PharmaSUG 2017 - Paper TT16

Dear Dave, Please See the .LST file for Our Validation Differences. Thanks, Bad Validation Programmer

Tracy Sherman, David Carr, Ephicacy Consulting Group, Inc.;
Brian Fairfield-Carter, InVentiv Health

ABSTRACT

Good validation programmers are a rare commodity: too often validation programmers view themselves as filling a subordinate role, one that requires little more than basic understanding of SAS syntax, and that does not require imagination or an ability or willingness to question or critically evaluate or investigate. Validation is often reduced to a rubber-stamp corroboration of specifications, and a reverse-engineering of production output; investigating and resolving discrepancies is too often viewed as the exclusive domain of the production programmer.

This paper will give examples of the good, the bad and the ugly among validation practices in statistical programming. It will also strive to ensure that the next time you are validating a dataset or TFL (table, figure or listing), you will understand the importance of and the reasons behind good validation practices and effective communication, and also understand why responsibility for investigating differences between production and validation output lies primarily with the validation programmer.

INTRODUCTION

The production programmer and the validator *should* have a symbiotic relationship, as their goals are so closely aligned: both play an equal role in ensuring that reporting requirements are met, and that reporting accurately portrays the underlying data. In practice this symbiosis is rarely in evidence, as many validators habitually implement practices that do little to further these shared goals. Our intent in this paper is to shine a light on deficient validation practices, using some real life examples, so validators can recognize when they are 'missing the bus', and also understand why appropriate practices make a difference. In a world of time constraints and competing demands, it's important to be aware of where corner-cutting might save modest increments on the validation side, but end up requiring substantial additional effort in the long run.

Validation that is done well is a true embodiment of team-work, and not only goes a long way toward keeping the production programmer sane, but also provides efficiencies for the company. If everyone looked for opportunities to be more efficient and team oriented it would make all our lives easier. And ultimately, streamlining the production of deliverables supports decisions and actions that offer better treatment options for those who need them.

Good validation programmers are extremely hard to come by, not because the role is typically filled by programmers lacking in programming skill, but because the roles and responsibilities of the validation programmer are so often vastly underestimated. We will outline seven simple, useful, and thought-provoking validation guidelines that could be used as guidance for all validation programmers. These guidelines include communication techniques (for example, demonstrating exactly which subject/parameter/visit you are not matching on and your deductions that produced the result you arrived at), as well as investigational methods (for example, reviewing source data (raw/SDTM/ADaM datasets) in light of study documentation, evaluating dataset specifications in light of the statistical analysis plan (SAP), and/or the protocol, and leaning on the study statistician for confirmation of your programming logic).

Simple, independent and minimalist code can be written to show the production programmer how you achieved your results; not only is it unnecessary, bewildering, and totally counter-productive to paste your entire validation code into an email, but it also violates the basic premise of 'independence' in validation. And if your strategy is to merely point the production programmer to a .LST file, then the listing must at a minimum include sufficient ID variables (i.e. in the PROC COMPARE output) to allow the production programmer to isolate specific records.

In practice, validation needs to follow a reductionist approach: illustrative code, used for discussion purposes and to test interpretations, needs to consist of the bare minimum necessary to query specific records and generate specific computed values; and the demonstration of discrepancies needs to focus on *discrepancy patterns*: i.e. specific visits, parameters, patients where discrepancies are concentrated.

After reading this paper, we hope to encourage you to implement some of the proposed guiding principles, to streamline the validation process, broaden the scope of validation responsibilities, to and ensure you approach validation with a team-focused mentality.

SEVEN GUIDING PRINCIPLES FOR VALIDATION AND COMMUNICATION

For the purposes of this paper, we propose that there are seven guiding principles behind validation and the communication of validation findings. These principles describe not only the full scope of validation responsibilities, but also the objectives that should drive validation methodology:

1. **Apply industry standards**: rather than being passive consumers of 'unassailable' specifications, validators should play an active role in developing and refining specifications, and should critically evaluate specs (SDTM, ADaM) against established principles and standards.
2. **Develop and apply knowledge of study documentation, analytical rules and objectives**: again, validators should challenge specs and ask if they realize the objectives as defined in the SAP and protocol. Validators shouldn't just say 'I wrote my validation program exactly per what you put in the spec', but should also ask whether the spec accurately represents the SAP.
3. **Develop a thorough knowledge of the data**: validators should not take on faith that the ground-work done by the production programmer is correct, but should double-check that data handling rules make sense in light of the source data, and handle any exceptions that the raw data may pose.
4. **Be pragmatic, systematic and consistent in communication**: validators should recognize that project leads (who are often the production programmers) have to receive validation comments from multiple programmers, so there isn't time to do a lot of deciphering of sprawling email threads, nor to accommodate multiple styles of communication. Communication of validation findings should be kept to the bare minimum required to 'prove your point'.
5. **Provide specific and supportable explanation for validation findings**: as validator, be prepared to state why you think that an abstract idea should be implemented according to your specific prescription. Support your explanation directly from source data (raw data, or validated SDTM data, or validated ADaM data, depending on context), rather than from some random WORK dataset that may suffer from confounding factors.
6. **Identify discrepancy patterns**: validators should learn to differentiate between discrepancies that illustrate unique differences in data-handling, and those that are merely instances of the same type of discrepancy.
7. **Identify dependencies and cascading/secondary effects**: validators need to identify interdependencies between related derivations, and trace the impact of discrepancies across all relationships.

Table 1 outlines seven guiding principles for programming validation and communication, and provides illustrative "Do's" and "Don’t's".

7 Guiding Principles for Validation Programming	**Do**	**Don't**
Apply industry standards	Review standards and/or have methods in place for checking standards (e.g. Pinnacle 21 for SDTM/ADaM/define)	Validate only what you see, and assume that specs already meet Industry standards; blindly implement specs in validation code without ensuring the specs are accurate.
Develop and apply knowledge of study documentation, analytical rules and objectives	Respond with the page of the section in the SAP/Protocol/Specifications that details the derivation you are not matching on, and provide your interpretation or paraphrase of that text; go through a 'hand-calculation' based on specific data points.	Make the unsupported assertion that 'the derivation is not matching the supporting documentation (SAP, etc.)', and pass responsibility back to the production programmer.
Develop a thorough knowledge of the data	Trace specific data points (i.e. for a target patient) through a derivation, and make sure expectations are met at each step. Proactively refresh datasets or output when pre-requisites have been updated.	Wander laboriously through a complex program hoping to miraculously spot a logic error. Observe that data and/or outputs are out of sync, but do nothing about it.
Be pragmatic, systematic and consistent in communication	Be specific, and provide sufficient background to make your case. Be aware of global time differences and explain discrepancies as they are seen in real time. You save time for team if you keep team logistics in mind.	Assume that the production programmer is intimately aware of the minute details of every derivation, and has nothing else to work on. Send a vague email to a programmer in a different time zone, and lose a 24-hour cycle without resolving any discrepancies.
Provide specific and supportable explanation for validation findings	Send examples of subjects/data points not matching, and explain why you think there is a difference, and what you think the correct outcome should be, as based on supporting documentation	Send an email pointing to the .LST file of the PROC COMPARE output and observe that 'there are differences'.

7 Guiding Principles for Validation Programming	Do	Don't
Identify discrepancy patterns.	Recognize where discrepancies are all of the same type, and provide a small number of examples. For example, if values on the validation side are consistently half of what they are on the production side, then it's likely that someone missed multiplying or dividing by 2 in the derivation algorithm.	Email the production programmer a huge list of identical discrepancies, observe that the numbers are not matching and that you don't know why, and propose that someone else needs to figure it out.
Identify dependencies and cascading/secondary effects	Understand how differences with some variables can have cascading effects on other variables (e.g. AVAL > AVALCAT1, AVALCAT2); start by communicating to the production programmer *only* the discrepancies at the top of the dependency hierarchy.	Inundate the production programmer with discrepancies across the whole dependency hierarchy; or worse, cite the secondary discrepancies but not the 'parent' discrepancies from which they derive.

Table 1. Seven guiding principles for validation programming and communication

APPLY INDUSTRY STANDARDS

Standards are intended to improve efficiency and accuracy by promoting uniformity. Applying standards requires judgment and interpretation, and as such there must be a certain division of labor between the production programmer and validator: interpretation by the production programmer is not necessarily correct or complete, and needs to be challenged and tested by the validator. Further, in a given project, the production programmer does not necessarily enjoy exhaustive knowledge of all reporting requirements across all domains, which means that a 'second set of eyes' is vital: specs need to be reviewed for completeness by the validator.

If you are validating an ADaM dataset such as ADVS, review the dataset specification and compare it to the current version of the document titled "Analysis Data Model (ADaM)" which is available for download at http://www.cdisc.org/adam. Are all the requirements of a dataset following 'Basic Data Structure' met? Are there additional variables or derived records, and if so, do they follow accepted conventions? If there are inconsistencies or things that seem questionable, it is your responsibility as validator to point these out to the author of the specifications. Don't trust the specifications are accurate as standards are constantly changing and people often make mistakes. Take the initiative to use the readily available tools such as Pinnacle 21 Community to validate an ADaM dataset or group of datasets. Performing these checks before TFL programming starts, can eliminate rework if issues are found after the fact. The most current version can be downloaded for free at https://www.pinnacle21.com/downloads.

You should also check that each TFL can all be programmed from the supposedly 'analysis-ready' dataset. It is not adequate to say that ADSL is 'validated' simply because there are no production/validation discrepancies, if there are population flags or subject-level covariates missing.

DEVELOP AND APPLY KNOWLEDGE OF STUDY DOCUMENTATION, ANALYTICAL RULES AND OBJECTIVES

Given turnover in project teams, the tendency to over-simplify in the face of interim or incomplete data (i.e. when generating DMC output early on in a project), and the resistance among some programmers to develop things from scratch (often accompanied by a tendency to take on faith work done by previous

team members), it's not uncommon to see programming specifications that include definitions for baseline that read "set BASE=VSSTRESN where VISITNUM=1". This begs the question: as a validator, when you see this, do you obediently implement exactly this derivation, and consider your job complete when you get a perfect match with production output? Hopefully your answer is an emphatic 'No'. When you run into a questionable derivation, or *any* non-self-evident derivation for that matter, the first thing you should do is look up exactly what the SAP has to say on the matter. Then you should implement the derivation on the validation side according to your interpretation of the SAP, and highlight the deficiency in the spec to the production programmer.

Regardless of what is written in the specifications, the Protocol/SAP/CRF are the 'ultimate authorities', and programming specs should be reviewed for consistency with these documents to ensure the analysis is done correctly. It's everyone's responsibility to point out inconsistencies; arguably the review of specs should be a task unto itself, but validation programmers provide a last line of defense against the potential disconnect between the SAP and analysis implementation. A quick email or call to the lead programmer or statistician, citing the section in the Protocol/SAP that is inconsistent with the specifications, is extremely beneficial to all parties involved. It may take a few extra minutes at the time it was noticed, but it will save the team many hours and a lot of stress if it is addressed well in advance of the end of the study or during a time-crunch deliverable.

DEVELOP A THOROUGH KNOWLEDGE OF THE DATA

Some of the most common discrepancies result from differences in record selection, or more accurately, from over-simplified assumptions about the structure of the underlying data which result in imprecise or ambiguous record selection. Consider for example what happens when you select the 'last' lab record prior to first dose of investigational product as baseline, but you assume that you only need to order records by USUBJID, PARAMCD and AVISITN (in other words, you assume that there could only be 1 record per patient per parameter per visit). When there are actually multiple records per patient per visit, record selection becomes arbitrary and ambiguous.

In any program, and for any problem, you will probably never bring as much focus, or as much attention to detail, as you do while making your initial effort at writing code. This means it is vital, as you're programming any derivation, to include defensive code (Sherman and Ringelberg, 2013), and test for assumptions, right from the outset, rather than assuming you'll go back and carry out these tasks later. At every point in a derivation there are a host of obvious questions that should be explored: is there a potential for missing values?; are there potentially multiple records within 'by' variables?; etc. Answering these questions during your initial coding effort will provide necessary insight when you reach the point of addressing discrepancies.

Knowing the *status* of the source data (SDTM/ADaM) is also vital to addressing validation problems. This includes [a] noting when the source data was last updated and ensuring both production and validation code has been run after any data refresh and [b] ensuring that any dependencies from other ADaM datasets have also been re-run against refreshed data prior to validation of that dataset.

Too often programmers find themselves enquiring why validation is no longer matching when it previously passed. This is where the investigator skills come in handy for the validator. The validator should look to see if there were updates to the specifications and if the production program had changed after the last validation date. If not, then it is probably due to the source data being refreshed. A quick rerun of the production and validation code can eliminate this as a potential reason.

BE PRAGMATIC, SYSTEMATIC AND CONSISTENT IN COMMUNICATION

As more companies are going global, with offices in many different time zones, it's important to recognize the potential cost of ineffective communication. For instance, if you ask a question of someone located in an overseas location such as India, it may take two working days (your work day plus the other programmer's work day) to get an answer.

To mitigate this potential inefficiency, it is vitally important to adhere to stringent communication standards, with the goal of making it impossible to be misunderstood. Adequate context, background and supporting information must be provided so that the recipient of the information is spared from thinking "I *think* what you mean here is…", and all possible effort must be made to investigate the underlying reasons for a discrepancy.

Specifically, where there is the potential for alternate interpretations of project documentation, start by referencing the relevant sections of (i.e.) the SAP, followed by your interpretation or paraphrase. Provide illustration in the form of specific records and data values, and if necessary, provide self-contained, independent code fragments (Fairfield-Carter, 2015) (i.e. that can be dropped verbatim into a SAS session and run 'as is', as opposed to excerpts from your validation program that require WORK datasets as pre-requisites).

Use complete sentences, and try to structure your arguments: 'if X and Y, then Z', rather than 'I used the last record and multiplied it so shouldn't the answer be 0.000342558?'. State the alternate assumptions that occurred to you, and why you rejected them and/or what the outcome was of your assumptions testing (i.e. 'I considered censoring at last dose date, but realized that for lots of patients this would fall much earlier than the true end-of-trial').

There are a few practical considerations that should be kept in mind: if you send a screenshot showing specific/illustrative records, the recipient cannot copy/paste ID values (such as USUBJID) into a SAS session to try and corroborate your results. Further, in PROC COMPARE output you should make it abundantly clear which is the production and which is the validation output (so PROC COMPARE should be run on datasets called something like 'PRODUCTION' and 'VALIDATION' rather than 'DEV' and 'VAL' or 'FINAL1' and 'TEST'). Many validation issues can be resolved quickly over IM; you may find that logging on for a few minutes into the other programmer's time zone can save an entire day's wait.

PROVIDE SPECIFIC AND SUPPORTABLE EXPLANATION FOR VALIDATION FINDINGS

Probably the most important principle behind effective and efficient validation is that of exercising detective skills in supporting observations. No synopsis of validation findings should ever be offered without at least some form of defensible evidence. The level of detail will of course depend on context and the complexity of the derivation, but should provide enough evidence to support why you think your answer is correct.

Look at the values on both the production and validation sides, and try to understand why the production side shows the value that it does. Assemble an illustrative example and provide supporting information, which could be as simple as the following email, Display1:

Hi,

In the ADQOL dataset, we are not matching on the numbers of observations. I have 20,515 and you have 21,902 records.
Looks like you have included the QSSTAT='NOT DONE' records and I am not. When I checked the dataset specs, it says to keep the 'NOT DONE' records as we need to report the number of questions not answered for each questionnaire in Table 14.3.1.5.
When we match on the observations, I will check into any variable differences.

Display 1. Offer Supporting Information for the Differences

The email above offers a potential reason for the discrepancy of the number of observations as well as illustrates that the number of observations should match before you start investigating value-level differences. This is a common mistake that junior programmers make when starting validation. PROC

COMPARE will spit out a ton of variable mismatches if the number of observations are not equal between the production and validation datasets, and record-count differences absolutely must be resolved before any attempt can be made to address value-level differences.

If you are validating a figure and multiple endpoints are not matching, a useful strategy is to copy the PROC PRINT output that supports your inference and add it to your outgoing email. A visual representation of your observation is sometimes worth a thousand words as shown in Display 2.

Hi,

I compared the data until Week 18.

Patients at Risk (Placebo): For W03, I get **35** subjects whereas on production output it is displayed as 34 patients. Below are the numbers from my output until Week 18 for both treatments.

	W0	W03	W06	W09	W12	W15	W18
Event/Cum. Events: Treatment	0/0	0/0	8/8	8/16	5/21	2/23	2/25
Patients at Risk	48	48	37	24	17	14	12
Event/Cum. Events: Placebo	0/0	0/0	4/4	6/10	5/15	2/17	2/19
Patients at Risk Placebo	35	35	30	22	17	15	13

Display 2. Create a Visual for Supporting Validation Observations

Sending an email pointing to the .LST file containing PROC COMPARE output is generally not efficient for either the production or the validation programmer. Somewhere along the line someone must take the plunge and assess what the PROC COMPARE output means, and what it is showing, and since this output is generated by the validation program, it makes the most sense for this to fall to the validation programmer (otherwise the production programmer is reduced to interpreting implementation on the validation side by proxy, by taking a 'back-bearing' from validation output). And since validation essentially amounts to leveling criticism, the validation programmer should provide evidence and justification. (Imagine a book review in the New York Times that just said "this is a terrible book", but provided no support for the opinion).

The exception is if the PROC COMPARE output is self-explanatory, such as when the number of decimals is off by one place. Validation compare code should typically include the LISTALL option along with an ID statement to ensure all records are in sync with production. The LISTALL option can capture valuable pieces of information such as duplicate records which can cause data mismatches on its own.

The ultimate goal should be to save every team member time and energy and to support each other at all times. We are all trying to achieve the same result; the highest quality product, produced in the shortest amount of time.

IDENTIFY DISCREPANCY PATTERNS

Inefficient problem-solving generally involves exhaustively eliminating non-working alternatives, and we often see this enacted in validation programming when piecemeal, isolated, 'hail-Mary' updates are made in the hopes that the number of discrepancies will somehow be reduced. Efficient problem-solving, in contrast, works by identifying the layers of abstraction that define any problem, and looking for patterns that result from some underlying property. In validation programming, this involves looking for patterns that expose collections of discrepancies as belonging to a particular 'type'.

Record-count differences might seem at the outset to follow no particular pattern, and may result from discrepancies across multiple variables, but almost always turn out to be isolated to specific combinations of key variables (for instance, comparing production/validation frequency counts by PARAMCD and

AVISITN often isolates the specific parameters and/or visits where problems exist). Record-count differences mask discrepancies at the level of individual variables, so there's an obvious hierarchy or order that needs to be followed in deciphering and reducing PROC COMPARE output.

Value-level differences are naturally and conveniently grouped by variable, but it's always worth looking for further patterns within individual variables before going back to the program code and experimenting with updates, and this will usually identify and resolve issues more quickly (sometimes instantly). For example, in a summary variable that gives 'n (%)', if your frequency counts match but your percentages don't then you can conclude that the difference must be isolated to the denominator. Similarly, if all values on the validation side are exactly half those on the production side, then one programmer or the other likely just missed multiplying or dividing by 2 in the derivation.

As a further example, consider an instance where you notice that all the 'No' counts in a frequency table are off by a few patients but you are matching on the 'Yes' counts. Further, the first variable that is displayed in the COMPARE output is a flag variable, such as MESIRFFL (Patients with Measurable Disease), which is only populated in the dataset with a 'Y'. Right away, you realize the trend of mismatches is related to how you are both counting subjects that are missing data and are not considered an 'N'. Instead of wasting your time going through each flag variable, you can send the mismatch for the one variable and have confidence that the rest will likely resolve once the first variable matches.

Other trends you might see in a time-to-event analysis dataset are where all the AVALs are off a couple of days. Most likely one of you is using the treatment start date as opposed to the randomization date as the 'start' date. Looking for these kinds of trends is far more productive than staring at program code hoping to spot the exact statements that are causing problems, and can save you many hours of work over the long run.

IDENTIFY DEPENDENCIES AND CASCADING/SECONDARY EFFECTS

When reviewing the validation output for an ADaM BDS (basic data structure) dataset, it's helpful to understand that some variables can have cascading effects on other variables. For example, if AVAL/AVALC is not matching, then dependent variables such as AVALCAT1, AVALCAT2, BASECAT, CHG, PCHG, and SHIFTy, just to mention a few, will most likely not match as well. Getting the number of observations to match should always be the priority, but this should be followed closely by resolving any discrepancies in AVAL. Once these are matching, it should be fairly easy to investigate differences in dependent variables; but investigating differences in dependent variables when AVAL is not matching is probably a waste of time.

Similarly, if you notice in the validation output of a summary statistics table that the N's are not matching for all the parameters, then you can deduce that record-selection criteria differ, and that there's little point in worrying about differences in descriptive statistics until the 'N' counts are resolved. Following a little investigation, you can email the production programmer and let them know what N *should* be for that particular table, based on analysis population and any secondary criteria specified in the SAP or table shells. Contrast that with simply sending an email that says "the N's are not matching" (the latter approach offering no investigation and no insights, and inevitably wasting a lot of time).

Partial-date imputation and the derivation of Adverse Event treatment-emergence offers a further example of secondary discrepancies: if there are differences in imputed onset dates, then there's little point in citing differences in the treatment-emergence flag until these differences are resolved.

CONCLUSION

The way to be effective and efficient when validating in this global industry is to keep the team in mind by thinking of ways to mutually benefit each other. You can do this by providing, thoroughly explaining and supporting validation observations in as much detail as you can muster. To do this, you should know the data inside and out, learn how different variables can have cascading effects on other variables,

recognize validation output trends, study the industry standards, look for inconsistencies between the study documentation and the production output and by using globally minded communication tools.

REFERENCES

Sherman T. and A. Ringelberg. 2013. “Defensive Programming and Error-handling: The Path Less Travelled.” *Proceedings of the 2013 Pharmaceutical Industry SAS Users Group Annual Conference.* Chicago, IL. http://www.lexjansen.com/pharmasug/2013/TF/PharmaSUG-2013-TF24.pdf

Fairfield-Carter, Brian. 2015. “Team-work and Forensic Programming: Essential Foundations of Indestructible Projects.” *Proceedings of the 2015 Pharmaceutical Industry SAS Users Group Annual Conference.* Orlando, FL. http://www.lexjansen.com/pharmasug/2015/TT/PharmaSUG-2015-TT01.pdf

ACKNOWLEDGMENTS

We would like to give a big shout out to Ephicacy Consulting Group, and in particular, Ganesh Gopal, for his support and encouragement in conference attendance, as well as our family, friends and colleagues.

CONTACT INFORMATION

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

Name: Tracy Sherman
Enterprise: Ephicacy Consulting Group, Inc.
E-mail: shermantracy@gmail.com