Resolving Question Text Substitutions for Documentation Purposes Using the Blaise 4.8 API Component

Jason Ostergren, Helena Stolyarova, and Danilo Gutierrez, The University of Michigan

Overview

The Health and Retirement Study (HRS) is a national longitudinal survey on the health concerns and economics of aging and retirement. The HRS utilizes a CAI instrument for biennial interviews of one to three hours in length given to around twenty thousand participants. One of the prominent features of the HRS CAI instrument is that it makes uncommonly heavy use of text substitution in its question wording. In some HRS questions, the entire wording consists of text substitutions, and oftentimes multiple substitutions are used in succession and/or layered inside of others. While this situation provides for a more dynamic and streamlined interview experience, for the purposes of documentation it presents a steep challenge for anyone attempting to resolve and document the various permutations of question wording.

There are various reasons why HRS finds text substitutions so necessary. At a basic level, they allow HRS to provide exact wording to the interviewer rather than offering a parenthetical that forces the interviewer to choose the correct words on the fly (e.g. “What is your husband’s name?” vs. “What is [your/his/her] [husband’s/wife’s/partner’s] name?”). Some substitutions are very closely tied to particular bits of data like gender and tend to be fairly simple. Others reflect more complicated and dynamic data gathered from multiple variables or the flow of a particular interview. What is more, these various kinds of substitutions can be layered on top of each other to produce even more complicated structures.

HRS has tried a number of schemes to resolve these text substitutions in the past. These range from manually tracing the logic of the substitutions to parsing the Blaise instrument source code in various ways. These past attempts have each proved deficient in some way. While no scheme is ever likely to be perfect, by making use of improvements in the Blaise API, HRS has now written a program that can resolve these text substitutions accurately and efficiently. The key advance involves tracing variable assignments through parameters between blocks and procedures using the API.

Text Substitutions in the HRS

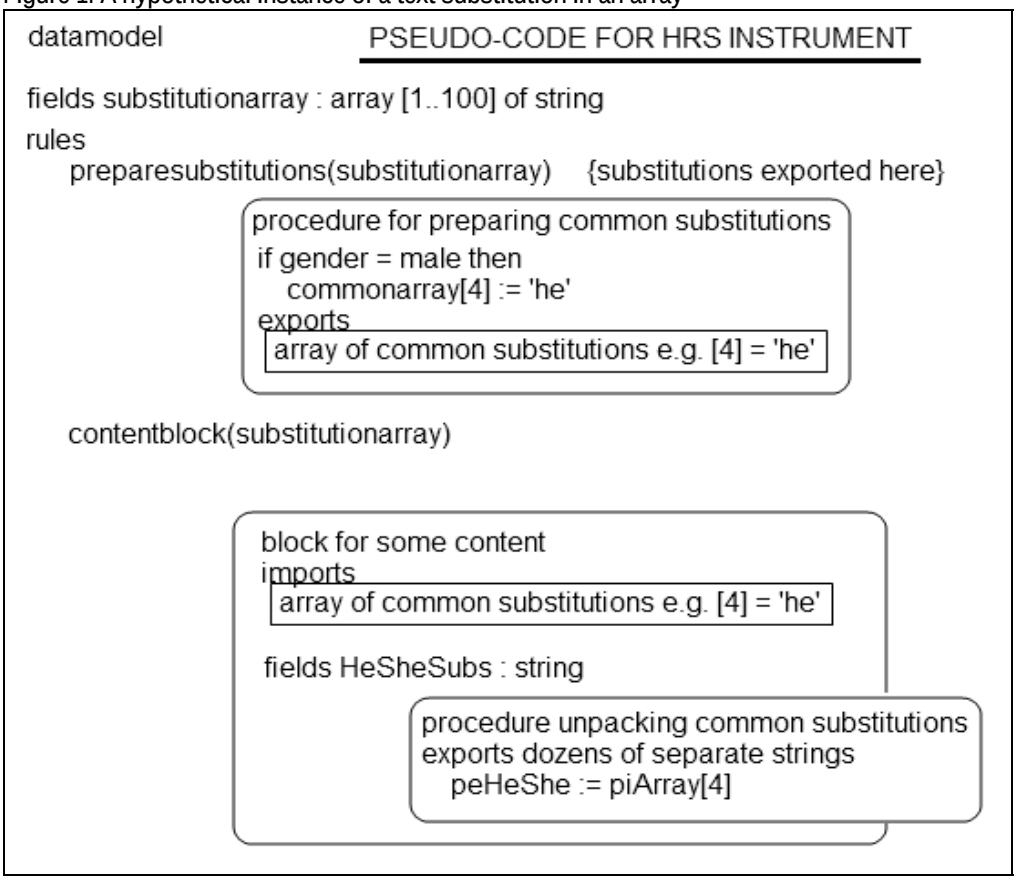
To see how this works, it is first necessary to understand how HRS handles text substitutions in the code of its instrument. Depending on how one counts, the present iteration of the HRS instrument contains between 6,000 and 15,000 text substitutions in its primary interview language. When Blaise was adopted as the platform for the HRS interview in 2002, an array of problems related to text substitutions presented themselves and the solutions chosen continue to drive the structure of our instrument. In particular, HRS chose to move all code for text substitutions into procedures and pass the necessary arguments and the text output as parameters. Procedures were chosen over blocks in part because they do not require maintenance of instance fields to call them and because HRS had no desire to store the internal variables used in generating the text substitutions in the database.

The problems which HRS confronted fell mostly into three categories which included the degree of clarity of the flow code, the ease of translation, and performance. The HRS instrument possesses a high degree of complexity in its flow logic and thus code readability has been a real problem in the struggle to debug and maintain the instrument. Pulling all of the text substitution code out of the flow rules helps to keep these rules readable to the programmer, decreasing the likelihood of flow bugs. Additionally, HRS has Spanish language translators who work in the code but are not as proficient with Blaise as the main programmers. Sectioning off all the text substitutions into procedures which are placed together near the

top of each include file has made translating this text somewhat easier and minimized the incidence of flow bugs being introduced by a translator.

Performance was also a serious problem in the early versions of the HRS Blaise instrument, in large part due to an excess of generated parameters, some of which were in effect global text substitutions – that is common substitutions referring to the respondent's spouse or partner, for example, which are used across different content areas. To improve performance in the handling of these common substitutions, HRS built a procedure which was called at the beginning of the instrument that packaged up the 40 or so strings into an array which was exported to a field at the datamodel level. This array was then passed into each content block (of which there are approximately 25) and another common procedure was used to unpack this array into a set of substitution strings for use in that block (see figure 1). This meant there would be no generated parameters from outside content blocks while also requiring only one field to be passed into each block, as opposed to up to 40 separate substitution strings.

Figure 1. A hypothetical instance of a text substitution in an array



Although all of this makes the instrument more reliable and efficient, the results of this scheme when it comes to handling metadata are mixed. In some ways, sectioning off these substitutions makes them easier to digest manually for the staff tasked with documenting them, but this is not at all an easy task given the size of the HRS. For automated documentation systems, this has proved a nightmare because it has simply been difficult to trace the content of a substitution among all the blocks, procedures and parameters though which it may have passed.

When it first adopted Blaise, HRS found itself almost immediately in need of ways to resolve text substitutions. Question wording with accurate substitution text was needed for IRB approval,

documentation, translation, quality assurance and other purposes. Manual attempts by programmers to document substitutions, such as placing the resolved text in the question text of the substitution variable proved to be impractical to maintain. It also proved too difficult for non-programmers assigned to document the instrument to determine the possible outcomes of the substitutions in the web of parameters through which they passed.

Resolving Text Substitutions: General Approach

Ideally, a program which resolves substitutions properly would operate as described in the following paragraphs. It would walk through the statements of the Blaise instrument in order, stopping at questions which are asked and which contain carets in the question text of the language being resolved. It would begin by correctly parsing the name of a substitution variable out of some question text, done by matching text following the carets which indicate substitutions. Some care needs to be taken here and also at later points to parse this correctly keeping in mind potential sources of error such as the fact that fully qualified variable names may be used and that characters like periods at the end of sentences may be adjacent to the variable name.

Once the variable containing the substitution is named, the correct instance of that variable must be identified since identically named variables can exist in different blocks. The program must begin searching for the variable in the immediate block. If no such variable is present, it must then begin to search upwards through the parent block and on to its parents as necessary. If the variable was a parameter, the program must identify the name of the variable passed in by the calling block and begin the process again based on that variable. If a fully qualified variable name is provided at any point that ends the search.

Once the actual substitution variable is identified, the program must find and evaluate any values assigned to that variable. Once processed, these become the resolved content of the text substitution – for example 'husband/wife/partner' – and can be plugged into the documentation of the question text at the location of the variable. In a simple case, there may be only a few strings such as 'husband' and 'wife' to locate. In more complicated cases, variables may be nested in the assignments which themselves have to be recursively traced and resolved and then incorporated into the larger resolution process of their parent substitutions. Additionally, it is necessary to filter out assignments for texts in languages other than the one presently being resolved (in HRS there are six "languages" – three each of English and Spanish for respondents, proxies, and proxies for deceased respondents– which each may have specialized substitutions).

The process of finding these assignments involves walking through all statements looking for direct assignments to that variable as well as assignments coming from export parameters of blocks or procedures. When parameter assignments occur, a recursive process of identifying assignments to the export parameter in question inside the other block or procedure begins. Assignments can also come from fully qualified references to the variable or from generated parameters. Finally, in some cases in HRS, assignments can come from parts of the instrument which are not even on the route at the point where the question with the substitution being resolved was asked (but which could nonetheless appear in the actual interview). Incidentally, HRS has made no attempt to account for this last eventuality, even though it occurs frequently in the first section of the HRS instrument due to the complexity of HRS and the way the rules engine works.

Beyond the mechanics of tracing the substitutions as described above, various additional obstacles exist which have to be handled. For example, some substitutions simply display a user input value without ever being assigned a value in the rules, which means there is nothing from the rules to use. However, if description texts exist in these situations, it makes sense to substitute them. Another problem is handling

array indexes which can make matching variable names difficult, for starters. It can be useful to try to filter the results by array index if a definite number is given, but it is often the case that a counter variable is provided, or worse that more than one array index is involved or that some expression such as 'blockA[i – 1].Q1' is used. HRS found that it was not worth tracing any indexes more complicated than a simple number. It also seems advisable to avoid tracing mostly irrelevant variables like counters when resolving text substitutions.

Implementation: Blaise API Use

HRS has been using the Blaise API to gather metadata since 2001, but only with more recent versions of the API has HRS been able to use it to write a program along the lines described above. The API provides a `RulesNavigator` object which can be used to walk through the entire statement hierarchy of the instrument. It also provides field metadata for statements that put a field (including blocks and procedures) on the route, which includes items such as question text that can be searched for the carets indicating substitutions. The functionality of this is well known and will not be described here, but the information that is gleaned from it is the key to this process. What follows is a description of the new kinds of data that must be gathered from the `RulesNavigator` in order to resolve text substitutions effectively.

Given the large number of places in the process where tracing parameters is critical, it is vitally important to glean information about parameters from the API. Three pieces of information are vital. These are the name of the parameter inside a block or procedure (that is, its name in the parameters section) and the corresponding name of the variable outside which is passed in as an argument to the block or procedure call. Finally, it is necessary to know the direction of the parameter (import, export or transit) – that is, whether the data passed by the parameter is flowing into or out of the block or procedure in question.

To obtain this information, it is necessary to look at the `StatementText` and the `Field` object for the statement calling the block or procedure. If there are parameters involved, the `StatementText` string will contain the text "{Parameterlist:}" followed by a comma delimited list of names of the parameters on the outside of the block or procedure call. By splitting this list by commas, an array can be obtained which will correspond to the order of parameters listed in the `Fields` collection *inside* the block or procedure. These `Fields` can be obtained from the method `get_DefinedFields()` on the `Field` object used above. Then it is necessary to loop through this set of fields and obtain each `LocalName`, which will provide, in order, the inside names that correspond to the variable names from the "Parameterlist." While looping through the set of `Fields`, it is vital to collect their `ParameterKind`, which will require casting them to `IBlaiseField2`. It may also be helpful to verify that they are parameters by checking that `FieldKind == BlFieldKind.blfkParameter`.

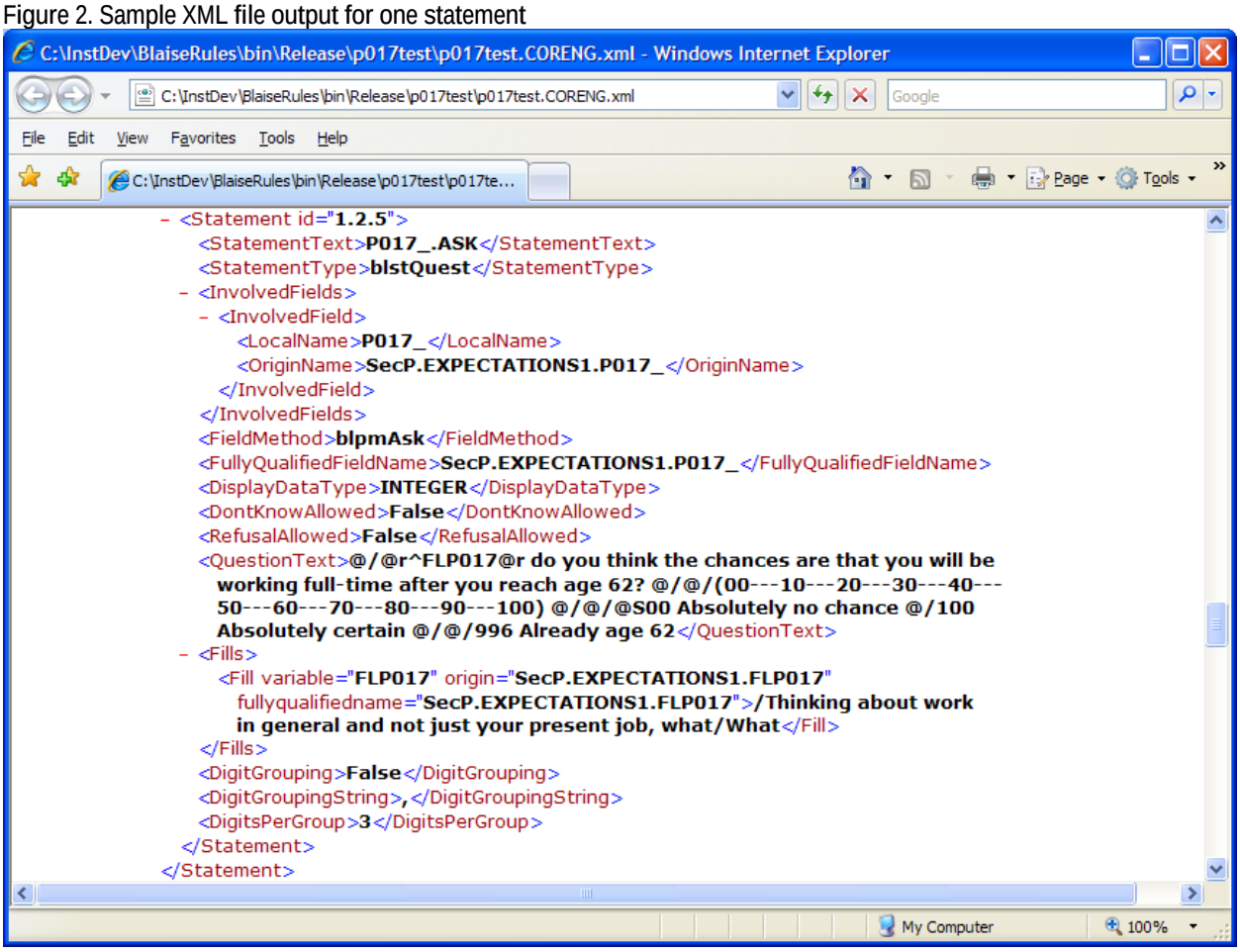
With these data about parameters in hand, it is possible to trace the originating variable of a text substitution upwards through layers of parameters by matching the outside parameter and continuing the search in the parent block. It is also possible to track down assignments to a variable by matching a variable receiving an assignment from the export parameter of a block or procedure with that export parameter so that the assignments to that export parameter can be tracked down in turn.

Finally, in rare cases the API functions in such a way as to prevent tracing these substitutions. Basically, the problem is that a procedure called from inside a loop at any structural depth lower than the loop block is treated differently by the API than a procedure called outside of a loop or in the looped block itself. The `Field` object for the procedure call statement is null in the former case but not in the latter. This prevents us from gathering information about the procedure's parameters as described above. The ideal solution would be for procedures called inside of loops to be handled just like those outside of a loop. It

is hoped that this paper has laid out a sufficient case for the benefits of this functionality to spur interest in resolving these remaining quirks.

There is one implementation issue that deserves special mention with regard to the sort of text substitution program described above. HRS found that it was not practical or efficient to directly use the `RulesNavigator` object to do the tracing described above. In particular, the `RulesNavigator` does not lend itself to free movement within the statement tree, and, at least in many of our early attempts, had a propensity to max out system resources resulting in extremely slow execution or crashes due to the large size of HRS datamodels. Instead, the text substitution program that HRS has built walks through the `RulesNavigator` in a forward-only fashion, but it simultaneously generates a parallel XML document which incorporates all relevant information and is arranged in a similar hierarchical fashion. At each point where a text substitution needs to be resolved, the XML document is used to trace originating variables and assignments exactly as described above. The resolved text substitutions are then incorporated into the XML document as nodes connected to the variable containing the substitution.

Figure 2. Sample XML file output for one statement



XML File Overview

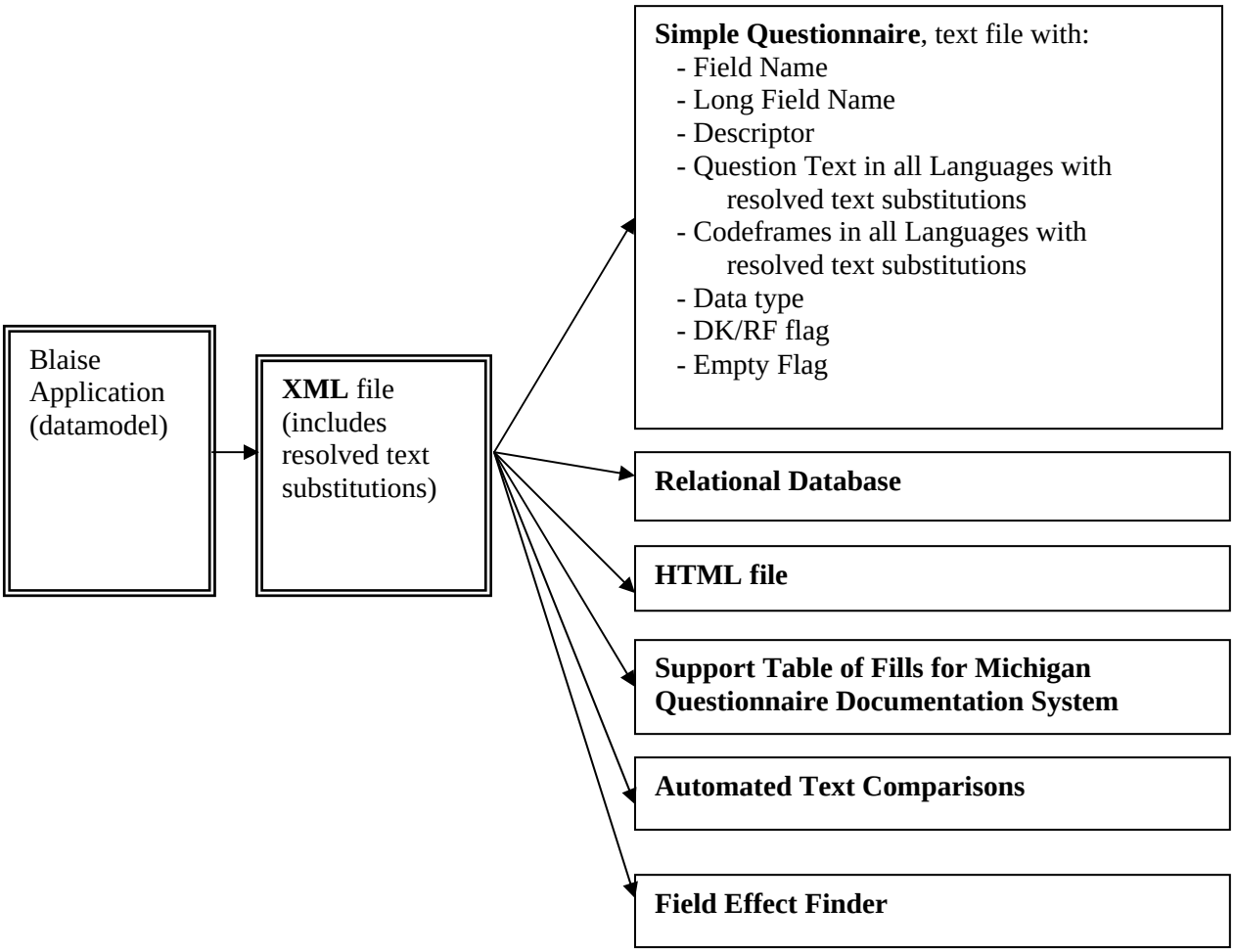
The XML document is handled entirely in memory, despite consuming some 200 megabytes in the case of HRS. The entire process requires between 5 and 10 minutes of execution time on a typical computer. The originally intended output of this process is a delimited text file containing only the substitution information, but a side benefit of generating the XML document is that it can also be saved to disk at the

end of the process and used for other purposes. It turns out to be particularly valuable both because of the resolved text substitutions that are contained within and because the XML seems more accessible than the `RulesNavigator` object to programmers and non-programmers alike. Important uses for this XML output are described below.

Potential Uses of the XML File

One critical function is to accurately resolve text substitutions for input to a metadata documentation system. As a byproduct of the process which handles text substitutions, HRS retrieves a variety of valuable metadata such as field names in ask order as processed by language. The table below describes potential uses of this information.

Figure 3. Potential uses of the XML file with resolved text substitutions for metadata documentation system



Simple Questionnaire

The Simple Questionnaire (SQ) is a formatted output text file designed to document the set of questions in the order and the form they are presented to a respondent, including question wording with accurate text substitutions. The SQ does not contain "ask rules." The procedure to produce an SQ was previously a challenge since many HRS questions use text substitutions. The method described above significantly simplifies the process. In a current version of our SQ generating program, we are using a forward-only

data reader to read the XML file created from the Blaise datamodel. By construction, each question is represented by a separate node in the XML file, with the following attributes attached to the node:

- Field Name
- Long Field Name
- Descriptors
- Question Text with resolved text substitutions
- Codeframes with resolved text substitutions
- Data type
- DK/RF flag
- Empty Flag

As the program reads these attributes for each question node, it produces a formatted output that accurately represents the question texts, codeframes and other prompts that are presented to interviewers during the interview. The output of this process is an easy-to-read text file. The advantage of this approach where we divide the processing from the presentation is that each can be separately modified to accommodate further needs and changes in the final product. This gives us a great deal of flexibility. We can omit, reformat or replace parts of the metadata from the XML file as required for different presentations.

HTML File

The XML file can also be converted to XHTML format by using appropriate XML-to-HTML filters or by adding tags to the nodes. This is a means of altering the presentation in a portable and familiar way. It also allows the metadata to be reorganized in versatile ways.

Portability

Another advantage of a widely-used format like XML is that it can fairly easily be directly imported or programmatically converted to a relational database or other useful storage format. This is helpful for users who want access to this metadata in a way that they are more comfortable with than either the Blaise API or our XML output.

Automated Text Comparisons

This process also makes possible the automation of text comparisons that allow reviewers to easily determine when and how changes were implemented which turns out to be very important for making best use of resources in translation work, among other things. As with the Simple Questionnaire, a small program can be written based on the XML to compare targeted parts of the metadata. There are two work processes in HRS that benefit greatly from the use of such products:
1. Translation comparisons during development. Automated text comparisons between different builds of the datamodel allow translators to easily determine the changes in English that need to be applied to Spanish. Tracking these changes has typically been a very involved, cumbersome, and time-consuming task.
2. Documentation of changes between released datamodel versions. Datamodel changes that affect interviewers need to be summarized and disseminated during data collection periods.

Field Effect Finder

As a side effect of our text substitution process, we get very useful information about how fields and parameters are related in the datamodel. It is possible for a field to influence different parts of the datamodel under different names through parameters. For example, you might have a condition depending on a parameter called *piSex*, which actually represents data in a variable called *Respondent.Gender*. We include nodes in the XML which tell you about these relationships. You can write a simple program that takes a particular variable as an input, reads through the XML, and finds all of the places that this variable of interest affects, even under a different name.

Support Table of Fills for MQDS

This was, in fact the main justification for this programming effort. HRS was concerned that an older mechanism that was being used would not be maintainable in the future. As previously mentioned, extracting text substitutions from the datamodel and using them accurately in documentation has always been a challenge at HRS. This new process to produce accurate text substitutions (fills) has allowed us to replace an older, less accurate, more complex and less flexible process. One of the main output products of this process is a delimited text file containing only the fills. A table called the "support table" is generated from this delimited file and is used as input for the Michigan Questionnaire Documentation System (MQDS), which HRS uses to document the survey. The support table content now has much more accurate metadata than ever before. The resulting output from MQDS now needs less editing, leading to a lesser workload for the editors of this documentation. The documentation's purpose varies, but is mainly used to get study approval from the Institutional Review Board (IRB) at the University of Michigan.

IRB documentation, End-User Documentation and Quality Assurance

While producing documentation for IRB study approval was the original purpose of the new process and products, it will likely become equally important in producing metadata for quality checks and end-user documentation. Some of the other things the new process allows us to do are:

- produce field descriptors and quickly check for duplicates and errors throughout our large instrument
- produce field names in ask order sequence for checking integrity and completeness in documentation
- produce question text and codeframe text and check them for omissions and errors
- produce ask rules for use in easy-to-read formats

Conclusion

In short, HRS was in need of a new product to save time and increase accuracy in metadata processing of text substitutions for IRB documentation to obtain study approval. The application described here took advantage of features currently available in the `RulesNavigator` class of the Blaise API which make it possible to trace the effects of variables throughout the structure of a datamodel. This effort was combined with another longstanding HRS project which converted metadata from the API into an XML document in order to make that information more widely accessible. The intention was to provide a source of metadata with text substitutions in the XML which could be exploited by small, easy-to develop programs for a variety of uses as described here and others that have not yet been imagined.