A Systematic Approach to Debugging in the Blaise Environment: An Author's Perspective

Peter Sparks, The University of Michigan

1. Introduction

There has been much written about testing Blaise instruments using a variety of systematic approaches, databases and logs of changes, version control, automated instrument creation, testing cycles, analysis utilities, and so forth. But what about the author writing programming code that follows the survey specifications? What tools and methods are available at the author's fingertips to make this development quick, robust, and accurate?

Blaise provides a rich environment for development of survey instruments, Manipula/Maniplus scripts, Cameleon, Basil, CATI, and CAWI. The author has access to the source code, metadata, test data, as well as the modelib, configuration files, datamodel properties, "watch window," menu files, keystroke files, Manipula debugger, and Blaise API (Blaise Component Pack), and so on. Additionally, DLLs (dynamic link libraries) can be developed to enhance the existing tools. For all of this, though, debugging can still present a challenge.

This paper presents methods for writing Blaise code (instruments and scripts) that will make later debugging easier, as well as a practical approach to using the existing tools to aid in debugging, testing, and the creation of new tools using the Blaise 4.8 environment.

2. Standards to Minimize Debugging

"A good offense is a good defense." The first steps to debugging should always be to reduce the amount of checking needed in the first place. The points below have been thoroughly covered in other materials and are just provided as a reference.

2.1 Clean and Standardized Specs

Ideally, every survey should have clear, standardized specifications that clearly define everything that is to be programmed. This is used by everyone involved and is the final authority for changes. In general, the specs should cover

1. Defined logic flow for all values including missing
2. Ranges defined
3. Checks/signals defined
4. Checkpoints defined
5. Fills defined
6. Question text layout
7. Help, booklet, and other references

2.2 Standardized Programming Conventions

The programmer, likewise, should follow standard coding conventions. Not only does this make it easier to isolate problems, but fellow programmers can more readily assist in case of a difficult problem without overcoming the additional burden of poor coding. This, of course, is not a complete list.

1. Source code formatting
2. File naming and content
3. Headers and comments
4. Variable naming
5. Block and Procedure use

6. Layouts

3. Write Easier-to-Debug Blaise Code

The steps needed to write better code are not hard, but do require discipline, and in fact heavily rely upon consistency. The points given in 2.2 are not trivial, but they do have tremendous results when done well.

In general, first follow the coding standards for your organization. Then apply these concepts as you are able within those confines. The items below are focused on Blaise as the programming language.

1. Create each section as a separate block and file
   Physically separating each top level section as its own file makes it easier to work on, track in version control software, and is a logical unit by itself. Do not put more than one section in the same file.
2. Put a comment header on each file. Include revision dates.
   This allows a revision history to be maintained, and if the survey is printed also gives more information.
3. Don't mix Fields, Auxfields, Locals multiple times
   The key is that you want to be able to locate information quickly, and having a jumble is self-defeating, even when working with the Control Center Explorer.
4. "Forward" passing of assignments
   If you don't use parameters to blocks, then do assignments to blocks "below" the current level instead of grabbing an assignment from down below. The "above" will be consistently on the rules path and will not create a "generated parameter."
5. Minimize overuse of procedures
   Procedures do cause the survey to run slower, and procedures embedded within procedures can quickly multiply and cause problems. Because procedures do not store data they also can be harder to debug, and documenting procedures is also more difficult. Use your best judgment.
6. Use fields for potential data items (counts, internal checkpoints)
   Although the client may not think of it, always consider using a field for counts of things (family members, children, item lists, ...) instead of auxfields/locals. The data will be stored and available instead of having to be recreated (potentially with errors) afterwards. Having the results stored in the data also make it easier to debug some routines.
7. Use auxfields for fills, some counts
   You have more control over auxfields than locals, and as a bonus can see fills in the DEP watch window. Those counts that are easily recreated can be made as an auxfield instead of a fill.
8. Use locals for loop indices, or those items that always reevaluate\
   Locals are always reevaluated if on route, cannot be seen by the watch window, and are never stored. Relegate them to loop indexes and sometimes counts.
9. Define parameters on blocks as needed
   Parameters minimize the number of generated transit parameters. These have the highest overhead and are on-par with locals. If you do use parameters, be sure to explicitly define INPUT, EXPORT, or TRANSIT.
10. Always .KEEP fields that are assigned
    Don't assume that a field will be kept correctly ... make it explicit at the point just before the assignment at the same level or one higher than the assignment. Implied KEEPs are at the bottom of the rules section, and sometimes not at the place you would expect.
11. Create fills and other assignments as local as possible
    Don't put all your fills at the uppermost level and do assignments at the bottom, but define your fills for that block if possible. This will make it easier to track down and observe in the watch window.
12. Create internal checkpoints where used
    Likewise, keep internal checkpoints (assigned variables used for logic pathing) as local to where they're used as possible. Whenever possible make these fields and not auxfields/locals.
13. Define ranges for everything

Don't be a sloppy programmer! The specifications should give explicit widths and ranges for everything. Define widths for fills like STRING[28] instead of STRING (implied 255 characters). Use ranges like 0..12 instead of INTEGER (width of 18!). When it comes time to work with the data you won't have to deal with impossibly long fields, and the analysts will appreciate it as well.

14. Create a type file/library
    Put all your types into one place, name them consistently, order them consistently, and place them into one file. Anytime you have to modify a type you'll know exactly where it is, and you will not duplicate types unnecessarily.
15. Use comments
    Comments are handy for matching very long ENDIFs to their parent IF/ELESIF statement. They are good for describing complex algorithms or special study constraints. They are good at documenting changes to an instrument after data collection has started. In fact, use comments as need but be sure to make them accurate. Place comments appropriately per your organization.
16. Unique language fills
    If you work on multi-lingual surveys you have to program fills per language. I would heavily recommend that a distinct fill is created, named appropriately, for each language instead of reusing a fill for the languages. For example, create xHeShe and xElElla for the fills (he/she) and (él/ella) respectively. Otherwise you will need to heavily use the ACTIVELANGUAGE command in the rules, later documentation of your survey becomes more complex, and debugging each fill requires a debugging run in each language instead of just once for the main logic. Yes, it is a little more overhead per memory but a much cleaner approach with less coding.

4. Know Your Work Environment

One of the greatest obstacles to debugging and doing it well is knowing the tools available to do the job. The Blaise system is very large, complex, and covers many different platforms. To cover all the debugging features of Blaise in detail would create a very large paper. What follows is a discussion of the different environments, how it can be accomplished, and tips and suggestions for making better use of the feature.

4.1 Delta

This tool is an enhanced datamodel browser that can compare two datamodels. It provides a graphical rules viewer of the datamodel and allows the user to navigate through the datamodel according to the displayed IF/THEN/ELSEIF, blocks in the rules. A selection of a node in the tree view also focuses the appropriate place in the rules viewer.

Delta's value is that it has a rich view of the datamodel rules, and that it can compare two datamodels, such as from one release and then a second. This is very valuable when source code is not available, or documentation of changes has not been up-to-date in the source code.

Delta is XML-based, and so new stylesheets can be used to view the information from the datamodel, such as the provided **iexml.xsl** stylesheet found under the C:\Program Files\StatNeth\Blaise 4.8 Enterprise\Bin directory. This means that the entire XML from Delta could be retrieved, or other stylesheets to be used by Delta can be created.

4.2 Blaise XSD schemas

This utility exports a schema for the entire datamodel that describes types, fields, codes, question text (and language identifier), ranges, data sequence of questions, and so forth related to the data of the datamodel.

Note: it is missing some details, such as the width of fields, checks/signals or other features found only in the rules.

This information could be a great source of information about the datamodel without resorting to writing special BCP utilities. The XML can be imported into databases, spreadsheets, or stylesheets written to take advantage of the information, and so forth.

See the Blaise Data Centre for the related data XML.

4.3 X-Tool

The Blaise help has this entry:

> "X-tool is a software package for the analysis of experiments embedded in ongoing sample surveys. With X-tool you can test hypotheses about differences between design-based estimates of finite population parameters observed under different survey approaches."

Practically speaking, I don't have a current use for X-Tool in the context of debugging Blaise applications.

4.4 Manipula/Maniplus

Manipula has been the robust tool for extracting data and manipulating from a Blaise database, and most recently it works with datamodel information, and can be called from DEP and Basil.

It is incredibly versatile, and as a result the focus is on features that make debugging easier rather than features of the language.

4.4.1 Manipula Debugger

The debugger is basic. The greatest value is to step line by line through a script and examine variable via the "evaluate" dialog, tool tips, or the Manipula watch window.

The things you can do with the debugger are:

Set/clear a breakpoint/clear all breakpoints
Step over a procedure
Trace into a procedure
Evaluate a variable via a dialog & tool tip
Display multiple variables (watch window)

4.4.1.1 Manipula watch window

This is not the same as the DEP watch window, and is set by a literal field name, typed, or inserted from the evaluate dialog.

4.4.1.2 Evaluate window

The fieldname has to be typed and a pick list of prior viewed fields can be selected. Valid fieldnames can be added to the Manipula watch window.

4.4.1.3 Cursor tool tip window

Hove the cursor over a variable while the script is in "stopped" mode and a small tool tip will display the variable's contents.

Remember that this debugger is only for Manipula/Maniplus (not DEP). Be sure to check the "debug information" box under datamodel properties before working with the debugger. Be sure to set the "checkrules" global setting as appropriate. Breakpoints do not always work.

4.4.1.4 Display()

The DISPLAY command can either wait for user input or is updated dynamically as the script runs. It can be used at critical places to easily show multiple values, though this use has been superseded by the watch window. It can display html.

4.4.1.5 Temporary Files

Temporary files are useful to grabbing information from Blaise database and showing them in grid format to the user in Maniplus. Multiple keys defined on a temporary file allow great flexibility in sorting data, and using the INMEMORY setting keeps things fast.

4.4.1.6 Write to a file/MESSAGE()

Customized messages can be written to a physical file to create a log of actions. This is most useful when a lengthy or complicated manipula script is run and the problem occurs intermittently.

Either create your own log file, or use the MESSAGE() command to write to the default error file. A BOI could also be defined to write to a database for better sorting/filtering.

Manipula standard errors include Calcerror, Overflow, RangeError, SubscriptError, ReportImportError. Other settings that influence the file are

MESSAGEFILE = 'FileName' (global setting)

MAXMESSAGE = N (global setting)

Be sure to rename the messagefile in the global setting to something other than "*.msg" so Outlook (Microsoft email program) won't try to open it by default.

4.4.1.7 DLLs

Custom DLLs offers the greatest flexibility of any programming, but require the most work. Manipula has the capability of calling DLLs, which then can do things like create log files/databases, interact with users, read/set system resources, influence interaction with Blaise, and much more.

Be sure to keep to standard programming practices when creating the dll, and release any resources after execution is finished.

4.4.1.8 Late Binding

Manipula scripts generally have needed a datamodel defined before the script can be prepared/run. In 4.8.1.1399, generic scripts can be created that do not need the datamodel. However, all interaction with the datamodel has to be done through manipula datamodel commands.

Take advantage of this and create a library of manipula scripts that handle common situations: load preload data, manage Blaise databases, get counts/status on common fields, and so forth. Debug the scripts once and use them in a variety of situations.

4.4.1.9 Registry Commands

The recent Blaise release has also opened the Windows registry to Manipula. This opens the possibility of transferring data between manipula runs so preferences can be stored and hopefully reduce errors in similar processes.

Warning: be careful, and be sure of what you're doing! Clean up after yourself!

4.4.1.10 Alien Router/Procedure Object

Manipula can now be called from an alien procedure/router call from the datamodel. This allows complex calculations to be place outside the DEP into a standard library. Update the library, release to projects, and there's no need to prepare the datamodel again.

Note: the language has been extended to allow Manipula to recognize the environment is called from.

4.5 Cameleon

This scripting language is intended to be used to create data dictionaries for information exported from Manipula. Debugging Cameleon scripts has always been difficult, and the best methods simply have been to insert normal (non-command) text in the output and see what the output looks like.

Like other languages, use consistent indenting when possible. Consider writing new routines in Manipula because of the datamodel information now available.

4.6 Basil

Basil is meant for a self-administered interview. It bridges the gap between Blaise IS and DEP, but has present special difficulties in debugging since all the commands are stored in the text fields of the DEP using pseudo-xml. The layout on the screens is controlled by panels, and position of controls on the screen are all relative to panels.

This means all panel and controls have to be programmed without a syntax checker at design time and errors occur only during runtime.

Develop by making each panel a different color during the design work to make it easy to position controls in a panel, and then use consistent colors when releasing to testers/production.

It's best to only work on a few screens and get them correct, then copy everything as needed. Copy as much as possible and type as little as possible. Make everything consistent.

Basil heavily relies upon Manipula in the background to handle all the processing of commands from the DEP. Use debugging techniques for Manipula.

4.7 Hospital

Hospital hasn't changed much, but a new visualizer interface has been provided.

This utility allows the user to scan through the datafile and examine those things that are a problem. As the cursor moves over the top of each of the items in the datafile the information related to that area is displayed to the right.

The Blaise help states, "This graphical view may help you finding errors and understand the cause of them."

4.8 Control Center (DEP/Cati)

Listed below are the more useful items found within the control center.

4.8.1 Run - Run (DEP)

Some of the more useful parameters:

Show watch window

Choose the files to run with (DEP config, menu file, ...)

Surveys often have "gate" variables that prevent from code executing prematurely, and once set it cannot be easily reset (i.e., for household listings, lists, randomization). Reset these by using assignments in menu items, manipula scripts, or "discard all changes" after saving the entry session at critical points.

4.8.2 Database Browser

Allows the user to look through the raw data of a Blaise database, data view, or BOI. Note a new feature of the database browser is that the details view does not show empty fields in the form. Type multi-part keys by using a semicolon between each of the keys, such as 10001;23. Right-click on the tree structure to use the Find feature to quickly locate and select the fields you want to use. Save the view after making your selections (Control Center - <open database browser> - File - Save As).

4.8.3 Structure Browser

The structure browser, like Delta, shows all the information from the datamodel.

Right-click and select details when in the Elements view. Clicking on the top level of the datamodel will bring up the rules for that level, and clicking on any block will bring the rules for that block. You can also drag & drop any field in this view to a Blaise editing window and you can retrieve the full name of the field directly into the code. No typing = reduced errors = less debugging.

When in the Statements view with details, browse to the statement/field you are interested in looking at (sorry, the Find is not enabled here). Click the "Fields involved" tab, then click the details button at the bottom right. This will bring up the same details as in the Elements view within the Statements view, but for blocks allows you to easily browse through the list.

4.8.4 Datafile Management (Rename, Copy, Move, Create, Delete)

Use these tools to manage Blaise databases and avoid the network file copy. You will avoid accidental problems with the databases.

4.8.5 Tools - Environment Options

The Environment Options are important for creating that workspace that leads to better programming.

Select each of these options for a better work environment.

General

- Check for changed files
- Use tab sheets
- Syntax extensions (i.e., put in .bla, .inc, .prc, ...)

Editor

- Auto indent mode
- Find text at cursor
- Create backup
- Syntax highlighting (uses the syntax extensions, modify Blaise.sht file)
- Show Explorer (double click on includes to open files)

Make use of bookmarks (CTRL-K# where # is 0 to 9 to set a bookmark, then CTRL-Q# to go to the bookmark). Use the Enhanced search and the Explorer for navigating quickly. Always use project files.

4.8.6 Configure Tools

Add commonly used utilities here with shortcuts to reduce time retrieving them.

4.8.7 Internet Workshop, Internet Service Manager

Necessary for creating Blaise IS surveys. (Another paper ...)

4.8.8 Oldeb Db Workshop, Oledb Db Command Builder

Necessary for creating BOI files. (Another paper ...)

4.8.9 Cati Specification, Cati Management

Needed for working with the CATI system. (Another paper ...)

4.8.10 Blaise Data Centre

This GUI interface is superior to the default database browser in that it can export data in text, BDB, or XML formats outside the control center, including internal form keys. It can filter records, group results, create sub databases, compare case data and BOI versioning, and save the export routines as projects. No knowledge of Manipula needed.

The viewing and exporting of data outside Manipula is a tremendous asset. Versioning (with BOI) and comparison of data is especially useful.

4.8.11 Bascula

A utility to do weighting and analysis. Not used in the debugging context.

4.8.12 Menu Editor

The menu editor allows additional debugging tools to the programmer via the menu items. These include

Invoke DLL Procedure

Start executable

Invoke DLL Procedure (Advanced)

Invoke COM object method

Start Manipula

Keyboard action

Goto field

Assign field

Depending upon the need, the author could write information to a file, perform a series of assignments, interact with specific testing software, or other needs. Organizations in the past have made use of the menu for specific testing software. A large number of custom functions can be sent as parameters detailing the environment the item was executed.

4.8.13 Diagnostic Tool

This is a wealth of information that is more useful to StatNeth than to the typical programmer. It does show what licenses you have in your installation (Blaise/Components/Internet/CATI/Basil) for registry and file entries.

4.8.14 Blaise Emulator

The emulator has been updated to work in a non-CATI mode. It can generate random data for all fields in a way that is consistent with the rules of the datamodel. Please note than when using these settings (stored as DMD2.teo file in the example below), you will have very active use of the drive where the data is located. This is because it appears that BTEmula flushes the data to the disk after each change.

There will be some paths that are highly unlikely to be reached randomly. You should create data files for these situations, then change the UseScripts and RandomOrder settings, and specify the [DataScripts] section. Run the special cases first, then change your script (set UseScripts=0), then rerun against the same datafile. All the rest of the values will then be filled.

Set properly, this tool has the potential for revealing holes in logic flow that need more careful testing.

```
[EmulatorSettings]
DataFile=DMD2
MetaFile=DMD2
WorkDir=.
BATCHMODE=1
DELAYRANDOMASSIGN=0
MININTERVIEW=0
MAXINTERVIEW=0
MINQUESTTIME=0
MAXQUESTTIME=0
DisableCATI=1
GenerateRandomData=1
GenerateRemarks=1
UseScripts=0
RANDOMORDER=0

[DataScripts]
Count=3
Script1=DMD2.sc1
Script2=DMD2.sc2
Script3=DMD2.sc3
```

4.9 CATI

The CATI management block now has an auxfield block with history information of the current case before it's written to the database.

The CATI specification program now has a log file of all specification changes. This is very useful to track down when/who made a change to the scheduler.

The CATI management program now has a log file for daybatch exclusions. Unknown users now are identified as <LoginName - ComputerName>.

4.10 Database Monitor

This has been updated to show the name of the machine on which the process is running plus the name of the user that is running the process.

4.11 Blaise IS

Blaise IS pretty much offers the same capabilities as found within Blaise DEP, with a few exceptions, such as manipula scripts, extra DLLs, and so forth are not available on the client side. The same issues as Basil still apply to Blaise IS because all the commands are stored in the text of the survey. Normal tools, such as the Blaise Watch Window, are still present. Extra HTML tags can be used within Blaise IS as normal.

4.12 Modelib

The modelib is used to set the style of pages being shown to the interviewer within the DEP. It interacts via the LAYOUT instructions given in the DEP, and by the default grids and input panes, as well as the configuration file.

It can preview the general layout of each page before it even gets to the point of being tested. Additional information can be found by right-click (other than the entry box) on the page and choose Page/Question properties. This will give information on the grid, info, and question properties for that page.

4.13 Configuration File

The configuration file is nearly the same as the Modelib except that no additional layouts can be created. The configuration can change certain parameters, such as the margins, status bar, and help/audit trails.

It also allows a preview of the layout of the instrument the same way as the modelib.

It is very useful to open datamodel and create a configuration file based upon the datamodel to see what original settings were in the modelib with that datamodel.

A common source of problems for layout/audit/help/status is the interaction between the configuration file and the datamodel. Make sure the configuration file is current for the datamodel. DEP will not warn of an invalid configuration file.

5. Additional Debugging Tools

5.1 Watch Window

The Watch Window has long been a very useful tool for revealing the values of variables as they are being changed. And long has been the mystery of why some values are set, some are skipped, and otherwise unexplained behavior.

The updated watch window has a new option, Show Changed Fields. As a field is entered all other fields that are changed are also displayed. For the example below, the field USE.USE15 has a value entered. It shows that

fields USE.Count_USE_Never, USE.Count_Use_Once, USE.CSE21a, DIWD.DIWDChkPt have all been updated, and the parameters into the procedure GetTimeCodeToSound have also been updated.

This information should be helpful to narrow the possibilities when tracking down a difficult problem.

```
*** General ***
Field changed: USE.USE15 = Once
Blocks checked: 5
Procedures executed: 2
*** Check rules information, 12:37:32 PM ***
Check <main block>
Check USE
> field: USE15 (value: 1)
Check DIWA
> referred field: USE.USE15
> referred field: USE.Count_USE_Never
> referred field: USE.CSE21a
Procedure GetTimeCodeToSound
> parameter: piTimeCode (value: 34)
Procedure GetTimeCodeToSound
> parameter: piTimeCode (value: 10)
> parameter: peSound (value: SOUND (..\MM\430pm.wav))
Check DIWD
> referred field: USE.Count_USE_Never
> referred field: USE.Count_USE_Once
Check DIWE
> referred field: USE.Count_USE_Never
> referred field: USE.CSE21a
> referred field: DIWD.DIWDChkPt
```

5.2 Audit Trail

The audit trail has long been very useful to track down user's interaction with the DEP and determine what series of keystrokes were used to result in a reported problem that comes back to the author and is very useful information.

It is possible to retrieve a backup of the case in XML from the audit.dll, but there are concerns: it is "open" (but so is the audit trail), it creates an additional file, and there is a slight pause as the file is created, and the file extension is .ASC.

The new settings in the .AIF file are

```
BackupRecord=1
AuditFolder=\\DataSrv\Shared\2003
AuditTempFolder=c:\Temp
```

5.3 Alien Routers & Procedures

Alien routers are best used for collecting data in a special format that is not easily obtainable via the DEP, such as using a custom date control. As a debugging tool this has limited value because it is a "real" question that is placed on-route, but will only be processed once it is on-route. The advantage of alien routers as debuggers is that they have access to the data of the datamodel.

Alien procedures are a bit more useful for debugging if there's a real need to dive into the BCP, but will be limited to a few parameters passed in from Blaise into the procedure.

Potentially a great deal of information (access to the entire Blaise datamodel, data, rules checker, ...) can be retrieved and utilized when these alien routines are triggered. However, given that Blaise has more useful means at its disposal (audit trails, xml backup of data, enhanced watch window, calls to Manipula through the Alien call) that writing a custom DLL for debugging purposes may now be more effort than its worth.

6. Creating New Tools

First and foremost, before any new tool is created make sure you cannot do the same task with existing tools. As mentioned before, the Blaise development environment is very rich in details and has plenty of possibilities.

In the past the options for creating new tools were solely using the Blaise Component Pack and interacting with the datamodel. Since the release of Blaise 4.8, the ability to run Manipula scripts from the DEP and the greater flexibility of BOI files has decreased the need of writing .Net applications using the BCP.

Potential tools for use within the Blaise environment include:

Manipula/Maniplus
BCP
Delphi and .Net programming languages
Other programming languages

I would recommend first seeing if existing tools can be enhanced by using their results, then if Manipula can handle the task, and then finally delve into the BCP as needed.

When creating new tools, always keep object oriented programming practices in mind, and don't be afraid to explore new tools that are constantly appearing, such as the Smart Client Software Factory by Microsoft (http://msdn.microsoft.com/en-us/library/aa480482.aspx).

Do your best to have very clear specifications on

- Who are the users
- What is the timeline and budget
- What platform will the utility run on
- What are the expected outputs (reports, results - be very explicit)
- What the interfaces (design and interaction)
- What sort of deployment is expected

Be sure to get all the specifications first, provide an estimate of hours based upon those specifications, and then only start programming once those details are set. Guaranteed, otherwise, the project will not meet the needs of the users, will run out of time or money, will have to be restarted, there will be scope creep, the interface will be poorly designed and unusable, the reports will be insufficient, and so forth.

It's nothing new, but always bears repeating. Spend most of your effort up front in the design and you will save a tremendous amount of effort debugging later.

6.1 Suggested Custom utilities

Custom utilities can always be created are a welcome addition to the suite of tools available to and written by the authors. Some of these may not be exactly a "utility," such as taking advantage of the macro facility within Microsoft Word to perform a complex search & replace operation on source code.

Utilities can be very single-purposed, or general. Below are examples of utilities that an author would find helpful if someone were to write them. Note: these do not necessarily exist except as concepts in this paper.

Utility	**Description**
Check finder	Locations places in the source code where an inadvertent check was created instead of an assignment. For example, A = 1 is a check, while A := 1 is an assignment. Both are valid syntax.
Fields crosswalk	List the fields/auxfields/locals/parameters that are defined, and how they are used: assigned, ASK, KEEP, SHOW, lookup, SEARCH, ... By examining this list it should be possible to find all unused fields that

	are no longer needed, or old code that no longer serves a purpose.
Layout checker	This would scan the modelib/config file and report any overlaps of panes or fields, or other potential problems where part of one pane/field is overlaid by another.
Completeness checker	Scan the datamodel and report on the fields. This help makes sure each field has been fully defined: Field tag, description, languages, and so forth.
Logic "holes" checker	This may be infeasible per the Turing proof that a program cannot determine if another program will ever run to completion. However, it may be possible to locate some logic holes where a question can never be asked. This is a complex problem, and probably is better revealed by the empirical method via the BTEmula program.
Logic Trace	These are addressed in part by the audit trail, the new "fields changed" watch window option, but a full logic trace may be accomplished by modifying the audit trail dll, or possibly by using an alien procedure to capture every action of Blaise rules checker.
Datamodel Version comparer	Blaise allows versioning of the datamodel, and hence the database. The current database tool, Data Centre, does comparisons between data records on the same database (versioned or non-versioned). The Delta tool performs comparison between two datamodels. However, it may be more useful to have the results in a different format than provided by Delta.

7. Conclusion

Blaise environment is very rich in terms of what it can accomplish for computer-assisted surveys, and as a result is a very complex environment to thoroughly test an instrument.

8. References

http://www.blaiseusers.org

Sparks, Peter, "Testing Tips & Tricks," 2007

Sardenberg, Amanda F.; Gloster, John W., "Testing a Production Blaise Computer-Assisted Telephone (CATI) Instrument," 2001, U.S. Census Bureau. Covers the systematic approach to testing an instrument, but not necessarily the tools the author would use to debug an instrument before releasing it for internal testing.

Levinsohn, Jay R.; Rodriguez, Gilbert, "Automated Testing of Blaise Questionnaires," 2001, Research Triangle Institute. Use of keystroke files to verify the same path up to the point being tested.

Sjödin, Sven, "Whatever Happened To Our Data Model? Documenting Change in Continuous Surveys," 2000, National Centre for Social Research, UK. Using TADEQ to document datamodels, and then being able to run comparison between datamodels.

Hofman, Lon; Wings, Hans, "MANIPLUS, A powerful environment for managing Blaise III applications," 1995, Statistics Netherlands, Introduction of Maniplus