

Data-driven Validation Rules: Custom Data Validation Without Custom Programming

Don Hopkins, Ursa Logic Corporation, Durham, NC

ABSTRACT

One of the most expensive and time-consuming aspects of data management is the programming of validation rules to check for data values that may be incorrect. Validation rules – also known as “edit checks” – are essential for cleaning data that has been captured on paper forms (or via any method that allows invalid values to be recorded). The pharmaceutical industry, for example, relies heavily on validation rules to clean research data captured on paper forms at hospitals and clinics. There are many kinds of validation rules, including range checks, valid value checks, missing value checks, and checks for consistency between two or more variables. Implementing these kinds of checks through custom code is standard practice in many organizations, despite the high cost of creating and validating custom programs. A data-driven approach – especially one that handles multivariate consistency checks – can radically decrease the time and expense involved in creating custom validation rules. This paper describes a data-driven system for creating and applying validation rules to locate questionable data values in SAS datasets.

INTRODUCTION

Although paperless methods for entering data are increasingly common, there are still many contexts in which data is first captured on paper and then transferred to computer through some kind of transcription process. Unfortunately, capturing data in this way offers two golden opportunities for introducing error. Mistakes can be made when the original paper forms are filled out, and additional mistakes can be made when the data are transcribed from the forms to produce computer files.

To reduce the rate of error in transcribed data files, they are often subjected to a cleaning process, the purpose of which is to locate and correct erroneous values. There are a variety of methods for searching out erroneous values, but one common technique is to define and apply a set of “validation rules” (also known as “edit checks”). A validation rule is a condition that involves one or more data values. If the data values fail to satisfy the condition, they are considered “questionable” and are candidates for closer scrutiny.

Creating and applying validation rules is an important standard practice in the pharmaceutical industry, which generates large amounts of research data, most of which is collected on paper forms in hospitals and clinics. Pharma companies conduct thousands of studies each year. For the most part, validation rules have to be created independently for each study, because the kinds of data collected vary greatly from one study to the next. As a result, pharmaceutical companies and clinical research organizations often have programmers on staff who spend a major portion of their work time implementing validation rules. Each study gets its own set of customized validation programs. This practice is expensive and time-consuming.

An alternative to customized programs is to represent validation rules as “data” that can be interpreted and applied by a generic program. There are four essential components for such a system:

1. A schematic way of representing validation rules.
2. A component that allows users to enter and maintain validation rules.
3. A component that interprets and applies the previously entered validation rules to a set of “target” datasets.
4. A component that outputs information about data values that fail the validation rules.

There are many ways of implementing such a system. This paper describes one approach, which makes use of Microsoft Excel for entering and maintaining validation rules, SAS macro language for interpreting and applying them, and Proc SQL for reporting on rule failures.

The system discussed in this paper was developed for internal use at Ursa Logic Corporation. The goal of the development project was to simplify, streamline, and standardize the process of creating customized validation rules for large collections of data stored in SAS datasets. To facilitate the presentation, the system is referred to in this paper as the Data Checker.

Before turning to the system itself, it may be helpful to say a few words about the variety of rules supported by the Data Checker.

TYPES OF VALIDATION RULES

Validation rules can be divided into two categories: those that involve one variable (univariate rules) and those that involve multiple variables (multivariate rules). There are three common types of univariate rules.

A **range check** is a rule stating that a data value must fall within a given range. Here are some examples:

- WEIGHT must be between 100 and 200 pounds.
- BIRTH DATE must be between 01JAN1965 and 31DEC1990.

A **valid-value check** is a rule stating that a data value must equal one of a set of specified values. Here are some examples:

- SEX must equal “M” or “F”.
- STATE must equal “NC” or “SC” or “GA” or “FL”

A **missing-value check** is a rule stating that a variable is required, (i.e., it cannot have a missing value).

Univariate rules by themselves can locate a large percentage of the errors in a database, but there are many situations in which it is also useful to compare the values of two or more variables to determine if they are mutually consistent, or if they meet some criteria when combined. Here are some examples of multivariate rules:

- LENGTH times WIDTH must equal AREA.
- PHYSICAL EXAM DATE must be less than or equal to INVESTIGATOR SIGNATURE DATE
- WEIGHT at follow-up visit must not differ by more than 25% from WEIGHT at baseline visit.

Multivariate rules can be divided into three types based on how many datasets and records are involved in the rule. A **within-record** rule involves variables that are all on the same record. It is the easiest type of multivariate rule to implement. Greater programming challenges are presented by **cross-record** rules, which involve variables that are on different records within the same dataset, and **cross-table** rules, which involve variables in multiple datasets.

A RULE SPECIFICATION SCHEME

One requirement for the Data Checker was that it be capable of handling all the types of validation rules described above. The first step in achieving that goal was to devise a scheme for specifying rules as data. The specification format had to be simple enough to be used by the intended users, but rich enough to express the full range of common validation rules, and the specifications had to accomplish all of the following:

- Identify the variable(s) involved in a rule.
- Identify where the variable(s) could be found
- Identify the condition that the variable(s) were expected to meet
- Specify an appropriate message to alert users of a rule failure

During analysis work for the Data checker a choice was made to limit the scope of the system to rules that involve only one or two datasets. This constraint simplified the specification scheme appreciably without giving up much in the way of needed functionality. It is possible to conceive of rules that combine variables from three or more datasets, but this rarely happens in practice.

Given the limitation to two datasets, it was determined that all univariate rules and multivariate rules could be fully characterized using the following set of fields:

RuleID:	A unique identifier for the rule.
Description:	A free-form description of the rule.
DatasetA:	The name of the first target dataset involved in the rule. (Or the <i>only</i> dataset, in the case of univariate and within-record rules.)
FilterA:	Used to define a filter condition for DatasetA. Filters are useful for rules that apply to only a subset of the records in a dataset.
KeyVariablesA:	A comma-delimited list of the variable(s) that form unique keys for records in DatasetA.
RuleVariablesA:	A comma-delimited list of the variable(s) from DatasetA that are involved in the rule.
DatasetB:	The name of the second target dataset involved in the rule.
FilterB:	A filter condition for DatasetB.
KeyVariablesB:	A comma-delimited list of the variable(s) that form unique keys for records in

DatasetB:	
RuleVariablesB:	A comma-delimited list of the variable(s) from DatasetB that are involved in the rule.
MergeVariables:	A comma-delimited list of the variables that are used to join records from DatasetA to records in DatasetB.
GoodCondition:	The condition that the target variable(s) are expected to meet.
FailureMessage :	A message that is output when data values are found that fail the rule.

Note that some of the fields are optional and can be left blank. The fields associated with Dataset B and the field containing merge variables are used only for cross-record and cross-table rules. The fields for defining filters are used for rules that apply only to a subset of the records in a dataset.

SAMPLE RULE SPECIFICATIONS

To illustrate the specification scheme in action, two sample rules will be described in detail – one univariate rule and one multivariate rule. The sample rules will be used throughout the remaining sections to clarify programming techniques used in the Data Checker.

SAMPLE WEIGHT RANGE RULE

Health-related studies often include weight restrictions in the eligibility criteria for potential subjects. Expressed in English, a typical weight range rule would look something like this:

Subject weight must be equal to or greater than 100 pounds and less than or equal to 200 pounds.

Assume that the value for subject weight is stored in a variable called WEIGHT which is found in a dataset called PHYSEXAM. Assume also that, in this particular study, weight may be recorded in either pounds or kilograms. A variable called WEIGHTU identifies the unit of measurement. It is set to "L" for pounds and "K" for kilograms.

The sample validation rule applies only to records that have weight recorded in pounds. (A separate rule would be created for records that have weight in kilograms.) Expressed using the Data checker specification scheme, the rule would look something like this:

RuleID:	Phys-4
Description:	Subject weight in pounds should be between 100 and 200.
DatasetA:	PHYSEXAM
FilterA:	WEIGHTU = "L"
KeyVariablesA:	SUBJECT,VISIT
RuleVariablesA:	WEIGHT
Dataset:B	
FilterB:	
KeyVariablesB:	
RuleVariablesB:	
MergeVariables:	
GoodCondition:	100 LE WEIGHT LE 200
FailureMessage :	Subject weight is not within the expected range (100 to 200 lbs).

Only one dataset is involved in this rule, so the four fields for

specifying information about Dataset B are blank, as is the field for specifying merge variables.

SAMPLE DATE COMPARISON RULE

The data collected for health-related studies typically include a large number of dates. Many validation rules are created to check relationships among dates – to confirm, for example, that clinic exams occurred in the expected order, or to verify that a physical exam took place before the investigator signed off on the data collection forms. Expressed in English, a typical date comparison rule would look something like this:

The physical exam must be dated on or before the date of the investigator's signature.

Assume that the date of the physical exam is stored in a variable called PHEXDT in the dataset PHYSEXAM, and the date of the investigator's signature is stored in a variable called SIGDATE in the dataset INVSIG. Expressed using the Data Checker specification scheme, the rule would look something like this:

RuleID:	Phys-1
Description:	Physical exam date should precede or equal investigator's signature date.
DatasetA:	PHYSEXAM
FilterA:	
KeyVariablesA:	SUBJECT,VISIT
RuleVariablesA:	PHEXDT
Dataset:B	INVSIG
FilterB:	
KeyVariablesB:	SUBJECT
RuleVariablesB:	SIGDATE
MergeVariables:	SUBJECT
GoodCondition:	B.SIGDATE GE A.PHEXDT
FailureMessage :	Date of physical exam is after date of investigator's signature.

In this case, all fields are used except for the filters. Note that in the specification for *GoodCondition*, variable names are preceded by "a." or "b." to identify which dataset they belong to. The prefixing of the variables is an important part of the syntax for multivariate rules, as will become clear in the sections below.

PERSISTENT STORAGE FOR RULES

The rule specification scheme described above is the first essential component of the Data Checker. The second is a method of entering, storing, and maintaining rules using the conventions of the specification scheme.

Because the rules are represented as values in a set of fields, any technology that allows data to be entered through a table-like interface would be suitable for implementing the second component. One possibility is to store the rules in SAS datasets and build a rule-entry application using SAS AF. This approach could be used to provide a user interface that is extremely user-friendly, with selection lists for names of datasets and variables, as well as immediate error-trapping.

Spreadsheets are also good candidates for entering specifications arranged in a tabular format. The spreadsheet interface is familiar to many computer users and a serviceable rule-entry system can be set up in a matter of hours.

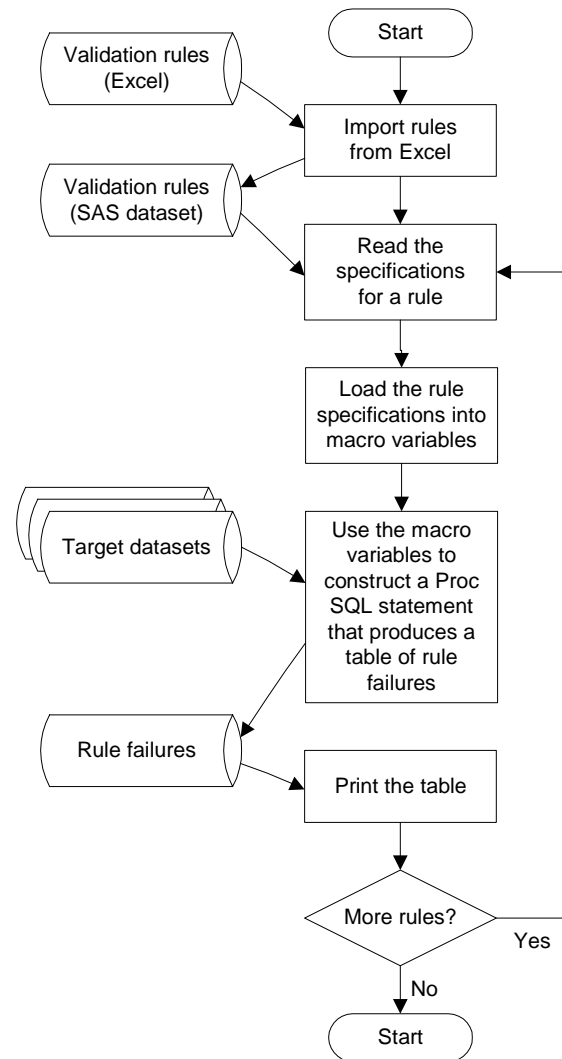
In order to minimize the development timeframe, the spreadsheet

approach was adopted for the Data Checker. Microsoft Excel was used to create a tool for entering validation rules. The Excel spreadsheet serves as input to the third component of the Data Checker, which was developed using Base SAS.

BASIC SYSTEM LOGIC

With specification syntax in place and a method for entering and storing validation rules, only two components remain to be considered: a component that can interpret rule specifications and apply them to target datasets, and a component to output information about data values that fail validation rules.

For the Data Checker, both of these components are provided by a single program written in Base SAS. The logic of the program is shown in the flowchart below.



IMPORTING RULES FROM EXCEL

Importing the validation rules from Excel to SAS is straightforward using Proc Import. Assume that the rules are contained in an Excel spreadsheet called *RULES.XLS* in the directory *C:\PROJECT\DATA*. The statements below copy the spreadsheet data to a SAS dataset called *RULES*. The

GETNAMES statement tells SAS to use text values in the first row of the spreadsheet as variable names.

```
proc import out=work.rules
  datafilez= "C:\project\data\Rules.xls"
  dbms=EXCEL2000 replace;
  getnames=YES;
```

LOOPING THROUGH RULES

After the rules are captured in a SAS dataset the Data Checker must loop through the rules and apply each in turn to the relevant target datasets. The looping is accomplished using the SAS macro language.

The macro *ApplyRules*, shown below, starts by counting the rules and storing the result in *&ruleCount*. It then executes a macro loop, processing one rule with each pass through the loop. During each pass the macro loads the specifications for the current rule into a set of macro variables. It then uses the variables to construct a Proc SQL statement that produces a table containing all data values that fail the rule. It executes the Proc SQL statement and then prints the result table.

```
%macro ApplyRules;
  * Set a macro variable with rules count;
  proc sql noprint;
    select count(*) into :ruleCount
    from rules;
  quit;

  * Loop through the rules;
  %do i = 1 %to &ruleCount;
    * Load rule specifications into macro
    * variables;
    proc sql noprint;
      select ruleID, description,
        datasetA, filterA,
        idVariablesA, ruleVariablesA,
        datasetB, filterB,
        idVariablesB, ruleVariablesB,
        goodCondition, failureMessage
      into :ruleID, :description,
        :datasetA, :filterA,
        :idVariablesA, ruleVariablesA,
        :datasetB, :filterB,
        :idVariablesB, ruleVariablesB,
        :goodCondition, :failureMessage
      from rules (firstobs=&i);
    quit;

    * Construct a Proc SQL statement to
    * produce a table of rule failures;
    proc sql
      create table ruleFailures as
      [select clause ...]
      [from clause ...]
      [where clause ...]
    quit;

    * Print the table;
    proc print data=ruleFailures;
      title "Rule ID: &ruleID";
    run;

  %end;
%mend ApplyRules;
```

CONSTRUCTING PROC SQL STATEMENTS

The bulk of the work in the Data Checker is accomplished by constructing a customized Proc SQL statement for each stored validation rule. Before delving into the macro code that is used to construct these statements, it may be useful to see examples of what the macro code needs to accomplish.

For the sample weight range rule, the Data Checker needs to generate Proc SQL code that looks like this:

```
proc sql;
  create table ruleFailures as
  select a.SUBJECT, a.VISIT, a.WEIGHT,
    "Subject weight is not within the
    expected range (100 - 200 lbs)."
  as failureMessage
  from datalib.PHYSEXAM
  (where=(WEIGHTU = "L")) a
  where not (100 LE WEIGHT LE 200);
quit;
```

When executed, the statements above will produce a table of rule failures, if there are any. The table will look like this:

subject	visit	weight	failureMessage
11MN16	1	98	Subject weight is not within the expected range (100 – 200 lbs).
11MN16	3	212	Subject weight is not within the expected range (100 – 200 lbs).

For the date comparison rule, the Data Checker needs to generate Proc SQL code that looks like this:

```
proc sql;
  create table ruleFailures as
  select a.SUBJECT as a_SUBJECT,
    a.VISIT as a_VISIT,
    a.PHEXDT as a_PHEXDT,
    b.SUBJECT as b_SUBJECT,
    b.SIGDATE as b_SIGDATE,
    "Date of physical exam is after date
    of investigator signature."
  as failureMessage
  from datalib.PHYSEXAM a, datalib.INVSIG b
  where a.SUBJECT=b.SUBJECT
  and not (B.SIGDATE GE A.PHEXDT );
quit;
```

Note that, in this case, the select clause renames most of the selected variables by adding a prefix of either "a_" or "b_". The prefixes are added because the validation rule involves variables from two datasets. In the result table, shown below, the prefixes identify which variables were pulled from each dataset. The "a_" variables came from the first dataset (PHYSEXAM) and the "b_" variables came from the second dataset (INVSIG).

a_subject	a_visit	a_phexdt	b_subject	b_sigdate	failureMessage
11MN16	1	8/1/94	11MN16	10/16/93	Date of . . .
11MN16	2	12/9/93	11MN16	10/16/93	Date of . . .
11MN16	3	2/9/95	11MN16	10/16/93	Date of . . .

The two examples illuminate the central task of the Data Checker. For each stored validation rule, the macro code must dynamically generate appropriate Proc SQL statements based on the specifications for the rule. This is accomplished by inserting

macro statements in the three parts of the code that need to be customized: the *select*, *from*, and *where* clauses. Before tackling the customized parts, however, the macro begins by generating the *proc sql* statement itself and the *create* clause, which are invariant:

```
proc sql;
  create table ruleFailures as
```

CONSTRUCTING THE SELECT CLAUSE

The *select* clause identifies variables that will be included in *ruleFailures*, the result table. The macro inserts the *select* verb followed by the names of all dataset variables needed by the rule. These names were previously loaded into four macro variables: *&idVariablesA*, *&ruleVariablesA*, *&idVariablesB*, and *&ruleVariablesB*. The macro must scan each of these macro variables, extract strings that represent names of dataset variables, and construct new strings that have the appropriate syntax for the *select* statement.

For the sample weight range rule, the four macro variables have the following values:

```
&idVariablesA:    SUBJECT,VISIT
&ruleVariablesA:  WEIGHT
&idVariablesB:    [blank]
&ruleVariablesB:  [blank]
```

Using these values the macro must construct the following string:

```
a.SUBJECT, a.VISIT, a.WEIGHT,
```

The task is a bit more challenging for the sample date comparison rule, because the dataset variables in the selection clause must also be renamed. In this case the four macro variables have the following values:

```
&idVariablesA:    SUBJECT,VISIT
&ruleVariablesA:  PHEXDT
&idVariablesB:    SUBJECT
&ruleVariablesB:  SIGDATE
```

Using these values the macro must construct the following strings:

```
a.SUBJECT as a_SUBJECT,
a.VISIT as a_VISIT,
a.PHEXDT as a_PHEXDT,
b.SUBJECT as b_SUBJECT,
b.SIGDATE as b_SIGDATE,
```

The macro code that performs these string manipulations makes use of several macro language techniques. To aid in understanding these techniques and how they produce the desired result, a portion of the code is presented in the text box below, side-by-side with an interpretation in pseudo code.

The macro logic assumes that each of the macro variables is either blank or contains a list of one or more variables names separated by commas. The *%scan* function is used to extract a variable name from a given position in the list.

The code block presented below deals with *&idVariablesA* only. In the complete program, this code block is followed by three similar code blocks that handle the variable names contained in *&ruleVariablesA*, *&idVariablesB*, and *&ruleVariablesB*.

After the dataset variables have been inserted in the *select* clause, there is only one variable remaining to be added – a container for the failure message. This variable is inserted by the following code fragment, which appears immediately after the four code blocks just described:

```
"&failureMessage" as failureMessage
```

Macro code:	Interpretation:
<pre>%let j=1;</pre>	set <i>&j</i> to 1
<pre>%let var= %scan(%quote (&idVariablesA),1,%str(,));</pre>	set <i>&var</i> to the string in the 1 st position of the list in <i>&idVariablesA</i>
<pre>%do %until (&var eq);</pre>	do until <i>&var</i> is blank
<pre> a.&var</pre>	in the Proc SQL statement that is being constructed, insert "a." followed by the value of <i>&var</i>
<pre> %if &datasetB ne %then %do; as a_&var</pre>	if <i>&datasetB</i> is non-blank then do insert "as a_" followed by the value of <i>&var</i>
<pre> %end;</pre>	end
<pre> ,</pre>	insert ",",
<pre> %let j=%eval(&j+1);</pre>	add 1 to <i>&j</i>
<pre> %let var=%scan(%quote (&idVariablesA),&j,%str(,));</pre>	set <i>&var</i> to the string in the <i>&j</i> th position of the list in <i>&idVariablesA</i>
<pre>%end;</pre>	end

For the two sample rules, this resolves as follows:

```
"Subject weight is not within the expected
range (100 - 200 lbs)." as failureMessage
```

```
"Date of physical exam is after date
of investigator signature."
as failureMessage
```

CONSTRUCTING THE FROM CLAUSE

Next to be tackled is the *from* clause, which identifies the target datasets that are involved in the rule and (hence) will provide input to the SQL procedure. To construct this part of the SQL statement, the macro needs the values that were previously loaded into *&datasetA*, *&filterA*, *&datasetB*, and *&filterB*.

The macro inserts the *from* verb followed by the name of the first dataset. It then checks to see if a filter has been specified for the dataset. If so, it constructs and inserts a *where* clause using the contents of *&filterA*. Finally, the macro inserts "as a" to create an alias for the dataset. (The alias is used in the *select* and *where* clauses).

If the rule involves a second dataset, the same techniques are used to insert the name of the second dataset, a *where* clause (if a filter is specified), and an alias of "b".

Shown below is the macro code that accomplishes all of this.

```
from
  datalib.&datasetA
  %if %quote(&filterA) ne %then %do;
    (where=(&filterA))
  %end;
  as a

  %if &datasetB ne %then %do;
    , datalib.&datasetB
    %if %quote(&filterB) ne %then %do;
      (where=(&filterB))
    %end;
  as b
%end;
```

Returning again to the two sample rules may help to clarify what is going on here. For the sample weight range rule, the macro variables are set as follows:

```
%datasetA:    PHYSEXAM
%filterA:     WEIGHTU = "L"
%datasetB:    [blank]
%filterB:     [blank]
```

On the basis of these specifications, the *from* clause generated by the macro looks like this:

```
from datalib.PHYSEXAM
  (where=(WEIGHTU = "L")) a
```

The specifications for the sample date comparison rule involve two datasets but no filters:

```
%datasetA:    PHYSEXAM
%filterA:     [blank]
%datasetB:    INVSIG
%filterB:     [blank]
```

These settings produce a *from* clause that looks like this:

```
from datalib.PHYSEXAM a, datalib.INVSIG b
```

CONSTRUCTING THE WHERE CLAUSE

The last bit of SQL that needs to be customized is the *where* clause. For rules that involve only one dataset, this clause is used for one purpose only, i.e., to find the subset of records that does NOT meet the specified "good" condition. Only one macro substitution is needed to construct the clause. The relevant setting for the sample weight range rule is:

```
&goodCondition: 100 LE WEIGHT LE 200
```

The Data checker must use the macro variable to construct the following *where* clause:

```
where not (100 LE WEIGHT LE 200);
```

The situation is more complex for rules that involve two datasets, because the *where* clause serves two purposes. As above, it subsets records using the value of *&goodCondition*, but it also performs the joins necessary to combine data from the two datasets. The macro variables involved in this construction are listed below, along with the values assigned to them for the sample date comparison rule:

```
&mergeVariables: SUBJECT
&goodCondition:  B.SIGDATE GE A.PHEXDT
```

The Data checker uses these macro variables to construct the following *where* clause:

```
where a.SUBJECT=b.SUBJECT
  and not (B.SIGDATE GE A.PHEXDT);
```

Techniques used to construct the *where* clause are similar to those used for the *select* clause, described above. The *%scan* function is used to extract strings from the list in *&mergeVariables*, and each string is then used to construct a new string. In this case, the new string has the form "a.[string] = b.[string]". These equivalencies – one for each merge variable – are the parts of the *where* clause that accomplish the joins between variable records. After the joins are constructed, the negation of *&goodCondition* is added, along with a semi-colon to finish off the SQL statement. The complete code block for constructing the *where* clause is as follows.

```
where
  %if &datasetB ne %then %do;
    %let j=1;
    %let andToken=;
    %let var=%scan(%quote
      &mergeVariables),1,%str(,);
    %do %while (&var ne);
      &andToken a.&var=b.&var
      %let j=%eval(&j+1);
      %let var =%scan(%quote
        &mergeVariables),&j,%str(,);
      %let andToken=and;
    %end;
  and
%end;

  not (&goodCondition)
;
```

EXTENSIONS TO THE BASIC SYSTEM

The Data Checker presented above is capable of detecting and reporting many kinds of potential data errors. In many data management shops it can be used as is to eliminate work that is currently being accomplished through customized, single-use reporting programs. However, some shops may need to extend the basic system to add additional functionality.

Two particularly useful extensions are resolvable message tokens and rule builders. Both of these extensions have been implemented at Urso Logic using Base SAS and macro language statements. The programming will not be detailed in this paper, but a description of the concepts and logic involved may be instructive for those who are interested in doing something similar.

RESOLVABLE MESSAGE TOKENS

The Data Checker allows users to specify a failure message – a text message that is automatically included on each record in the *ruleFailures* table. Recall that the failure messages specified for the two sample rules were:

Subject weight is not within the expected range (100 – 200 lbs).

Date of physical exam is after date of investigator signature.

In practice, it is often useful to customize failure messages with values taken from the target record. Consider, for example, the *ruleFailures* table that was generated by the sample weight range rule. The table is reproduced below:

subject	visit	weight	failureMessage
11MN16	1	98	Subject weight is not within the expected range (100 – 200 lbs).
11MN16	3	212	Subject weight is not within the expected range (100 – 200 lbs).

Note that the failure messages are identical for the two questionable data values. This may not always be desirable.

Sometimes failure messages are copied verbatim into email messages or memoranda that are generated to request confirmation or correction of questionable values. It may be useful in such cases for the Data Checker to produce a more complete English-language statement of the problem. The messages for the two records above might be expanded to the following:

The weight recorded for subject 11MN16 at Visit 1 is 98, which is not within the expected range (100 – 200 lbs).

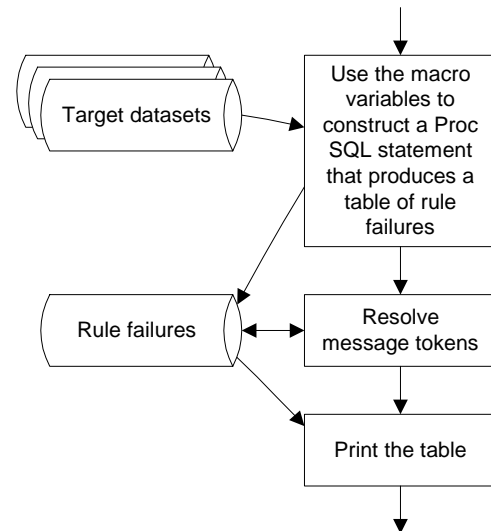
The weight recorded for subject 11MN16 at Visit 3 is 212, which is not within the expected range (100 – 200 lbs).

These new messages have been customized with three data values taken from the target record: the subject number, the visit number, and the subject weight. Customized failure messages such as these can be created by embedding standard tokens in the message specification and adding code to the Data Checker to resolve the tokens at run-time. The following is an example of a tokenized message specification:

The weight recorded for subject [subject] at Visit [visit] is [weight], which is not within the expected range (100 – 200 lbs).

Any number of different conventions can be employed to denote message tokens. In this example, tokens are indicated by square brackets containing a field name. The Data Checker resolves a token by replacing it with the value of the indicated field.

One way to implement resolvable message tokens is to develop a stand-alone macro that can read a dataset, look for a field called *failureMessage* and perform the replacements necessary to expand the message text. The macro is invoked immediately after the *ruleFailures* table is created. A Shown below is a portion of the Data Checker system flowchart with the additional step added.



RULE BUILDERS

Imagine a study in which each subject is examined six times over a period of twelve months. The validation rules for the study database might include a rule like the following:

Physical exam dates for a subject will be in ascending chronological order.

This rule is expressed easily and concisely in English, but in order to represent it using the basic Data Checker specification scheme, it would have to be split into five separate rules:

- The date for exam 1 is less than the date for exam 2.
- The date for exam 2 is less than the date for exam 3.
- The date for exam 3 is less than the date for exam 4.
- The date for exam 4 is less than the date for exam 5.
- The date for exam 5 is less than the date for exam 6.

Entering specifications for five rules is not too demanding, but suppose there were 100 exam dates that needed to be in sequential order, or that the exact number of exams was unknown at the time the rules were created. In a situation like that, it would be handy to be able to create a "rule builder" that could generate the required validation rules at run-time, based on the exam visits that were actually recorded in the database.

This scenario is an example of a situation that arises often enough to justify some additional programming support – rule

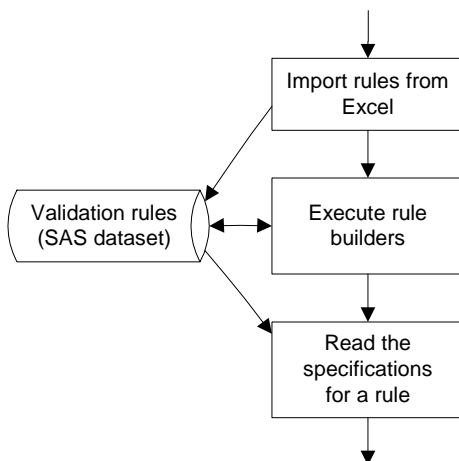
sets that are easier to generate by a program than to enter by hand. The Data Checker developed at Ursa Logic includes a facility that allows rule builders to be defined and used in such situations.

Rule builders are implemented as stand-alone macros. Each rule builder takes a single rule specification as input and expands it into specifications for multiple rules. To use a rule builder, the user creates a rule specification, as usual, but instead of entering an expression in the *GoodCondition* field, the user enters a call to the appropriate rule builder macro. Here is an example of a rule specification that uses a rule builder:

```

RuleID:          Phys-3
Description:     Physical exam dates should be in
                 chronological order.
DatasetA:       PHYSEXAM
FilterA:
KeyVariablesA:  SUBJECT,VISIT
RuleVariablesA: PHEXDT
DatasetB:
FilterB:
KeyVariablesB:
RuleVariablesB:
MergeVariables:
GoodCondition:  %ASCENDING
FailureMessage  Physical exam dates are not in
:              chronological order.
    
```

To incorporate rule builders, the Data Checker adds a pre-processing step after the rules are imported from Excel, but before they are used to generate Proc SQL statements. A portion of the system flow chart is shown below with the additional step inserted:



During the pre-processing step, the Data Checker loops through the rule specifications. For each rule, it loads the rule specifications into macro variables, then checks to see if *&goodCondition* contains a call to a rule builder macro. If so, it invokes the macro.

The rule builder's responsibility is to take the information loaded into the macro variables and use it to generate an entire set of validation rules. The newly generated rules are then added to the list of rules that will be used to construct Proc SQL statements.

The example above makes use of a rule builder called *%ASCENDING*. This macro generates a set of rules to make sure that a target variable (PHEXDT, in the example) is in

ascending order when the records are sorted according to the specified key variables. Because *%ASCENDING* is completely generic, it can be reused over and over again across projects to simplify the process of specifying rules for fields that have ascending data values across records.

CONCLUSION

A system like the Data Checker can radically reduce the time and expense required to implement data validation rules for data cleaning projects. As with any system that is designed for reuse across multiple projects, careful validation is necessary to ensure that the system performs as intended across the full range of possible uses. The payoff for this effort is that a great deal of project-specific programming – and the attendant validation work– is eliminated.

CONTACT INFORMATION

Don Hopkins
 Ursa Logic Corporation
 2625 McDowell Road
 Durham, NC 27705

Work Phone: 919.490.9025
 Fax: 919.490.9088

Email: hopkins@ursalogic.com
 Web: www.ursalogic.com

TRADEMARK NOTICE

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.