Configuration Management Tips

Whether you’re just getting started with version control techniques or an old pro, this article has some good suggestions for you to improve your overall plan.

Many organizations have adopted the use of source configuration management (SCM) tools. While having the right tool for the job is the mantra of professional mechanics everywhere, knowing how to use the tool is equally important. Hammers, screwdrivers, and wrenches are largely self-explanatory; such is not the case with SCM tools. This article will review how SCM delivers higher quality software and aids developers working together; show how it should be used to gain maximum advantage; and how it fits into the bigger CM picture.

SCM and the development effort

When used properly, SCM tools are great aids in debugging software and delivering quality products that inspire confidence. SCM provides a mechanism whereby every product can be characterized as a collection of known inputs. The ramifications of that mechanism are two-fold. First, it provides the concept of a control set when debugging. Second, it provides the ability to know, with 100% confidence, which products are affected by a newly discovered bug.

In science, the fundamental necessity for performing any test or experiment is a control group. Through the use of a control group, scientists are able to isolate a single variable from the set of all variables that could cause an observable change, and then measure the change that said variable actually causes.

As an embedded engineer, I have had countless experiences where a new bug is discovered when trying out some feature that is in the process of minor tweaking. Often, the question asked is, “did something on the hardware break or has my software introduced a new bug?” Pre-development (or prototype) hardware is error-prone and very sensitive. When things go wrong or new bugs are discovered, you can’t be sure if the hardware is somehow to blame or the software is at fault. This is particularly
Another useful debugging technique CM provides is the ability to try out and test prior executables for the presence of a newly discovered bug.

What should be under version control?
- Source files
- Makefiles
- Scripts
- Associated documentation

The first question any SCM user has to answer is: “what should be version controlled?” Strictly speaking, anything that is an input to the process of creating your product should be version controlled. However, I would add the following caveat: tools from outside the organization that are not expected to change over the course of the project should not be version controlled. For example, your compiler, linker, and debugger are static over the course of the average project. Therefore, I would not put them under SCM. I do suggest that they be archived in some fashion, but they will not need the amount of rigor that an SCM tool provides.

Another manner in which SCM assists developers working together is that it allows each team member to view the other members’ work. When one developer is about to make changes to a file, he can immediately see if another developer is working on the same file. Once the other developer has finished making changes, and added them back into the SCM database, the first developer can easily compare the new version to the old version and see the exact nature of the changes.

Maximizing effect
Like any tool, SCM should be used on the problems it was designed to fix and in the manner it was designed to operate. To truly gain the benefits mentioned in the preceding paragraphs, SCM must be used in a planned, organized method. What follows is a compilation of common sense practices to using SCM that are often obvious only in retrospect.
recreated from one day to the next, given identical inputs and tools. In more direct terms, if one has to check out a file to be able to reproduce a build, something is being incorrectly version controlled.

Some items that may not be obvious, but nevertheless should be placed under SCM are the scripts used to invoke the tools. Likewise any command-line options passed to the tools should be placed in directive files passed to the tool, which can then be placed in the SCM database. This leads to the first rule of maintaining a build process:

Source + Tools = Product

The build process should be cast in stone with no room for deviation. Any and all command-line options should be in some form of file passed in with the command-line. The steps of calling each tool and script should be in batch files as much as possible. Ideally, one command typed at the command-line will execute every step necessary for creating the executable. Anything that cannot be automated in some kind of script should be fully documented in the CM plan (discussed later). The test of a good build process is: “can an engineer not on the development team recreate the product your team is developing?”

As a matter of completeness, any project-related documentation should also be placed under version control. Just as reviewing the evolution of source code over the course of a project is useful, reviewing the evolution of such things as requirements, designs, policies, and practices can also be useful. By placing these documents in the SCM database, the ability to review where the project has been from a managerial perspective is greatly facilitated.

The first rule of using an SCM tool is to do all development work within the provided workspace.

Working in the SCM workspace
- Always work within the workspace
- Only use one workspace per unit of work
- Keep the workspaces synchronized with the codeline

The first rule of using an SCM tool is to do all development work within the provided workspace. Many engineers new to SCM prefer to stick to their old habits of ad hoc SCM and develop their files in their own subdirectories. Once their work is complete, they check out the files they changed, copy their personal files to the workspace, and check everything back in. This approach is flawed in many ways. Part of the power of SCM is the history of a file it records as the file incrementally changes through the course of development. I usually suggest that anything that compiles is ready to be checked in. Anyone inclined to develop outside the SCM workspace is also going to be very reluctant to check in anything until it is 100% complete. The next flaw with such an approach is the risk to the project should the guilty developer be unexpectedly unavailable at a critical time. If the developer has his work spread across multiple subdirectories using various home-brew back-up naming schemes, there is little chance another developer will be able to step in and figure out what is the “correct” version with which to proceed.

Next, each developer should use a unique workspace for each unit of work: problem report/change request (PRCR), bug fix, or feature. If a developer is working on two separate PRCRs he should be using two separate workspaces. If two developers are working on the same unit of work, they should again have their own workspaces. Any exchange of files should only be done through the SCM system via check-in. Sharing a workspace either between developers or among PRCRs can introduce confusion as to the state of files. The SCM tracks development by check-ins from the workspace. When multiple development efforts are taking place in a workspace, the ability to make distinctions as to which check-in was for which PRCR is compromised. In effect, the development efforts become coupled.

Lastly, developers should always keep their workspace synchronized with the codeline. I can attest that constantly re-synchronizing with an ever-changing mainline is difficult. It is like trying to hit a moving target, but to not do so is an invitation for error. It is quite a waste to spend half a day debugging some problem that has already been fixed and made part of the latest codeline. Worse still, is to write code that works within the constraints of some other module that has been redesigned or even removed. Furthermore, to forego merging and synchronizing until all the development work is complete doesn’t remove the necessity of synchronization; it only puts it off until the end, when the workspace and the codeline are most divergent.

Growing the SCM database
- Always have a mainline codeline
- Branch when appropriate
- Design the codeline before the project starts

Two approaches to codeline development are the mainline model and the promotion model. The mainline approach has a single branch that evolves forever. It is the ultimate destination for most (not necessarily all) changes made in all the other branches. By virtue of that fact, the mainline approach creates a common point from which all new development will sprout. All developers know the mainline is the most up-to-date, and the best place to begin new work.
Conversely, the promotion model abandons every branch once it is no longer needed. This creates a moving point for beginning new development—something that may be difficult to communicate. It also creates extra difficulty for developers to stay synchronized with the codeline. If the branch they were working from becomes obsolete, they must relocate all their work to some other branch.

I also suggest that the mainline always have a fully working and tested build as its latest release. This gives all developers a known-good baseline to start whatever development effort they need. This is something that is impossible with the promotion model as the branch in which to start new development is always changing.

Branching is, perhaps, the most controversial topic in SCM because everyone seems to do it differently. SCM tools support branching in a myriad of different ways, and organizations use branching in still a myriad more. My advice for branching is to branch only when absolutely necessary. Maintaining branches is a lot of work. Propagating changes from a branch to the mainline and back again is a never-ending chore. It is most prudent to keep all this effort to a minimum by just keeping branching to a minimum.

Branching should be used only when incompatible policy exists for user’s needs. For example, the product release group wants a check-in policy that is very controlled and rigorous. The marketing research group wants a check-in policy that allows frequent, easy check-ins of nearly anything that compiles. While these groups may be working with the same source modules, their needs are radically different. Another example is that of separate development teams using a common code base and creating common products, but for different markets. They may be working on the same product, but a Web-based Vehicle Registration System for the Commonwealth of Kentucky is going to have some radically different code than that for the State of Indiana.

Don’t copy when you should be branching. I have seen some engineers think they can avoid the effort and hassle of branching by just copying. Copying has all the maintenance requirements of branching without the automatic merging support pro-
vided by the SCM tool. If a bug is found and fixed for the Indiana project, and it applies equally to Kentucky and 28 other states, the effort of propagating the bug fix is very high when it must be done manually as compared to an automatic merge facility provided by most SCM tools.

One practice that requires a lot of branching, but is well worth the effort is that of the Branched Development Model (BDM). When using BDM, a branch is created for every development effort. All developers have complete freedom for check-in and check-out in their development branches. Once their work is complete, they notify the build boss (discussed shortly) that their work is ready for merge. The build boss then merges all the development branches en masse back to the mainline for release. Often this merge effort takes place in still another branch before it merges into the mainline. This practice is very beneficial when multiple developers are making many small changes to the same sets of files. BDM also helps in supporting the practice of always having a mainline with a known-good release as the latest version.

**Highbrow SCM**
- Designate a “build boss”
- Make use of change packages
- Releases and release notes

The **build boss** is the person responsible for all the operational SCM decisions. He “owns” the mainline and is responsible for any check-ins that occur to it. He ensures that the build and release process is working and all products can be reproduced. Likewise he is responsible for creating releases. Should a release fail, it is his responsibility to track down the offending files, contact the responsible developer(s), and work out a solution. The build boss enforces the SCM policy, and when developers request to circumvent stated policy,
All in all, the build boss is responsible for the smooth operation of the SCM system. His value cannot be understated and every development team should have one.

he decides if it is warranted. All in all, the build boss is responsible for the smooth operation of the SCM system. His value cannot be understated and every development team should have one.

A *change package* is a logical grouping of all the files that were modified through the course of some unit of development work (for example, PRCR). Rarely does any but the most trivial bug fix require modification of only one file. Rather, the bug fix spans multiple files that are inter-dependent to get a working executable. Trying to build with three of the four files that were changed will likely not even compile. Since using the modified versions of the affected files is an all-or-nothing proposition, it makes sense to track them as a single unit of change. This creates the capability to undo the entire change should a problem or question concerning the PRCR ever arise. This concept is directly supported by such SCM tools as Continuus/CM, ClearGuide, and others. Of course it can be manually supported by any SCM user through such mechanisms as release notes (discussed next).

A *release* is a formalized build that creates a milestone product. The product itself can be for internal consumption only, or for delivery to a customer. Typically a release involves applying a label to all versioned files. The label provides a means for getting a specific version of a file by name rather than number. This is a convenience for developers by logically grouping specific versions of all the files that create a build. The manual alternative would be to keep a list of the version number for every file needed to build a given executable.

The *release notes* are a simple ledger created after a version label is applied to the source code. It lists all the files that were modified, the version used in the previous release, the version used in the current release, the order in which branches were merged into the mainline, and a brief description of the reason for the change, or a reference to the PRCR that the work was performed against.
Once the effort of using the SCM tool has been reigned in, it is time to take a careful look at the high-level concepts and how your organization is supporting them.

The release notes do not contain any unique information; rather, they summarize in one convenient location everything that makes this release different from the last one. Figure 1 shows a sample Release Note from a hypothetical project.

The first line identifies the label to which this note applies. Again, by using the label, all the components that make up this build can be retrieved with 100% confidence. It is not necessary to keep track of the relevant version number of every individual file. The next section lists all the files that make this release different from the previous one. Any modules that were added, removed, or changed are listed with: the version number they had in the previous release, the version number for the current release, and a list of all branches from which the changes were merged (this project uses the BDM). The third section is a high-level description of what was fixed or otherwise changed and why. Notice the Feature ID is an implementation of the Change Package concept. If something proves wrong with the work done for the Phone effort, it is possible to undo the change for each of the seven files that were affected by Phone. A new release would then be performed to ensure a working mainline is always preserved. The last two lines identify a point of contact and a date of creation. The point of contact is usually the build boss. He is the one most intimately familiar with the release since he is the one actually going through the process of building and releasing the software.

The big picture

This article has largely been about techniques for using your SCM tool efficiently and effectively. However, the tool itself is less than half the whole configuration management picture. Configuration management encompasses the following seven aspects:

- Identification: identifying the input components
- Control: the policy for allowing changes to the input components
- Status accounting: recording vital statistics (for example, change requests) about the product
- Audit and review: verification of the product being derived from a known set of inputs
- Manufacture/build: the construction of product from its input components
- Process management: ensuring the CM policy is being designed and followed
- Team work: managing the interactions among multiple developers

Once the effort of using the SCM tool has been reigned in, it is time to take a careful look at the high-level concepts and how your organization is supporting them. Different SCM tools support different concepts at varying levels. For example, RCS is heavily centered on the identification and control aspects, but has minimal built-in audit and review capabilities. Chances are, the concepts that the tool you are using most strongly supports are the ones your organization most strongly supports. However, effort should be made to supplement any configuration management aspects not inherently part of your tool’s paradigm.

The best place to start the rest of your configuration management effort is the CM plan. The CM plan gathers in one location all the policy decisions for implementing configuration management on your project. Once the policy decisions have been made, a foundation exists to create the procedures which will be used over the course of the development effort. A full discussion of a CM Plan is outside the scope of this article. However, I suggest reading “Configuration Management (CM) Plans: The Beginning to Your CM Solution” (by Nadine Bounds and Susan Dart) as an excellent place to start further research.

Allan Folz is a staff engineer at Logikos Inc. a software out-source and consulting company located in Indiana. He has developed software for a variety of applications in the automotive industry in both product-intent and R&D settings. Allan received his BS in electrical engineering from Rose-Hulman Institute of Technology in Terre Haute, IN. He encourages correspondence and can be reached via e-mail at afo@logikos.com.

References


