Paper 1793-2018

Runtime Validation of SAS® Jobs: A Threesome of Error-Throwing Macros

Quentin McMullen, Siemens Healthineers

ABSTRACT

SAS® programming can feel fraught with danger. Each time a program is executed, there is a risk that an error in the source data or the code will lead to undetected erroneous results. Runtime validation is an offensive programming technique designed to decrease that risk by adding code to detect and report errors as a program runs.

This paper presents three utility macros for runtime validation of SAS jobs:

- %DupCk detects duplicate key values in a dataset.
- %Assert detects values in a dataset that are invalid or unexpected.
- %CheckRecordCounts detects the accidental deletion of records.

By increasing the detection rates of common errors, these utilities increase the programmer's confidence in their results. Principles of runtime validation are discussed, as are principles of macro design encountered during the development of the macros.

INTRODUCTION

When I started programming in SAS®, I was afraid I would make a mistake. My biggest fear was I would make a mistake, and I would not detect that mistake until it was "too late." "Too late" might mean I had delivered incorrect results to an internal customer, or worse, to an external customer. I was afraid of the undetected error. To reduce the risk of making an undetected error, I learned different ways to check my results, such as using the FREQ procedure or PRINT procedure to produce check tables, and reviewing the SAS log to check record counts. But I soon realized that most of my methods for checking results were manual. As a self-aware programmer, I recognized that I was lazy (efficient). I knew that even if I took the time to review a set of check tables after I developed a program, I wasn't likely to repeat that review *every time I ran the program*. I needed a way to automate my checks. In time, I learned to automate these checks, through the implementation of offensive programming techniques. This paper presents three offensive macros, designed to protect against surprises in data and code.

What is offensive programming[1]? It's actually a type of defensive programming. If defensive programming is a collection of techniques employed to handle problems in data, code, and other surprises in the computing environment, then offensive programming is a subset of those techniques that say "if a problem is encountered, generate an error." Offensive programming is a way to make sure that unexpected conditions are immediately raised to the attention of the programmer. A key tenet is that if a program encounters such a condition, it should "fail loudly," rather than quietly continue and risk returning incorrect results (Miner, 2008).

The SAS log is your friend. If SAS detects an error in your code, it will write an error message to your log (I use the generic term error message, to include actual ERROR: messages as well as WARNING: messages and even bad NOTE: messages). SAS will often continue to execute code and generate results despite errors that have been encountered. As a programmer, you are responsible for reviewing the log after submitting code, to make sure that SAS has not generated any error messages. Most programmers learn the maxim that *SAS results should not be trusted until the log has been reviewed*. Log review is such a critical step of the SAS programming process, that many papers have been written about methods for automating review of the log (see Hughes, 2017 for a review).

While SAS will automatically detect many types of errors in your code, including compile-time syntax errors and execution-time errors, there are many errors which SAS cannot detect automatically, which are primarily errors in logic. These are errors that occur when you have submitted code that is syntactically

[1] https://en.wikipedia.org/wiki/Offensive_programming

correct, and the data being processed satisfy the expectations of the code, but you may have simply written incorrect code, or the data may not satisfy your expectations. Because SAS cannot know the *intent* of your code, SAS cannot detect such errors in logic. As you write code, the intent of your code, and your expectations regarding the code and data, are defined only in your head. However, you can write extra code that will verify that the expectations in your head have been satisfied. If your expectations are not satisfied, then there is an error in your expectations, your code, or your data. Regardless of the source of the error, the most critical step of error resolution is error detection. Offensive programming methods in SAS allow you to generate error messages in the log when logical errors occur.

OFFENSIVE PROGRAMMING IN SAS (NO MACROS YET)

The concept of offensive programming is a native part of the SAS language. While SAS tends to be quietly fault-tolerant by default, it also provides many options to support the development of programs that will fail loudly when errors are encountered.

As an example of fault tolerance, consider that in the DATA step, there are many errors that will result in only a NOTE: being written to the log, or no message at all. Famously, if code references a variable that does not exist (perhaps because of a typo in a variable name), rather than generate an error message SAS will create the misspelled variable and write only a NOTE to the log:

```
1    data want ;
2      set sashelp.class ;
3      x=NoSuchVar ;
4    run ;

NOTE: Variable NoSuchVar is uninitialized.
```

To the offensive programmer, this is madness. If code references a variable that does not exist, why would it not generate an error? Prior to SAS 9.4, the only solace was that at least a NOTE is generated in the log, and an offensive programmer could choose to treat that note as an error message. In 9.4, SAS introduced a system option, VARINITCHK, which can be used to change this NOTE into an actual ERROR:

```
1    options varinitchk=error;
2
3    data want ;
4      set sashelp.class ;
5      x=NoSuchVar ;
6    run ;

ERROR: Variable NoSuchVar is uninitialized.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.WANT may be incomplete.  When this step was
stopped there were 0 observations and 7 variables.
```

SAS provides many system options which can make it more likely that mistakes in code (or data) will generate error messages in the log. Some of my favorites are shown in Table 1.

Table 1. Error Handling System Options

System Option	Description (from SAS 9.4 Systems Options: Reference)
dkricond=error	sets the error flag and writes an error message to the SAS log when a variable is missing from an input data set during the processing of a DROP=, KEEP=, or RENAME= data set option
dkrocond=error	sets the error flag and writes an error message to the SAS log when a variable is missing from an output data set during the processing of a DROP=, KEEP=, or RENAME= data set option

fmterr	specifies that when SAS cannot find a specified variable format, it generates an error message and does not allow default substitution to occur
mergenoby=error	Specifies that an error message is issued when MERGE processing occurs without an associated BY statement.
msglevel=i	specifies that INFO: notes are written to the log when a MERGE statement would cause variables to be overwritten
noautocorrect	specifies that SAS does not automatically attempt to correct misspelled procedure names, misspelled procedure keywords, or misspelled global statement names
varinitchk=error	specifies that the DATA step stops executing and writes an error message to the SAS log when a variable is not initialized
varlenchk=error	specifies that an error message is issued when the length of a variable that is being read is larger than the length that is defined for the variable

While the number of such error handling options has grown over time, even with such options set to their most conservative setting, there are still many log notes which the offensive programmers would prefer to treat as errors. For example, if a numeric expression includes a character variable as an argument, SAS will convert the character values to numeric values (using a default informat), with only a NOTE to the log: "Character values have been converted to numeric values." The strict offensive programmer would treat this note as an error, as it is safer to code an explicit character to numeric conversion rather than allow SAS to guess that the conversion is intended. For the offensive programmer who is willing to use undocumented SAS features, there is an undocumented option, DSOPTIONS=note2err, which will turn this and many other potentially problematic notes into errors:

```
1    options dsoptions=note2err ;
2    data want ;
3      x = '1' * 1 ;
ERROR: Character value found where numeric value needed at line 3 column 7.
4    run;

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.WANT may be incomplete.  When this step was
stopped there were 0 observations and 1 variables.
```

The option can be turned off with DSOPTIONS=nonote2err. While DSOPTIONS=note2err is still undocumented, Andrew Ratcliffe (2009) has published an interesting blog post where he provides some insights on its use and history. Apparently it was designed as an internal option for use by SAS testers.

When SAS detects an error, it notifies the programmer by writing an error message to the log. An error is the log is the primary way that SAS will "fail loudly." Through features such as the system options described above, the SAS language allows the programmer to decide which conditions should constitute an error. The macros presented below are simply additional error detection tools.

%DUPCK

Duplicate values can ruin your whole day. When working with SAS data sets, it's important to know what variable (or group of variables) identifies a unique record. A PATIENTS data set might have PATIENTID as a unique identifier; a VISITS dataset might have PATIENTID VISITID as a unique identifier. If somehow your PATIENTS dataset has multiple records with the same PATIENTID, you have a problem. That problem could become magnified as you work with the data. One common place where unexpected duplicate values cause problems is the MERGE statement. When merging two data sets, it is important to know that at least one of the two data sets is unique by the BY variables used in the merge, because the MERGE statement does not handle many-to-many merges nicely (Tilanus, 2008).

Early in my programming career, when I realized the importance of unique BY values in a merge, I got in the habit of using PROC SORT NODUPKEY to confirm that the BY variables I expected to be unique really were unique. I would sort by the BY variables, without creating an output dataset, and read the log to check that the number of duplicate variables was zero. Thus my log from merging two datasets might look like:

```
1    *Confirm Patients data set is unique by PatientID ;
2    proc sort nodupkey data=Patients out=_null_ ;
3      by PatientID ;
4    run ;

NOTE: There were 3 observations read from the data set WORK.PATIENTS.
NOTE: 0 observations with duplicate key values were deleted.

5
6    data want ;
7      merge Patients Visits ;
8      by PatientID ;
9    run;
```

When I reviewed the log, if the PROC SORT NODUPKEY note indicated that one or more observations with duplicate key values had been deleted, I would know there was a problem in my data.

Adding PROC SORT NODUPKEY to check for duplicate values was a small step toward offensive programming, because it added a NOTE to the log which reported the number of duplicate records. But it would be hard to argue that the note constitutes “Failing Loudly.” I wasn’t likely to re-read the log note every time I resubmitted the code. I really wanted an error message in the log if duplicates were detected. As a SAS programmer, an error message in the log is the best way for SAS to let me know that something has gone wrong.

Luckily, SAS makes it easy to produce user-generated error messages, using the PUT statement (or PUTLOG statement introduced in version 9). BY-group processing can be used in the DATA step to check for duplicate values of the BY variable(s). The below step will check the PATIENTS data set for duplicate values, and will write an error message to the log if any duplicate values are detected:

```
*Confirm Patients data set is unique by PatientID ;
data _null_ ;
  set Patients ;
  by PatientID ;
  if not (first.PatientID and last.PatientID) then
    putlog "ERROR: Duplicate records found " PatientID= ;
run ;
```

When that code runs, if there are any records with duplicate values, they will generate an error in the log, e.g.:

```
1    data _null_ ;
2      set Patients ;
3      by PatientID ;
4      if not (first.PatientID and last.PatientID) then putlog "ERROR:
Duplicate records found "
4  ! PatientID= ;
5    run ;

ERROR: Duplicate records found PatientID=2
ERROR: Duplicate records found PatientID=2
```

The below log shows a merge of the PATIENTS and VISITS data sets, with an offensive DATA _NULL_ step added to confirm that there are no duplicate PatientID values in the PATIENTS data set:

```
1    *Confirm the Patients data set is unique by PatientID ;
2    data _null_ ;
3      set Patients ;
4      by PatientID ;
5      if not (first.PatientID and last.PatientID) then putlog "ERROR:
5  ! Duplicate records found " PatientID= ;
6    run ;

NOTE: There were 3 observations read from the data set WORK.PATIENTS.

7
8    data want ;
9      merge Patients Visits ;
10     by PatientID ;
11   run;
```

When the offensive DATA _NULL_ step runs, it validates my assumption that there are no duplicate PatientID values in the PATIENTS dataset. The absence of error messages in the log confirms that there are no duplicate BY-values. After I run my code, I don't need to use my eyes to read the individual log notes to confirm that 0 duplicates were detected. If my log has no error messages, then there are no duplicates.

Once I was in the habit of checking for duplicate values, I decided to write a macro that would generate a DATA _NULL_ step to check for duplicates. As is typical of the macro language, the benefit of writing a macro is not that it provides some new functionality which cannot be had from the SAS language alone. The benefit is that it provides me with an easy way to check for duplicates, because I will need to type less code. Once I have made it easy to check for duplicates, I am more likely to check for duplicates in the future rather than just assume that there are no duplicates.

The initial version of the macro looked like:

```
%macro dupck
  (data
  ,by=
   )
;

data _null_ ;
  set &data(keep=&by) ;
  by &by ;
  if not (first.&by and last.&by) then
    put "ERROR: Duplicate records found " &by= ;
run;

%mend dupck;
```

The macro is basically a wrapper to generate the DATA _NULL_ step. It is called like `%dupck(Patients, by=PatientID)`, and if it detects duplicate values, the log would look like:

```
1    options mprint ;
2    %dupck(Patients, by=PatientID)
MPRINT(DUPCK):   data _null_ ;
MPRINT(DUPCK):   set Patients(keep=PatientID) ;
MPRINT(DUPCK):   by PatientID ;
MPRINT(DUPCK):   if not (first.PatientID and last.PatientID) then put
"ERROR: Duplicate records found " PatientID= ;
MPRINT(DUPCK):   run;

ERROR: Duplicate records found PatientID=2
ERROR: Duplicate records found PatientID=2
```

There are important limitations to that initial version. In particular, it cannot handle multiple BY-variables. The below enhanced version of the macro handles a list of BY-variables, and does some minimal parameter validation:

```
%macro dupck
  (data        /* (r) name of dataset to be checked for duplicates */
  ,by=         /* (r) space delimited list of by variables */
  ,out=_null_  /* (o) name of output dataset holding duplicates */
   )
;

%local lastvar ;

%if %missingRequiredParameters(data by) %then %goto mexit ;

%*Create a view, to honor any parenthetical options passed with &data ;
data __dupckdata (keep=&by ) / view=__dupckdata ;
  set &data ;
run ;

%let lastvar=%sysfunc(scan(&by,-1,%str( ))) ;

data &out ;
  set __dupckdata ;
  by &by ;
  if NOT (first.&lastvar and last.&lastvar) then do ;
    putlog "ERROR: Dataset %superq(data) is not unique by &by: " (&by)(=) ;
    output ;
  end ;
run ;

proc delete data=__dupckdata (memtype=view) ;
run ;

%Mexit:
%mend dupck;
```

The design of the macro is discussed below, but at its core, it’s still just a wrapper for a DATA _NULL_ step that uses BY-group processing to detect duplicate values. Invoking the macro to check for duplicate values before the merge might look like:

```
%dupck(Patients, by=PatientID)
%dupck(Visits, by=PatientID VisitID)

data want ;
  merge Patients Visits ;
  by PatientID ;
run ;
```

If that code runs and there are no error messages in the log, I know there are no duplicate values. The close reader might notice that I added a second call to %DupCk, to check that the VISITS data set is unique by PatientID VisitID. Strictly speaking this is not necessary for the success of the MERGE step. But logically, I expect my VISITS dataset to be unique by PatientID and VisitID. It is helpful to confirm this expectation, because if it is not met, there is either a problem in my data, or a problem in my understanding of my data. In either case, I would like to be notified of that problem as soon as possible. By making it easy to check for duplicates, I make it more likely that I will check for duplicates, and thus more likely that I will detect duplicates if they exist.

MACRO DESIGN ISSUES (AS ILLUSTRATED BY %DUPCK)

Like most macros, the job of %DupCk is to generate some SAS code, in this case a DATA _NULL_ step which will check for duplicate values of a list of BY-variables. Macros are primarily tools to be used by SAS programmers, to make it easier to generate SAS code with less typing. When I develop a macro, I assume that the programmers who will be using the macro are experienced with the SAS language. If they are not yet experienced with the SAS language, they should be writing their own SAS code, without adding the complexity of using the macro language to generate SAS code. That assumption helps guide many macro design decisions.

MACRO NAMES

Macro names should identify the purpose of the macro. Generally I choose longer names that describe the purpose of a macro. I might have been tempted to call this macro %DuplicateCheck. However, in this case I erred on the side of a shorter name. I did this because I thought this would be a macro I called a lot, and if I made it a shorter name, it would be easier to type. I also liked the symmetry with the IntCk function.

PARAMETER TYPES

Keyword macro parameters are better than positional parameters. One benefit of keyword parameters is the macro can define a default value for the parameters. Almost more importantly, keyword parameters are easier to use when invoking a macro, because you don’t need to remember the order of the parameters. SAS functions use positional parameters, and I often find them annoying. I’m one of those people who can never remember the order of parameters to TRANWRD() and TRANSTR(). I would rather remember the name of a parameter than the order of a parameter, because programmers are used to remembering names (names of datasets, variables, procedures, etc.).

The only benefit to positional parameters I am aware of is that you do not need to type the parameter name when invoking the macro. If a macro has only a single positional parameter, I feel it’s acceptable to use a positional parameter for the primary parameter (often &data), as there is less burden on the macro user because they do not need to remember an order. In the rare case of a macro with multiple positional parameters where the order does not matter (e.g. because they are arguments to a commutative operation such as addition), multiple positional parameters might be acceptable. One nice feature of the macro language is that even if a macro defines a parameter as positional, it can still be invoked as a keyword parameter. Thus the following three calls all succeed, and generate identical SAS code:

```
%dupck(Patients, by=PatientID)
%dupck(data=Patients, by=PatientID)
%dupck(by=PatientID, data=Patients)
```

PARAMETER NAMES

Macro parameter names should use terms from the SAS language, for increased ease of use. When I first learned the macro language, I remember seeing macro calls like `%DoSomething(dsn=mydata)`. I remember wondering what that DSN meant, until it hit me, “oh, data set name.” For a brief time, I wrote macros like `%DoSomething(dsn=,dsv=)`. I felt I had learned the secret parameter-naming convention of macro authors. Then years later I was lucky to take a macro course taught by Ian Whitlock. He asked me to consider how much easier my macros would be to use, if I used parameter names that would be obvious to any SAS programmer. So instead of using DSN as a parameter name for data set name, use DATA, because the SAS language uses DATA as a parameter name on every PROC step. Instead of using DSV as parameter for data set variable, use VAR, because every SAS programmer knows what a VAR statement is. And a BY statement. If I read `%DupCk(mydata, by=id)` I have a pretty good idea about what the code generated by the macro will do, without even looking at the macro definition.

PARAMETER VALIDATION

Some macro developers believe that a proper production macro should have error handling code (defensive programming) to validate parameter values passed by a user and prevent any SAS code errors caused by a user passing an invalid value (such as the name of a data set that does not exist) from generating errors in the log. I disagree. Guided by the belief that macro users should be experienced SAS programmers, for the most part I think the macro user should be trusted to interpret most log errors produced by a bad macro call. For example, when invoking %DupCk I expect that the macro user will pass the name of a data set that exists, and a BY-variable that exists, and that the data set will be sorted by the BY-variable. What happens if the user makes a mistake? They will get an error in the log, but one that they are familiar with, because they are SAS programmers:

```
%dupck(NoSuchData, by=id)
ERROR: File WORK.NOSUCHDATA.DATA does not exist.
NOTE: The SAS System stopped processing this step because of errors.

%dupck(sashelp.class, by=NoSuchVar)
WARNING: The variable NoSuchVar in the DROP, KEEP, or RENAME list has
         never been referenced.

NOTE: Variable NoSuchVar is uninitialized.
ERROR: BY variable NoSuchVar is not on input data set WORK.__DUPCKDATA.
NOTE: The SAS System stopped processing this step because of errors.

%dupck(sashelp.class, by=Sex)
ERROR: BY variables are not properly sorted on data set WORK.__DUPCKDATA.
last=0 Sex=M FIRST.Sex=1 LAST.Sex=1 _ERROR_=1 _N_=1
NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: There were 2 observations read from the data set WORK.__DUPCKDATA.
```

There would be little benefit to me writing macro code to check that the data set name passed by the macro user actually names an existing dataset, and write a custom error message if it doesn't exist, because when the SAS code executes it will do that for me, and will return an error message that the user already knows how to interpret.

Some might argue that if the input data set passed to %DupCk is not sorted by the variables listed in the BY parameter, the macro should sort the data set rather than return an error. But that is not the way the BY statement works in the SAS language. If the macro changed the sort order of the input data set, many users might see that as an unwanted side effect of using the macro. What if you want to allow a user to check a data set for duplicates without having a sorted data set? A possible enhancement to the macro would be to add a feature to use a hash table to identify duplicate values, as described by Dorfman & Henderson (2017), rather than BY-group processing. Hash tables do not require data to be sorted, but there are trade-offs between the BY-group approach and the hash table approach. To provide flexibility for users, we could add a new macro parameter KEY, and make both the KEY parameter and the BY parameter optional. If the user specifies BY variables, the macro would use BY-group processing to check for duplicates; if a user specifies KEY variables, it would use a hash table.

Often the most parameter checking I will do is to check that all of the required parameters are not null. For that, I use a utility macro %MissingRequiredParameters. It is a function-style macro that takes a list of macro variable names as an argument, prints an error message to the log if any macro variable has a null value, and returns a count of the number of macro variables that are null. If the count of null macro variables is greater than 0, the main macro will exit early. The macro definition is:

```
%macro MissingRequiredParameters
  (paramList /*space-delimited list of required macro parameters*/
   )
;

%local
  i
  param_i
  MissingCount /*count of missing parameters*/
  MacroCalledBy /*name of macro that called this macro*/
;

%let MacroCalledBy=%sysmexecname(%sysmexecdepth - 1) ;

%let MissingCount=0 ;
%do i=1 %to %sysfunc(countw(&paramList,%str( ))) ;
  %let param_i=%scan(&paramList,&i,%str( )) ;

  %if %eval( %length(%superq(&param_i))=0 ) %then %do ;
    %put ERROR: (%nrstr(%%)&MacroCalledBy)
/*     */Missing required parameter: %nrstr(&)&param_i ;
    %let MissingCount=%eval(&MissingCount + 1) ;
  %end;
%end ;

%if &MissingCount %then
  %put ERROR: (%nrstr(%%)&MacroCalledBy) Macro exiting. ;

&MissingCount  /*return value*/

%mend missingRequiredParameters ;
```

If the invocation of %DupCk does not specify a value for both the DATA parameter and the BY parameter, it will exit early and write an error message to the log:

```
1    %dupck(data=Patient)
ERROR: (%DUPCK) Missing required parameter: &by
ERROR: (%DUPCK) Macro exiting.
```

The decision about how best to check if a macro variable has a null value is surprisingly complex. See Chung & King (2009) for an excellent discussion of this issue.

VARIABLE LISTS

A SAS BY statement supports multiple BY-variables. If I write a macro with a BY parameter, it should also support multiple BY-variables, to allow an invocation like: `%dupck(Visits, by=PatientID VisitID)`. To support this, the macro uses the SCAN function to identify the name of the last variable on the BY-list, to use in BY-group processing, and uses the parenthesis notation in the PUT statement that generates the error message: `(&by)(=)`,to write the value of each BY variable in the error message. One limitation of the current design is that it expects the list of BY variables to be a space-delimited list of variable names. While this is the most common type of variable list, SAS does allow variable names with spaces in them (named literals), such as “My Var”n, which would not be correctly parsed by the macro.

DATA SET OPTIONS

In SAS a data set name can be passed with options, such as drop= keep= where=. When a macro has a DATA parameter, ideally it should be able to accept any valid data set specification, which includes honoring parenthetical options. The user of the macro knows that the SAS language allows them to specify options for data sets, and macro design should not prevent them from doing so. Because the macro introduced its own parenthetical option `(keep=&by)`, it starts by creating a VIEW which honors the user's options, then adds the KEEP option. With that hurdle out of the way, the user may pass it a data set name with a WHERE option like: `%dupck(sashelp.zipcode(where=(statecode='CO' and CountyNm='Denver')),by=zip)`. An alternative approach for allowing users to generate a WHERE option would be to add a WHERE parameter to the macro. But with that, the user would only have the ability to generate a WHERE option; the macro wouldn't support all the other dataset options available in the SAS language. By accepting options passed in the standard SAS way, a user could pass a convoluted data set specification such as `%dupck(sashelp.zipcode(firstobs=10 obs=100 rename=(zip=MyZip) where=(statecode='CO' and CountyNm='Denver')),by=MyZip)`. The user should be allowed to pass any valid data set specification.

CLEAN UP

Macros should always clean up after themselves, so as to avoid causing side-effects that might be a problem for the macro user. Potential side-effects to avoid include changing the user's system options, and polluting the user's work library with temporary data sets that are conceptually internal to the macro. %DupCk cleans up after itself by using the DELETE procedure to delete the view that was created by the macro, __dupckdata. For more on testing macros for potential side-effects, see Frank DiIorio's (2010) macro %WhatChanged.

%ASSERT

%DupCk made it easy for me to detect one specific problem which could cause undetected errors in my SAS programs: duplicate values of an ID variable which I expect to be unique. Of course there are many other types of mistakes which can cause undetected errors; among the most common problems in DATA step programming are invalid (or unexpected) values and coding mistakes which lead to surprising results. As an offensive programmer, I want a way to detect when such mistakes occur and generate an error message in the log. Luckily, after that macro course I took with Ian Whitlock, he mentioned off-handedly, "I wish SAS had an ASSERT statement. Someone should write an Assert Macro." And I asked, "What's an ASSERT statement?"

Many languages have an ASSERT statement, which allows you to state a Boolean expression which is expected to be true, and if the expression is false, it will generate an error. SAS does not have an ASSERT statement, but it's possible to achieve similar functionality. Consider a simple age calculation[2]:

```
data want ;
  set have ;
  Age = (VisitDate - DOB) / 365.25 ;
run ;
```

What could go wrong? Well, lots. The DOB could be missing, or could be in the future. The VISITDATE could be missing, or could be in the future. Both DOB and VISITDATE could be non-missing and within the expected range of values, but the VISITDATE could be before the DOB, resulting in an embarrassing negative age. One simple check would be to confirm that the calculated value for AGE is at least 18. This could be done with a conditional PUTLOG statement:

```
data want ;
  set have ;
  Age = (VisitDate - DOB) / 365.25 ;
  if Age < 18 then putlog "ERROR: Invalid age " Age= ;
```

[2] This is one simple way to compute age. For a review of alternatives, see Carpenter (2011).

```
run ;
```

That will throw an error message if the AGE which is derived is less than 18 (which includes potential embarrassments like negative values for AGE, and missing values). The Boolean expression above evaluates whether an invalid value has been computed (Age < 18), and writes an error to the log if the expression is true. An assertion evaluates whether a valid value has been computed (Age >= 18), and writes an error to the log if the expression is false. If SAS had an ASSERT statement, it might look like:

```
data want ;
  set have ;
  Age = (VisitDate - DOB) / 365.25 ;
  assert (Age > 18) ; *Write an error if age is NOT greater than 18 ;
run ;
```

Not having an ASSERT statement, we can code a simple one line-macro:

```
%macro assert
  (assertion  /* (R) Boolean expression */
  ,msg=       /* (O) Message to print to log, if Boolean is false */
   )
;
  if NOT (&assertion) then
    putlog "ERROR: Assertion (%superq(assertion)) is FALSE. " &msg;
%mend assert;
```

Which would be used like:

```
data want ;
  set have ;
  Age = (VisitDate - DOB) / 365.25 ;
  %assert(Age > 18)
run ;
```

The macro has a single required positional parameter, ASSERTION, which is a Boolean expression that the programmer expects to be true. Any valid SAS expression can be passed as the assertion. It has an optional keyword parameter, MSG, to write a customized error message to the log. The macro generates a DATA step IF statement to evaluate the expression, not a macro %IF statement, because it is evaluating a DATA step language expression, which will usually reference variables in the program data vector. If that expression is NOT true, the macro writes an error message to the log.

Consider an example where you have a date of birth that is in the future, causing computed age to be negative:

```
data want ;
  DOB = "03Nov2101"d ;
  Age = (today() - DOB) / 365.25 ;
  %assert(Age >= 18)
run ;
```

When the code runs, the asserted expression will evaluate to false, and the assertion will write an error message to the log:

```
ERROR: Assertion (Age >= 18) is FALSE.
```

The classic programming text The Pragmatic Programmer (Hunt & Thomas, 2000) has a great section on “assertive programming.” Assertions validate data (and code) at runtime, as code is executing. Assertions can be coded to confirm any expectation (or assumption) a programmer has about their data. Hunt & Thomas propose a simple rule: “Whenever you find yourself thinking ‘but of course that could never happen,’ add code to check it.” (p. 122)

It is useful to consider two logical types of assertions: pre-conditions are assertions used to validate the inputs to some algorithm, and post-conditions are assertions used to validate the outputs from some algorithm. Consider a slightly different age calculation example, computing age of a participant at an enrollment date. A more extensive use of assertions might look like:

```
data want ;
  set have ;
  %assert("01Jan1900"d <= DOB <= today(), msg=(ID DOB)(=)  )
  %assert("01Jan2015"d <= EnrollDate <= "31Dec2016"d
         ,msg=(ID EnrollDate)(=)
          )

  Age = (EnrollDate - DOB)/365.25;

  %assert(18 <= Age <= 120
         ,msg= "Participant not between 18 and 120 years old: " (ID Age)(=)
          )
run ;
```

Two assertions of pre-conditions are used to validate that each of the inputs to the age calculation algorithm are within the expected range. One assertion of a post-condition is used to validate that the computed AGE is within the expected range. All three of the macro invocations use the MSG parameter to generate a custom error message. Thus if the AGE assertion detects an out of range age, it will generate an error message like:

```
ERROR: Assertion (18 <= Age <= 120) is FALSE.
Participant not between 18 and 120 years old: ID=101 Age=-2.001368925
```

Note that %ASSERT, like %DUPCK, does not provide some new, exciting, complex functionality which is difficult to code in SAS without the macro language. All it generates is a simple IF statement! But it makes it easy to code an assertion, and any time you make it easier for a programmer to code something, it is more likely that they will do it. Perhaps more importantly, by implementing the concept of an ASSERT statement, it encourages you to identify all of the expectations you have about your data (and algorithms), and automatically confirm those expectations every time the program is run. This can provide a peace of mind which cannot be obtained through the manual review of “check tables” or data listings, particularly for programs that will be run repeatedly, processing source data that changes over time.

In addition to providing run-time validation, %Assert also serves an additional purpose of documenting the intent of code. Suppose I am reading code, and come across a MERGE step like below:

```
data C ;
  merge A B ;
  by id ;
run ;
```

Immediately, a number of questions enter my head: is this intended to be a one-to-one merge, a one-to-many merge, or a (disastrous) many-to-many merge? Should all of the records in A and B match, or is it expected that there may be mismatches? Of course SAS does not care about the *intent* of code, but programmers do. Compare that code to the below step, with assertions added:

```
data C ;
  merge A(in=ina) B(in=inb) ;
  by id ;
  %assert(first.id and last.id)
  %assert(ina and inb)
run ;
```

Adding the %Assert calls has made it clear that the programmer has designed this to be a one-to-one merge (first assertion), and that all records in A and B should match (second assertion). The %Assert calls have made the code more readable. They communicate information not contained in the main code.

Note that %DupCk is really an example of a specific assertion, that is, asserting that there are no duplicate values of some variable. I wrote %DupCk years before %Assert. If I had developed %Assert first, I might have replaced the central IF statement in %DupCk:

```
if NOT (first.&lastvar and last.&lastvar) then do ;
  put "ERROR: (USER) Dataset %superq(data) is not unique by &by: " (&by)(=)
;
```

with a call to %Assert:

```
%assert(first.&lastvar and last.&lastvar
       ,msg="Dataset %superq(data) is not unique by &by: " (&by)(=)
        )
```

McMullen (2012) describes %Assert in more detail, and provides a longer version of the macro with additional parameters that provide enhanced functionality (e.g. ability to specify whether false assertions should be reported as an error, warning, or note; limit the number of log messages written by failed assertions; and create a new variable which flags records that failed an assertion).

A warning: Once you get in the habit of using assertions, you may develop into what Hunt & Thomas (2000) refer to as a “paranoid programmer,” one who sees risks in every piece of code. In this case, “paranoid” is a term of endearment. For example, consider a simple statement which computes the tax for some item, based on whether the item is taxable, the tax rate, and the price:

```
if taxable = 1 then tax = taxrate * price ;
```

When you see such code, you will start seeing all sorts of things that could go wrong (either with the data you are processing now, or the data your program will be processing tomorrow, or next year). It looks like TAXABLE is expected to be a Boolean value, but what if there are missing values? What if TAXRATE is missing, or negative, or someone enters a TAXRATE as a percentage rather than a proportion? All of these possibilities risk creating an embarrassing miscalculation of the TAX for an item. As an offensive programmer, when you hear a little voice in your head say “That could never happen,” your reaction will be to code an assertion, to prove that it never happens, and make sure that when it does happen, your program will throw an error.

%CHECKRECORDCOUNTS

McMullen & Black (2011) argue that documenting record counts at the end of a SAS job is an important quality control step, which should be automated. Typically, a SAS program starts by reading one or more source datasets, and then processes the data in some way, before creating a final derived dataset. It is common for a program to apply inclusion or exclusion criteria which result in some source records being dropped from the derived dataset. Unfortunately, it is easy to accidentally drop more records than intended (with a mistake in a WHERE statement, subsetting IF, or inner join), or even accidentally add records (with a surprise many-to-many Cartesian product join, or multiple OUTPUT statements in a DATA step). For the offensive programmer, accounting for record counts provides a level of confidence that if such problems occur, they will be detected.

Consider the following log, from a program which generates a data set for a clinical trial with 100 records, then drops records with duplicate IDs, missing values for treatment assignment, or a score that is out of range:

```
1    data study ;
2      do id=1 to 33, 33, 35 to 100 ;
3        site =ceil (ranuni(3)*5) ;
4        tx   =round(ranuni(3)) ;
5        score=round(ranuni(3)*100) ;
```

```
6        if id IN (3,4)    then tx=. ;
7        if id =9          then score=200 ;
8        output ;
9      end ;
10   run ;

NOTE: The data set WORK.STUDY has 100 observations and 4 variables.

11   proc sort nodupkey data=study out=study1;
12     by id;
13   run;

NOTE: There were 100 observations read from the data set WORK.STUDY.
NOTE: 1 observations with duplicate key values were deleted.
NOTE: The data set WORK.STUDY1 has 99 observations and 4 variables.

14   data study2;
15     set study1;
16     if not missing(tx);
17   run;

NOTE: There were 99 observations read from the data set WORK.STUDY1.
NOTE: The data set WORK.STUDY2 has 97 observations and 4 variables.

18   data final;
19     set study2;
20     if NOT (0<=score<=100) then delete;
21   run;

NOTE: There were 97 observations read from the data set WORK.STUDY2.
NOTE: The data set WORK.FINAL has 96 observations and 4 variables.
```

Because the SAS log helpfully reports the counts of records in each dataset, it is possible to manually review the log to account for each record. The key word in that sentence is “manually,” which is anathema to lazy programmers. Any manual task is not likely to be performed reliably or repeatedly. The accounting can be automated by modifying the program, so that every time records are dropped in a step, the dropped records are output to a separate dataset. The log below is from the program, after it has been modified to create output data sets that hold all of the records that are dropped during processing:

```
1    data study;
2      do ID=1 to 33, 33, 35 to 100;
3        site=ceil(ranuni(3)*5);
4        tx=round(ranuni(3));
5        score=round(ranuni(3)*100);
6        if id IN (3,4)    then tx=.;
7        if id =9          then score=200;
8        output;
9      end;
10   run;

NOTE: The data set WORK.STUDY has 100 observations and 4 variables.

11   proc sort nodupkey
12     data=study
13     out=study1
14     dupout=drop_dupID (label="Dropped records: Duplicate ID")
15   ;
```

```
16     by id;
17   run;

NOTE: There were 100 observations read from the data set WORK.STUDY.
NOTE: 1 observations with duplicate key values were deleted.
NOTE: The data set WORK.STUDY1 has 99 observations and 4 variables.
NOTE: The data set WORK.DROP_DUPID has 1 observations and 4 variables.

18   data
19     study2
20     drop_badTx (label="Dropped records: Missing TX")
21   ;
22     set study1 end=last;
23     if missing(tx) then output drop_badtx;
24     if not missing(tx);
25     output study2;
26   run;

NOTE: There were 99 observations read from the data set WORK.STUDY1.
NOTE: The data set WORK.STUDY2 has 97 observations and 4 variables.
NOTE: The data set WORK.DROP_BADTX has 2 observations and 4 variables.

27   data
28     final
29     drop_badScore (label="Dropped records: Invalid Score")
30   ;
31     set study2 end=last;
32     if NOT (0<=score<=100) then do;
33       output drop_badscore;
34       delete;
35     end;
36     output final;
37   run;

NOTE: There were 97 observations read from the data set WORK.STUDY2.
NOTE: The data set WORK.FINAL has 96 observations and 4 variables.
NOTE: The data set WORK.DROP_BADSCORE has 1 observations and 4 variables.
```

The cost of creating output data sets with dropped records is typically only a few lines of code. After creating the dropped records data sets, accounting for the record counts becomes a simple reporting problem. We have the source data set, the final dataset, and separate data sets of dropped records. We expect that the number of records in the final data set is equal to the difference between the number of records in the source data set and the number of records that were dropped. If that is not the case, then some records have been dropped but are not accounted for.

%CheckRecordCounts accepts a list of data sets as a parameter, and then loops over the list, counting the number of records in each data set. It compares the record count of the first data set to the sum of the record counts of the remaining data sets. It uses a function-style macro, %MTCNTOBS (Hamilton, 2001) as a helper to do the actual record counting.

```
%macro CheckRecordCounts
  (data /*(R) space-delimited list of dataset names*/
   )
;

%local i data_i count_i cumulative ;
```

```
   %put ;
   %put %sysfunc(repeat(-,55)) ;
   %put Record Counts ;
   %put %sysfunc(repeat(-,55)) ;

   %do i=1 %to %sysfunc(countW(&data,%str( ))) ;
     %let data_i=%scan(&data,&i,%str( )) ;

     %*Count the number of records in the ith dataset;
     %let count_i=%mtcntobs(data=&data_i) ;

     %*Write to the log report ;
     %put &data_i %sysfunc(repeat(%str( )
                               ,50 - %length(&data_i) - %length(&count_i)
                                )) &count_i ;

     %*Decrement the cumulative count ;
     %if &i=1 %then %let cumulative=&count_i ;
     %else %let cumulative=%eval(&cumulative - &count_i) ;
   %end;
   %put %sysfunc(repeat(-,55)) ;
   %put ;

   %if &cumulative ne 0 %then
     %put ERROR: Record counts are inconsistent. ;

   %mend CheckRecordCounts ;
```

%CheckRecordCounts writes a little report to the log, listing the number of records in each dataset, e.g.:

```
39   %CheckRecordCounts(Study Drop_DupID Drop_BadTx Drop_BadScore Final)

--------------------------------------------------------
Record Counts
--------------------------------------------------------
Study                                                100
Drop_DupID                                             1
Drop_BadTx                                             2
Drop_BadScore                                          1
Final                                                 96
--------------------------------------------------------
```

If all of the dropped records are not accounted for, an error message is written to the log. Using the same sample data, if the list of datasets passed to %CheckRecordCounts did not include Drop_BadScore, the log would show:

```
39   %CheckRecordCounts(Study Drop_DupID Drop_BadTx Final)

--------------------------------------------------------
Record Counts
--------------------------------------------------------
Study                                                100
Drop_DupID                                             1
Drop_BadTx                                             2
Final                                                 96
--------------------------------------------------------

ERROR: Record counts are inconsistent.
```

Invoking %CheckRecordCounts at the end of a program provides the offensive programmer with an automated way to confirm that no records have been accidentally deleted (or created) during processing. Like %Assert, %CheckRecordCounts has a secondary communicative purpose. The macro call communicates the programmer's expectation regarding the branches in their code that drop records, and it documents the number of dropped records in the log. The practice of writing dropped records to output datasets also allows further possibilities, such as analysis of the dropped records themselves to identify any interesting patterns.

LOG ERRORS VS. RETURN CODES

The offensive macros presented above are all designed to write an ERROR message to the log if a problem is detected (duplicate values, a failed assertion, or a lost record). Such user-generated log messages are useful because the SAS log is the critical information source for understanding whether or not a SAS submission was successful. There are some authors, such as Hughes (2016), who argue that there are benefits to SAS code modules employing return codes to indicate when an error has been encountered, either in addition to log messages or as an alternative to log messages. Hughes points out that use of return codes can allow the design of dynamic programs which can not only detect errors but can react to them (re-submitting code that failed, or submitting an alternative algorithm). For those who prefer to use return codes when implementing offensive programming, the macros presented here could easily be adapted to set a return code. For example, %ASSERT could be adapted so that a false assertion sets the SYSCC macro variable used to store the return code for a SAS job:

```
%macro assert(assertion, msg=) ;
  if NOT (&assertion) then do ;
    putlog "ERROR: Assertion (%superq(assertion)) is FALSE." &msg ;
    call symputx("SYSCC","9") ;
  end ;
%mend assert ;
```

The offensive programming approach recommends that programs fail "loudly." You are free to decide what loudly means to you. For some people, loudly means when a program fails there is an error message in the log. For others, failing loudly means ending with a non-zero return code. For others yet, failing loudly means immediately aborting the SAS session.

CONCLUSION

Programs will fail. You will not be the first programmer in the world who will write a program that cannot fail. Failure is always an option. Programs that fail loudly are not a problem. The failure is noticed when it occurs, and can be investigated and resolved. Programs that fail quietly are a nightmare, particularly if they quietly generate erroneous results. Such quiet failures can lead to undetected errors making their way into results, reports, and decisions.

The offensive programming macros presented in this paper provide an automated defense against undetected errors in your code and data. They encourage you to define your expectations about your data and algorithms, and they validate that those expectations are satisfied every time your program runs. By increasing the likelihood that errors will be detected, they increase your confidence in your results.

REFERENCES

Carpenter, A. (2011). "Your Age In People Years: Not All Formulas Are the Same." Proceedings of the Pharmaceutical SAS Users Group 2011 Conference. Avaialble at https://pharmasug.org/proceedings/2011/CC/PharmaSUG-2011-CC20.pdf.

Chung, C.Y. & King, J. (2009). "Is This Macro Parameter Blank?" Proceedings of the SAS Global Forum 2009 Conference. Available at https://support.sas.com/resources/papers/proceedings09/022-2009.pdf.

DiIorio, F. (2010). "%whatChanged: A Tool for the Well-Behaved Macro." Proceedings of the SouthEast SAS Users Group 2010 Conference. Available at http://analytics.ncsu.edu/sesug/2010/BB01.DiIorio.pdf.

Dorfman, P.M. & Henderson, D. (2017). “Beyond Table Lookup: The Versatile SAS Hash Object.” Proceedings of the SAS Global Forum 2017 Conference. Available at http://support.sas.com/resources/papers/proceedings17/0821-2017.pdf.

Hamilton, J. (2001). “How Many Observations Are In My Data Set?” Proceedings of the SAS Users Group International 26th Conference. Available at http://www2.sas.com/proceedings/sugi26/p095-26.pdf.

Hughes, T.M. (2016). *SAS Data Analytic Development: Dimensions of Software Quality*. Hoboken, NJ: John Wiley & Sons, Inc.

Hughes, T.M. (2017). “Pinching Off Your SAS Log: Adapting from Loquacious to Laconic Logs To Facilitate Near-Real Time Log Parsing, Performance Analysis, and Dynamic, Data-Driven Design and Optimization.” Proceedings of the SouthEast SAS Users Group 2017 Conference. Available at http://analytics.ncsu.edu/sesug/2017/SESUG2017_Paper-209_Final_PDF.pdf.

Hunt, A. & Thomas, D. (2000). *The Pragmatic Programmer*. Reading, MA: Addison-Wesley.

McMullen, Q. & Black, J. (2011). “Don’t Lose Your Data! Tracking and Reporting on Dropped Records.” Proceedings of the NorthEast SAS Users Group 2011 Conference. Available at: https://www.lexjansen.com/nesug/nesug11/ds/ds06.pdf.

McMullen, Q. (2012). “%Assert() Your Way To Sleep-filled Nights: A One Line Data Validation Macro.” Proceedings of the NorthEast SAS Users Group 2012 Conference. Available at https://www.lexjansen.com/nesug/nesug12/cc/cc31.pdf.

Miner, A. (2008). “Fail Early, Fail Loudly.” Retrieved from http://oncodingstyle.blogspot.com/2008/10/fail-early-fail-loudly.html.

Ratcliffe, A. (2009). “NOTE: More About NOTE2ERR (a.k.a. Be Of Good Type).” Retrieved from http://www.notecolon.info/2009/10/note-more-about-note2err-aka-be-of-good.html.

SAS Institute Inc. (2016). *SAS 9.4 System Options: Reference, Fifth Edition*. Cary, NC: SAS Institute Inc.

Tilanus, E.W. (2008). “Set, Merge, and Beyond.” Proceedings of the SAS Global Forum 2008 Conference. Available at http://www2.sas.com/proceedings/forum2008/167-2008.pdf.

ACKNOWLEDGMENTS

I am grateful to the following people for taking the time to provide feedback on a draft of this paper: Tom Arnold; Elizabeth Axelrod; Scott Bass; Ron Fehd; and John King. Any remaining mistakes are of course my own.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Quentin McMullen
Siemens Healthineers
quentin.mcmullen@siemens-healthineers.com
qmcmullen@gmail.com