PhUSE 2008

Paper AD08

An Approach to Using Version Control with SAS®

Paul Crean, ICON Clinical Research, Dublin, Ireland
John Woods, ICON Clinical Research, Dublin, Ireland

ABSTRACT

In a controlled, regulatory environment, it is essential to have an automated version control system for the tracking of program changes. The process of manually tracking changes is error-prone, and relies heavily on the individual programmer to be as detailed as possible in their change description. A version control system lets you compare programs to previous versions, and provides an exclusive file locking functionality when editing, making it a safer environment for projects of all sizes.

This paper will describe a manageable, cost-effective approach to version control, with some pros and cons of early stage implementation in a global CRO.

INTRODUCTION

Version control (aka source control) is a set of tools designed to safeguard and manage changes to individual files over time. The tools track what change was made, when it was made, who the change was made by, and why they made it. The tools enable a programmer to compare and retrieve earlier versions of a file, and to understand the evolution of a piece of code.

This paper will discuss in more detail the following:

- Reasons for implementing version control.
- An overview of version control theory and current approaches.
- How ICON has implemented version control.
- Benefits ICON has achieved post-implementation.

REASONS FOR IMPLEMENTING VERSION CONTROL

As this is a controlled, regulatory environment, there are a number of reasons for implementing an automated version control system in the Pharmaceutical research and development industry.

21 CFR PART 11 COMPLIANCE

A version control system helps to ensure the integrity and authenticity of programming and validation processes, and that these processes are in compliance with this FDA regulation, as it will automatically track each user who made changes to a file from a given time point (e.g. from when the file was first submitted to the version control system at the beginning of a project).

AUDITING

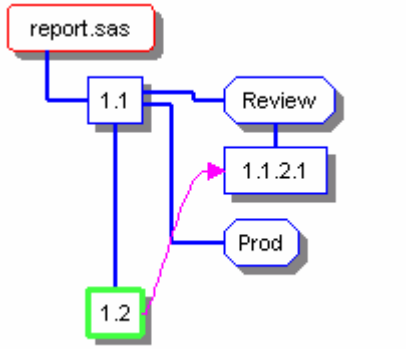
QA can be confident that all updates to program outputs after a client release can be tracked by the system itself. Simple reports can be generated from a SQL database which holds the metadata of all files in the repository. E.g. an auditor can run a query on the history of changes made to programs (what they were and who made them) since a previous client release, and then compare these to validation documentation completed for the same period.

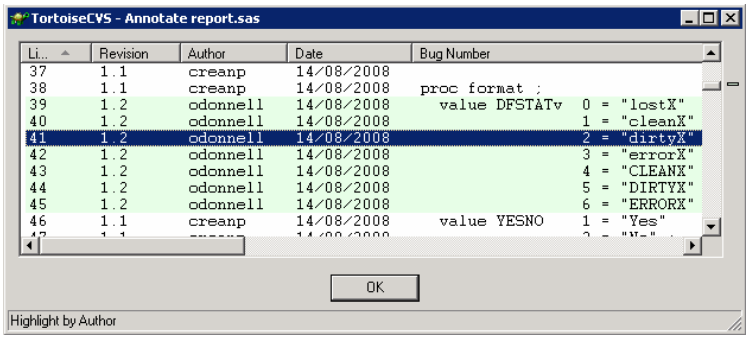
STUDY MANAGEMENT

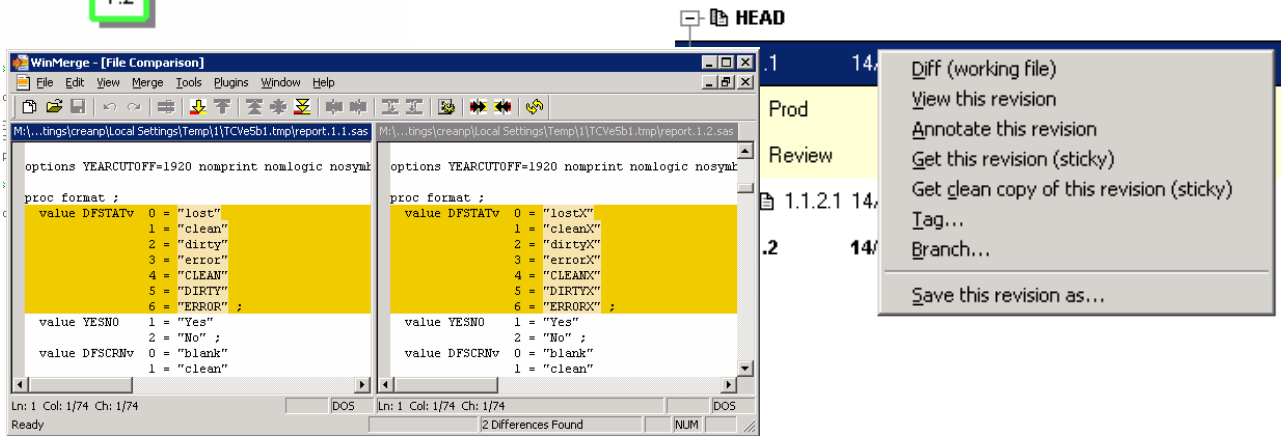
Once statistical programs have been developed and validated, it is important that some control over them can be maintained throughout a study. With project programmers in different geographical locations (Europe, India and USA, as well as home users), it makes sense to have an automated process where program development is controlled. A version control system enables the reproduction of output from any stage of a project, so nothing is ever destroyed.

CHANGE HISTORY

Understanding the development of a piece of code is simplified by the use of history, annotation/blame, and diff tools. Any revision of a file can be compared to the current working copy of that file or any other committed revision.







For more complex tasks, most version control tools support associating a bug/issue ID with multiple changes to a file or group of file. Integrating version control with an issue tracking tool allows us to track and audit changes at the programming task level in addition to the individual file level.

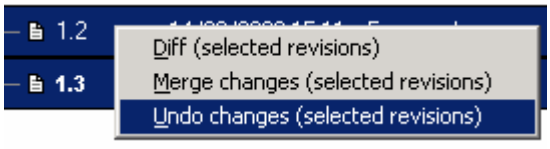
COHERENT SNAPSHOTS

Using the tag feature, you can create snapshots at milestones in the lifecycle of a development project. In this way you can roll back the entire project to earlier milestones, and the rolled back code will be a coherent snapshot of the project at that time.

EASY BACKUP AND RESTORE

Once a file is committed to the repository, restoring to an early revision is straightforward and does not need intervention from IT.

Version control tools manage space better than traditional backup tools, as only the change deltas between multiple revisions of the same file are stored.



OVERVIEW OF VERSION CONTROL CONCEPTS

At its most basic level, a version control system consists of a repository (or database) and a work area (or sandbox).

THE REPOSITORY

The repository is the area in which managed files and the information required to reconstruct any revision of a managed file is stored. The user never works directly on the files in the repository, but uses a client to check out a copy of the files to the work area where changes can be made. When a file is checked out of the repository, the latest revision of the file is distributed to the work area by default.

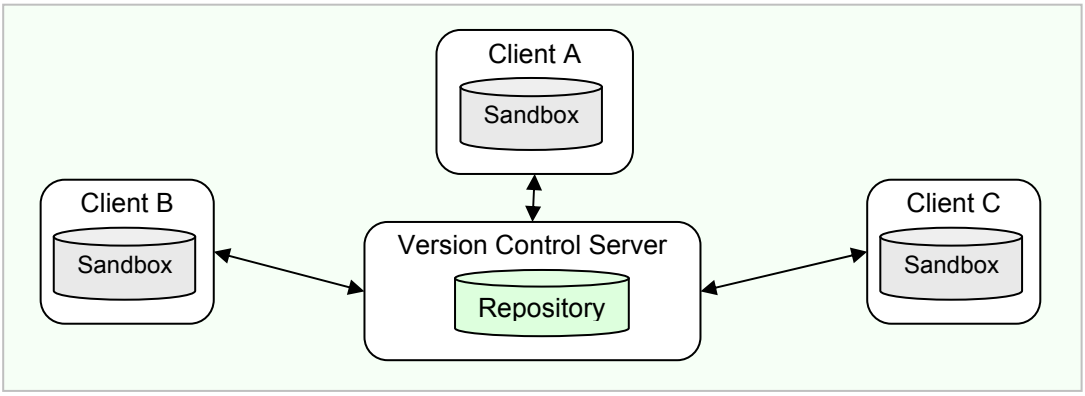
Once the changes are complete, the updates to the file are committed to the repository. After the updated file is committed, the repository will contain the updated file and a delta of the changes made since the last commit. A file

that has had multiple changes committed will have multiple deltas stored in the repository.

In addition to the latest revision of files and their associated deltas, the repository stores metadata for each set of changes committed to the repository. The metadata includes the date and time of the commit to the repository, the username of the user who performed the commit and the reason for the change that was entered by the user.

THE WORK AREA (SANDBOX)

The work area is where programmers write new code and change existing code. The copy of the code in the repository is considered the master copy, and the copy in the work area is the working copy.



When you commit your changes to the repository you can enter a log message. These log messages can be viewed when you look at the history log of that file. Multiple files/directories can be committed at the same time; the log message that you enter will apply to all of the files committed.

The master copy remains unaffected by any changes made in the work area until the work area changes are committed to the repository.

Work areas are normally personal and restricted to a single user, but a shared network work area can be used if managed carefully.

CHECKING OUT AND COMMITTING

Depending on the version control model implemented, a programmer must check out or reserve a file before they can edit it. This will create a working copy of the file in the work area.

The programmer makes their changes to the file in the work area and on completing the changes he/she commits those changes back to the repository. If a programmer is unhappy with their changes they can abandon the changes or revert to an earlier version.

BRANCHING AND MERGING

Branching and merging are important concepts in version control and enable parallel development in a software project.

At certain points in the development process it may be necessary to create a copy of the project to allow for testing, or a bug fix, to create a new release, or to test new functionality. A version control system allows you to create and maintain parallel lines of development, or branches. Changes made to files in one branch are completely isolated from the versions of those files in another branch.

When the changes are complete they can be incorporated into another branch by merging.

TAGGING

When you create a tag in version control it attaches a symbolic name to the current revision of the file or group of files being tagged. By tagging a group of files or a project you can keep a track of which versions of the files were associated with each other at a particular point in time.

THE CURRENT ALTERNATIVES

At an early stage in the evaluation process, the decision was taken to restrict the search to open source tools. Their obvious advantage is the saving in licensing costs, but more importantly open source tools offered ICON greater flexibility in customisation and integration.

TRENDS IN OPEN SOURCE VERSION CONTROL TOOLS

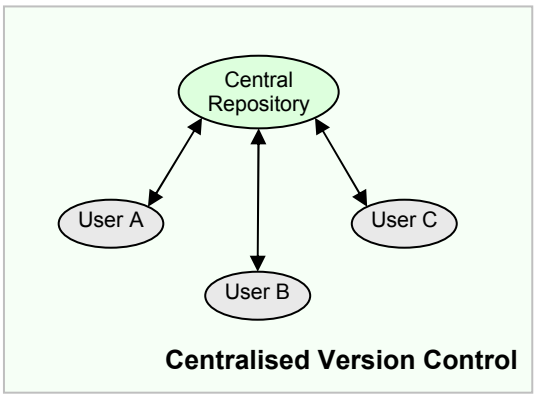
Initially the de-facto standard open source tool for projects was CVS. CVS was developed in the 80's and was based on an older system called RCS. In the late 90's open source developers were frustrated with the limitations of CVS, and a number of competing tools were developed that were designed to be compatible with CVS while either "fixing" it (Subversion, aka SVN) or "improving" it (CVSNT).

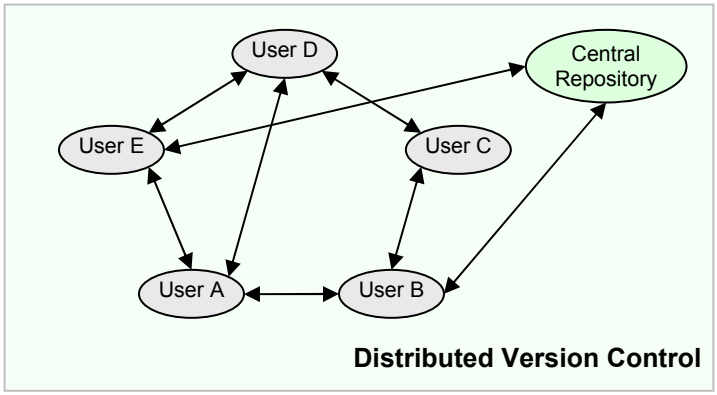
These tools, like CVS, were centralised client-server implementations but added functionality such as atomic commits, rename support, and improved performance. Development of CVS trailed off dramatically after the release of Subversion and the final version of CVS (v1.12.12.1) was released in 2005.

Subversion and CVSNT are the two most widely used open source centralised version control tools.

In recent years a new approach has become popular – distributed version control. The traditional approach of centralised version control requires that a user checks files out of a central repository to a local work area to make changes to a file. The user then connects to the central server to commit their changes to the repository. All history and change deltas are stored in the central repository, and the user must connect to this server to use version control functionality.

In the decentralised approach, there are multiple repositories. Rather than checking out a file to work on, an individual developer or a development team checks out (or clones) an entire repository (including all branches and history) and uses this cloned repository as their work area. Changes can be committed to the local repository and when complete the changes from the local repository can be merged into the mainline repository. This peer-to-peer approach removes the requirement to connect to a central server to perform version control tasks.





The most popular open source distributed version control tools are Mercurial, Bazaar and Git.

CONTROL MODELS

In a multi-user, multi-location programming environment the challenges of sharing and controlling access to project files must be considered. These challenges include how we share code without stepping on each other's toes and how we avoid the frustration of another programmer overwriting our code.

Discussion in version control literature often centres on the "lock-modify-unlock" and "copy-modify-merge" models.

The "lock-modify-unlock" model requires that the programmer request an exclusive lock on the file that's being changed before they can modify it. When they've made their changes, they commit them to the repository and release the lock on the file. No other users can make changes to the file while an exclusive lock is held. This model is a variant of the "Reserved" method; some tools also offer a number of non-exclusive reservation methods.

The "copy-modify-merge" model works without a locking or reservation mechanism. Multiple programmers check out a copy of the project files to their own personal work area, and they work on the files in parallel. Changes to files are

merged back into the central repository. If multiple programmers try to commit changes to the same file the version control tool will help the users manage any merge conflicts.

Most "best practice" documents recommend that a shared area not be used, and that programmers work on their own personal local work area. In the real world this isn't always the best approach, and the implementation team must decide whether it's best to use a "Centralised" or "Decentralised" work area.

Deciding on the correct combination of the "reserved vs. unreserved" and "centralised vs. decentralised" models is critical to integrating version control into an organisation's development process.

BRANCHING MODELS

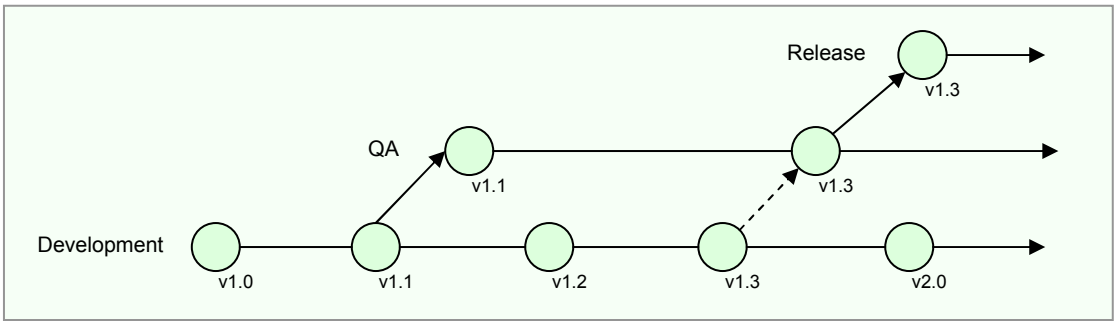
Branching functionality allows for flexible parallel development. Variations of branching policy run from the never-branch approach through a structured promotional-branch approach (Development – QA – Release) to branching by release, component or task. Branching, because of its flexibility and because it is so tightly linked to a team's development preferences, is considered a controversial topic in source control management.

The more common branching approaches in commercial software development are:

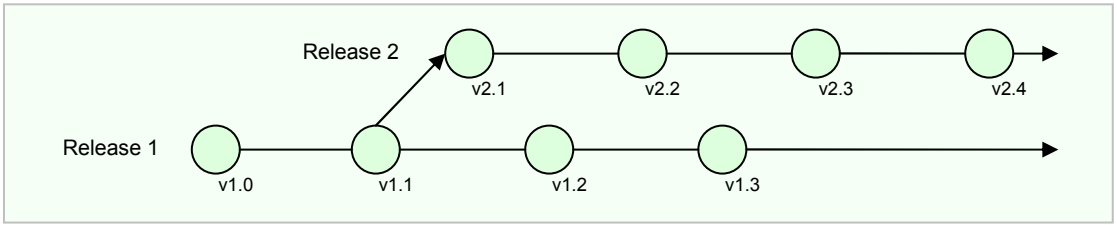
- **Never branch:** All revisions to files are committed to the HEAD revision. This is often used in small projects as it is the simplest to implement. As the number of developers increase, the risk of unstable code increases because there is no isolation.

- **Code-promotion branches:** Three branches are maintained - Development, QA (or test), Release – and all development occurs on the Development branch. When a piece of code is ready for QA, it is merged (promoted) into the QA branch. If the QA review is successful, the code is merged into the Release branch and ultimately deployed. No development is allowed on either the QA or Release branches.

  The QA and Release branches are tightly controlled to maintain the integrity of the promotion model.



- **Branch by release/version:** When a project is ready for release, a branch is created for the new release and development for the current release is continued on the current branch. Any bug fixes to the old version may be merged into the new release branch.



A couple of things to consider when you are designing your branching strategy are isolation/control and complexity.

Creating more branches creates greater isolation between developers; this can be very useful where there are problems with instability in the mainline code. By instituting a gatekeeper who reviews code before merging code into the HEAD revision from private branches, stability of the HEAD revision is always guaranteed.

Branching is a balancing act because creating more branches also creates complexity and overhead. Unless your

branching strategy is clearly defined, developers will get confused and “spaghetti branching” may result. Additional branches increase the amount of merge and conflict resolution tasks that a developer needs to carry out in order to integrate their code changes back into the mainline, leading to “merge mania” or “merge paranoia”. Depending on the tool used and branch structure, merging can be a complex task needing a lot of manual intervention; excessive merging will add considerable delays to your project.

For a detailed discussion of branching theory and the origin of terms such as “spaghetti branching”, see:

http://www.cmcrossroads.com/bradapp/acme/branching/

HOW ICON HAS IMPLEMENTED VERSION CONTROL

The implementation of version control is part of a larger project to overhaul the development workflow at ICON.

TOOL EVALUATION

Three version control tools were evaluated at ICON – Subversion, CVSNT, and Mercurial. These tools were chosen for evaluation because they had a lightweight shell tool that integrated with Windows Explorer (TortoiseSVN, TortoiseCVS and TortoiseHG) which avoided the need to use either a command line or a full client.

Mercurial is an impressive product and offers a lot of flexibility in terms of implementation; it can be implemented as a centralised, decentralised or hybrid version control system (a mainline repository with multiple local decentralised repositories). Unfortunately TortoiseHG was a very new product at the time of evaluation, so it was decided not to take the risk of implementing it.

Subversion was designed to be a replacement to CVS; it replicates most of the functionality of CVS and fixes some of the flaws in its design. At ICON, we spent a lot of time evaluating Subversion but ultimately rejected it because it was missing important functionality such as file locking. Additionally, there are problems with the implementation of branches and tags in Subversion. In Subversion branching and tagging are implemented as a directory copy, which is fast, but merging between branches is awkward and tags are not immutable so a coherent snapshot cannot be guaranteed.

CVSNT was developed as an alternative to CVS. It was initially designed to add support for case insensitive filenames in Windows, but over time functionality such as atomic commits, authentication with active directory, an audit database and branch and directory level access control were added.

Out of the three products tested, CVSNT and TortoiseCVS most closely matched our requirements and were chosen for implementation.

We made contact with March Hare Software, the sponsor of the CVSNT project and engaged their consulting services to implement and support the software. On implementation we switched from the open source versions of CVSNT and TortoiseCVS to the commercial CVS Professional package of tools which contains enhanced versions of CVSNT and TortoiseCVS.

PROCESS REVIEW

Installing a version control tool is technically straightforward. There are plenty of guides available on the web that will walk you through the install process. Essentially you download and run the server package, create a repository and set access control rights on it, then download a client and connect to the server to check files out to your work area. This simple set up is sufficient for a single developer, but is not effective for a larger team.

Many programmers who have experienced a version controlled environment express frustration with the tools and describe them as an impediment to the development process. These programmers will often work outside the managed workspaces and find ways to avoid using version control. This is symptomatic of a poorly implemented version control system. This happens for many reasons including installing the software tools without considering the business process that surrounds them, forcing the tools to fit a work process that is not compatible, or developing an overly complicated process that offers no obvious benefit.

Early in the evaluation process it was realised that, from a technical perspective, choosing the version control tool

would be relatively simple. The open source tools that are available have enough features to meet the needs of most teams and are stable enough to be used in a commercial development environment. The difficulty was finding a tool that would integrate well with the development process at ICON.

A review of the development process was undertaken as part of the version control project. As part of the review, the current development process was evaluated to see if there were any improvements that could be made to the process. Improvements to the process were found in the following areas:

- **Environmental Changes:** The development environment was split into three areas (Development, Quality Review and Release) to provide isolation and greater control. Structures were already in place that provided isolation between these stages; this standard implementation was introduced to create company-wide consistency.

  Code is exported from the release area to a tightly controlled production area for actual processing.

- **Standardised Project Structure:** A standard directory structure for new projects was implemented.

- **Macro Library and Code Reuse:** The concept of code reuse is that sections of code written and validated once can be used in multiple projects, consequently reducing the development and validation time for new projects. A software library of validated macros is being developed and stored in the version control system.

CODE MANAGEMENT

A primary goal of the implementation was to create as controlled an environment as possible while minimising obstructions to the development process. The ability to collect detailed, accurate auditing information was another primary driver.

A shared central work area was implemented for each project. This allows all users involved in a project to see the changes to a piece of code as they are made; as opposed to the more common approach of programmers keeping changes private until they are committed to the central repository. The “shared” approach matched our development process; using the “private” approach would have complicated the process and may have caused programmer frustration.

Implementing a shared work area complicated achieving the goals of audit and control. From an audit perspective, when a file is checked out to a central work area and a number of team members make changes to the file, the team member who commits the file to the repository is recorded as the person who made the sum of the changes. There is also the possibility that programmers may overlap (and overwrite) each other’s work in the working copy of a file, and because a working copy is effectively outside a version control system we have no way to control this.

To solve these problems, we implemented a system of exclusive locking.

When a programmer wants to make a change to a file, they must reserve a lock to edit the file by executing the “cvs edit” command. This marks the file as “reserved” in the repository, and modifies the file system ACL to prevent other users from changing the work area file. All users can see what changes the programmer is making to the work area copy. When the programmer is finished with the file, they release the lock by either committing their changes to the repository or by abandoning them.

Controlled files cannot be edited without acquiring an exclusive lock.

The exclusive locks guarantee that only one programmer can alter and commit a reserved file, effectively solving the audit and control weaknesses of a shared work area. This implementation is a variant of the “Reserved/Centralised” model (see Version Control Models above).

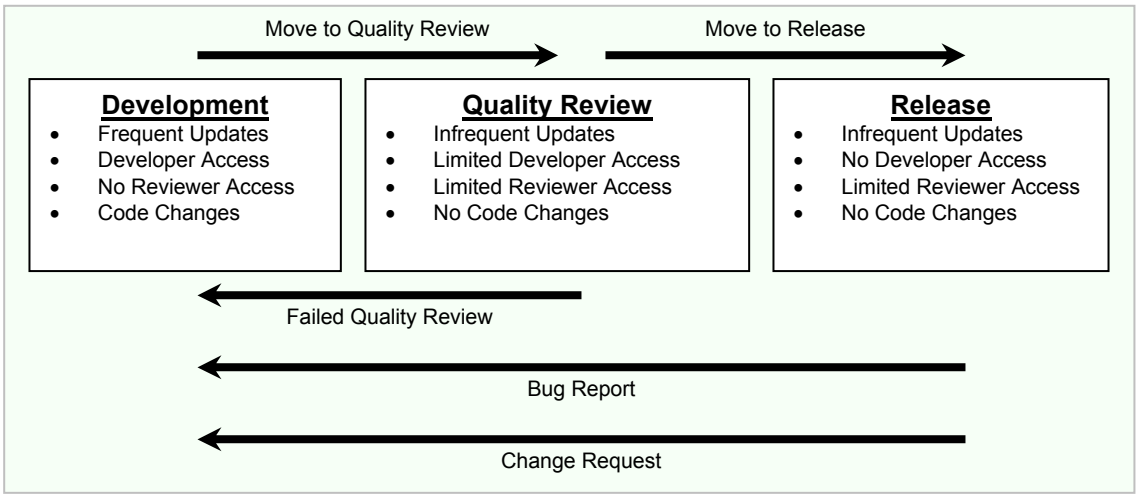
CODE PROMOTION

The process of code promotion is implemented by using branches in version control (see branching models above).

There are three branches in use – Development (HEAD), Quality Review and Release.

All code is committed to the Development branch, and when it is ready for promotion to Quality Review it is merged into the higher-level branch. Upon successful review, the code is merged to the Release branch. Code in the Release branch is regularly exported to the Production area. Changes to code (bug fix, etc.) always occur on the

Development branch and must follow the promotion process before they are released to production.



The following access control is in place:

- Developers have full access rights to the Development branch, and have rights to merge their changes from Development to Quality Review. They cannot make changes to code in the Quality Review branch and have no rights to Release branch.

- Reviewers have read rights to the Quality Review branch and can merge from the Quality Review branch to the Release branch. They have no rights to the Development branch, and cannot make changes to code in either Quality Review or Release.

- The Production area is maintained separately, and neither developers nor reviewers can make changes to code here.

TECHNICAL IMPLEMENTATION

Version control at ICON has been implemented as follows:

CVSNT Server: The CVSNT server runs on Windows 2003 on a virtual machine. The repository is stored on the virtual disk associated with the server. No direct access is allowed to the server.

TortoiseCVS: The TortoiseCVS client is made available as a published application through Citrix. This is available to all programmers.

CVS Workspace Manager: The CVS Workspace Manager is available to CVSNT administrators. The tool is used to create new projects in the repository and the shared work area.

Security: CVSNT offers a number of ways to authenticate users. The SSPI protocol is used at ICON to authenticate CVSNT users through Active Directory. CVSNT administrators use the CVS Workspace Manager to assign repository rights to Active Directory users and groups; rights are set at the repository, project, branch, tag and individual folder level.

Audit: CVSNT writes audit data to a relational database (including MSSQL, Oracle, MySQL) detailing CVSNT repository activity. This data includes detailed session information, commit metadata, and tag metadata. In order to ensure that all repository interactions are recorded CVSNT will not allow a repository action to take place if it cannot write to the audit database.

Disaster Recovery: A Windows port of Unison (a file synchroniser) is part of the CVSNT Windows service (Unix installations can run Unison separately). We create hourly snapshots of the repository files using Unison, and the latest snapshot and the audit database are backed up nightly. Replication to a standby server may also be triggered on each commit to the repository; we are evaluating this option and may implement it when repository activity increases.

CHALLENGES IN IMPLEMENTATION

Once the design of the new development process was complete and the version control policies were in place, the technical implementation was straightforward.

The process review and the decisions on policy took quite a while as everyone involved in the final design had to develop a strong understanding of version control theory. Without this knowledge, it would have been impossible to design a solution that would have integrated with the tools. The development process was redesigned a number of times as our knowledge of version control theory improved.

The process that we settled on is now being tested in a small number of pilot projects. We expect some changes to the process based on the experience of the programmers.

USER PROCESS

PROJECT SETUP

At the beginning of a project, the version control administrator creates the project's shared work areas (development, quality review and release) on a network drive. The development work area contains the standard project structure and is pre-loaded with the standard validated ICON macros and code. Depending on the project's sponsor, this directory may also include sponsor-specific standard validated macros and code.

The CVSNT administrator creates the promotion branches (Dev, QR, Rel) and sets the ACLs (Access Control Lists) for the project.

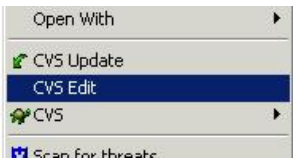
As the programmer creates new project files, they mark them for addition to the repository and commit them when ready.

DEVELOPMENT

The process for users is only a few extra steps when developing or updating project programs, but the little bit of extra time it takes is hugely compensated for by the information that CVSNT has readily available on file history and status within a project. Users can do all their version control tasks through the TortoiseCVS shell extension.

Files in the work area that are under version control are checked-out with a status of read-only ("unmodified read/only"). Users can open these files to review them, but they are prevented from saving changes to them.

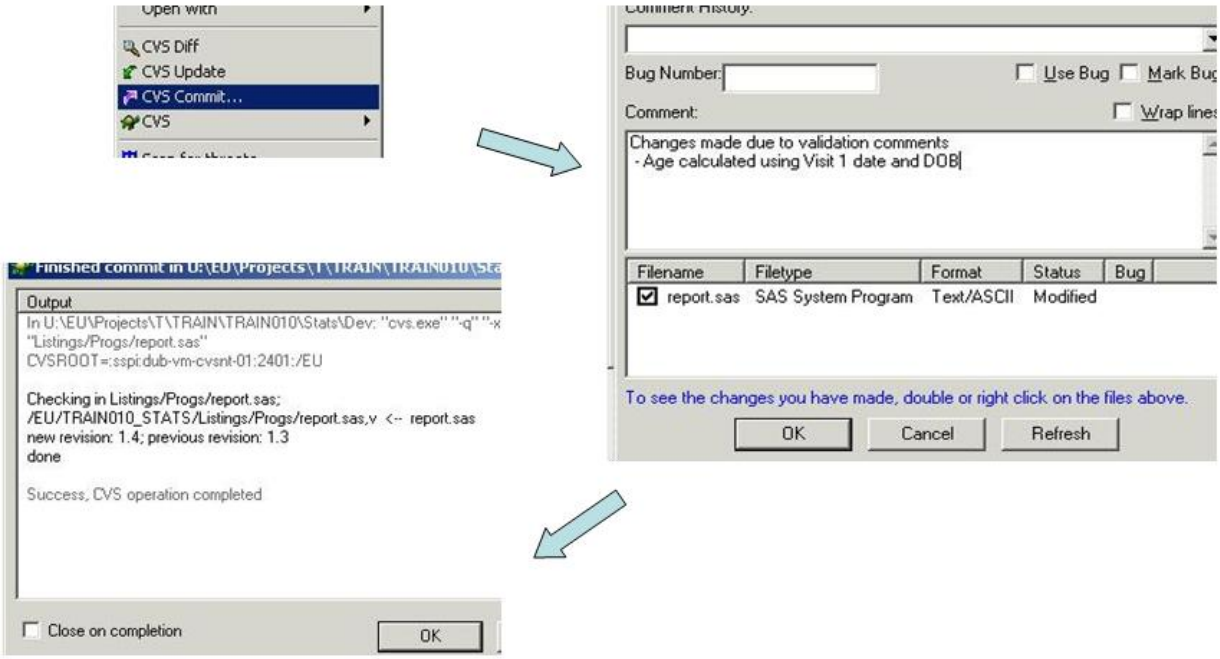
To make changes to a file, the TortoiseCVS shell is used to execute the "cvs edit" command. This requests an exclusive edit. The edit command sets the file to read-write in the work area for the current user (using a filesystem ACL). In the repository the server process marks the file as being edited by the current user (the "editor"). The file's status in the work area becomes "edited". Other programmers may review the file, but they cannot save changes.



When the programmer has completed their changes, they commit them through TortoiseCVS. This changes the file status back to read-only in the work area, and signals that you are no longer editing the file by removing the "edited" flag in the repository.

If the programmer decides to abandon their changes, they can revert the file back to its repository version by issuing the "cvs unedit" command through TortoiseCVS. This releases the exclusive lock.

On executing the "CVS Commit" command from the right-click menu the user is asked by the system to enter a reason for the change. This comment (along with other metadata including the user-name, date and time of change) will be visible in the history log of this file. CVSNT will then commit the updated file to the repository as the latest version, and inform the user of this with an OK box.

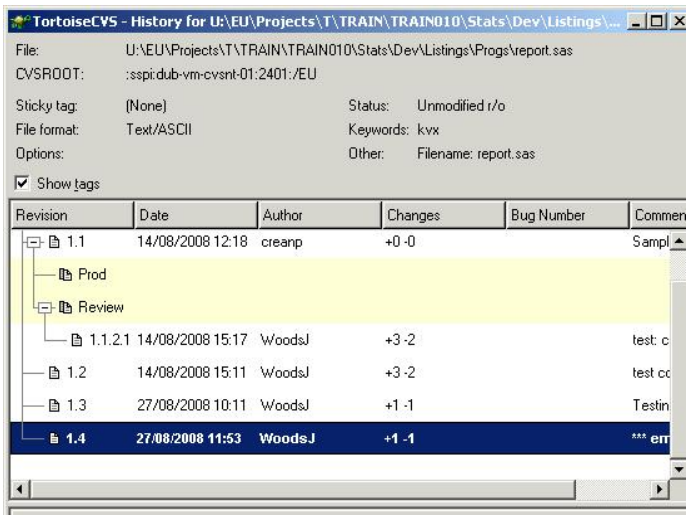


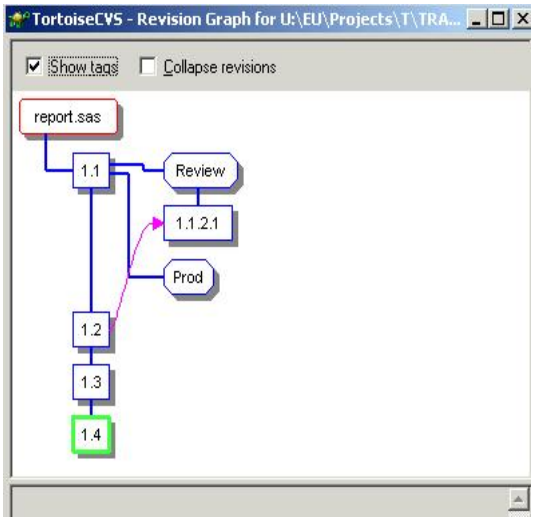
PROMOTION

When files are ready for validation, the developer can promote them to the validation branch using the CVS Merge command. On different branches, again there is the option of restriction, for example, validators only having read and execute rights on the files in the validation branch. Once validated, files can be promoted to the production branch, and labelled as part of an overall package.

HISTORY TOOLS

To see the evolution of a program, users can choose to view either the History log or a graphical Revision Graph. Both are easily accessible from the Tortoise CVS menu by right clicking the file in CVS Studio.



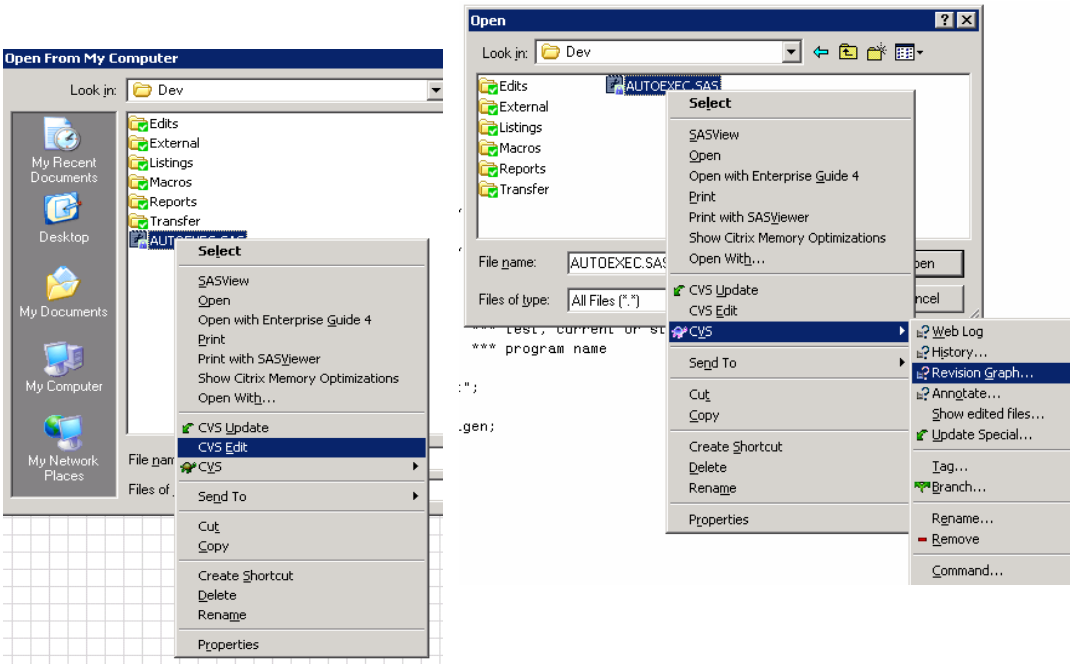


While the History window shows the current status of the file (location and version number), it also gives names, dates and reasons for changes made (user comments). The Revision graph shows a simple graphical history of the file from it's origin.

Diff and Annotate tools are available to compare the revisions of a file. Using these tools in combination with the history tools, programmers can trace the progression of a piece of code over time, and rollback to an earlier version if necessary.

INTEGRATION WITH SAS

TortoiseCVS is available through any SAS file dialog that uses the standard Windows file API.



A lot of SAS dialogs do not use the Windows Common Dialog API, so no interaction with the version control tools is possible through these dialogs.

To simplify training, we encourage users to perform all version control activities through Windows Explorer and TortoiseCVS and then to switch back to the SAS programming environment to perform development tasks.

We are considering extending the Enterprise Guide application to allow the user to execute a subset of the CVSNT commands from within the application.

FUTURE PLANS

On completion of this rollout it is planned to extend the use of version control tools globally, by implementing a version control server in the US.

While the implementation of version control will assist in programming tasks, we plan to expand the basic functionality of the tools by implementing:

- **Management tools:** There are a number of reporting and monitoring tools on the market that read statistics from the version control system. The reports produced help managers allocate work to programmers based on current activity. Project managers can use the report to see how a project is progressing and to identify bottlenecks in the process.

  At ICON, we plan to use the audit database to create an in-house monitoring solution.

- **Collaboration and code review tools:** Traditionally if a programmer had problems with getting a piece of code to work they would walk over to the resident expert's desk and ask them to look at code. Now because we work in multiple offices, in multiple time zones, that process is a mix of missed phone calls, voice mails and emails.

  We are looking into a lightweight peer review tool that integrates with the version control system. This tool can be used informally for collaboration and on a formal basis to assist the code validation using a moderator to authorise or reject programming code in a project.

- **Bug and issue tracking tools:** The CVS Professional tools integrate easily with Bugzilla and other bug and issue tracking systems. We will evaluate a number of tools to see how this functionality could be used at ICON.

- **Release Manager:** The CVS Professional suite includes Release Manager, an application for deploying projects from the repository.

- **Validation Documentation:** We are currently testing a validation documentation tool which generates the necessary documentation for a program, and we are looking into implementing this with version control so that all the documentation associated with any release version of a program is readily available (see paper "Automating Production of Validation Documentation" by Tangi Sanseau).

It is hoped that the tight integration of these tools with the version control system will simplify (and speed up) programming tasks, simplify communication on large-scale projects, and ease the task of managing projects.

CONCLUSION

We have recently installed and tested this software, and are now looking at a pilot project, before rolling it out across the company. This project is still very much in the early stages, but the benefits are already clear to see.

CONTACT INFORMATION

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

Paul Crean
ICON Clinical Research
South County Business Park
Leopardstown, Co. Dublin, Ireland
Work Phone: +353 (0) 1 291-2000
Fax: +353 (0) 1 291-2200
Email: Paul.Crean@iconplc.com
Web: www.iconclinical.com

John Woods
ICON Clinical Research (as above)
Work Phone: +353 (0) 1 291-2418
Email: John.Woods@iconplc.com