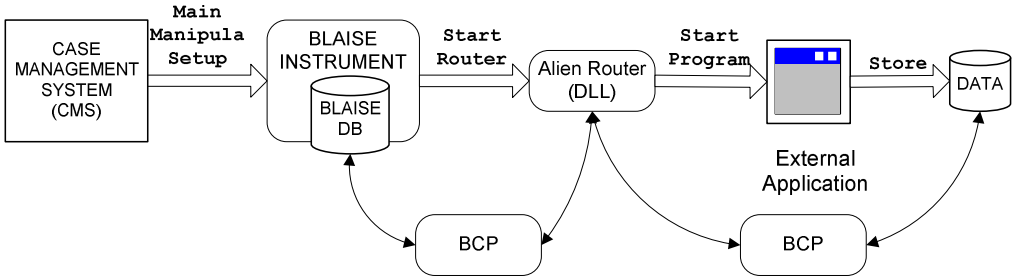

Figure 3 Basic Alien Router Architecture

4.2. External Application Window as Always on Top

The call from Blaise to the alien router and the call from the alien router to the external application are blocking, meaning the router does not continue to run until after the external application exits, and the Blaise instrument does not continue to run until after the alien router exits. While blocked, the Blaise instrument will not

respond to user input. Thus if the external application window is somehow moved behind the Blaise window, the Blaise instrument will appear to ”hang” or ”freeze”.

An easy way to prevent this is to set the external application windows as ”Always on Top”, by setting the TopMost property of the external application window to True. When the TopMost property is set to True, the window will always be in front of other windows. If the external application is not a .NET application, other methods can be used to make the window top-most.

4.3. Consistent Look-and-Feel

When an external application is invoked from a Blaise instrument via an alien router, it runs in its own window and has its own look-and-feel (visual representation) which may be very different from the Blaise instrument. This could create confusion for inexperienced field interviewers and respondents. To reduce confusion and improve the user experience, the visual properties of external applications can be set so that they appear to be an integrated part of the instrument. From the user’s perspective, the external application should appear as a Blaise pop-up dialog, not a separate application.

An example of a visually consistent external application is shown in Figure 4. In this case, the external application called from an alien router shows 14 houses and allows the respondent to select a race for each house in the neighborhood. The ”x” marks the resondent’s house. Also shown in Figure 4, the instructions to navigate the external application is clearly provided in the Blaise instrument question text. The user can move from one house to another using the standard ”TAB” key and label each house with different letter keys.

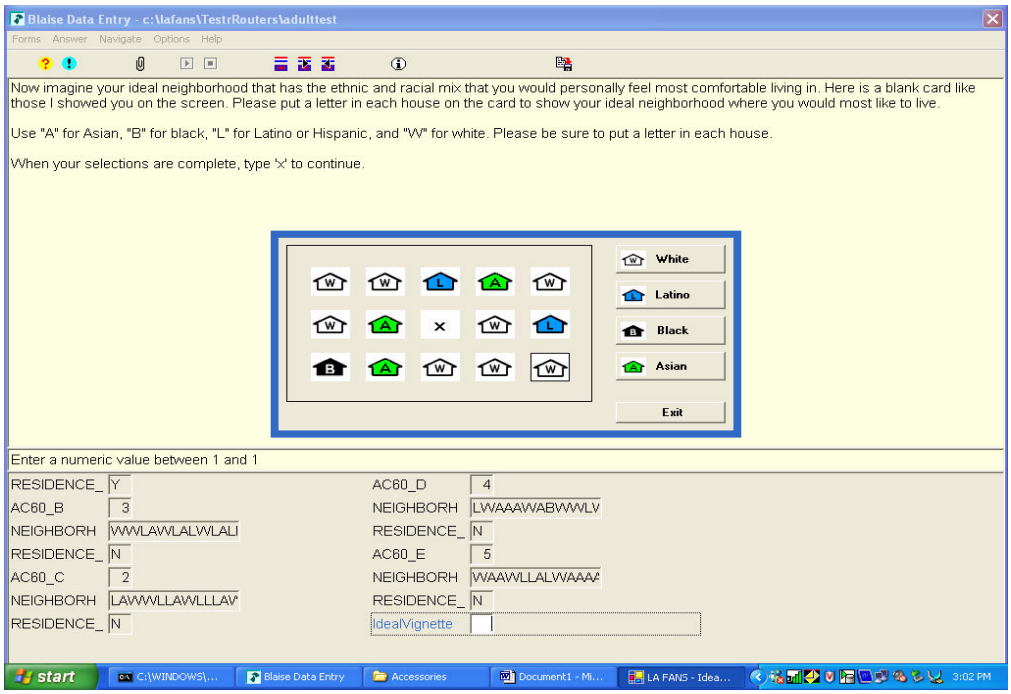

Figure 4 Visually-Consistent External Application

The external application’s window size and position can be specifically set using:

```
this.StartPosition = FormStartPosition.CenterScreen;
this.ClientSize = new System.Drawing.Size(480, 232);
```

These properties can also be set in the visual designer.

To prevent the user from prematurely closing the external application, the window's border can be set using:

```
this.FormBorderStyle = FormBorderStyle.None; //C#.NET
```

or

```
Me.FormBorderStyle = FormBorderStyle.None   'VB.NET
```

This property can also be set in the visual designer as shown in Figure 5.

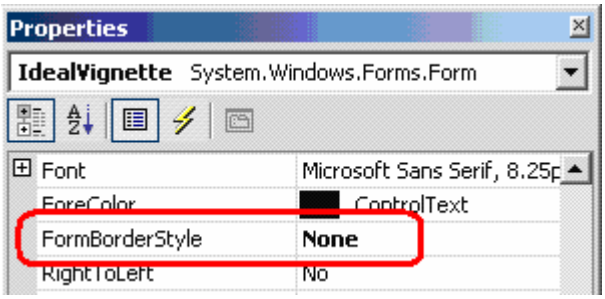

Figure 5 Setting FormBorderStyle Property

This level of seamless graphical presentation is easily achievable with external applications developed in-house. For third-party applications, for example the Event History Calendar (EHC), this level of integration is usually unachieveable or would take significantly more effort.

4.4. Complex Data Types in Blaise Code

In order to simplify and componentize the Blaise programming effort, we elected to use complex data types for router-calling variables in Blaise. This is best shown by example. The Blaise code is shown in Listing 1.

One of our external applications performed address verification. A user-supplied address is collected in Blaise and passed to an external application that uses a third party address verification API to ensure the supplied address is valid. In the Blaise instrument, we created a block called BAddressVerification. The BAddressVerification comprises other fields defined as blocks: AddressSearchElements, AddressOutputElements, and VerifyAddress.

AddressSearchElements contains the input address, which is collected by the Blaise instrument. VerifyAddress is essentially a wrapper variable for the call to the alien router. The results of the address verification by the external application are written to the sub-fields of AddressOutputElements.

When address verification is required in an instrument, the Blaise programmer simply creates a variable of type BAddressVerification.

Since the fields of the data type are well-defined, and the call to the alien router is within this block, the alien router has a relative reference to all the names of the sub-fields within the block and can easily address them.

Other benefits and implementation details of using such block-within-block complex data types are described in the next section.

4.5. Common Router Implementation

In order to increase code reuse and maintainability, we used a common implementation amongst our alien routers. While we did not go so far as to create a base class for our alien router implementation, such a class could have easily been developed.

Source code for a typical alien router is shown in Listing 2. This router is written in C#.NET. In order to use the BCP API, the Blaise BCP DLL is referenced and included:

```
using BlAPI4A2;
```

The entry point for all of our alien routers is the Run method, which we'll describe in detail.

The first thing the Run method does is store a class-level reference to the database and DEP state variables that were passed in. This allows the router to use these variables in all of its methods.

```
this._data = data;          // BlAPI4A2.Database
this._state = state;        // BlAPI4A2.DepState
```

Then the Run method looks at the AlienRouterStatus property of the state variable to determine what the reason is for this call to the Run method. The router will ignore all AlienRouterStatus types except BlAlienRouterStatus.blrsPreEdit.

```
if (this._state.AlienRouterStatus !=
   BlAlienRouterStatus.blrsPreEdit) return;
```

Next, the router gets the path of the current Blaise database from the DataFileName property of the _data variable and stores it in a variable called BlaiseDBPath. This path is used to locate the log file in the same directory as the Blaise database.

```
this.BlaiseDBPath = this._data.DataFileName.Substring(0,
    this._data.DataFileName.LastIndexOf(@"\"));
```

For debugging purposes, the router sends logging information to a log file. The CreateLogFilePath function sets the LogFilePath variable to the full path of the log file for this instance of the router. The name of the log file is generated using the name of the router class. Once the LogFilePath variable is set, the Log method can be used to send strings to the log file.

The variable called data that is passed in to the Run method is of type BlAPI4A2.Database. This object contains, via a number of index indirections, a reference to the current field in the Blaise instrument. In the case of our AddressVerification router, this will always be a reference to the AddressStart field of a VerifyAddress subblock of an instance of an AddressVerification block.

Since we know that the AddressStart field is two levels down from the top of the AddressVerification block, we can use the Parent property twice to find the root of this instance of AddressVerification. Having this, we can refer to any element within the AddressVerification block by name relative to this location.

The first Blaise field that the Run method looks at is the VerifyAddress.AddressStart field.

```
if (this._data.get_Field(this.activeField.Parent.Parent.Name +
```

```
          ".VerifyAddress.AddressStart").Text != "1")
```

This field is used to indicate whether or not the external application has already been run for this particular instance or not. After the external application runs, the router sets this field to “1”. On subsequent calls to the router, the router looks at this field and if it is not “1” then the router will run the external application. If this field is already set to “1” that means the external application already ran and the router will simply return without running the external application.

The router uses XML files to communicate the input parameters to the application, and to retrieve the output results from the application. Before launching the external application, the router updates the input XML file. The specifics of the XML file are beyond the scope of this paper.

The external application is launched using the System.Diagnostics.Process class.

```
    System.Diagnostics.Process program = new
       System.Diagnostics.Process();
    program.StartInfo.FileName =
       @"Rti.Data.AddressVerification.exe";
    program.Start();
    program.WaitForExit();
```

The StartInfo.Filename property is set to the name of the exectuable to run and the Start method is called. The router then calls WaitForExit, which causes the router to block on the return of the external application. Other application spawning mechanisms could have been used. We chose this method primarily for simplicity and because we did not want the router or Blaise to receive focus until the external application has exited.

After the external application exits, the router reads the output XML file and loads the returned values into Blaise variables.

```
this._data.get_Field(this._activeField.Parent.Parent.Name +
   ".OutputAddress." + addressElement).Text =
   outputNode.InnerText;
```

The router sets the current field to ”1” to indicate the external application has been run once.

```
this._data.get_Field(this.activeField.Parent.Parent.Name +
   ".VerifyAddress.AddressStart").Text = "1";
```

Finally, the router moves the Blaise instrument to the next question.

```
          this._state.AlienRouterAction =
             BlAPI4A2.BlAlienRouterAction.blraNextQuestion;
```

When code execution exits the Run method, the router is closed down and control returns to Blaise.

5. Case Study: Alien Router for Randomization of Questions

Many Blaise instruments require randomization of questions in various ways. In this section, we present one example we implemented by using an alien router.

The requirements were to randomize the order of 6 groups of 3 questions each and also randomly populate fills for hypothetical people used in the questions. More specifically, we were given the following tasks:

- Randomize the order in which the 6 question groups appear
- Randomize the order of the 3 questions within each group
- Randomize the names used for the people in the questions (based on the language of the interview)
- Construct the text of the randomized questions depending on the gender accociated with the randomly selected name

Figure 6 illustrates the randomization of questions and question groups.

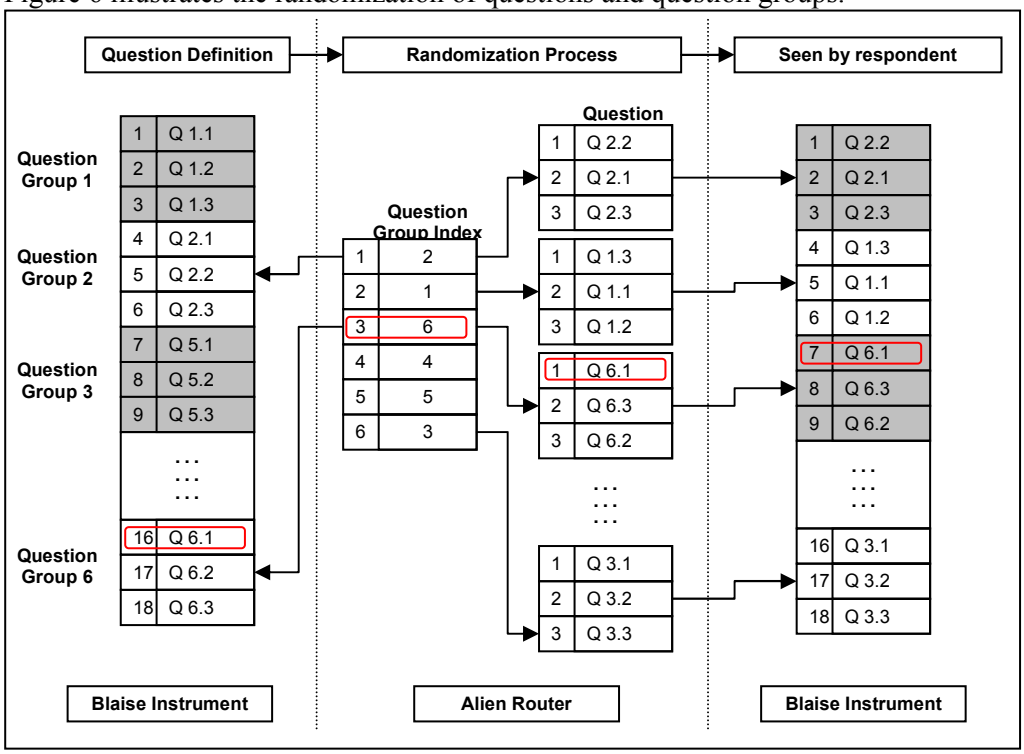

Figure 6 Question Randomization

Each question group is of a different Blaise type, but the number of response options is the same for all 18 questions. To construct the question text, we use a procedure in a Blaise instrument, while a VB.Net alien router calculates randomization numbers. The numbers are stored in the Blaise database where the Blaise instrument uses them to administer the questions in the correct order.

Several variables were added to the Blaise instrument for this purpose.

- Array RandomNumber[1..18] comtains an index into the question definition array. The index of RandomNumber itself is the order in which questions are asked. For example, if RandomNumber[7] contains the randomly selected value 16, then the seventh question asked of the respondent will be the 16th question defined, or question 1 from group 6 (labeled Q 6.1 in Figure 6).
- Array RandomName[1..18] contains the name that was used for the corresponding question. Continuing the example above, if RandomName[7] is "Daniel", then the name "Daniel" will be used in group 6, question 1.
- Array RandomGender[1..18] contains the gender for the corresponding name. In our example, RandomGender[7] is male.
- Array RandomField[1..18] actually asks the questions. Fields from this array are added to the route in the Blaise instrument and will be asked during an

interview. Text for the question and response options will be created "on the fly". The value entered in RandomField[7] in our example will be saved in a variable accociated with the first question 1 from group 6.

- Field RndSelection indicates the status of the call to the alien router. If all arrays defined above were filled with values, the router sets the value of the RndSelection field to "1" and will not be called again to calculate random numbers.

In the Blaise datamodel we add a type with fills for the response options. We create a procedure GetQuestAndType to create the response options ”on the fly”. Input parameters to the procedure are the question number, randomly selected name, and gender of selected name. Return values are the constructed question text and the type. In our example for the seventh question asked, the text for group 6, question 1 was created to use the name "Daniel". The resulting question asked of the respondent is shown in Figure 7.

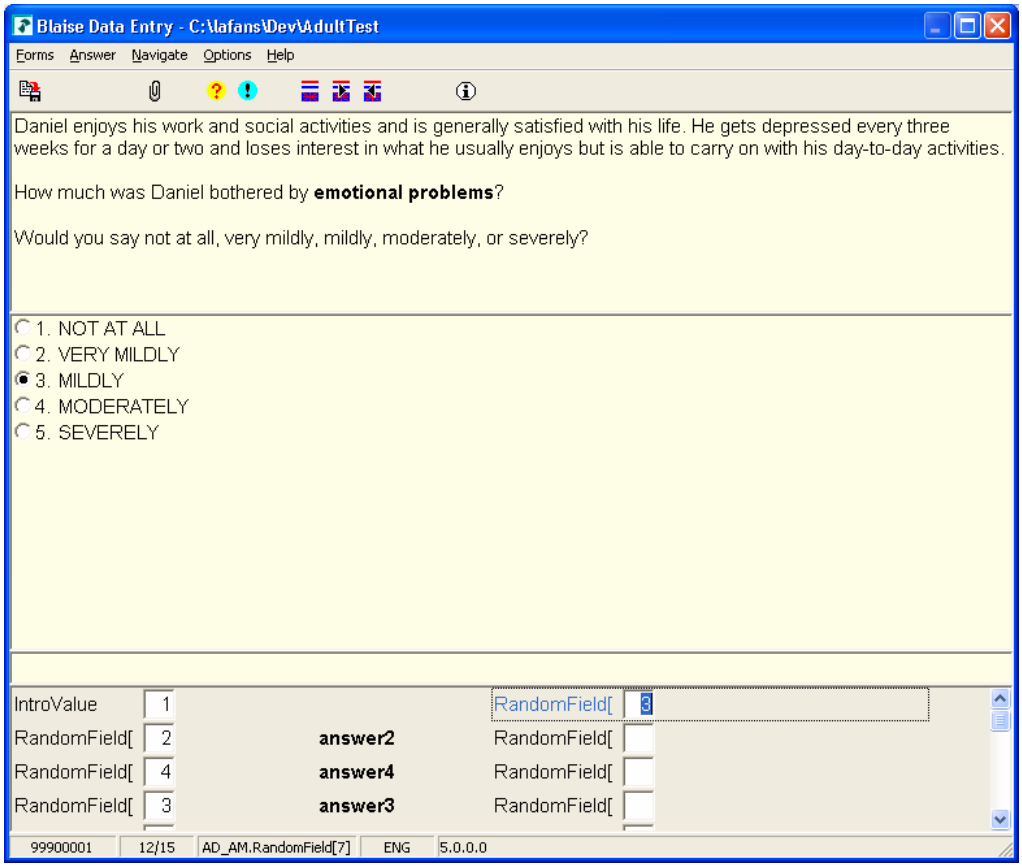

Figure 7 Randomization of questions

The values for the fields used in our example are shown in Figure 8. Analysts can later use these variables to reconstruct the order of questions and what exactly was asked.

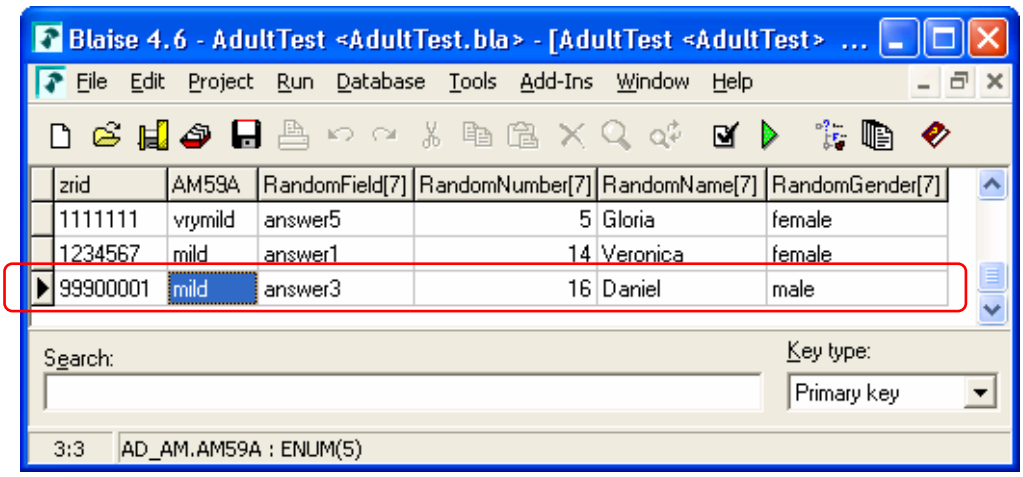

Figure 8 Randomization Data in Database browser

The Blaise source code is shown in Listing 3. See Listing 4 for a key excerpt from the VB.Net alien router.

6. Conclusion

If a study requires invocation of an external application during the Blaise interview, the following factors should be weighed:

- Does the application need to return data to the Blaise interview for future processing, and/or should it be stored in the Blaise database along with the interview data?
- Is complicated data processing needed?
- Does an external application need to be controlled?
- Is access to relational databases needed only for reading? For writing?
- Does collected data need to be in a format that is ready for analysis?
- How much time is available for development?

Depending on the requirements of a particular study, one of the discussed methods may be used. Our experience shows that many types of external applications can be integrated into Blaise surveys using these three methods.

Listing 1 Blaise Code for Address Verification Alien Router

```
BLOCK BAddressSearchElements
  FIELDS
    Street "What is the street address?": STRING[100], EMPTY
    City "What is the name of the city?": STRING[50], EMPTY
    State "What is the name of the state?": STRING[2], EMPTY
  RULES
    Street
    CITY
    { Call procedure to get a state from look-up table }
    GetState(State)
ENDBLOCK

BLOCK BAddressOutputElements
  FIELDS
    Street "QAS verified street address": STRING[100], EMPTY
    City "QAS verified city": STRING[50], EMPTY
```

```
    State "QAS verified state": STRING[2], EMPTY
    Zip "QAS verified zip": STRING[5], EMPTY

  RULES
    Street.SHOW
    City.SHOW
    State.SHOW
    Zip.SHOW
ENDBLOCK

BLOCK BVerifyAddress
  FIELDS
    AddressStart: 0..1, EMPTY

  ROUTER StartRouter
ALIEN('AddressVerificationRouter.AddressVerificationRouter','Run')

ENDBLOCK

BLOCK BAddressVerification
  FIELDS
    SearchAddress: BAddressSearchElements
    OutputAddress: BAddressOutputElements
    VerifyAddress: BVerifyAddress

  RULES
    SearchAddress.Keep
    IF SearchAddress.Street <> '' THEN
      VerifyAddress.AddressStart := EMPTY
      OutputAddress.Street := EMPTY
      OutputAddress.City := EMPTY
      OutputAddress.State := EMPTY
      OutputAddress.Zip := EMPTY
    ENDIF

    SearchAddress
    VerifyAddress.StartRouter
    OutputAddress

ENDBLOCK
```

Listing 2 C#.NET Code for Address Verification Alien Router

```
using BlAPI4A2;
namespace AddressVerificationRouter
{
  public class AddressVerificationRouter
  {

    private BlAPI4A2.Database _data;
    private BlAPI4A2.DepState _state;
    private BlAPI4A2.Field _activeField;
    private string BlaiseDBPath;
    private string LogFilePath;

    public void Run(BlAPI4A2.Database data, BlAPI4A2.DepState state)
    {
      try
      {
        // save the main Blaise objects as instance fields,
        //so they can be referenced in other methods
        this._data = data;
        this._state = state;

        //only handle BlAlienRouterStatus. blrsPreEdit
        if (this._state.AlienRouterStatus !=
```

```
            BlAlienRouterStatus.blrsPreEdit) return;

        // get the current directory of the Blaise database
        this.BlaiseDBPath = this._data.DataFileName.Substring(0,
              this._data.DataFileName.LastIndexOf(@"\"));
      CreateLogFilePath();

      // retreive the current field
      this.activeField = this._data.Screens.LayoutSetCollection
          [this._state.LayoutSetIndex].ParallelCollection
          [this._state.ParallelIndex].StoredPageCollection
          [this._state.StoredPageIndex].QuestionCollection
          [this._state.QuestionIndex].Field;


      // detect if the user is trying to move forward
      // or backward through the survey and call the
      // MoveForward or MoveBack routine acordingly.
      // The local blaise field AddressStart is either
      // blank or zero when moving forward, and is a 1
      // when moving back.
      if (this._data.get_Field(this._activeField.Parent.Parent.Name
+
          ".VerifyAddress.AddressStart").Text != "1")
      {
          // create the input file
          Log("Before BuildInputXml");
          this.BuildInputXml();

          // start the address verification program
          Log("Before StartVerificationProgram");
          this.StartVerificationProgram();
          Log("After StartVerificationProgram");

          // read the output and write to Blaise fields
this._data.get_Field(this.activeField.Parent.Parent.Name +
   ".OutputAddress." + addressElement).Text =
   outputNode.InnerText;

this._data.get_Field(this.activeField.Parent.Parent.Name +
   ".VerifyAddress.AddressStart").Text = "1";

          // proceed to the next question
          this._state.AlienRouterAction =
            BlAPI4A2.BlAlienRouterAction.blraNextQuestion;

      }
      }
      catch (Exception Ex)
      {
        this.Log(Ex.Message);
      }

    }


    private void CreateLogFilePath()
    {
      LogFilePath = BlaiseDBPath + @"\Log";
      DirectoryInfo di = new DirectoryInfo(LogFilePath);
      if(!di.Exists) di.Create();
      LogFilePath += @"\";
      //name the log file according to the class name
      //Pull out the class name
      LogFilePath +=

this.ToString().Substring(this.ToString().LastIndexOf(@".")
          + 1);
      LogFilePath += @"_Log.txt";
```

```
    }

    private void Log(string message)
    {
      StreamWriter writer = new StreamWriter(LogFilePath,
         true);
      writer.WriteLine(DateTime.Now.ToString() + ": " +
         message);
      writer.Flush();
      writer.Close();
    }

    }
}


private void StartVerificationProgram()
{
  try
  {
    System.Diagnostics.Process program = new
         System.Diagnostics.Process();
    Log("Changing directory to " + @"c:\test");
    Directory.SetCurrentDirectory(@"c:\test");
    program.StartInfo.FileName =
         @"Rti.Data.AddressVerification.exe";
    program.Start();
    program.WaitForExit();
    Log("Changing directory to " + BlaiseDBPath);
    Directory.SetCurrentDirectory(this.BlaiseDBPath);
  }
  catch (Exception Ex)
  {
    this.Log(Ex.Message);
  }

}
```

Listing 3 Blaise Code for Randomization of Questions

```
BLOCK BCallRouter
   FIELDS
     IntroValue: 1..1 {STRING[1], empty}
     ROUTER CallRandom ALIEN('Random.RandomRun','Run')
ENDBLOCK
TYPE
   TRandomField= (answer1 (1) "^Ttext[1]"
                , answer2 (2) "^Ttext[2]"
                , answer3 (3) "^Ttext[3]"
                , answer4 (4) "^Ttext[4]"
                , answer5 (5) "^Ttext[5]")
   INCLUDE "GetQuestAndType.prc"
   INCLUDE "AD_AMFields.inc"
   FIELDS
        RndSelection "Random selection made": 1..1
        RandomField "^Qtext[i]": array[1..18] of TRandomField
        RandomNumber : array[1..18] of integer
        RandomName : array[1..18] of string
        RandomGender : array[1..18] of Tgender
  LOCALS
    I, J, K, L : INTEGER
  AUXFIELDS
```

```
    Qtext: array[1..18] of TstrLong
    Ttext: array[1..5] of string
RULES
  NEWPAGE
  { Call the alien router to randomize order of AM54a-AM59c }
  AM54A_INTRO.CallRandom
  FOR i:= 1 TO 18 DO
    RandomNumber[i].keep
    RandomName[i].keep
    RandomGender[i].keep
    GetQuestAndType(RandomNumber[i],RandomName[i],
           RandomGender[i],Qtext[i])
    RandomField[i].ASK
  ENDDO
 { Initialize values in the corresponding fields }
 FOR j:=1 TO 18 DO
   IF  RandomNumber[j]=1 THEN
     IF RandomField[j] = RESPONSE THEN
        AM54A := ord(RandomField[j])
     ELSEIF RandomField[j] = DK THEN
        AM54A := DK
     ELSEIF RandomField[j] = RF THEN
        AM54A := RF
     ENDIF
   { ... }
   ELSEIF RandomNumber[j]=18 THEN
     IF RandomField[j] = RESPONSE THEN
        AM59c := ord(RandomField[j])
     ELSEIF RandomField[j] = DK THEN
        AM59c := DK
     ELSEIF RandomField[j] = RF THEN
        AM59c := RF
     ENDIF
   ENDIF
 ENDDO
```

Listing 4 VB.Net for Randomization of Questions in Alien Router

```
  'Read names from external file
  LoadNameAndGender(intTotalNames, Names, BlaiseDBPath)
  'Initialize random-number generator
  Randomize()
  'Populate array with randomly selected numbers for names
  PopulateArrays(rndName, usedName, 17, 18)
  'Populate array with randomly selected groups
  PopulateArrays(rndGroup, usedGroup, 5, 6)
  'Populate array with randomly selected questions in group
  For i = 0 To 5
    PopulateArrays(rndQuestion, usedQuestion, 2, 3)
    For k = 0 To 2
      rndField(i * 3 + k) = (rndGroup(i) - 1) * 3 +
                            rndQuestion(k)
      rndQuestion(k) = 0
      usedQuestion(k) = False
    Next
  Next
  'Write data into Blaise database (array: order of
questions)
  For i = 1 To 18
    sField = BlkName & ".RandomNumber[" & i.ToString & "]"
    db.Field(sField).Text = rndField(i - 1).ToString
  Next
```

```
'Get Active language variable
If sDep.LanguageIndex = 1 Then  'English
   Lang = 1
ElseIf sDep.LanguageIndex = 2 Then  'Spanish
   Lang = 2
End If
'Write data into Blaise database (arrays: names and gender)
For i = 1 To 18
  GetNameAndGender(Lang, Names, rndName(i - 1) - 1,
                   strName,intGender)
  sField = BlkName & ".RandomName[" & i.ToString & "]"
  db.Field(sField).Text = strName
  sField = BlkName & ".RandomGender[" & i.ToString & "]"
  db.Field(sField).Text = intGender.ToString
Next
```