PhUSE 2013

Paper AD01

A Guide to Deploying Subversion for Version Control of SAS Programs in the Pharmaceutical Industry

Tim Williams, UCB BioSciences Inc, Raleigh, USA

ABSTRACT

This paper discusses configuration and deployment strategies based on over a decade of experience with open-source version control of SAS® programs. UCB Biosciences Inc. first deployed Subversion (SVN) in 2010 and has recently reworked processes to improve efficiency and usability. Lessons learned during three years of use by both internal staff and external contractors provide valuable insight into successes, compromises, and challenges. System validation, access management, integration of validated macro code, and programming environments are considered.

Version control is not a panacea. SVN is a Source Code Management (SCM) tool that does not provide end-to-end traceability for programs, data, and outputs. By playing to strengths and recognizing weaknesses, SVN becomes part of an overall strategy to increase productivity and improve upon audit trail and regulatory requirements.

INTRODUCTION

A quick search of past conference proceedings (www.lexjansen.com) yields several papers that cover SVN as an SCM solution for SAS programs. Instead of restating the benefits of version control, this paper focusses on lessons learned deploying, maintaining, and supporting SVN in a validated pharmaceutical development environment. This paper is an evolution of my 2007 SAS Global Forum paper “Better SAS Programming Through Version Control” where CVS was the SCM. The experience documented here is based on client-side Tortoise SVN 1.6, a shared work area using NetApp storage accessed through Windows 2008 Server, and Linux on the repository side. While my observations are based on these circumstances many of the concepts are extendable to your choice of operating system and configuration. To allow a more concise description, Subversion commands are referenced using command line actions in lieu of Tortoise SVN menu choices.

PREPARATION

You have decided, or it has been decided for you, that version control is a good thing and SVN is the tool of choice. What now?

Consider your IT environment before rushing headlong into the project: determine what operating systems are available for both client and repository; if test, development, and production servers can be provided; and if you have installation privileges on the non-production machines. If SVN is already in the company you may be able to leverage existing support agreements and expertise. In-house experience with scripting languages like PERL or Python is essential for configuring SVN rules and automations.

Spend time with the programming teams. Get to know the folder structure, process flow, and requirements. SAS programming often takes place in a shared folder structure on a file server. Proposals to change existing models will meet with some resistance. Are separate development and production areas needed? Does code move incrementally into production or can it all move in one step? How does the team manage changes to code after a production run? Construct proof of concept demonstrations that show how SVN functions in these real-life scenarios. Early demonstrations are used to identify your user requirements and to develop validation test cases. Include senior programmers in this phase. Managers make deployment happen and long-term success will only be achieved if you obtain support from the programming ranks.

“The architect should strive continually to simplify...”
Frank Lloyd Wright (1869 - 1959), 1908

SVN provides many opportunities to customize. Do you avail yourself of every available customization and pare down after you find out what does not work? Or is it best to start with only the essentials and add features incrementally? Pursuing the route of minimal features and restrictions allows faster deployment, but fosters a Wild West / anything goes culture from the outset. Adding rules later is difficult, if not impossible in some cases, and will negatively impact

user satisfaction. The best approach is to base your level of customization and complexity on core user and regulatory requirements, balanced by the capabilities and limitations of SVN and your IT environment.

"Just because something is tradition doesn't make it right."
- Anthony J. D'Angelo, The College Blue Book

The introduction of SVN into the programming process provides an opportunity to establish new habits and improve existing processes. Challenge existing views on folder structure and work area locations. Why is the work area shared? What prevents separating data folders from the programs? What is the best way to run programs in production? How are the programs and outputs protected from changes after a production run?

It may be helpful to widen your audience with a proof-of-concept demonstration to a cross-section of programmers. A "Lunch and Learn" presentation is a great way to obtain valuable feedback and gauge reactions to your proposal.

VALIDATION

Dr. Teri Stokes nicely states the reasons for validating computerized systems:

> **"Passing audits or inspections is the LAST reason why anyone should want to validate a computerized system. Making sure the system operates to perform the work activities that users actually need it to do is the FIRST reason to validate. Checking that the system is reliable in handling and protecting data for both normal circumstances and for anticipated problem situations in the GXP work process is the SECOND reason. Verifying that the system works as expected with network communications and interaction with other systems and infrastructure is a THIRD reason to validate. Reasons one, two, and three are directly related to return on investment (ROI) for the cost of the system. After ROI there is the FOURTH reason to validate - compliance for audits and inspections."**
> - Dr. Teri Stokes

Validation plays a role throughout the system's lifecycle. Become familiar with industry and company requirements in the planning stages so you can deploy a validated system that is better-prepared for audits and easier to maintain. Does the mention of open source software make your Quality Assurance team break out in a cold sweat? Get an idea of their expectations and requirements. Providing examples of other companies that are successfully using SVN may help allay concerns. At a minimum, discuss the following topics with QA:

- What are the expectations for the validation? Is a full System Development Life Cycle (SDLC) evaluation needed and to what depth?
- Are Standard Operation Procedures (SOP), guidance documents, and templates available along with examples of completed documents from similar systems?
- Does QA require a Vendor Audit or Supplier Questionnaire when obtaining software from a vendor? Slow vendor response times are normal so start this step early. Making payments to vendors contingent upon completion of the QA requirement will help speed things along.
- What validation activities are required post go-live? How is change control managed? Ensure Change Control procedures are in place prior to go-live. When will the first Periodic Review occur?
- Develop a matrix of staff responsibilities for review and approval of validation documents. Have an introductory meeting for these staff (and their alternates) and set expectations for turn-around times.

Help is also available online where you can find industry requirements, best practices, and examples of System Development Lifecycle documents.

Other validation tips:

- Use certified binary installation files from a vendor like Collabnet. These files provide greater confidence in the installation and allow you to focus the validation effort on core requirements and customizations. Confirm that the system functions in your environment as expected, instead of trying to evaluate every version control option. If your teams will only use TSVN, and not command line SVN, focus your efforts on TSVN. Command line functionality can be added and validated at a later time using Change Control procedures.
- Obtain consultant support for installation and configuration, including workshops on best practices based on your requirements. Collabnet, WANdisco, and others provide consulting services in these areas.
- Avoid adding applications that replicate existing features. Even if TortoiseMerge is not your favorite code comparison tool, validating UltraCompare or WinDiff is likely not worth the extra effort during initial deployment.

Programmers are most interested in the audit trail of program development accessed through the SVN commit history. When using Apache server, SVN administrators and support staff can take advantage of additional logging for trouble shooting and debugging. Apache's `error_log` captures any errors encountered when serving SVN requests. An additional `CustomLog` should be configured to capture all access requests to SVN repositories. This highly customizable `access_log` records every request to SVN repositories and may also interest your QA department.

A simple `CustomLog` can be configured in `httpd.conf` using the line: `LogFormat "%h %u %t %r`

Which results in log entries similar to:

`%h` **Remote host**	`%u` **User**	`%t` **Time**	`%r` **First line of request**
10.1.123.456	twilliams	[30/Jan/2013:12:20:32 +0100]	**CHECKOUT** /*<repoPath>*/*<product>*/*<study>*/*<analysis>*/dev/adam/**ad_pe.sas** HTTP/1.1
10.1.321.654	jsmith	[02/Feb/2013:09:14:06 +0100]	**PUT** /*<repoPath>*/*<product>*/*<study>*/*<analysis>*/dev/validate/adam/**v_ad_pe.sas** HTTP/1.1

The full list of customizable log content is available in the Apache online documentation (see References).

REPOSITORY CONFIGURATION AND SECURITY

Decisions on IT architecture, validation considerations, and user requirements will drive configuration choices for the SVN repository. Programmers employed directly by your company may work concurrently on a number of investigatory products. External contract staff often focus on a single product. In this scenario each product is setup in a separate repository for flexible security. Mirroring each repository to a fail-over server provides rapid recovery in the event of catastrophic server failure. Development of validated macro code takes place in an isolated repository, restricted to only those programmers developing and releasing this protected code. This repository, along with one for managing administration of SVN (hook scripts, utilities, and training materials), is under an entirely different path (`SVNParentPath`) due to its unique functional and security requirements.

There are two facets to SVN security: *authentication* and *authorization*.

Authentication answers the question, "Who are you?" Integration of SVN authentication with Active Directory using **Apache server**, **mod_dav_svn**, and **LDAP** is a leading choice to consider. The added complexity compared to **svnserve + LDAP** is counter balanced by the large benefit of detailed logging for each request made to the Apache server. With LDAP our staff rely on the same username and password they use for login to our network and other applications, without the need to maintain a separate SVN identity. AD authentication frees the SVN administrator to focus on other tasks by making the IT department responsible for password length, composition, and expiration requirements. Setup of AD authentication is well worth the extra effort.

Authorization answers the question, "Are you authorized to access this resource?" SVN `access` files contain user names and paths which grant or deny access to repositories. Establishing a common `access` file for similar repositories helps minimize maintenance. When using AD for authentication the usernames in the `access` file must match with those entered by the user at a SVN prompt. If your company does not enforce the use of standardized usernames (for example, one system requires 'twilliams', another "TWilliams"), you will spend a lot of time chasing down access issues when users change their company password and forget to use the proper case for their SVN username. Consistent use of all upper or lowercase letters for SVN is recommended if usage varies within the company.

Use of AD and consistent usernames has another benefit. If training is recorded in a database, those records can be queried with a script to add the list of trained staff to a versioned copy of the `access` file. The file is deployed on the repository server by running a `svn export` command that specifies the new revision number of the `access` file. PERL or other scripting languages provide a convenient way to glue these tasks together.

Paths listed in `access` files further refine which teams can access specific trials and analyses and can restrict access during breaking of treatment codes. I recommend configuring a group for internal staff that has read-write access to all active clinical trials repositories. Access for external staff can be granted to individual studies or even to specific analysis folders within a study. The lower you go in the structure, the greater the maintenance effort. Granting an external company access at a higher level (at the study level or even the product level) results in less maintenance but provides more access than necessary. Base your decision on the number of studies, your level of trust in the contractors, the maintenance effort, and your risk tolerance.

Hook Scripts

Hook scripts triggered by key events are installed in each SVN repository. A hook either permits, terminates, or suspends an action when conditions are met. One of the most useful hooks is the `pre-commit` hook which executes before a transaction is committed to the repository. We use `pre-commit` hooks to:

- Require a Commit Message for every commit
- Parse each .SAS program to ensure the SVN comment header is present
- Allow only certain file types in the repository
- Enforce naming conventions for tags
- Prevent `svn delete` and `svn commit` from production areas
- Allow over-ride of restrictions by administrator accounts.

A `pre-commit` hook can also detect case-sensitive name clashes when the client is on a Windows OS and the repository is Linux. Linux happily stores foo.sas and Foo.sas as two distinct files in the same folder. A `svn checkout` or `svn update` to a Windows work area throws an error when attempting to write the second file to the folder. The hook can prevent a file name clash by detecting the potential future error before it is committed to the repository.

The maintenance effort for maintaining the hook scripts for each repository is reduced by using SVN to version the scripts and deploy them using the `svn export` command. Scripted exports ensure consistency across all repositories.

DEVELOPMENT, VALIDATION, PRODUCTION?

There is value in separating code development and validation from a production folder structure. Moving programs from development to production allows you to "start clean" by leaving behind old log files, data, and test programs.

The move to production must be efficient. An easy way to move to production is to tag the development structure and use that label to checkout all folders and content to production. The disadvantage of this approach is that it is more difficult to support incremental moves to production where only some programs are production-ready and others are not. One solution is to promote code from development to production within the SVN repository (using the TSVN Repository Browser), then `svn update` the production folder. Another approach is a program that identifies revision numbers for validated files and exports these select files to the production folder using `svn export`. A similar `svn export` process can be used to create a browseable code library for validated macros.

The inevitable corrections to a production checkout are made in development, validated, and merged into production. It is a simple matter to configure a `pre-commit` hook to prevent any commits from locations other than development. Preventing *modification* of a production checkout is more difficult. Restricting access to production with security settings can only go so far; at a minimum, the person performing the checkout must have write access to the production folders, and the same must be true for the programmer who runs the programs, if logs and outputs are written to the same area. Possible approaches include: restricting the number of programmers with access, scripting the production checkout, and relying on process rules dictating no changes in the production environment. `TSVN Check for Modifications` or `svn status` are useful commands for detecting modifications in production.

Programmers often find the process of merging changes into production complex and confusing, leading to delays. Merging is improved in SVN 1.8, where SVN chooses the type of merge to perform and many of the code conflicts reported in SVN 1.6 and 1.7 are avoided.

Outputs generated in production must be clearly differentiated from earlier development and validation runs without modifying the programs themselves. Any modification of the code nullifies a program's validated status. If production is separated from development by checking the programs out to another folder structure, this switch between environments is easily identified programmatically based on the path to the programs. For example, if the path to the program contains a folder named "production", a macro variable is set to "FINAL" and the value is resolved in output titles or footnotes. Paths to programs can be captured for code executed in batch (`sysin`) or interactive mode (`sas_execfilepath`) and assigned to a SAS macro variable. The path is evaluated for a folder name that indicates production `%index(&myPath,\production\) ne 0)` and, if the condition is met, a macro variable is set to "FINAL" for use in all outputs.

The use of production and non-production right-click menu options is one way to flip the designation from draft to final for situations where the code remains under the same path for both development and final execution (no checkout to production). The "Production submit" menu choice passes a macro variable to the SAS session, while other choices for development and validation execution leave the item undefined. It is advisable to separate the right-click options to prevent accidental Production execution. The code behind each option would look similar to:

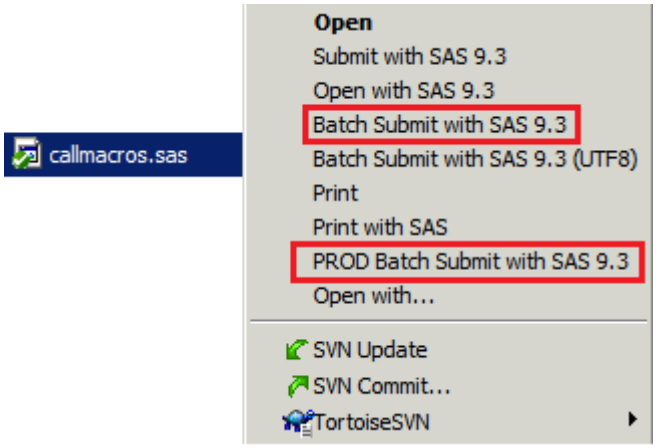
Batch Submit with SAS 9.3:

```
"<path to SAS .EXE>\sas.exe" -sysin
"%1" -nologo -config "<path to SAS
config>\SASV9.CFG"
```

PROD Batch Submit with SAS 9.3:

```
"<path to SAS .EXE>\ sas.exe" -set
drftfinl "FINAL" -sysin "%1" -nologo
-config "<path to SAS
config>\SASV9.CFG"
```

Windows Right-Click Menu Display



WORK AREAS

In traditional version control each programmer has their own private work area where they create, commit, and execute programs. In contrast, many SAS environments use a shared folder structure where all programmers work simultaneously, partitioning their work more-or-less successfully among the team. Programs, logs, data, and outputs may all be within the same folder structure. Moving from this shared work area to one more suited to version control often presents challenges for the organization. TSVN can be used on shared network drives but do not expect support for this approach from the user community, because you are deviating from version control doctrine.

Shared SVN work areas have several drawbacks. One of the most obvious is that it provides the opportunity for programmers to commit work that is not their own. The `svn:needs-lock` property is not a solution in this scenario. `needs-lock` is designed for traditional private checkouts and its function in shared work areas is counter-intuitive. Obtaining the lock on a file prevents anyone other than the lock owner from committing a change to the repository but makes the file writeable for anyone with access to the work area. If using this approach, it is best to commit your changes and release a lock as soon as you can; thus returning the file's read-only property. Be aware that the read-only restriction is easily circumvented by manually changing the file's Read-only attribute. SVN is a more open system by design and does not follow an exclusive locking model by default. Consider other systems if exclusive locking is critical to your methodology.

Private work areas prevent programmers from committing content that is not their own. This traditional model is not without its own problems: it becomes more difficult to detect if a programmer has uncommitted modifications and management of multiple work areas becomes much more complex for support staff.

Another problem with work areas on Windows Servers is that TSVN icon overlays often do not display correctly. There are several causes, including Windows OS limitations and TSVN configuration options. Consult online SVN support forums for details on this topic and never take icon overlays at face value. In our latest 64-bit Windows 2008 Server installation we discovered that the very useful SVN columns in the Explorer view are no longer present. This is a Windows OS limitation and there is no work-around. Limitations of the Explorer window and problems with icon overlays mean commands like `TortoiseSVN Check for Modifications`, `svn status`, and `svn commit` are relied upon for file status information.

WHAT TO VERSION

An important aspect in planning your deployment is deciding what types of files to version. Like most SCM's, SVN excels at managing ASCii text files like SAS programs. In SVN's early days one of its advantages over its primary open-source rival, CVS, was its ability to version binary files (CVS has since caught up in this area). Does that mean that SVN *should* manage binary files? MS Word documents, Excel Spreadsheets, and SAS datasets may become very large, so committing them to a repository is resource intensive and inefficient. The general tenet of versioning is to only manage the components that contribute to regenerating outputs from sources. Following this logic means source SAS datasets should be versioned. However, due to their size and number, it often becomes impractical for all but the smallest of projects.

Versioning SAS datasets is also problematic from a process perspective. Every time a program alters a dataset, SVN sees the file as modified and in need of committing to the repository. If datasets are versioned in development and then tagged, they become part of the checkout to production which exponentially increases checkout times. This checkout includes data that should be regenerated in production from source data. Even if datasets are only

committed after a production run, large datasets take a very long time to commit, due to SVN's lengthy binary differencing process. Commits can easily take hours for a few gigabytes of binary files.

It is feasible to version other file types that directly support SAS program runs like .BAT and .TXT files or Excel files that contain titles, footnotes, and program status. TSVN includes scripts to invoke differencing of Microsoft Office and Open Office files in their native applications. The scripts have limited value, particularly for Excel, where inserting a column will incorrectly identify all columns to the right of the insertion as new content. A caution about Excel: if SVN keywords appear in Excel workbooks that have an active keyword resolution property, the file will become corrupt and irretrievable.

Versioning SAS log files is debatable. Like datasets, a log file versioned in development is modified whenever its parent SAS program is run, even if the program was not modified. Committing logs during development adds much overhead and little value. Consider adding log files only after a successful production run. This allows you to compare logs between different production runs over time, but also complicates merging changes into production.

A `pre-commit` hook script is an easy way to prevent the committing of files based on paths (development or production) and file type (named extensions). `pre-commit` can prevent the commit of .SAS7BDAT from a folder that has a "development" or "dev" folder name in the path or only allow it from a path that contains "production" or "prod".

WORKING IN THE FOLDER STRUCTURE

A standard folder structure template can itself be versioned in a SVN repository and used to create new structures for analyses. When the folders include template files the programmers receive a head start on a new project. By versioning the folder structures and template files you can track the evolution of changes over time. A setup script adds the structure to the proper study repository along with template SAS programs and links to validated macros; making the setup fast, accurate, and consistent. Programmers may add subfolders after the structure is checked out to a work area.

Version control is a paradigm shift for some programmers. It removes the need for backup copies of files and folders that have traditionally given programmers peace of mind. Programmers often continue to make unnecessary copies because the history is not directly visible in the work area. Ensure programmers become familiar with the `TSVN Show Log` command to view all previously committed revisions and with commands to compare and retrieve earlier versions.

.svn folder

Every folder managed by SVN 1.6 and earlier has a hidden subfolder named .svn that contains administrative files used by the system. Turning off the display of hidden files and folders helps to protect content within .svn from accidental or conscious modification but removes the visual reminder that this content exists.

Programmers must be made aware of the grave consequences of moving folders in the work area. Files within .svn contain pointers to where content is stored in the SVN repository. If a folder and its contents are copied from one location to another using Windows Explorer (outside of SVN commands) the folder contents in the work area will commit to an incorrect location in the repository. Consider what happens if you are working on two different studies and want to re-use code by copying folder contents from source Study 1 to destination Study 2: commits from Study 2 are mistakenly made to the Study 1 repository. This problem is difficult to identify until Study 2 moves into production and code from the copied folder is discovered as missing. Two studies now have serious problems: commits are missing from the intended study (Study 2) and are incorrectly present in the Study 1 repository. Take every opportunity to reinforce the message: **never copy entire folders from one location to another without SVN commands**.

Corruption of .svn folder content is often observed in a shared work area. A likely cause is multiple programmers executing SVN commands simultaneously in the same folder. Competition for locks in .svn content, often for the `entries` file, causes a working copy lock that may be fixed by the SVN Cleanup command or may require replacement of the entire .svn folder. Working within .svn folders is a task best left to the SVN Administrator or support staff. The ultimate prevention is to use individual, not shared, work areas. If this is not practical in your situation, encourage programmers to execute SVN commands on individual files instead of recursively processing multiple folders.

SAS Programs

Use of SVN keywords in program comment headers is covered in other papers, so I will not go into details about the format or content here, other than to say that all programs should contain these keywords. Providing a header template as a hot-key combination or keyboard macro is a good practice because the keywords are case-sensitive.

Keyword resolution is added automatically to .SAS files as part of the `svn add` process if the `auto-props` option is set for the .SAS file extension in the client's subversion configuration file `conf` :

```
enable-auto-props = yes
[auto-props]
*.sas = svn:keywords=Author Date Revision HeadURL Id
```

If practical, make this setting the default for all SVN clients by using a shared configuration file for all users or by running a script that sets the configuration for each new user. Failure to automatically set keyword resolution causes confusion when a versioned file is copied or renamed outside of SVN commands. When the resulting file is added to SVN the old keyword values remain and do not update.

Committing and "the Blame game"

Successful versioning is as much about people as it is the system or process. Programmers who have experience with version control know that regular committing, especially during very active development phases, is a safety net against unanticipated problems. Failure to add files and commit modifications is one of the most commonly observed problems. Version control introduces a change in programming culture where commits to the repository must become part of the programming *flow*. Acceptance of SVN often takes a precipitous event like the use of `svn update` to return a file deleted by accident.

Some programmers are reluctant to commit modifications after learning that every committed line is viewable in the audit trail along with their user id and a timestamp. Perhaps they think that only their best code should become part of the SVN record and do not want early drafts or experimentation to be visible in the history. By not committing early and often the most useful features of version control will not be available. The safety net provided by version control actually frees programmers to try out new techniques and procedures. If a change does not work, revert the code back to a previous revision and try something else. In my experience no one has time to go through code revision-by-revision. The focus of code review should be on versions used for production runs and not the steps taken to get there.

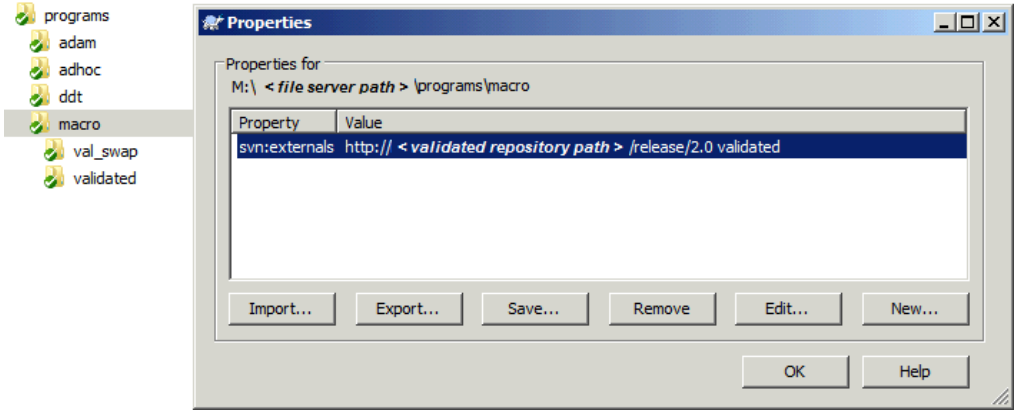
The aversion to committing changes is worsened by SVN's developers' choice of "Blame" as the name for the function that displays who is responsible for the changes to each line in a file. The potentially humorous name has a definite chilling effect on programmers. A better term is "responsibility". Blame is useful when investigating compliance with double-programming requirements, where code is recreated independently by a second programmer. TortoiseSVN's TortoiseMerge application is a great way to detect identical sections of code. Blaming one revision against another allows you to see who is responsible for each line.

Validated SAS Macros

Validated macros stored in a protected repository can appear within a project's work area folders. This is achieved by using the SVN property `svn:externals`. The property is added to a folder and specifies a link to the validated release in the protected repository, as well as the name of the subfolder where the files will appear. In this example the property is set on the \macro folder and creates a subfolder named \validated that contains validated files from "Release 2.0" provided by our Programming Standards group:

```
svn:externals http:// <validated repository path>/release/2.0 validated
```

This is how the property appears in the TSVN Properties window:



A major advantage over remote libraries for validated programs is that the programming team can easily identify the versions of macros used in the study and review the code. Setting the `svn:needs-lock` property during creation of the release makes the files "read-only" in the work area and reminders programmers not to alter these special files. Any programmer can circumvent the read-only property, but security settings prevent them from committing any modifications to the restricted repository. Setting `svn:externals` is part of the scripted setup of new folders and always ensures teams have the latest release of validated macros.

A programming team changes to a different release of validated macros by changing the release number specified in the property. Granting this power to the programmer is good because automatic update to a new release may break existing code, require re-validation of programs, or cause unexpected results. The decision to move to a new release is made by team consensus and implemented by the lead programmer.

Over-ride of individual macros in the \validated folder is made possible by configuring the macro search order as:

1. \val_swap
2. \validated
3. \macro
4. SASAUTOS

(See TSVN Properties window, above)

In this configuration, a macro in \val_swap overrides the current release of the macro of the same name in the \validated folder. This configuration provides programming teams with the flexibility to mix-and-match macros from multiple releases. Since the macros are validated together as a set, overrides are discouraged.

INTEGRATION WITH OTHER TOOLS

The **TSVN executable** TortoiseProc.exe and the optional **SVN command line client for Windows** facilitate execution of SVN commands from within toolbar buttons of applications like UltraEdit. They can also be used to call SVN from SAS by employing the `x` command or `call system()`. When creating buttons for TSVN commands in other applications it is a good idea to use the same icons that appear in the TSVN drop-down menus in Explorer. The images are available here:

http://tortoisesvn.tigris.org/svn/tortoisesvn/trunk/src/Resources/
Login: guest
Password: *leave blank*

The SAS Enhanced Editor does not expose the path and program name to the programming interface, making the addition of custom SVN toolbar buttons a challenge. You can only access the program path and file name after the code is executed. The situation is further complicated when multiple editor windows are open. I look forward to someone solving this problem in a future PhUSE paper.

Some SVN commands have unintended consequences when used outside of the regular SVN interfaces. For example, executing a SVN Revert command from an UltraEdit toolbar button on an unversioned file deletes the file from the work area.

Among the most useful command line commands are those that determine work area status (`svn status`, `svn info`). These commands can be inserted into logic that terminates a production run if any files have a SVN status of "modified" because all programs must be committed prior to a final run. Resist the temptation to automate `svn add` and `svn commit` commands: such automations remove the ability of programmers to enter Commit Messages that provide critical context and status information.

TRAINING AND SUPPORT

Hands-on training sessions are best for first-time users. Work through the basics in a training environment that mimics the real world; right down to authentication, program execution, changing to different releases of validated macros, and moves to production.

Minimum training and support includes:

- Instructor-led course
- User Manual, customized for your methodology and environment
- Frequently Asked Questions, updated regularly.

If your company has multiple office locations, I highly recommend designating a senior programmer at each office to act as local front-line support. If you hire external contractors, identify a contact within the company who will act as local support and will join in regular meetings to provide user feedback and updates. Do not rely on a no news is good news approach. Sit with programmers regularly to discuss what is and is not working for them. Revise methods as needed with a major review of the system no longer than one year post go-live.

Additional support considerations:

- Commercial Support
  Vendors like Collabnet and WANdisco provide multiple levels of support for SVN. Consider the level of technical expertise in your organization and make your decision accordingly. A higher level of support in the first year is beneficial, backing off to a lower level in subsequent years based on experience and number of problems.
- Online Support
  SVN has very active and supportive user communities. However, if you step outside the normal deployment situations, be prepared for less than helpful feedback, which may include: "Do not do that", "That is not what SVN is for", or the inevitable "It is open source – modify the source code to make SVN do what you want." If you are extending SVN into less common usage scenarios, like work areas on network shares, you may benefit from commercial support.
- Training
  Version control basics are not enough. Team leads must advocate for the system and review the team's work on a regular basis. Emphasize that the number and frequency of commits has no correlation with productivity or code quality.

CONCLUSION

SVN is a good SCM choice for SAS programs and it continues to evolve and improve. SVN 1.7 introduced a single .svn folder at the checkout root instead of within each checked out folder, greatly simplifying work area administration. The `svn move` command, system performance, and client-to-server communications are all improved in recent releases. Additionally, an enhanced `merge` command avoids many of the problems experienced in past versions.

A successful SVN deployment is dependent upon IT environment, regulatory requirements, process flow, and work culture. Some of these factors are likely beyond your control. Be prepared to make tough, and sometimes unpopular, decisions after fully understanding your teams' programming process. Extra time spent in the early stages is invaluable later. Deploying version control can be a positive disruption that facilitates process improvement if carefully planned in advance.

Coordinating development between many remote programmers is a primary use of version control, but is only part of the story. The ease of repository setup on a desktop or laptop also provides the benefits of code comparison, roll-backs, and its other advantages for small groups or even individuals working alone.

REFERENCES

The Apache Software Foundation. *Apache HTTP Server documentation, Version 2.4*. *Apache Module mod_log_config*. http://httpd.apache.org/docs/2.4/mod/mod_log_config.html. Access date: 26-Aug-2013

Mason M. 2006. *Pragmatic Version Control Using Subversion*. 2nd Ed. Pragmatic Bookshelf.

Collins-Sussman B, Fitzpatrick B, Pilato C. *Version Control with Subversion. For Subversion 1.7*. http://svnbook.red-bean.com/en/1.7/ Creative Commons Attribution License. Access date: 26-Aug-2013

Küng S, Onken L, Large S. *TortoiseSVN. A Subversion Client for Windows*.
http://tortoisesvn.net/docs/release/TortoiseSVN_en/ Access date: 26-Aug-2013

Stokes T. *Why Validate a Computerized System? Successful Computer Validation - A Monthly Blog.*
http://blog.gxpinternational.com/2013/01/why-validate-computerized-system.html Access date: 26-Aug-2013

Williams T. *Better SAS Programming Through Version Control.* Proceedings of
the SAS Global Forum 2007 Conference.

Commercial Support
Collabnet http://www.collab.net - Certified SVN installation binary, tiered support packages
Wandisco http://www.wandisco.com - SVN and TSVN downloads and support packages.

ACKNOWLEDGMENTS

I wish to thank the SAS programmers and staff at UCB Biosciences Inc. for their continuing support and comments.

CONTACT INFORMATION

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

Tim Williams
UCB BioSciences, Inc
P.O. Box 110167
Raleigh, NC 27613
Work Phone: +1-919-767-5997
Email: tim.williams@ucb.com / NovasTaylor@gmail.com