Aegis

A Project Change Supervisor

Peter Miller

Australian Geological Survey Organisation

ABSTRACT

Many CASE systems attempt to provide everything, including bubble charts, source control and compilers; even if you don't like one of the tools, you are stuck with it. In contrast, UNIX[1] utilities provide many components of a CASE system − compilers, editors, dependency maintenance tools (such as *make*), and source control tools (such as RCS). You may substitute the tool of your choice if you don't like the ones supplied with the system. Aegis adds software configuration management to these tools. True to UNIX philosophy, Aegis does not dictate the choice of any of the other tools (although it may stretch them to their limits).

1. Introduction

Aegis performs some of the tasks increasingly referred to as software configuration management (SCM). It supervises the development of changes to a project and the integration of those changes back into the master source of the project.

There are existing programs, such as RCS or CVS, which could do some of this task. The difference is that Aegis does not allow changes to be unconditionally added to the master source. It enforces a number of requirements, each designed to ensure that the project does not "go backwards" because of a change.

The word aegis was chosen as the name because of its meaning:

aegis (ee.j.iz) *n.*, a protection, a defence.

In Greek mythology, the god Zeus had a shield called Aegis, which provided a supernatural defence. While Aegis does not claim supernatural powers, it does provide a way of managing changes to a software project and a solution to many of the problems encountered when a team of developers write software. Some common examples of these problems include:

- bugs which refuse to die;
- lost changes, from developers "tripping over" each other;
- not knowing who changed the source, or why;
- using the wrong versions of the sources to build the project;
- not having a working copy to demonstrate to anxious management, or anxious customers.

While Aegis can help solve these problems, and many others, it cannot solve every problem, it is not a silver bullet.

2. What is SCM?

Software configuration management (SCM) is a large and increasingly complex discipline. It can be briefly described as consisting of a number of parts, which include:

Manifest Control
: It is necessary to know what all the source files of a project are, and where they can be found. It is also necessary to know when they were added or removed.

1. UNIX is a trademark of Bell Laboratories.

Version Control
: It is necessary to know which version of each source file is used. It is necessary to be able to recreate earlier versions of the project from this information.

Build Control
: It is necessary to know how to construct the object of the project from the source files.

Change Control
: It is necessary to know who performed each change and when, who initiated each change and why.

Quality Control
: It is necessary to know that the changes made to your project meet your quality criteria. It is essential that changes do not "break" an otherwise working project.

The last item on this list is frequently absent from SCM systems, and is a major focus of Aegis' design.

2.1 Development Model The master source of a project, and all the implications flowing from it, such as object files and executables, and all the tests, is called a *baseline*, to use common SCM terminology.

Aegis is designed to try to ensure that the baseline always *works*, where "works" is defined as passing all the tests in the baseline.

All file history tools include two functions: you can "check out" a file for editing, and you "check in" the file when you are finished. The concept may be generalized for sets of files. The problem with using such a simple process is that the "check in" is unconditional. Aegis breaks the "check in" into several steps, so that inadequate or defective alterations to the baseline are far less likely.

In Aegis, the unit of change to the baseline is, un-originally, called a *change*. Each change must be atomic, it must leave the baseline in a working state, and must not depend on any other change being performed simultaneously. For example, when the interface to a function is altered, the change must also include alterations to every call of that function.

Aegis tracks all the source files included in a change, and sufficient history information for each file so that when a change is finished, an algorithm similar to that used in RCS-Merge or CVS-Update may be employed to resolve any problems caused by the ability to simultaneously include the same source file in several different changes.

Developers may not directly edit the baseline. The baseline is updated by a user called the *integrator*, who integrates the baseline with the change, and then validates the result, before accepting it as the new baseline.

3. Change Control

Change control in Aegis is implemented as six states which a change must pass through. Various criteria must be met to leave one state and advance to the next. See Figure 1 for an overview of the change states.

3.1 Awaiting Development Not all members of the team may create changes. This is controlled by an access list, and thus may be as restricted or open as your project requires. Once a change has been created, it is in the *awaiting development* state. A change consists of little more than a description in this state.

3.2 Being Developed Not all members of the team may develop changes. This is controlled by an access list, and thus may be as open or restricted as your project requires.

A variety of methods may be used to assign changes to developers, but at some point, either from receiving instructions to do so, or browsing and finding one, a developer assigns a change to herself. Once a change is assigned to a developer, a development directory is created for it, and it advances to the *being developed* state.

This state is the coal face. Project source files can be edited in this state only. Source files which are to be edited are copied from the baseline into the development directory.[2] Aegis is used to copy the files, so that it knows which files are being modified by this change. Files can also be created or deleted by a change. Again, Aegis is used to do this, so that it knows what is happening. You don't have to issue two commands this way, one to tell UNIX to do it, and another to tell Aegis that you have done it.

Build Once you have edited the source files, the change must be built. *Building* is the process of manipulating or translating the source files in

2. Only those files which are to be edited need to be copied. The baseline acts as a cache for object files not present in the development directory.

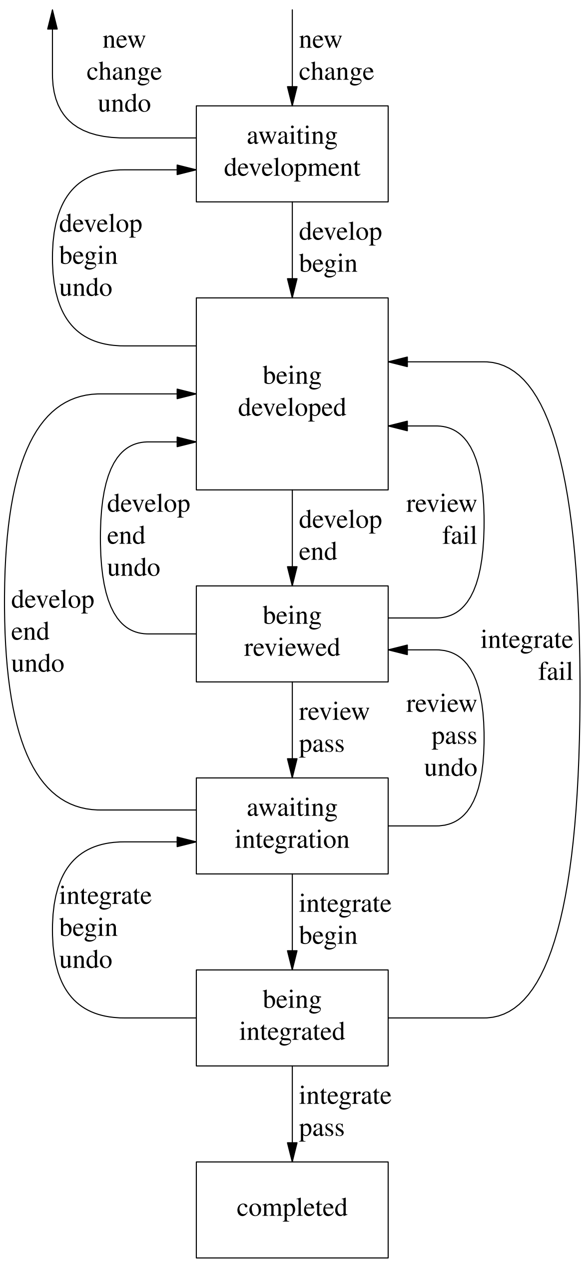

Figure 1: Change States and Transitions

some way to produce the object of the project. For programs this usually means compiling the source files and linking them into an executable program. The build is performed via Aegis, so that it can look at the exit status and know if the build succeeded or not. A successful build is a requirement for leaving the *being developed* state and advancing to the next.

There are no facilities in Aegis for describing how to build the project. Instead, Aegis delegates this task to a *dependency maintenance tool* (DMT). This delegation is stored as a UNIX command to be performed when a build is requested. There is no provision for Aegis to understand *any* dependencies, as these are expected to be described in the DMT's configuration file, itself a source file of the project.

Typically a DMT is a program like *make*(1), however this old-faithful is not able to cope with the demands placed upon it by Aegis.[3] The major problem is the two-directory structure used: when the DMT is looking for files, it must first search in the development directory and then in the baseline. It is best if the DMT can do this transparently, because it makes the rules much easier to write. The *cook* program, also written by the author, is a DMT known to work with Aegis.

Test Each change must be accompanied by at least one test. Except for the way tests are created, so that Aegis knows they are tests rather than ordinary source files, they are treated identically to source files: they may be modified and even deleted by later changes, by the same commands as used for non-test source files, and they are subject to the same process.

Tests are Bourne shell scripts. They are executed via Aegis so that it can examine the exit status, and know if each test has passed or failed. Having tests, and passing them, is a requirement for leaving the *being developed* state and advancing to the next.

As changes are integrated into the baseline, the tests which accompany each change accumulate into a regression test suite. A developer may optionally run all or part of this regression test suite to make sure that her change has not broken any existing functionality. Because this regression test suite grows steadily, it is not practical to run all of it for every change; so Aegis is designed to make it relatively easy to run a (hopefully representative) subset.

Difference Once a change builds and tests successfully, it is *differenced*. This is the process of creating files showing the difference between the baseline and the development directory, for each file in the change. This allows reviewers to examine all the edits made by the developer, not just the ones they can find. The difference command is configurable, as appropriate for each

3. Even GNU Make 3.65 is not up to the task; the VPATH semantics are too limited.

project.

Conflicts This difference stage is also where problems with out-of-date files are resolved. If two developers copy the same revision of the same file into two different changes, one of them will be integrated into the baseline before the other, hence the possibility of one or more files being out-of-date. A three-way merge between the common ancestor, the file in the development directory, and the current file in the baseline, is performed.[4] This produces a merge of the two competing edits, which the developer should then examine to make sure the automatic merge has produced sensible results. This merge tool is also configurable, as appropriate for each project. The out-of-date file is then marked as up-to-date, and the change will require another build and test, to ensure that the merge has not broken anything.

3.3 Being Reviewed

Once the files are up-to-date, built and tested, the developer may advance the change to the *being reviewed* state. At this point, all the source files in the change are locked, preventing any other changes with files in common from advancing to the *being reviewed* state. If any change file is already locked, the developer will be told to try again later.

The style of review is not dictated by Aegis. The only requirement is that an authorised user tell Aegis that the review passed or failed. Not all members of the team may review changes. This is controlled by an access list, and thus may be as restricted or open as your project requires.

A number of review schemes have been observed. Two extremes are presented here:

- A single team member is responsible for coordinating all reviews. Each review is performed by a panel of four team members in addition to the developer of the change. Only the review coordinator, after receiving the paperwork from the review panel, may pass or fail reviews.
- Any developer may review any change; this is done informally. Aegis prevents a developer from reviewing her own change, to avoid an obvious conflict of interests.

Many other review styles are possible, but the one best for your project will probably fall between these extremes.

4. This is the same algorithm as used by CVS-Update and RCS-Merge.

Reviewers know several things about a change in the *being reviewed* state, because of the requirements for getting there.

- The change is known to build successfully.
- The change is known to have tests and to have passed them.
- The source files in the change are known to be up-to-date with respect to the baseline.

This allows reviewers to concentrate on completeness of the code, completeness of the tests, and standards issues, etc.

If a change fails review, it is returned to the *being developed* state for further work by the original developer. The reviewer is not responsible for fixing problems found by the review.

3.4 Awaiting Integration

Once a change passes review it is advanced to the *awaiting integration* state. This state is a queue. Only one change at a time is integrated into the baseline, even though all changes in this state have no files in common. This allows clear indications of which change is at fault, should the integrator discover that there are problems. See Figure 2 for a diagram of how files flow through this model.

3.5 Being Integrated

Not all members of the team may integrate changes. This is controlled by an access list, and thus may be as open or restricted as your project requires.

To integrate a change with the baseline, an integration directory is created by copying the baseline, or more usually creating a logical copy using links. The change is then applied to this integration directory.

The integration copy is then built and tested. This is to ensure that it was not just some quirk of the developer's environment that allowed the change to get this far,[5] and also to have all files in the new baseline consistent with each other.

The integrator may choose to run a representative subset of the regression test suite, in addition to the tests which accompanied the change.

In addition to rejecting a change because it fails to build or test, an integrator may also act as a reviewer; this is a good place to watch the watchers, and monitor the quality of reviews.

5. For example, a weird environment variable setting, or a bogus *cc* command in the PATH which always exits success.

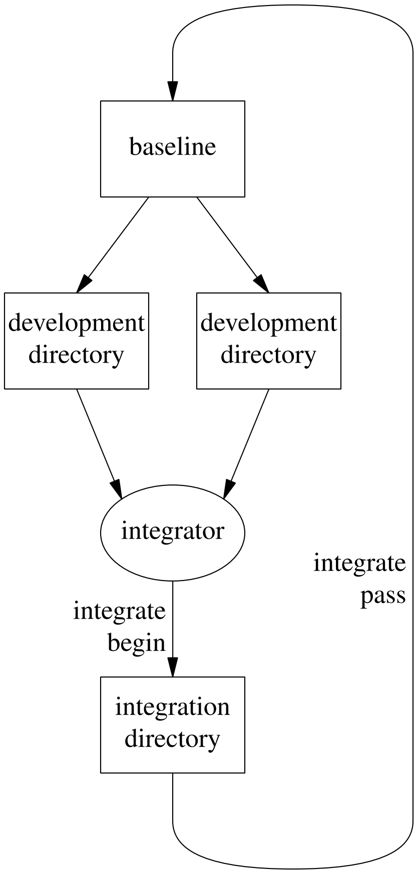

Figure 2: Flow of Files through the Model

Integrators may thus veto a change even though it builds and tests successfully.

If a change fails integration, the integration directory is removed, and the change is returned to the *being developed* state for further work by the original developer. The integrator is not responsible for fixing any problems found.

3.6 Completed

Once a change builds and tests successfully, it may be advanced to the *completed* state. The file histories are updated at this point, and the file locks released. The old baseline is removed, and the integration directory is renamed to be the baseline. The development directory is also removed.

Unlike earlier states, a change in this state cannot be reversed. If you subsequently wish to remove the change, you will need to create another change and repeat the whole process with all edits in reverse.

A change consists of a description and a list of files and versions in this state. A full history of state transitions has been kept, including who performed them and when.

4. Quality Control

Quality is addressed in a number of ways. As you can see from the above description of how Aegis manages change control, the mandatory testing and reviewing are steps in this direction. Exactly *what* is tested or reviewed is up to you, but the places exist in the process for them to be done. They can be as elaborate or simple as your project requires. Note that there is more to software quality than these two items, and they are not the only places where tests and reviews can take place.

4.1 Does it work?

A major advantage of Aegis is the ability to answer the question "Does it work?". This question is asked from a number of perspectives:

- The developer wants to know "does this change work?". Aegis provides the answer with tests for each change. Developers have always tested their code, but Aegis provides ergonomic advantages, never forgets to test something, and the tests are preserved for future use.
- The integrator wants to know "does this change break anything else?". Aegis provides the answer with a constantly growing regression test suite, and also makes the developer's own tests available to the integrator.
- The project leader, and management further up the tree, want to know "does the project work?". They want to be able to touch and feel progress towards the target, and they want some confidence that the project will not cease to work at random (but usually disastrous) times. Aegis provides the answer here in the form of a baseline which always works, and is always available for demonstrations.

The various mandatory tests and validations are configurable for each project (and in some cases, for each change). You may use as many or as few of the safeguards provided by Aegis as you need for each project.

4.2 No Back Door

Another issue is whether there is a "back door" into the process, so that a developer who finds the process tedious can avoid it and just "fix" the baseline directly. With Aegis, there is no way to circumvent the process.

Access to the baseline is read-only for the development team, including the integrator. Access is protected by the UNIX group and *umask* mechanisms. A UNIX group and umask is associated with each project, and any commands Aegis executes will arrange to have the appropriate group and umask, to ensure that all users in that group have access (even if the user has a different default group).

All developer commands are run as the developer, and hence have the developer's permissions. All integration commands are run as the *project owner* who is usually not the integrator (so the integrator can't edit, even if she wants to). It is only ever necessary to login as the project owner to perform actions beyond Aegis' scope, such as recovering after a disk crash.

5. Manifest Control

Aegis remembers where all the source files are. They are initially created in changes, and only exist in the baseline after a successful integration. The location of the baseline and all development directories are known, and the names of all the source files in them are known.

6. Version Control

As described above, Aegis delegates the file history maintenance to the package of your choice. All Aegis requires is the ability to determine the latest version number for each file's history at integrate pass, so that the random string (it need not look anything like a number) may be used later to extract an earlier version, should it be needed.

The version string for each source file in the baseline and each development directory is known, so that the difference and merge facility described above can work.

In addition, a project version may be specified when copying files from the baseline into a change. Thus an earlier version of the project may be recreated, in order to reproduce a bug, for example.

7. Build Control

The change control description mentioned that build control is delegated to the dependency maintenance tool (DMT) of your choice.

7.1 Capabilities The DMT needs to be able to cope with the fundamental concept of two directories. This is a "search path" for every file, no matter what the file is used for.

The baseline contains all the implications flowing from the source files, this typically means the object files from compilations and the linked executable. It could also include documentation and manual entries formatted from appropriate source files.

The development build may thus compile a minimum of code, and link the rest from the baseline, minimizing disk usage and compile time across all developers.

There is a catch: the dependency maintenance tool must be able to detect when an include file in the development directory logically invalidates an object file in the baseline, necessitating re-compilation of a baseline source file, and leaving the object file in the development directory for linking.

Experience has shown that the various *makedepend* programs do not work very well. What is most needed is the ability to determine the include files "on the fly". This implies the ability to give DMT rules like

```
%.o: %.c `includes-of %.c`
        $(CC) -c %.c
```

where *includes-of* is a program to be invoked when the rule is matched, rather than when it is read (note the back quotes).

7.2 Dependencies The DMT is expected to know *all* project dependencies. This functionality is completely delegated, and so Aegis knows *nothing* about any dependencies.

The configuration file for the DMT is a project source file, and therefore is altered by the same process.

8. Directory Structure

Aegis attempts to dictate as little as possible about the directory structure of the projects it supervises. There is one mandatory file, and one mandatory directory. The mandatory file contains Aegis' configuration information for the project, the mandatory directory contains all tests for the project. The configuration file, and all tests, are treated as source files, and are subject to the same change process.

The source directory tree of each project may be as deep or as shallow as required.

The placement of project directories is completely configurable. Each project may be owned by a unique user if desired, and Aegis can manage many projects simultaneously. Security is through the UNIX groups mechanism, so it can be as open or restricted as required.

When a change is being developed, it has its own development directory. This development directory is a subset of the baseline. Only those source files which need to be edited are present in the development directory.

9. Applicability

There are some projects which are well suited to supervision by Aegis, and there are others which are not.

Ideally suited projects are programs which take a set of input files, process them, and generate a set of output files. Test cases may be easily generated, and actual output compared with expected output.

Another class of programs have full-screen text interfaces or GUIs and thus are not so well suited to supervision by Aegis. Because tests are Bourne shell scripts, only the functionality accessible from the command line can be automatically tested. In these cases there are three options:

- Change the program to optionally accept fake input and to write screen dumps to files, thus providing a testable case.
- Change the program to allow access to the functionality from the command line, thus providing a testable case. Note that this cannot test the user interface.
- Do not do any automated testing. This may be configured for a single change or for a whole project. You still get the supervision aspects of Aegis, but no regression test suite.

The least suitable class of programs for supervision by Aegis are stand-alone programs. Operating systems and embedded systems are members of this class. The program in a hand-held calculator, for example, would be extremely difficult to test from a shell script. It is possible to test this class of programs with the right hardware, but it is usually impractical.

10. Summary

This paper has given a very short overview of Aegis, and described a few of its strengths. Things to remember about Aegis include:

Aegis is designed to be a small piece in a larger system, like many other UNIX utilities.

Aegis is a project change supervisor, it performs part of what is becoming known as software configuration management (SCM). This provides control for manifest, versions, building, changes and quality.

Aegis is not a history tool, such as RCS. It is layered above such a tool.

Aegis is not a dependency maintenance tool, such as *make*(1). It is layered above such a tool. Any dependencies which cannot be expressed in the rules file of the DMT, cannot be expressed by Aegis.

Aegis is not a bug tracking system, it has no mechanisms for tracking bugs and telling you which are fixed and which are not. However, there are notification hooks to liaise with such a system.

Aegis does not draw Gantt charts, bubble charts, flow charts, or any other pretty pictures. It does not itself generate any code. It is not a CASE system, it is a component of a CASE system.

Aegis attempts to dictate as little as possible about each project. It dictates very little directory structure, and it does not dictate test content or the review method. Reviews and tests in addition to those required by Aegis may be performed.

Aegis is free. This means that it has an excellent cost/benefit ratio, compared to commercial products, even if it doesn't have all their features.

11. Availability

You can get Aegis by anonymous FTP from

Host:	`ftp.nau.edu`	
Dir:	`/pub/Aegis`	
File:	`aegis.2.3.tar.gz`	*the full source*
File:	`aegis.2.3.ps.gz`	*the User Guide*

Aegis is distributed under the terms and conditions of the GNU General Public License.

12. References

The following free software, and their documentation, are referred to in this paper:

- Miller, P. A., "Aegis - A Project Change Supervisor," *AUUG '93 Conference Papers,* 1993, p. 169-178.

[1] RCS 5.6
Copyright © 1982, 1988, 1989 by Walter F. Tichy.
Copyright © 1990, 1991 by Paul Eggert.

[2] CVS 1.3
Copyright © 1986, 1988-1992 Free Software Foundation, Inc.
Numerous authors, principally Brian Berliner.

[3] GNU Make 3.65
Copyright © 1988-1993 Free Software Foundation, Inc.
Numerous authors, principally Roland McGrath.

All of these programs may be fetched by FTP from your closest GNU archive site. Within Australia, you can find them at *archie.au* in the */gnu* directory.