Using Blaise for Implementing a Complex Sampling Algorithm

Linda Gowen and Ed Dolbow, Westat Inc.

Introduction

In 2012 Westat was awarded a contract to conduct a very large household survey. The survey included a roster of all household members. The roster collected demographics data along with many other study specific variables. These data from the roster were used in a complicated selection algorithm to choose household members to complete extended questionnaire instruments. The goal of the survey designers was to conduct "extended" interviews for no more than 2 adults, 2 children ages 12-17 and one child aged 9-11. In addition, they wanted to select persons with particular characteristics. In cases where there were multiple members, with these similar characteristics, they wanted a random selection applied.

Since the roster instrument and the extended interviews were being developed in Blaise, the system architects wanted to explore using Blaise to implement this complex algorithm. They were avoiding installs of additional software to the interviewer tablets that would have to be maintained and licensed. The algorithm required generating random numbers for implementing various sampling rates and sorting. In addition to developing these functions, we needed to be able to load test data to verify the system would correctly yield the sampling rates. We were not sure if Blaise was the "perfect" tool for these functions, but definitely thought it was worth a try.

This paper will explain our Blaise approach. It will describe how we used the Blaise Random Function to meet the requirements for applying specific sampling rates. Also, we will discuss how we programmed a bubble sort in Blaise code to conduct a sort and then finally we will discuss our test methods and the results.

Random Number Generator

The first function we discovered to be very helpful was the RANDOM function. It's simple to use. Here is the screen shot from Blaise help.

Purpose
Returns a random number.

Syntax

```
RANDOM [ ( N ) ]
```

N is an integer expression. Real expressions will be rounded to integer values.

If you specify **N**, the result **R** is an integer random number in the range 0 .. **N** - 1 (0 <= **R** < **N**). If you do not specify **N**, **R** is a real number in the range 0 .. 1 (0 <= **R** < 1).

Remarks
By default, each time a program is started, a new seed value will be determined for the random function. Because of this, the result of the random function differs systematically from one run to another. Only by using the SETRANDOMSEED instruction the same result from one run to another can be reproduced.

Example

```
X:= RANDOM ((A**2 + B**2) / C**2 + LEN ('1'))
Y:= RANDOM (5)
Z:= RANDOM
```

We did not seed the RANDOM function. We simply defined the receiving field, like the example below, to generate a uniform distribution between 0.001 and 1.000"

```
RAND1 (RAND1)
    {English text}
    "Person Random Number 1 generated following
    a uniform distribution between 0 and 1."
    : 0.001..1.000,EMPTY
```

We conducted an analysis in SAS on a dataset of 15,800 records generated by our Blaise program. We wanted to see if the random numbers were evenly distributed.

Mean	0.500607	Std Deviation	0.28960
Median	0.500000	Variance	0.08387
Mode	0.146000	Range	1.00000
		Interquartile Range	0.50400

The results prove the distribution of the random number was very good and reliable for using it for the purpose of random selection.

Quantile	Estimate
100% Max	1.000
99%	0.991
95%	0.951
90%	0.901
75% Q3	0.752
50% Median	0.500
25% Q1	0.248
10%	0.101
5%	0.049
1%	0.010
0% Min	0.000

Duplicate Random Numbers

In order to have a true random ordering of household members, we needed unique random numbers for each person. We realized early on, it was quite possible there would be duplicate random numbers among persons within a household. The odds are 1 in 1000 with the range .001 to 1.000 for one additional person in the household. The odds increase with each person in the household. So, we came up with the solution to check for duplicates every time a random number was generated after the first person. We designed the program to regenerate a random number up to 3 times if a duplicate was found among household members. In our test of 5700 households with 15,800 people, there were no duplicates. Our test included households with up to 12 people in them. We did not keep track of how often a number was regenerated. It would be interesting to know. Also, we could probably have increased the precision (places to the right of the decimal), to decrease the incident of duplicates. But, I think we still would have looked for duplicated to guarantee there were none.

Bubble Sort

The requirements for our household member selection were to generate a random number and sort the members by the random number generated. We decided that a bubble sort would work well in Blaise.

We programmed the simplest bubble sort algorithm modeled after the following:

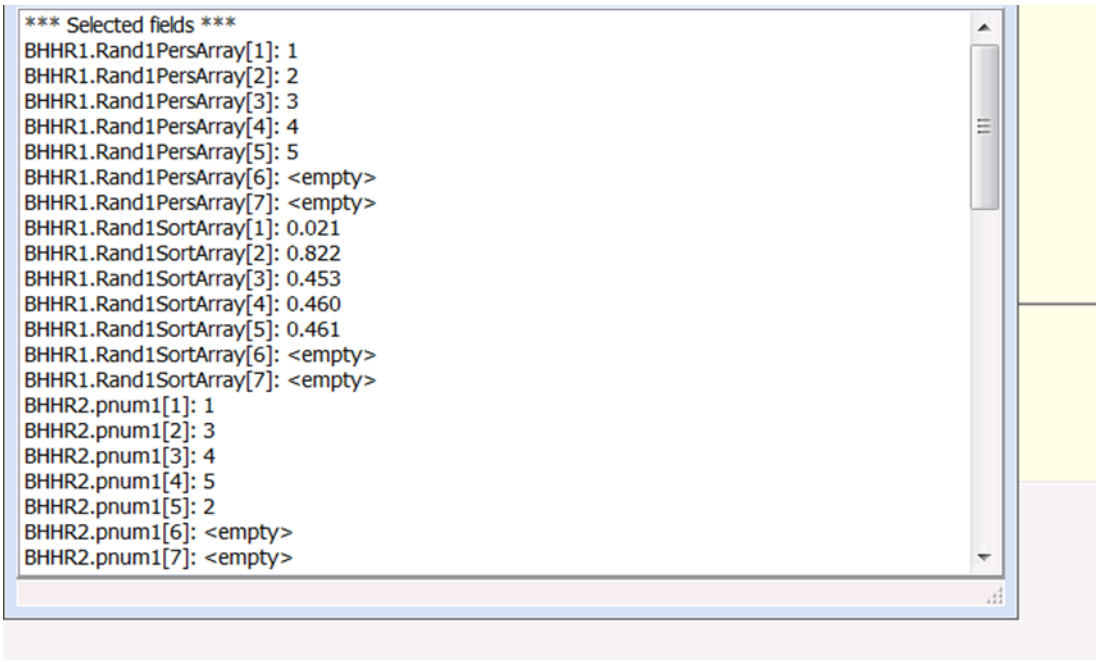
```
For I:=1 TO 10 DO  {simple sorting algorithm}
        For j:=1 to 9 DO
            IF SortedArray[J] > SortedArray[j+1]  THEN
              k:= SortedArray[J]
              SortedArray[J]:= SortedArray[j+1]
              SortedArray[j+1]:=k
             ENDIF
         ENDDO
      ENDDO
```

We replaced the upper bound of 10 for the outside "I" loop to be the number of members in the household. We replaced the upper bound of the inside "j" loop with the number household members minus 1. We didn't see a performance issue with this sort, so we went no further with trying to optimize it. There are many ways to optimize a bubble sort by checking for swaps, but we felt this would complicate the code when it wasn't necessary.

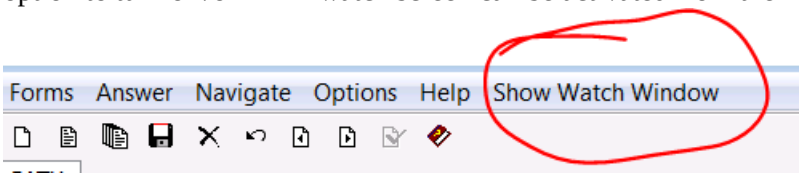
Testing

We conducted 3 levels of testing. (1) Unit Testing, (2) Bulk Load Testing, and (3) End to End Testing.

Unit Testing consisted of creating a testing environment with just the sampling algorithm code where we could control the many different parameters. We used the DEP Watch window extensively. Below is a screen shot from our unit testing. This screen shot shows 5 persons in RAND1PersArray, the random number generated for each of the persons stored in RAND1SortArray and the person numbers sorted in the Pnum1 array.



To activate the DEP Watch Window, the option "/!" has to be set when calling the DEP program. The option to turn off/on DEP watch screen can be activated from the Blaise Data Entry Screen.



The second testing we conducted is **Bulk Load Testing**. We took a set of 570 cases with fields set to values representing test scenarios and duplicated them 100 times. We created Manipula programs to set the values in fields and to duplicate the cases 100 times. Once the manipula programs loaded the data and the Blaise program automatically generated the random numbers for each case were ready to run our systematic analysis of the results. We conducted our analysis on the large numbers cases to see if the program was truly yielding the selections at the targeted sampling rates.

Final testing is **End to End Testing**. We integrate our sampling algorithm into the larger system which includes a (1) study management system (SMS) calling a (2) fully implemented instrument which collects the roster and then calls the (3) sampling algorithm. There are lots of data moving between the components of the system. There are data passed as parameters to the instrument from the SMS and data generated by the instrument collecting the roster. Both the data from the SMS parameters and the data collected by the instrument are used by the sampling algorithm. All of this data have to work together and can be complex. We have to make sure all of the components are using correct fields in the correct places with correct field types. Once all three types of testing are completed we can feel confident that our complex sampling algorithm will work in the field.

Conclusion

Everyone involved with the project was pleased at how well the sampling algorithm was implemented in Blaise both in performance and results. It was a huge advantage to stay with the same technology used by the data collection instruments. This way we avoided the additional complexities of maintaining and integrating different software and managing more licenses.