Case Scenario Library for Testing Large Blaise Applications

Rhonda Ash and Jason Ostergren, University of Michigan

1. Introduction

In the last couple of decades, the use of computers for conducting interviews has allowed an increasing amount of complexity and customization to be achieved in the mechanics of a survey instrument without necessitating a corresponding increase in the burden on the interviewer or respondent. Because of this trend, it is now possible to design legitimate specifications that nonetheless overwhelm our ability to develop reliable survey instruments using the presently available tools. The Health and Retirement Study (HRS) at The University of Michigan has such a design. In 2004, the average HRS respondent was asked 425 questions out of a possible 11,745 over the course of 87 minutes. These figures were the result of approximately 76,000 conditional statements and 220,000 active lines of code in the logic of the instrument. Variation did not simply extend to the logic, however. HRS made use of what may be an unprecedented number of “fills” which had the potential to customize each of these questions to take into account a variety of factors such as extensive preloaded data that HRS uses, or proxy wording, or deceased respondents. All of this in a survey development environment (Blaise) which includes no debugger and scarce resources with which to test in any but the most time consuming manual fashion. In order to cope with the contest between complexity of design and reliability in programming, HRS developed a number of tools designed to improve debugging and testing in a Blaise environment with the goal of achieving an appropriate level of reliability. This paper will trace the evolution of this set of tools and then dwell in detail on the latest, a library of “scenarios” that combines and builds on them.

2. HRS Overview

The HRS is a complex longitudinal survey covering a wide range of health issues and economic concerns related to aging and retirement. The average interview lasts more than an hour and is administered to over 22,000 respondents. The HRS began in 1992 and had been carried out for three waves at two-year intervals before it was merged in 1998 with its sister study AHEAD, which covered an older cohort and a somewhat different content area. At that time two new age cohorts were added, and a “steady-state” design was born which would avoid eventual obsolescence by bringing in new cohorts at regular intervals (another was added in 2004). That change dramatically increased the capacity of HRS to address the effects of events and policy changes within its scope. It also allows HRS to aspire to a very long lifespan with continual expansion and adaptation of its content as these changes occur.

The complexity of the design is compounded by the need for multiple language types, not only for Spanish interviews, but also for the special wording needed for proxy interviews. HRS conducts proxy interviews with living respondents when necessary and also attempts up to two exit interviews in waves following the death of a respondent. In 2002, the exit interview, which had previously been programmed as a separate instrument, was merged into the main instrument with a resulting increase in scheduling efficiency at the expense of yet more complexity in the code.

Further, the code for the HRS instrument is rather unwieldy due to a host of other factors. For example, HRS makes use of a large amount of preloaded data (as many as 400 variables) particularly for family members and co-resident people who may be helping the respondent or influencing the respondent's life financially. Most importantly, the spouse or partner of the respondent, although interviewed separately, nonetheless has much of their information loaded for confirmation purposes. These data are fed into a series of tables at the start of the interview. This poses a particular problem because no other content can be reached and thus tested without first passing through a series of elaborate tables and sequences based on this preload. The main content of the interview is divided into approximately two dozen sections, each of which need to be programmed separately. This is because we rely on different sets of analysts and researchers to determine what additions, deletions, and changes will be implemented in each section for each new wave. Changes typically involve alterations to question text or to the 7,000+ fills in order to clarify or customize the language. Additionally, substantial changes to the logic of these sections are frequently requested in order to add new content, save interview time by targeting questions better, or accomplish any number of other goals.

3. HRS Testing Tools

At this point, a sense of the scope of the problem of ensuring the reliability of the HRS instrument should be emerging. In 2001, the study was in the middle of a transition from SurveyCraft to Blaise as the survey programming language. What had been a well-established set of code had to be set aside along with all of the tools that the staff had become accustomed to for checking the integrity of the instrument. Though Blaise is a conventional-looking language, with moderately customizable tools and a decent application programming interface (API), it had shortcomings, particularly for a survey the size of HRS, which became apparent during that transition year.

Many of these early difficulties tended to result in greater complexity. Performance problems are a good example of this. Surveys ran very slowly at first because large amounts of data derived from preload had to be accessible to different parts of the instrument and generated parameters proliferated as a result. This played havoc with the selective checking mechanism and caused the time gap between questions to become unacceptably long. Eventually these problems were solved as project staff became more familiar with Blaise, but the solutions, such as adding copies of arrays or requiring certain sections to be locked out after completion, added further to the complexity of the instrument. And as complexity increased, so did the difficulty of testing the instrument.

Off the shelf, Blaise lacks the sort of debugging and testing capabilities and systems that are needed for a complex survey like HRS. Foremost among the testing needs of HRS was some way to run an interview with real or simulated preload. Without this capability, it was nearly impossible to determine whether the new Blaise code being written would work in a real-world environment, especially given the variety of different interview patterns which are determined by preload. Since Blaise uses a proprietary database technology that was (and remains) difficult for data processing staff without extensive programming knowledge to access, some sort of case management system was needed that could read preload in a standard database format and then load it into the Blaise database, at the same time performing a series of manipulations required by the HRS design. A program called SurveyTrak, developed for field use by The University of Michigan, proved too cumbersome to use during development because the design precluded the study staff from updating preload or Blaise code with a quick turnaround.

In the end, HRS began to develop its own case management programs which allowed for easy manipulation of preload and quick release of code fixes for testing. Without this tool, it is difficult to imagine the 2002 wave of HRS having made it into the field in a timely manner without a disastrous number of errors and bugs. Also, without this basic capability, some of the other important tools that will be discussed here would not have been possible.

As background for the Scenario Library tools that are the subject of this paper, what follows is an explanation of the development of HRS testing applications, particularly in the wake of the 2001-2002 development period. Once that first Blaise version of the survey was in the field, and a fair number of subsequent updates had been issued, we assessed the effectiveness of our programming and testing practices and determined that there were a number of areas where considerable improvement could be realized by the development of additional testing applications. There were two main issues that needed to be addressed at that time: the quality of error reporting and the ability to reproduce reported errors.

The solution to the reporting problem was the simpler of the two. Already in 2001, HRS had an error reporting database where problems were collected. However, the quality of these reports was decidedly mixed because it was completely manual. It was often unclear where in the instrument an error had appeared and what circumstances had caused it to appear. As a result, programmers and testers typically expended enormous amounts of time repeatedly attempting to key their way to the point in the instrument where the problem was alleged to have occurred. Oftentimes, things such as unreported alterations to preload by the reporter conspired to make it very time-consuming to find the reported error. Vague error reports are certainly a common problem, but in survey programming they should be far less burdensome than in other areas of software development. Blaise is a narrowly focused scripting language that is solely geared towards surveys, and the programmer and user of a Blaise instrument operate in a very circumscribed environment. It therefore ought to be quite easy to identify and reproduce the actions which produce an erroneous behavior. The first step toward this goal was to automate the reporting process. Using the Blaise API and the menu functionality of the Blaise data entry program (DEP), HRS produced a tool using Visual Basic 6 that would collect metadata directly from the DEP and send it to the bug database along with the audit trail file from the interview in progress. This would occur when the user selected an option that we added to the DEP menu, and at the same time a window would appear to prompt the reporter for a description as well.

This brings us to what has become the quintessential HRS testing tool which was the subject of an HRS paper at the 9th International Blaise User's Conference: a keystroke replayer. The Survey Research Center at The University of Michigan has been very interested in audit trails for some time from both a research perspective: what can be learned from "paradata," and from a quality assurance perspective: that is, audit trails can be used to monitor interviewer performance. HRS hoped to use them for another form of quality assurance: testing and debugging. The idea was to collect audit trails from error reports, as described above, or from other testing activities. A separate application would then attempt to reproduce the interview that generated the audit trail when a programmer had need of it, and possibly again later for a tester to verify that a fix, for example, was working. What is more, it was hoped that this would aid in other testing needs, such as language testing, by allowing certain cases to be replayed under different conditions. This keystroke replayer was developed in Visual Basic 6, and turned out to be the most

elaborate of all the HRS tools. In order to function, it required a number of separate programs including the DEP, all running in concert and maintaining communication and synchronization through Windows API calls as described in our previous conference paper. This program was completed in 2002, and did indeed produce a noticeable improvement in the quality and efficiency of debugging and testing during subsequent development periods.

However, there were a couple of problems with our otherwise successful keystroke replayer. First, new versions of Blaise from time to time bring with them small changes to the DEP that require reworking of the program. Second, due to the complicated nature of what we were doing, it proved difficult for some of the less technical users to become comfortable with the replayer. As a result, in late 2004, HRS embarked on a project to entirely rewrite the suite of testing tools and add new automated testing features.

One part of this process was to build a new case manager. The new version was written in a .NET language (C#) in order to take advantage of the .NET libraries and its better handling of objects. The most important feature of this rewrite is that it can easily switch among a variety of preload data storage formats including SQL Server, MS Access, and XML. Our practice now is to store preload in a common permanent SQL Server database in which new records (often modified copies of old ones) are allowed, but the updating or deleting of previously existing records is not. Temporary changes to preload are allowed when dumped to XML, because such files can be preserved for later reference. What this has allowed us to do is to track every preload change made by a user, thus eliminating a major source of variability in reproducing problems. Now, for example, with each error report, a copy of preload in XML format is stored along with the audit trail in our SQL Server bug database. This has actually had a dramatic effect on our testing efforts because testers have become less constrained when altering preload, allowing them to more fully evaluate the ability of the instrument to cope with different sets of preload.

The most important new features of the testing system were built in answer to the user-friendliness problems of the keystroke replayer. Since the DEP was not easy to work with from a programming perspective, HRS pursued two new strategies to supplement the replayer. They both rely on the fact that the Blaise API exposes the functions of the DEP, so it is possible to simulate its operation and remove the actual DEP from the equation altogether. First, we added an option to simulate a replay in the background, which has proved very popular with users due to its ease of operation. Second, a rudimentary DEP was built in VB.NET that used the API and displayed a user interface that could be manipulated at will. Both of these solutions eliminated the need to "send" keystrokes and query the DEP to maintain synchronization. As usual, there were problems despite the good reception of the new tools. We discovered that in certain obscure situations, the Blaise API does not operate precisely the same as the DEP. Statistics Netherlands was notified and they confirmed that we had discovered a new bug. A future resolution of this issue, resulting in a high degree of confidence that the behavior of the API matches that of the DEP, could even lead to the development of a full-fledged custom-built DEP. Despite these minor issues, the new tools have yielded even better results in 2006 than were obtained in prior waves. Overall, in the absence of built-in debugging features in Blaise, these tools have allowed HRS to release a complex survey instrument every two years with reasonable confidence in its reliability.

4. Scenario Library

Prior to the 2006 development period, work began on another testing tool to address the long-held desire for a way to handle HRS "scenarios" in an automated way. In HRS parlance, scenarios are constellations of preload data and response patterns that correspond to a particular category of respondents. For many years, these were developed by reading the specifications and hand-entering combinations of preload and response values into spreadsheets describing each significant category of respondents in each section. A tester was then required to interpret these scenarios and enter the values into the instrument in order to verify its accuracy. In combination with many of the tools and techniques described in the section above, HRS has developed a new set of tools to automate this process. This is described in detail below.

5. Scenario utilities

5.1 Overview and Purpose

With a multi-leveled interviewing application, including language controls, subject content, multiple interview modes, and re-interview constraints, it is very important to be able to test each dimension. Keystroke files provided by the Blaise process allow us to validate the application in a systematic manner. HRS has created a mechanism to store those files in a library that can recall them at a later date, alter the original test, and quickly guarantee instrument accuracy.

HRS was able to reduce error difficulties by conceiving of and developing testing utilities that combine into a system called the HRS Case Manager (HRS-CM). Together, these programs allow testers to make use of stored keystroke files that are produced as a by-product of Blaise test interviews. These files are categorized and stored for reapplication.

To ensure maximum coverage and reduce the time needed for testing, the keystroke files for specific scenario paths are given identifying information such as a scenario number, mode of interview, section of test focus, and several major variables that control flow. HRS commonly refers to the course that an interview follows through an instrument as a "path" or as the "flow."

5.2 Implementation

There are three components to the testing process, each requiring different ways of handling applications: 1) preload – information that we have collected in the past about a respondent's finances, family or medical history that needs to be brought forward into this wave's data and updated, 2) a Blaise datamodel – the program specifications that allow the collection of data, and 3) a scenario - the combination of preload, path, and content though the instrument that is later used for verification of accuracy. We will detail each component and describe the mechanism designed to incorporate it into a coherent system.

5.3 Preload process lifecycle

Preload can determine the path taken through the application. As an example, consider a respondent we have talked to before from whom we have collected baseline information

versus a respondent that we have never interviewed. These two respondents would take different, but overlapping paths through the application. We would not need to ask the baseline questions of the re-interview respondent, but we would need to ask them of the new respondent.

There are several major flow control fields that greatly influence the flow of the HRS application. These were considered when a preload library was developed. A set of common preload values were created from real data. These cases were then altered and any identifying data was changed. This set, which numbered around one hundred cases, became the basis for the HRS_CM scenarios. As the testing cycle advances, the testers have made additions to the preload library for specific testing needs.

These data are currently stored in several related tables on an SQL server, and accessed through a series of programs that allow update and retrieval for usage. Below we provide an example of a preload table with each row representing part of a case. Each case includes data for most of the 400+ variables.

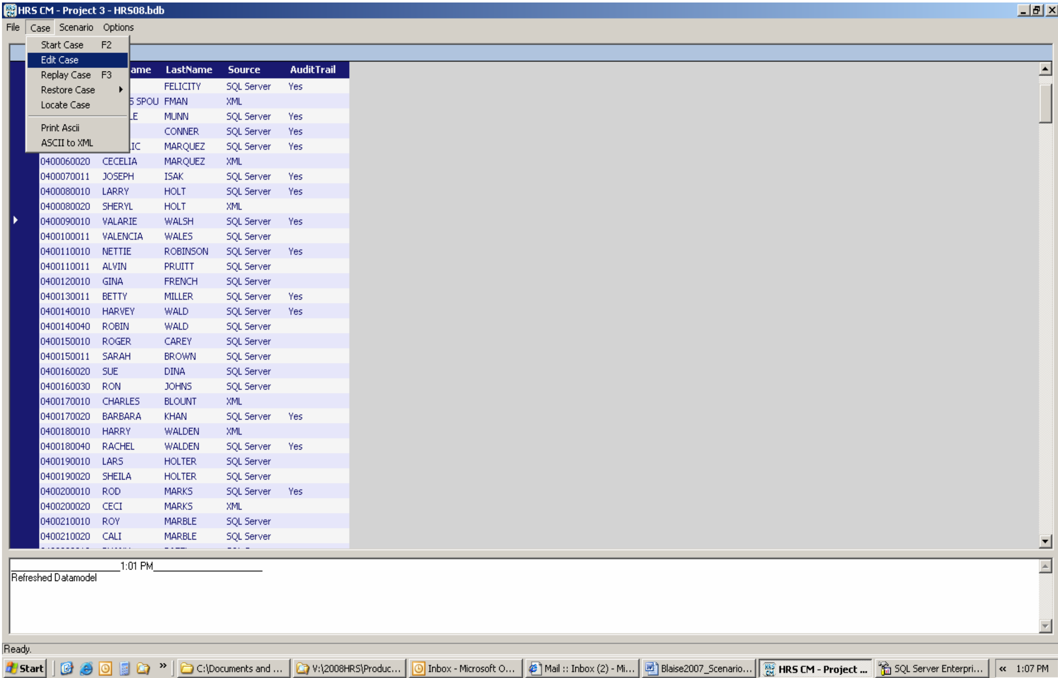
SQL Server Enterprise Manager - [Data in Table ' SSQL']

File Window Help

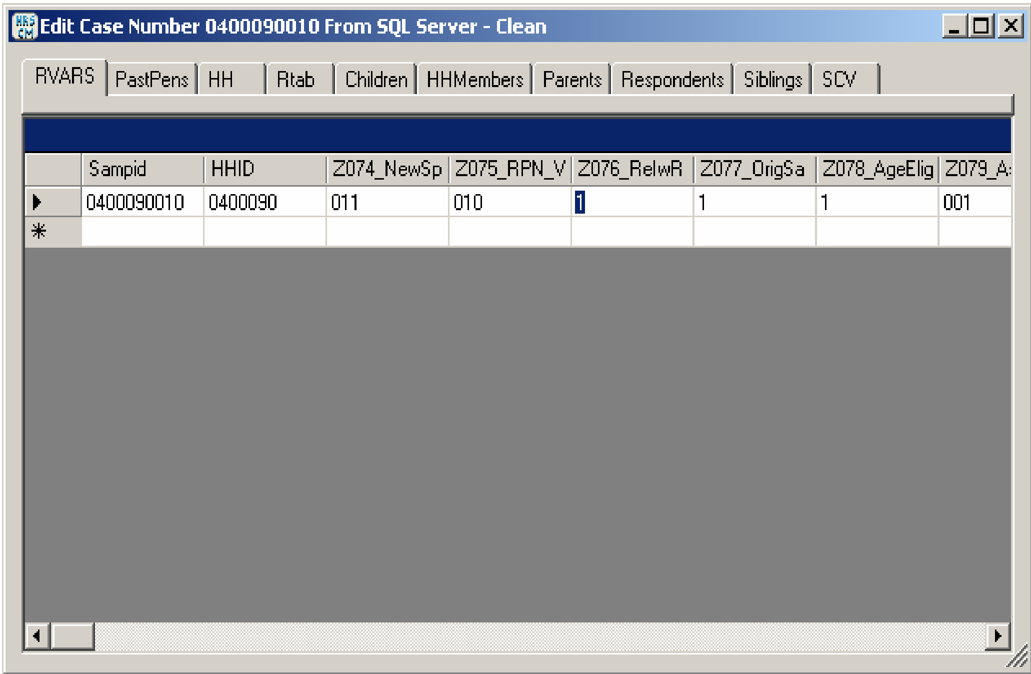
Sampid	HHID	Z074_NewSpSeed	Z075_RPN_V	Z076_ReIwR_V	Z077_OrigSamp_V	Z078_AgeElig_V	Z079_AskDOB_V	Z080_MarStat_V	Z081_SpPDie_V	Z082_WaveId_V	Z083_UnfMedCost	Z084_UnfNHHosp	Z085_L
0401130010	0401130	011	010	1	1	1	001	1	5	0400810010	3	1	1
0401130040	0401130	041	040	1	1	1	001	1	5	0400810040	2	2	2
0401140010	0401140	011	010	5	1	1	001	4	5	0400900010	2	1	3
0401150010	0401150	012	010	1	1	1	110	1	5	0400620010	3	1	3
0401150011	0401150	<NULL>	011	5	5	1	111	1	5	0400620011	2	2	2
0401160010	0401160	011	010	1	1	1	001	1	5	0400540010	2	3	1
0401160020	0401160	021	020	1	1	1	001	1	5	0400540020	2	2	2
0401170010	0401170	011	010	1	1	<NULL>	001	<NULL>	5	0400400010	3	3	2
0401170020	0401170	021	020	1	1	<NULL>	001	<NULL>	5	0400400020	1	3	3
0401180010	0401180	011	010	1	1	<NULL>	001	<NULL>	5	0401170010	3	3	2
0401180020	0401180	021	020	1	1	<NULL>	001	<NULL>	5	0401170020	1	3	3
0401190011	0401190	012	011	1	5	1	001	5	5	0400100011	3	3	1
0401200010	0401200	011	010	1	1	1	001	5	5	0401120010	3	3	1
0401210010	0401210	011	010	1	1	1	001	5	5	0401200010	3	3	1
0401220010	0401220	011	010	1	1	1	001	3	5	0400970010	1	3	3
0401230010	0401230	011	010	1	1	1	001	5	5	0400020010	3	3	1
0401240010	0401240	011	010	5	1	1	001	2	5	0400060010	3	3	2
0401240020	0401240	021	020	1	1	1	001	3	5	0400060020	1	3	3
0401250010	0401250	011	010	5	1	1	001	2	5	0400060010	3	3	2
0401250020	0401250	021	020	1	1	1	001	3	5	0400060020	1	3	3
0401260010	0401260	011	010	5	1	1	001	2	5	0400060010	3	3	2
0401260020	0401260	021	020	1	1	1	001	3	5	0400060020	1	3	3
0401270010	0401270	011	010	5	1	1	001	2	5	0400060010	3	3	2
0401270020	0401270	021	020	1	1	1	001	3	5	0400060020	1	3	3
0401280010	0401280	011	010	1	1	1	001	4	5	0400030010	2	1	3
0401290010	0401290	011	010	5	1	1	001	4	5	0400010010	2	1	3
0401300010	0401300	011	010	1	1	1	001	1	5	0400140010	3	1	1
0401300040	0401300	041	040	1	1	1	001	1	5	0400140040	3	3	3
0401310010	0401310	011	010	1	1	1	001	4	5	0400010010	2	1	3
0401320010	0401320	011	010	1	1	1	001	3	5	0400050010	1	3	3
0401330010	0401330	012	010	1	1	1	001	4	5	0400280010	2	2	1
0401330011	0401330	<NULL>	011	1	5	5	001	<NULL>	5	0400280011	1	1	1
0401340010	0401340	011	010	1	1	1	001	1	5	0400220010	2	3	1
0401340020	0401340	021	020	1	1	1	001	1	5	0400220020	2	2	2
0401350010	0401350	011	010	1	1	<NULL>	001	<NULL>	5	0400080010	3	3	2
0401350020	0401350	021	020	1	1	<NULL>	001	<NULL>	5	0400080020	1	3	3
0401360020	0401360	021	020	1	1	1	001	<NULL>	5	0400230020	3	3	3
0401360030	0401360	031	030	1	1	1	001	1	5	0400230030	1	1	1
0401370010	0401370	011	010	1	1	1	001	2	5	0400060010	3	3	2
0401370020	0401370	021	020	1	1	1	001	3	5	0400060020	1	3	3
0401380010	0401380	011	010	1	1	1	001	3	5	0400050010	1	3	3
0401390010	0401390	011	010	1	1	1	001	5	5	0400020010	3	3	1
0401400010	0401400	011	010	1	1	1	001	1	5	0400220010	2	3	1
0401400020	0401400	021	020	1	1	1	001	1	5	0400220020	2	2	2

When a case is called into use these data are copied into eight tables that have matching structure in the instrument prepared for this preload. This structure is ready to receive data.

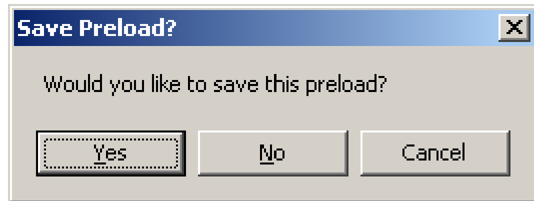
Below is a sample of the available preload records (or cases) from the SQL server that has been saved in the HRS_CM system. This is the main screen from which the various testing activities can be activated.



By selecting a case from the HRS_CM system and choosing EDIT from the menu options, a tester is given the ability to view the preload values and alter data to perform a specific test as shown here.



If a tester has altered the preload for a case and will need that preload setup for future testing, they have the ability to save it as a new record and add it to the permanent preload list. If they have a specific need that would never have a broader applicability, they can choose to save it only to their testing folder on the network instead of saving it permanently to the SQL Server. A dialog box will appear asking them for their choice.



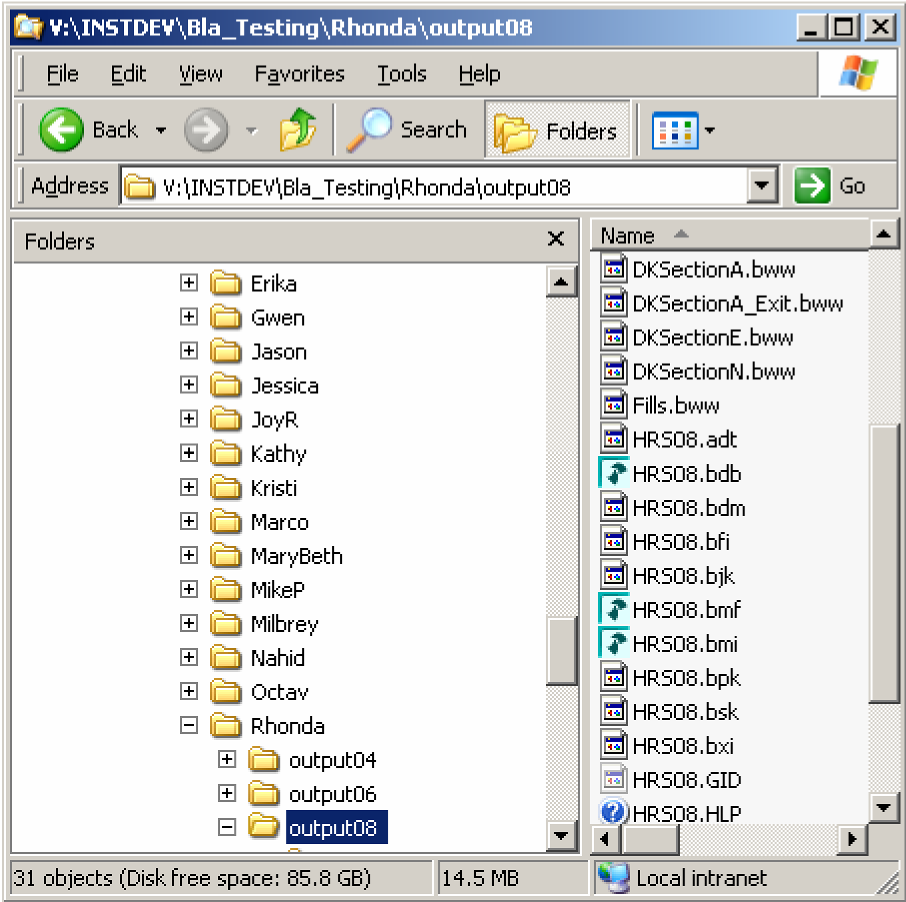
The temporary storage is done through the use of an XML file produced by this system (see below). This file contains changed and unchanged preload values.


```
<RVARS>
   <Sampid>0400430010</Sampid>
   <HHID>0400430</HHID>
        <Z074_NewSpSeed_V>012</Z074_NewSpSeed_V>
        <Z075_RPN_V>010</Z075_RPN_V>
        <Z076_RelwR_V>1</Z076_RelwR_V>
        <Z077_OrigSamp_V>1</Z077_OrigSamp_V>
        <Z078_AgeElig_V>1</Z078_AgeElig_V>
        <Z079_AskDOB_V>001</Z079_AskDOB_V>
        <Z080_MarStat_V>4</Z080_MarStat_V>
        <Z081_SpPDie_V>5</Z081_SpPDie_V>
        <Z082_WaveId_V>0055120010</Z082_WaveId_V>
        <Z083_UnfMedCost_V>2</Z083_UnfMedCost_V>
        <Z084_UnfNHHosp_V>2</Z084_UnfNHHosp_V>
        <Z085_UnfOthMedCos_V>1</Z085_UnfOthMedCos_V>
        <Z086_UnfPrescript_V>3</Z086_UnfPrescript_V>
```

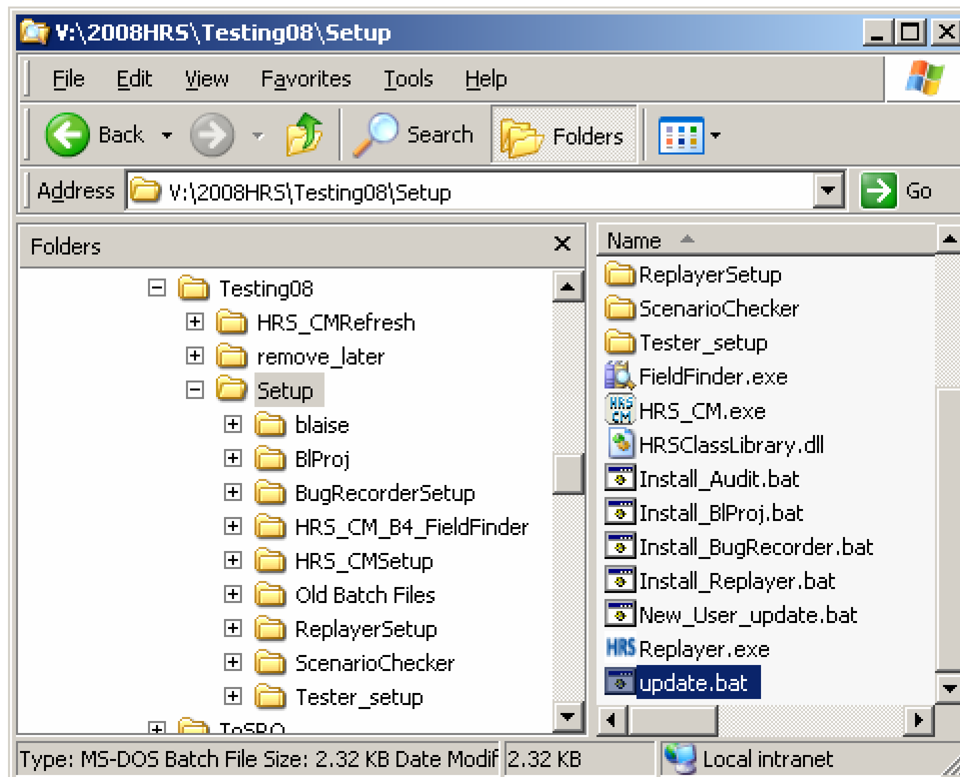
5.4 Datamodel process lifecycle

As the development phase progresses, changes and corrections are made to the Blaise source code. The programmer re-compiles and tests, then distributes the datamodel update for more extensive checking by testers. HRS has several programmers working at the same time in the application. Because of this the datamodels are time-stamped. This allows the testers to take note of when the datamodel was last updated and know when an issue is ready for re-testing.

HRS_CM is installed in a series of folder structures on a network. This structure allows each tester to have their own space, within which they can create their own testing environment without affecting other testers. This space includes saved watch-window settings associated with the tester's content area allowing quick confirmation of field content while testing, previous temporary preload files and played scenarios, along with the latest datamodel and instrument HELP file.



With a series of batch files that help move files from one location to another, we have been able to routinize the file updating processes. Below is an illustration of the folders used to set up a new tester with all needed files and a batch file that places files in the appropriate place and runs any installation application to activate programs.



Programmers also release new datamodels through the use of a batch file. This file moves all needed files to a "Refresh" folder that the testers can access to update the datamodel in their work folder. An example of that file is shown here.

```
@echo off
copy HRS08.bsk V:\2008HRS\Testing08\HRS_CMRefresh\HRS08.bsk
copy HRS08.bpk V:\2008HRS\Testing08\HRS_CMRefresh\HRS08.bpk
copy HRS08.bjk V:\2008HRS\Testing08\HRS_CMRefresh\HRS08.bjk
copy HRS08.bfi V:\2008HRS\Testing08\HRS_CMRefresh\HRS08.bfi
copy HRS08.bdb V:\2008HRS\Testing08\HRS_CMRefresh\HRS08.bdb
copy HRS08.bxi V:\2008HRS\Testing08\HRS_CMRefresh\HRS08.bxi
copy HRS08.bdm V:\2008HRS\Testing08\HRS_CMRefresh\HRS08.bdm
copy HRS08.bmi V:\2008HRS\Testing08\HRS_CMRefresh\HRS08.bmi
echo Copying of files is done!
```

Once the tester is in the HRS_CM system, the program references the master "Refresh" folder and compares the dates of the files with the files in the tester's own folder. If the two dates do not match, the tester is notified that they may need to "Refresh" (retrieve updated files) for their work folder.

All of this is to ensure that everyone involved remains in sync.

6. Scenario process lifecycle

6.1 Scenario creation and storage

As a first step in building a scenario in the HRS-CM system, the tester sets up an environment for a test by making any needed edits to the preload and checking that they have the latest datamodel. They then open the Blaise Data Entry Program (DEP) and enter keystrokes to correspond with the selected scenario. As a byproduct of the interviewing process, Blaise creates an .ADT file that contains all of the keystrokes required to reproduce that path. The tester can choose to save that ADT file into the Scenario database. To do this a Visual Basic procedure is called that opens that ADT file and loads its contents into a string in memory.

```
ADKFILE = globOutputFolder & "\" & globSampid & "." & globAuditExt
XMLFILE = globOutputFolder & "\" & globSampid & ".XML"
'saving the data stream
Dim wholetext, wholetext2 As String
wholetext = ""
Dim sr As StreamReader
If File.Exists(ADKFILE) Then
   sr = New StreamReader(ADKFILE)     'File IO
   wholetext = sr.ReadToEnd()
   sr.Close()
End If
wholetext2 = ""
If File.Exists(XMLFILE) Then
   sr = New StreamReader(XMLFILE)     'File IO
   wholetext2 = sr.ReadToEnd()
   sr.Close()
End If
```

Once loaded, the procedure connects to our SQL Server database, creates a new record and pushes the string from memory into the database. At this time, several questions are asked of the tester about the scenario being saved and the answers are stored. These questions include requests for a description of the scenario's purpose, a scenario code referencing attributes of the scenario, and a section letter that the scenario is focused on. The procedure also checks for any matching XML preload file in the tester's folder and saves it in the same manner. The code for this procedure is as follows.

```
    Cnn.Open("Driver={SQL
Server};server=SQL_SERVER;database=DevScenario;")
    rs.Open("select * from scenarios order by ScenarioID Desc", Cnn,
   ADODB.CursorTypeEnum.adOpenKeyset,
ADODB.LockTypeEnum.adLockPessimistic)
   rs.AddNew()
      If wholetext <> "" Then
         rs.Fields("ADKString").Value = wholetext
      End If
      If wholetext2 <> "" Then
         rs.Fields("XMLString").Value = wholetext2
      End                                                      If
```

6.2 Scenario retrieval and usage

Now that the preload environment and scenario keystrokes have been saved in the library, several other utilities were needed to allow testers to view descriptions and preload criteria, and to allow recall of a scenario that would satisfy a particular testing requirement. We describe several of these utilities below.

6.3 Scenario Finder

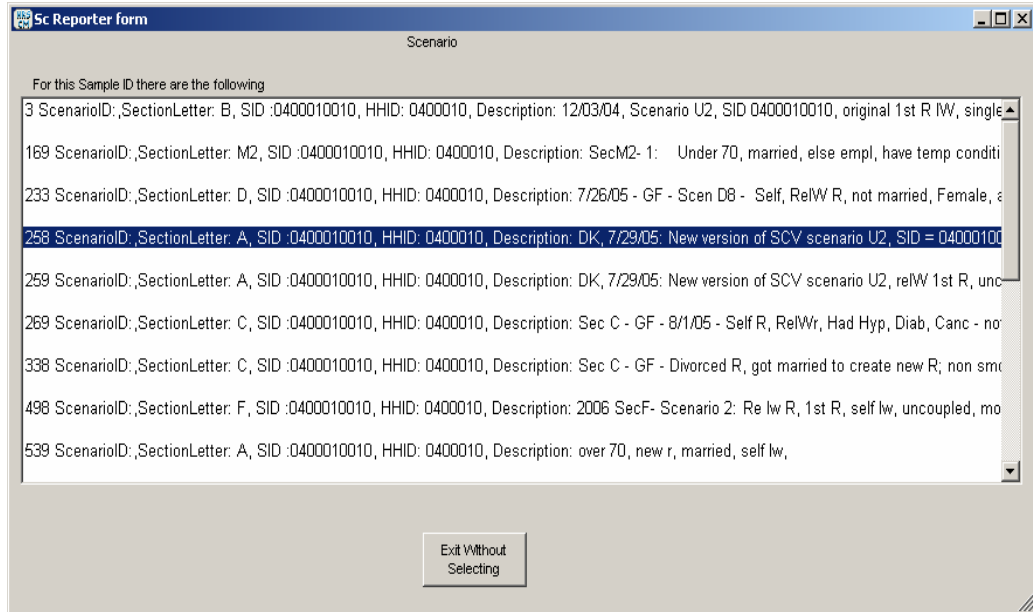
This utility allows the tester to recall a scenario based on specific criteria needed for their test. As noted previously, each scenario has a description, scenario code, and section letter attached to it, all of which allow the tester to quickly find a saved scenario that they wish to re-test.

Each attribute that was saved, such as user name or primary key, is available to the tester from a pull-down menu as a selection criterion. These options are used in the program that recalls the files from the database as shown.

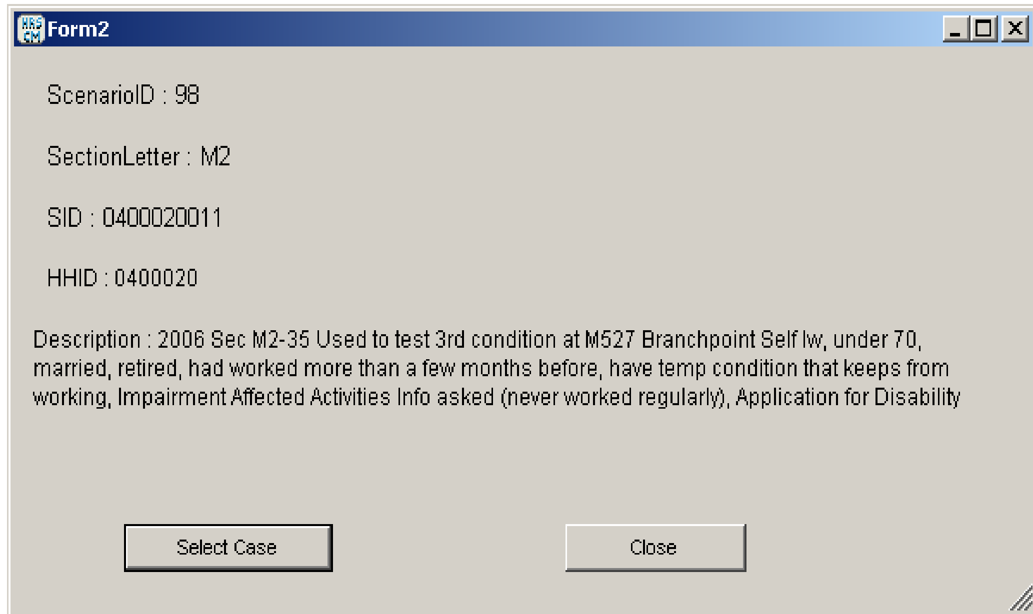
```
cnn.Open(globCnnScenario)

If globSearchType = SearchType.User Then
   rs.Open("select * from scenarios where CreatorName = '" &
    Environment.UserName & "' ",
   cnn, ADODB.CursorTypeEnum.adOpenKeyset,
    ADODB.LockTypeEnum.adLockPessimistic)
ElseIf globSearchType = SearchType.SID Then
   rs.Open("select * from scenarios where sid = " & globSampid & " ", cnn,
   ADODB.CursorTypeEnum.adOpenKeyset,
    ADODB.LockTypeEnum.adLockPessimistic)
Else
   rs.Open("select * from scenarios", cnn,
```

```
        ADODB.CursorTypeEnum.adOpenKeyset,
       ADODB.LockTypeEnum.adLockPessimistic)
    End If
```

In addition, it is possible to request scenarios based on whether a specified field or fields appear in it. The results are displayed in a form allowing the tester to select a scenario.



By highlighting a record the testers can view the recorded details of the scenario, seen in Form2.



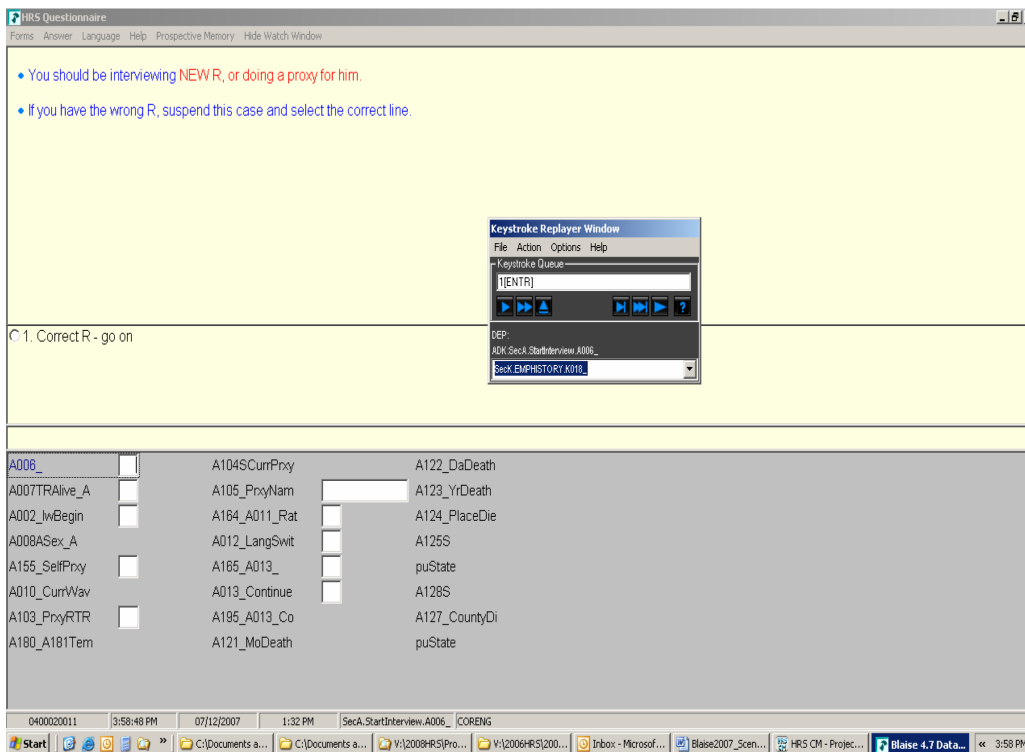
Once a record is selected, the ADT data is pulled into memory, and pushed into a text file, using the appropriate tester's file location. Below is the Visual Basic code that performs that task.

```
ADKFILEout = sAdkLocation & "\" & globSampid & "." & globAuditExt
 FileOpen(1, ADKFILEout, OpenMode.Output)
 PrintLine(1, rs.Fields("ADKString").Value)
 FileClose(1)

 If IsDBNull(rs.Fields("XMLString").Value) = False Then
    XMLFILEout = sAdkLocation & "\" & globSampid & ".XML"
    FileOpen(3, XMLFILEout, OpenMode.Output)
    PrintLine(3, rs.Fields("XMLString").Value)
    FileClose(3)
 End If
```

At this stage, the environment for the case has been recreated and the tester can either restore or replay the case.

6.4 Replayer

The Replayer mentioned earlier in this paper allows a tester to watch the keystrokes be re-applied to the data entry application and interact with the answers to questions, accepting them and moving on, or changing them on the fly.



Above is an image of the keystroke replayer window. The tester is able to replay to any point in the application, stop and alter keystrokes, then continue the interview,

monitoring the effects of their changes. Replaying saved scenarios can quickly ensure that a datamodel change has not affected other areas of the application. This is a major benefit for a questionnaire as large and complex as the HRS.

6.5 Restore

Quite often testers do not need to step through each keystroke individually, but rather need to reach a specific point in the application to re-test a sequence. The "Restore" utility allows the tester to select a location in the application and have all stored keystrokes applied using only the BCP in order to reach that point, without having to spend the time to walk through the entire interview.

7. Outcomes

We have moved from tracking and testing a limited set of scenarios that followed critical paths to being able to test flow and language, replicate data errors from the field, change and retest quickly, and share keystroke files for other testing needs.

Prior to these utilities being developed, we used spreadsheets to document major paths for testing. This technique had some major disadvantages. These became outdated quickly as the instrument was changed. Also, a tester might find a problem and often the programmer would not be able to replicate it, or a tester would not be able to test a path without the ability to change the preload appropriately. Given limited time, we were not able to completely test all modes of our application. As a result of incomplete testing, we often received reports from the field staff that there were problems with flow, screen text, or even "Hard" code errors that would stop the application and cause us to lose valuable field time. In 2004, HRS released nine revisions of the datamodel. In several instances interviewing had to stop while the program was corrected. In 2006, HRS issued only three field revision updates, of which two were planned in advance to update the Spanish language and to change the year reference. The ability for testers to save a critical path and then quickly re-test when changes have been made ensured a much higher degree of accuracy in the data collection application.

These utilities have benefited many parts of the instrument development process. For example, a translator can switch languages at any point in the application to check that the text for a given question is correct. Another benefit of this system is the ability to reproduce problems that are reported from the field. By recovering the .ADK and preload files from the field, we can place them in a tester's folder and run the same Replayer utilities to replicate the problem. This has proven to be a great time saver for getting quick resolution and turn-around with field issues.

Each section of the application now has a set of twelve or more saved scenarios within the library from each tester. These ensure that every part of the instrument is well tested. We now have the ability to select a variable and scan though the scenarios in the library looking for and reporting on any instance of that variable. This allows testers to find a scenario with the keystrokes that will get them to a specific location in the application. This has proven to be a great timesaver. Also, with this "field finding" ability we can check coverage of the scenarios in the library in relation to our real-time data and know that we are capturing the major paths in our testing.

The benefits go on. The long term effect of this library is that 2006 scenarios will be replayed on the 2008 datamodel after changes and updates have been made. This will ensure that no unintended consequences were introduced as the result of changes or corrections that were applied to some of the more difficult areas of the instrument.

These utilities have made it possible for us to shorten our pre-production cycle, allowing more time for investigators to incorporate the latest innovations. The testing staff has been able to reduce the amount of time they spend on testing and take on other responsibilities.

8. Conclusion

The project staff at HRS aims to produce a reliable and accurate survey instrument that mirrors the complex and continually evolving design of the researchers and to do so in an efficient and timely manner. The story of the development of the tools that HRS uses to achieve this goal is indicative of the complexity of instrument development on a survey of this scale. Over time, what has emerged is a substantial suite of applications that may be instructive for and even flexible enough to be used by other studies. It has certainly proved to be a fertile ground for the development and refinement of new and existing tools at HRS as the example of the Scenario Library described here illustrates. The Scenario Library builds on tools developed previously by combining them to produce a new functionality that has already proved quite valuable. We fully expect this tool to evolve and grow, like the others, according to the results obtained as it is used in practice. Ultimately, through the development of these tools, we aspire to continue to improve the reliability and accuracy of HRS with each successive wave, while becoming ever more flexible and efficient about incorporating new design ideas from researchers even when they are particularly complicated or given on short notice.