David Matthews · Greg Wilson · Steve Easterbrook

Configuration Management for Large-Scale Scientific Computing at the UK Met Office

Received: date / Accepted: date

Abstract Computational models used in scientific research can become large and complex, and may evolve over many years. Keeping the codes up-to-date to reflect the latest science requires considerable effort, and yet scientific programmers tend to be slow to adopt best practice software development tools. In this paper we report on the experiences of the UK Met Office in adopting a new system for software configuration management. The system, FCM, is based on existing open source tools, such as Subversion and Trac, but has been tailored for the specific needs of the modeling teams at the Met Office. The new system is widely regarded as a success. It replaced a plethora of different code management tools, and has allowed development teams to dramatically reduce their release cycles, introduce parallel development, and to improve the organisation of their codes. We identify a number of factors that led to this success.

Keywords Configuration Management · Software Development Tools · Scientific Programming · Climate Modeling

1 Introduction

Scientists using computational models face a dilemma: each new generation of commercial software development tools and techniques promises to increase productivity, but those tools and techniques take significant time to set up, master, and maintain. As commercial software development tools and techniques are tailored to the needs of commercial developers, rather than those of computational scientists, it is difficult to know which ones are worth adopting. So difficult, in fact, that most scientists choose to play it safe and not adopt any.

This paper describes how one group—the UK Met Office—escaped from this trap. By combining open source tools with some home-grown "glue", the Met Office created a system to manage millions of lines of source code for Numerical Weather Prediction (NWP) and climate simulation. The system allows scientific end users to assemble the modules they need for particular simulations with much less effort than was previously required. Just as importantly, it gives users confidence that the code they are running is the code they asked for and that it has been assembled correctly. The Met Office's experience in building, customizing, and deploying these tools offers interesting lessons for other groups faced with the same challenges.

D. Matthews
UK Met Office
FitzRoy Road
Exeter, Devon, UK
EX1 3PB

S. Easterbrook and G. Wilson
Department of Computer Science
University of Toronto
Toronto, ON, CANADA
M5S 3G4

2 Tool Support for Developing Scientific Software

The large computational models used in NWP and climate research have evolved continuously over decades. The current generation used at the UK Met Office add up to more than a million lines of Fortran, of which up to one third change each year. Coordinating these changes while maintaining correctness is hard enough in any software system of this scale [11], but scientific programmers must also face additional challenges, including the need to keep track of exactly which version of the program code was used in particular experiments, the need to rerun experiments with precisely repeatable results, and the need to build alternative versions of the software from a common code base for different kinds of experiments.

Software productivity is now a major bottleneck in scientific computation [17]. Moore's Law may have dramatically reduced the time it takes to run a scientific simulation, but has had absolutely no impact on the time it takes to write and debug the necessary software. Furthermore, as codes grow ever more complex, scientists are having to devote more of their time to the messy details of software configuration management—managing different versions, releases and configurations of source code, tracking defects, coordinating changes, and extracting and building the source code to create executables.

Scientific programmers appear to have been slow to adopt best practice tools to support these activities. Carver *et. al.* [10] identify a number of reasons for this. The people who develop the code are trained primarily in their scientific discipline (rather than software engineering) and have learned programming skills along the way. Their experience of tools developed by the software community is usually disappointing—for example, these tools don't cater for their need to develop high performance code for parallel architectures. The use of legacy code (the models are built over many years) and the need for optimization means that teams tend to use older programming languages, for which the latest software development environments are not available. In addition, scientific teams prefer to develop everything in-house, rather than take the risk of relying on external sources for tools that might not be supported over the many years they will be needed.

Open source tools address this problem to some extent, as they can be maintained and adapted in-house. However, open source tools tend to come with higher adoption costs [16], because the effort needed to install and configure them to local needs can be too high, especially for small teams.

Furthermore, new tools must match the scientists' working practices. In software engineering, the term *process improvement* is used to describe a concern with identifying and correcting weaknesses in how the activities of software developers are coordinated and managed. Good software development processes increase software quality and reduce development effort, largely by preventing mistakes, and hence reducing the amount of rework needed when mistakes are detected. But introducing process improvements without good supporting tools can be counter-productive, because the new processes typically increase the burden on the programmers. Conversely, the best tools in the world are of little use unless they are used properly - i.e. within the context of a good process. Hence, tool adoption and process improvement usually need to go hand-in-hand.

We believe that good tools can speed the adoption of good processes by making them more concrete and understandable, but the tools alone will not effect a process change. Successful process change depends on the impact of the new process on individual developers' productivity (as opposed to team productivity), how compatible the process is with existing working practices, and developer's perceptions of whether their co-workers are also adopting the new process [15]. This circularity makes it hard to know which route to take to improve software tools and processes used in the development of scientific software.

3 Background: The Need For a New System

The UK Met Office has been developing scientific software for almost as long as electronic computers have existed. Simulation and data analysis are now critically important to hundreds of scientists working at the Met Office, and indirectly to the millions of people who depend on this software for everything from weather forecasts to advice on climate change policy.

Researchers at the Met Office have had relatively good configuration management processes in place for many years: for example, all key software systems were under version control with well defined change review procedures. However, different teams used different processes and tools, which led to difficulties for staff who needed to move code (or themselves) from one group to another. For example, although there was limited use of standard tools such as CVS [7], most projects used tools specific to the Met Office. This meant that new staff or collaborators had to climb a steep learning curve before being able to contribute to Met Office efforts. Similarly, the Met Office had several build systems in place to support its large Fortran code base. Each of these build systems had some powerful features, but also some serious deficiencies, which made none of them suitable for general use.

The Met Office had recognised for many years that a new system was needed to support a common working practice for all applications and ease the learning curve for new staff. In 2004, a team of three IT specialists within the Met Office was assembled to tackle this issue (including one of the authors of this article). Working closely with the scientists and systems administrators, they produced the Flexible Configuration Management (FCM) system described below.

A key to the success of this effort was the combination of grass roots effort and management initiative. The team tasked with designing and implementing the system had gotten to know each other and the scientists who would be the primary uses of the system over the course of many years. This gave their work credibility that would be lacking for solutions offered by external vendors or imposed by senior management. Even so, because of the amount of work involved, and because two very different systems and teams had to be brought together, support from management was crucial to the project's success.

From the outset, the FCM team understood how important it was to convince all the stakeholders that the new system would be a significant improvement. While the team had suitable representation on the project board to help with this, they also maintained close contact with key individuals, including system managers and other influential managers, to listen to their requirements and keep them onboard with the project. In the end, success came down to producing a system which was clearly better than any of the existing systems and ensuring that it had the support of the key decision makers.

4 Open Source Building Blocks

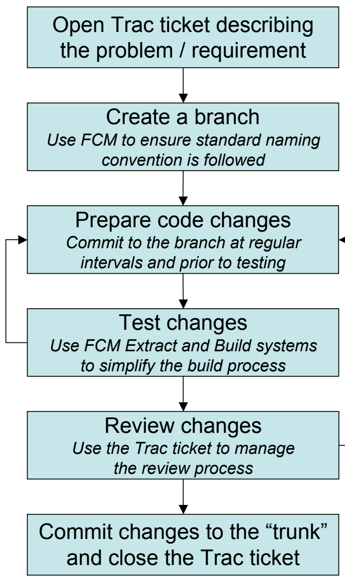
2004 turned out to be an excellent time to start work on FCM. The core of any CM system is version control; many groups had been using CVS, but it was clearly showing its age. Fortuitously, February 2004 marked the first major release of Subversion [14, 6], a new open source version control system that was explicitly designed to be "a better CVS".

The fact that Subversion was free was not essential to the Met Office. However, a number of Met Office systems are used externally by universities and other meteorological centres, and there were clear advantages to everyone being able to use the same version control tool. An initial evaluation showed that Subversion had the features and reliability required, so there was no need to evaluate commercial tools in detail.

There was, inevitably, some concern that Subversion had only just had its first major release. However, after spending time monitoring the Subversion mailing lists, the FCM team was able to gain confidence from the highly active developer community and the already large user base (confidence which they would have had difficulty gaining in the first release of any commercial product). It was clear that the Subversion team had not made this first major release until they were convinced it was ready. Subversion also had the backing of a commercial company (CollabNet). Overall the FCM team felt that they could trust Subversion with their mission critical source code and that help would be available should they run into problems. Subversion has since become very successful, with an ever-expanding user base and a strong development team, which gives confidence that it will not become an "orphan".

The second key tool needed was an issue tracker, and again the timing was good, as February 2004 marked the first public release of a lightweight software project management portal called Trac [8]. Like SourceForge [5] and other software project portals, Trac combines a wiki with an issue tracker and Subversion repository browser. While it was (and is) not as mature or widely used as Subversion, even in 2004 it had the key features the FCM team felt were needed to coordinate work on their applications.

Just as importantly, Trac's interface was very simple—much simpler, for example, than those of SourceForge or the widely-used open source issue tracker Bugzilla [1]. This was an important factor in gaining acceptance of the new system: not only did it lower the learning curve,



Open Trac ticket describing the problem / requirement
Create a branch
Use FCM to ensure standard naming convention is followed
Prepare code changes
Commit to the branch at regular intervals and prior to testing
Test changes
Use FCM Extract and Build systems to simplify the build process
Review changes
Use the Trac ticket to manage the review process
Commit changes to the "trunk" and close the Trac ticket

Fig. 1 The FCM Change Process.

it also meant that users saw the purpose and value of every single field[1] of every form they filled in[2].

Clearly, the relative immaturity of Trac was a concern. However, the availability of the issue tracker is not essential for developers to do most of their work so the FCM team felt happier accepting more of a risk. In any case, if serious problems were discovered with Trac they knew that they could extract all the data from the underlying database and migrate to a different solution (albeit with some pain).

Community support for these tools has been excellent. In the case of Subversion, the few minor problems encountered were issues already known to the developers. With Trac, the FCM team submitted a number of minor new bug reports and enhancement requests, to which they received quick responses. The team also fed back a minor bug fix of its own, allowing it to contribute in a small way to the community.

5 Configuration Management

The key software systems at the Met Office all follow a similar CM process:

– All changes to the system have to be associated with a ticket (this is what an issue is called within Trac).
– All changes have to be prepared in a version control branch.
– A reviewer has to sign off the changes made in that branch before it can be merged back into the main system, at which point the associated ticket is closed.

[1] Well, most of the fields...
[2] Well, most of the forms...

Subversion and Trac have proven to be very effective at supporting this CM process (See figure 1). Subversion provides support for branching, which allows parallel development and encourages developers to store intermediate versions of code in the repository (rather than on personal machines). Trac allows all bug fixes and enhancements to be recorded in tickets on the web, which hyperlink to associated code changesets and to wiki pages for changes that need to be documented in more detail. If all code changes are associated with a ticket then Trac makes it easy for anyone examining the code to trace each change back to its associated ticket, and thence to the documentation and discussion that led to the change.

It is important to note, though, that this process is not mandatory: Subversion allows users to make changes directly in the main code line, and unlike some commercial tools, does not require an open ticket to be closed (or at least updated) as part of that process. This flexibility means that smaller projects, with less strict requirements are also able to make effective use of the same tools. In particular, users who are required to use FCM on bigger, more stable codes do not then abandon FCM when working on smaller, informal projects.

This flexibility is important, but often overlooked by tool developers, who tend to build process enforcement into their tools. Many systems and teams start small, with little experience of industrial-strength software development; it is important that they be able to "ease into" more formal procedures. FCM (which was self-hosting from an early stage) is an example of this. During initial development the team used informal working practices with little reviewing and most changes happening on the main code line. As the system grew closer to going live, the team began to shift development work to branches and to review all changes. This shift was relatively pain free since they could continue to use the same tools, just in a slightly different way.

6 The FCM Tool Itself

Subversion's flexibility has been a key to its general success. However, with this flexibility comes additional complexity in that some common commands can be quite tricky to master. In order to achieve widespread acceptance at the Met Office, the FCM team felt a somewhat simpler interface was required. The first step was to define working practices to limit the ways in which Subversion could be used. For those familiar with Subversion, the key restrictions are:

1. FCM enforces a particular repository structure and branch naming convention.
2. Working copies are assumed to contain a single revision and repository location (no "mixed working copies").
3. Partial commits are not supported.

1. Create a branch
Subversion command:
svn copy -r 123 svn://server/my_repos/my_proj/trunk \
    svn://server/my_repos/my_proj/branches/dev/userid/r123_mybranch
Equivalent FCM command:
fcm branch -c -r 123 -n mybranch fcm:my_proj

2. Switch your working copy to a branch
Subversion command:
svn switch svn://server/my_repos/my_proj/branches/dev/userid/r123_my
Equivalent FCM command:
fcm switch dev/userid/r123_mybranch

3. Show the changes in your working copy relative to the base of your branch using a graphical viewer
Subversion command:
svn diff –diff-cmd graphic_diff \
    –old svn://server/my_repos/my_proj/trunk@123 –new .
Equivalent FCM command:
fcm diff -b -g

4. Merge in the changes from the trunk since you created your branch
Subversion command:
svn merge -r123:HEAD svn://server/my_repos/my_proj/trunk
Equivalent FCM command:
fcm merge trunk

Fig. 2 Example FCM commands

The team then built into FCM a lightweight rule-conformant layer on top of Subversion with a simpler interface designed to support the working practices and conventions they had defined.

The most important differences between the FCM interface and "pure" Subversion interface concern the processes of branching and merging, plus the use of a graphical difference and merge tool called `xxdiff` [9]. These are all key areas for the development process where it was essential to simplify the interface in order to make it accessible to the scientific user base. However, it is worth noting that the FCM interface was designed to match the underlying Subversion interface as closely as possible so that anyone who was already familiar with Subversion could easily switch to FCM and back.

Some examples of common FCM commands, together with the equivalent Subversion commands, are given in figure 2. In each case, FCM simplifies the command by relying on a standard organisation for the repository structure and standard naming conventions. For example, in figure 2a, a branch is normally created in Subversion by a complete server-side copy, requiring the user to specify the full URL for both the source and destination. in FCM, it is only necessary to specify the project, and a name for the branch. These examples illustrate why the FCM team felt that a simpler interface was essential in order to achieve acceptance of the new system.

7 A Build System for Fortran 9X Code

Most of the scientific code at the Met Office is written in Fortran 9X. Some of the systems are very large and can take a considerable time to compile so an efficient build system is essential.

FCM's build system is based upon GNU Make [4], a widely-used tool that keeps track of dependencies between files and recompiles, relinks, copies, or otherwise updates files that have fallen out of date. Since its appearance in the 1970s, Make has been one of the most widely used build tool—possibly *the* most widely used—in industry. Many commercial and open source applications rely on complex Makefiles which often use only a limited subset of Make's functionality. Some applications also rely on recursive Makefiles which can cause numerous problems, e.g. causing Make to be overly sensitive to changes in the source code, and hence greatly increase compilation time [13].

The FCM build system avoids these problems by relying on a single top-level configuration file written in a much simpler syntax than Make's. FCM then automatically generates the complex Makefile required to do what the user wants done. This makes life simpler for users, and makes it much easier for any defects to be fixed and new features added. As with the FCM wrapper around Subversion, this tool relies on users following naming conventions and other process mechanisms; put another way, it gives them a tangible reason to stick to those conventions, and feedback when they don't.

One of the challenges for Fortran 9X build systems is handling inter-module dependencies. Code needs to be compiled in the correct order and, for incremental builds, changes to a source file need to trigger re-compilation of any dependent files. Where possible, FCM analyses the dependencies of each source file automatically. To do this, it requires:

1. One program unit (e.g., module, subroutine, or function) per file.
2. All used routines defined within the Fortran module or in included interface files.
3. Source code comments to identify other dependencies (e.g., on Fortran 77 or C code).

The first two requirements are good practice that most developers follow anyway, so insisting on them was unproblematic. For most applications, the final requirement only applies to a small number of components, so it has not been difficult for developers to modify the source code to comply.

But saying that *most* developers do something isn't the same as saying that they *all* do it. When deploying FCM, the team was faced with several applications whose code did not follow these rules. This was certain to be an ongoing issue, since the Met Office needs to be able to integrate code from other organisations into its models with minimal effort (and preferably without modifying their original source). FCM supports this by providing "escape mechanisms" so that users can define dependency information manually. While this could potentially be so time consuming as to make FCM too costly to use, experience so far is that most Fortran 9X codes can be compiled using FCM with very little effort. Here and elsewhere, the lesson learned is knowing when to stop: a tool only has to handle enough common or expensive cases to be adopted, not all conceivable cases.

An innovative feature of the FCM build system is that it allows a build to inherit source code and object code from another build. This saves disk space and compilation time, both of which are important when dealing with applications as large as those at the Met Office. Typically an inherited build would be from a stable release of a system. Developers who have prepared code changes relative to this stable release can then inherit from the build and will only need to compile what is necessary as a result of their code changes rather than the entire model.

FCM's build system is also "smart" with regard to compilation and preprocessor flags. Changes to these can have as much impact on the compiled code as changes to the contents of source files, because the flags can change the behavior of the code, altering subroutine interfaces and code dependencies. However, most build systems do not include them in dependency calculations. As a result, users often waste time doing full recompilations just to ensure that they have the correct compile options in use, or lose valuable time chasing down bugs that are actually "just" mis-matches between memory layout, loop optimization, or synchronization directives. FCM deals with this by allowing users to specify flags for the entire system, at the directory level, or for individual files. It triggers a build of the appropriate scale when these flags change. It also has an optional pre-processing step prior to the dependency analysis in which changes to preprocessor flags trigger the appropriate re-compilation. These features are particularly useful when inheriting from another build.

One final build-related problem was related to Fortran 9X's type checking of arguments using subroutine interfaces. If subroutines are declared in modules then the interfaces get used automatically. However, working in this way can lead to cascading compilation issues for incremental builds where modifications to a commonly used routine can result in recompilation of almost the entire code, even though the subroutine interface has not changed. One way around this is to use standalone subroutines and then define the interface for a routine in the calling routine, typically by using an include file. Manually maintaining these include files can be error prone, so FCM avoids this issue by generating them automatically at build time. Further, the interface files only get updated if the subroutine interface is changed, so that no unnecessary compilation is triggered.

8 The Extract System

The FCM extract system provides the interface between the configuration management and build systems. This sounds a relatively simple task: just extract some code and then run the build system. The reality, though, is much more complex.

Firstly, users may need to pull in code from a number of different repositories. This is especially true if a number of systems share some common code, a practice we obviously want to encourage. By making it very easy to pull in such code at build time, the FCM extract system helps to encourage it.

Scientists at the Met Office often want to be able to pull together changes from a number of different branches containing proposed scientific changes which are still under development, so that they can evaluate new models or algorithms. If the changes overlap, the only way to do this is to create a new branch and merge the different changes into the branch. However, in many cases, the changes do not overlap; in these cases, the extract system will merge the selected branches automatically. This initially only worked if the changes never modified the same file, but FCM has now been enhanced to enable it to combine changes to the same file so long as there are no line clashes. This assumes that if the changes are to different parts of the file then they are probably safe to combine. This is the same assumption Subversion itself makes when performing a merge, so doing it as part of extraction doesn't introduce any additional risk.

The extract system generates the configuration file required by the build system. This means that users only ever need to deal with a single configuration file. Once this single file is set up appropriately then they can get their code extracted and built with a couple of simple commands.

Code can also be mirrored to a remote build system if required. This is particularly important at the Met Office, where scientists need to be able to run the extraction on desktop systems in order to include locally modified code before mirroring it to their supercomputer where it can be compiled and executed.

Finally, like the build system, the extract system is able to inherit from a previous extraction, hence saving disk space and reducing extraction time. This means that when scientists are testing a code change which only affects a handful of files the extract system only needs to extract those files rather than the thousands which may make up the entire system.

An example of a very simple extract configuration file for some Fortran code is given in figure 3. Running the command "fcm extract" against this file would result in the code tree being extracted from the Subversion repository and a build configuration file would be generated. Running the command "fcm build" would then result in any program units found in the source code being compiled into executables. Some points to note:

```
# Tell FCM this is an extract configuration file
cfg::type ext

# Extract and build the code in the same directory
# as this file
dest $HERE

# Recursively extract all the code from a branch
# in the repository
repos::my_proj::base fcm:my_proj-br/dev/userid/r123_mybranch
expsrc::my_proj::base

# Compilation options
bld::tool::fc ifort
bld::tool::fflags -g -check bounds -traceback -w95
```

Fig. 3 Example Extract Configuration File

1. If the Fortran code follows the FCM guidelines, FCM will automatically work out all the dependencies and compile the necessary code in the correct order. If the code does not follow the standard then FCM will need further information defined in the configuration file to help it do the right thing.
2. Remember that there are no other Makefiles or anything like that cluttering up the source tree - everything needed is defined in this file.

Real life cases would typically need more build options, and the more advanced extract features such as combining branches, mirroring, etc. would require further entries in this file. However, the example illustrates the point that a single file can be used to configure the entire extract and build process and that the FCM system does a lot of the hard work for you.

9 Migrating to FCM

Building a better mousetrap is one thing; getting people to use it is often quite another. Prior to FCM, most Met Office groups had reasonable CM tools and processes in place, which meant that they were trying to evolve existing practices rather than introducing entirely new concepts. In some ways, though, this made the task harder: they weren't just saying, "This is going to make your life so much easier," but, "Please throw away the tools which you've been using happily for the last ten years or more and start using these new unfamiliar ones instead." As Glass has observed [12], any new tool or process initially makes its adopter *less* productive; persuading potential users to work their way through that period was therefore crucial to FCM's success.

The FCM team therefore put a lot of effort into supporting the migration process—almost as much, in fact, as was put into building FCM in the first place. For example, the team wrote an import script which allowed all of the code stored in the most common existing version control systems to be put into Subversion with history intact. Another script imported tickets from a locally-written issue tracker into Trac. A lot of time was also spent documenting the system. Finally, the team prepared an extensive tutorial that became the basis of training workshops which were run for all the developers as the systems were migrated.

But FCM's developers didn't just import code: in some cases, as they were bringing code into FCM, they took the opportunity to refactor it, sometimes extensively. Fixed-format Fortran was converted to free format, code headers were updated and standardised, and directory structures were redesigned. These kind of changes are typically so disruptive as to be impractical while any development effort is going on, so the migration was a one-off opportunity to do some of these things without (much) additional aggravation.

Finally, most system managers could see the advantages provided by FCM and were keen to migrate. In the remaining cases, though, migration to FCM was effectively forced since it was the members of the new FCM team who had been supporting the old tools. Judicious use of both carrot and stick allowed FCM to reach critical mass within the research areas of the Met Office very quickly: the first official deployment was November 2005, most systems were migrated by March 2006, and migration of the main NWP and climate model was completed in November 2006.

10 FCM as a Service

The last thing which helped to ensure the wide adoption of FCM at the Met Office was that its developers did much more than simply provide it as a tool. Anyone with a requirement for a new system could (and can) ask the FCM team to set up Subversion, Trac, and associated tools for them. They then look after the system, making sure that it is regularly verified and backed up. They also keep it up to date with the latest developments to all the tools FCM is built upon and manage upgrades when appropriate.

One of the reasons this is manageable is that many maintenance procedures have been automated: it is not really any harder to manage fifty systems than it would be to manage five. The maintenance side of the service requires much less than one full time staff member.

11 Evaluation

All of the Met Office's key modeling systems have now been using FCM for at least a year. It is clearly a success, in part because most users are fairly indifferent to it: they are able to focus on science rather than software infrastructure (although some have been kind enough to say how much they like working with FCM and how much easier they find it to work with than its predecessors).

A few, however, really don't like FCM. This isn't surprising, considering the scale of the changes that have been made to working practices which have been in place for many years. The FCM team has tried to listen to these users and to get to the root causes of their problems. Some changes have been made (or are in the works) to deal with specific concerns, but in other cases users have just had to accept that there is some pain in adopting new working practices.

We have not yet attempted to assess the impact FCM in quantitative terms such as "X% improvement in productivity". However there is plenty of anecdotal evidence of its success:

1. Staff no longer have to get to grips with lots of different tools - all key systems use the same tools and follow very similar processes.
2. Lots of smaller systems, some of which previously had little or no version control, have chosen to adopt FCM.
3. Several systems with tens of developers now have the tools to support parallel development - this was very painful with their previous tools.
4. The Met Office Unified Model (MetUM), used for climate and weather prediction, has been able to introduce a new, much improved, directory structure and to make use of more Fortran 9X features - the previous tools had prevented this.
5. The MetUM has been able to introduce a 3 month release cycle since it adopted FCM - the previous code management and build tools made it much harder to create releases resulting in them taking up to 18 months.
6. The use of Trac means that code changes are now much better documented and, just as importantly, the documentation is much more accessible.
7. FCM now supports 50 different systems with over 230 total users, and the number is steadily increasing, so it must be doing something right.

One unexpected development is that many people have started to use Trac for other purposes, without either Subversion or FCM. As they have grown accustomed to its simple interface, they have started relying on it to manage activities and projects that are not code-based. Seeing people repurpose tools of their own volition is perhaps the most powerful proof that they find those tools worthwhile.

The Met Office has a number of collaborators [3] who are now adopting FCM so that they can use and develop the MetUM. FCM is also being used within the Met Office to manage several external models [4] and it is possible that FCM will be adopted more widely in the future by other users of these models.

[3] Including the Australian Bureau of Meteorology, the Commonwealth Scientific and Industrial Research Organisation, the South African Weather Service, and the Natural Environment Research Council.

[4] Such as the NEMO ocean model.

12 Conclusions

We regard FCM as a major success story at the UK Met Office. We attribute its success to a number of factors:

- Support for FCM came from both the grassroots and from management; we believe it would have been much less likely to succeed if either had been missing.
- The open source tools adopted have strong and growing development communities, reducing the risk of lack of support in years to come.
- The tools, especially Trac, have simple intuitive user interfaces. This reduces the learning curve and helps users to see immediate benefits.
- FCM encourages processes without enforcing them, which allows projects (rather than users) to migrate to new processes at their own pace. It also allows users to apply the tools to a much wider range of systems.
- FCM has concentrated on handling the common cases well, rather than trying to handle all possible cases. This simplifies the user interfaces, and encourages use of standard practices.
- The FCM team put a lot of effort up front into supporting the migration process, writing scripts to ease the transition, and developing extensive tutorial support.
- FCM is run as a service rather than a tool; ongoing support and expertise are available to all teams that adopt it.

FCM's developers are planning a number of improvements based on lessons learned, and expect that increased uptake by collaborators outside of the Met Office will lead to further changes beyond that. There are also some exciting new developments in the pipeline for Subversion and Trac which should benefit FCM users; as always, the team will have to be careful to balance the power of new features against the burden of learning how to use them, or find ways to wrap new features so that users get the functionality they care about most without being distracted by possibilities they don't need.

Finally, FCM is available for general use [3] under the Flexible Configuration Management License [2]. Although the Met Office is not able to provide support to outside users, the team is always happy to receive feedback, bug fixes, and enhancements from anyone who finds the system of use.

13 Acknowledgments

Our thanks to all the developers who have contributed to all the open source tools upon which FCM relies. The quality of some of the free tools available is simply amazing. Thanks also to the software system managers at the Met Office for their enthusiasm and patience whilst migrating to FCM which helped to ensure the success of the project, and to the other members of the FCM team at the Met Office: Matt Shin, who was responsible for writing most of the FCM code (currently approximately 15000 lines of Perl) and Jim Bolton, who managed the project and dealt with a lot of the migration issues.

References

1. Bugzilla. http://www.bugzilla.org.
2. Flexible configuration management license. http://www.metoffice.gov.uk/research/nwp/external/fcm/LICENSE.h
3. Flexible configuration management web site. http://www.metoffice.gov.uk/research/nwp/external/fcm.
4. Gnu make. http://www.gnu.org/software/make.
5. Sourceforge. http://www.sourceforge.net.
6. Subversion. http://subversion.tigris.org.
7. CVS. http://www.nongnu.org/cvs.
8. Trac. http://trac.edgewall.org.
9. xxdiff. http://furius.ca/xxdiff.
10. Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. Software development environments for scientific and engineering software: A series of case studies. In *Proceedings of the 29th ACM/IEEE International Conference on Software Engineering*, May 2007.
11. Paul F. Dubois. Maintaining correctness in scientific programs. *Computing in Science & Engineering*, 7(3):80–85, May 2005.
12. Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2002.
13. P. A. Miller. Recursive make considered harmful. *AUUGN Journal of AUUG Inc.*, 19(1):14–25, 1998.
14. C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2004.
15. Cynthia K. Riemenschneider, Bill C. Hardgrave, and Fred D. Davis. Explaining software developer acceptance of methodologies: A comparison of five theoretical models. *IEEE Transactions on Software Engineering*, 28(12):1135–1145, December 2002.
16. C. Spinellis, D.; Szyperski. How is open source affecting software development? *Software, IEEE*, 21(1):28–33, Jan-Feb 2004.
17. Greg V. Wilson. Where's the real bottleneck in scientific computing? *American Scientist*, 94(1), Jan-Feb 2006.