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

The UK Met Office's Flexible Configuration Management (FCM) system uses existing open source tools, adapted for use with high-performance scientific Fortran code, to help manage evolving code in its large-scale climate simulation and weather forecasting models. FCM has simplified the development process, improved team coordination, and reduced release cycles.

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. Because commercial software development tools and techniques are tailored to the needs of commercial developers, rather than those of computational scientists, it's 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 article describes how the UK Met Office escaped 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 lets scientific end users assemble the modules they need for particular simulations with much less effort than previously required. Just as importantly, it gives users confidence that the code they're 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.

1521-9615/08/$25.00 © 2008 IEEE
Copublished by the IEEE CS and the AIP

David Matthews
UK Met Office
Greg Wilson and Steve Easterbrook
University of Toronto

Tool Support for Developing Scientific Software

The large computational models scientists use for NWP and climate research have evolved continuously over decades. The current generation used at the UK Met Office contains 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,[1] 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, rerun experiments with precisely repeatable results, and 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.[2] Moore's law might have dramatically reduced the time it takes to run a scientific simulation, but it has not re-

duced 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 have been slow to adopt best-practice tools to support these activities. Jeffrey Carver and his colleagues identify several reasons for this.[3] 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 with tools developed by the software community is usually disappointing—for example, these tools don't cater to 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 aren't 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 because scientists can maintain and adapt them in-house. However, open source tools tend to come with higher adoption costs[4] 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, *process improvement* is used to identify and correct 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, hence reducing the amount of rework needed when mistakes are detected. But introducing process improvements without good supporting tools often fails because, as Robert Glass observed,[5] most new processes initially increase the programmers' burden. Conversely, the best tools in the world are of little use unless they are used properly—that is, 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 won't bring about 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 the developers' perceptions of whether their coworkers are also adopting the new process.[6] This circularity makes it hard to know which route to take to improve software tools and processes used in scientific software development.

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 there, as well as 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 the Concurrent Versions System (CVS; www.nongnu.org/cvs), most projects used tools specific to the Met Office. This meant that new staff or collaborators had 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 systems had powerful features, but also some serious deficiencies, which made none of them suitable for general use.

The best tools in the world are of little use unless they are used properly—that is, within the context of a good process.

For many years, the Met Office recognized its need for a new system 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 (including one of the authors of this article) assembled to tackle this

issue. Working closely with the scientists and systems administrators, they produced the Flexible Configuration Management (FCM) system.

A key to FCM's success has been the combination of a grass roots effort and management initiative. The team tasked with designing and implementing the system got to know each other and the scientists who would be the system's primary users 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. Although 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 leaders, to listen to their requirements and keep them onboard with the project. In the end, success came down to producing a system that was clearly better than any of the existing systems and ensuring that it had the support of the key decision makers.

Open Source Building Blocks

The year 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 (http://subversion.tigris.org),[7] an 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, several Met Office codes are used externally by universities and other meteorological centers, so 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.

The project team was initially concerned that Subversion had only just had its first major release. However, after spending time monitoring the Subversion mailing lists, the team gained confidence from the highly active developer community and the already large user base; such confidence would have been difficult in the first release of any commercial product. The Subversion team clearly hadn't 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 mission-critical source code and that help would be available should they run into problems. Subversion has since become successful, with an ever-expanding user base and a strong development team, which gives confidence that it won't become an orphan system.

The second key tool was an issue tracker, and again, the timing was good because February 2004 marked the first public release of Trac (http://trac.edgewall.org), a lightweight software project management portal. Like SourceForge (www.sourceforge.net) and other software project portals, Trac combines a wiki with an issue tracker and Subversion repository browser. Although it was (and is) not as mature or widely used as Subversion, even in 2004, it had the key features the FCM team felt it needed to coordinate work on various applications.

Just as importantly, Trac's interface is simple—much simpler, for example, than those of SourceForge or the widely used open source issue tracker Bugzilla (www.bugzilla.org). This was an important factor in gaining acceptance of the new system: not only did it lower the learning curve, it also meant that users saw the purpose and value of every single field (well, most fields) of every form they filled in (well, most forms).

Trac's relative immaturity was a concern. However, the issue tracker's availability is not essential for developers to do most of their work, so the FCM team felt comfortable accepting more of a risk. In any case, if developers encountered serious problems with Trac, the team knew 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 several 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 them to contribute in a small way to the community.

Configuration Management

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

- All changes to the system must be associated with an issue, or *ticket*, in Trac.
- All changes must be prepared in a version control branch.
- A reviewer must 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.

Subversion and Trac support this CM process effectively (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 change sets and to wiki pages for changes that developers need to document 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 hence to the documentation and discussion that led to the change.

It is important to note, however, that this process isn't mandatory: Subversion lets users make changes directly in the main code line and, unlike some commercial tools, doesn't 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, can make effective use of the same tools. In particular, users who are required to use FCM on bigger, more stable codes don't then have to 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 painfree because the developers could continue to use the same tools, just in a slightly different way.

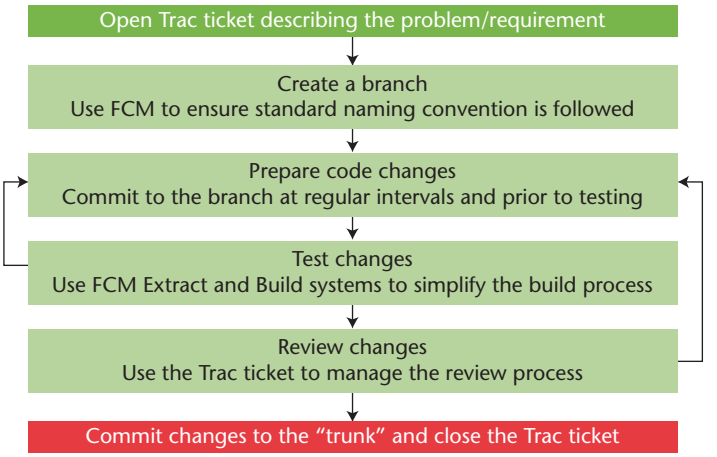

Figure 1. The FCM change process. Each change begins with the opening of a new ticket in Trac and the creation of a new branch on which changes can be prepared. The ticket is closed only after changes have been tested, reviewed, and committed to the trunk.

The FCM Tool

Subversion's flexibility has been a key to the FCM tool's general success. However, with this flexibility comes additional complexity in that some common commands can be tricky to master. To achieve widespread acceptance at the Met Office, the FCM team felt that the scientists needed a somewhat simpler interface. 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 that FCM

- enforces a particular repository structure and branch-naming convention,
- assumes working copies contain a single revision and repository location (no mixed working copies), and
- does not support partial commits.

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 a pure Subversion interface concern the processes of branching and merging, plus the use of xxdiff (http://furius.ca/xxdiff), a graphical difference and merge tool. These are key areas, where it was essential to simplify the interface to make it accessible to the scientific user base. However, it's 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.

Figure 2 gives some examples of common FCM commands, together with the equivalent Subver-

1. Create a branch
   Subversion command:
   svn copy -r 123 svn://server/repos/proj/trunk \
     svn://server/repos/proj/branches/dev/userid/r123_mybranch
   Equivalent FCM command:
   fcm branch -c -r 123 -n mybranch fcm:proj
2. Switch your working copy to a branch
   Subversion command:
   svn switch \
     svn://server/repos/proj/branches/dev/userid/r123_mybranch
   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/repos/proj/trunk@123 --new .
   ('graphic_diff' assumed to be suitable wrapper script)
   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/repos/proj/trunk
   Equivalent FCM command:
   fcm merge trunk

Figure 2. Example FCM commands. These show how FCM simplifies the `svn` commands by using a standard naming convention and repository structure.

sion commands. In each case, FCM simplifies the command by relying on a standard organization for the repository structure and standard naming conventions. For example, in Figure 2, 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.

The 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 large and can take considerable time to compile, so an efficient build system is essential.

FCM's build system is based on GNU Make (www.gnu.org/software/make), 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. 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—for example, causing Make to be overly sensitive to changes in the source code and hence greatly increasing compilation time.[8]

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 intermodule dependencies. Code must be compiled in the correct order, and, for incremental builds, changes to a source file must trigger recompilation of any dependent files. Where possible, FCM analyzes each source file's dependencies automatically. To do this, it requires

- one program unit (such as module, subroutine, or function) per file,
- all used routines defined within the Fortran module or in included interface files, and
- source code comments to identify other dependencies (such as on Fortran 77 or C code).

The first two requirements are good practices 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 hasn't 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 with code that didn't follow these rules. This was certain to be an ongoing issue because the Met Office must be able to integrate code from other organizations into its models with minimal effort (preferably without modifying their original source). FCM supports this by providing escape mechanisms so that users can define dependency information manually. Although 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 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 lets a build inherit source 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 the 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 code's behavior, altering subroutine interfaces and code dependencies. However, most build systems don't 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 they lose valuable time chasing down bugs that are actually mismatches between memory layout, loop optimization, or synchronization directives. FCM deals with this by letting users specify flags for the entire system (at the directory level) or for individual files and triggers a build of the appropriate scale when these flags change. It also has an optional preprocessing step prior to the dependency analysis in which changes to preprocessor flags trigger the appropriate recompilation. These features are particularly useful when inheriting from another build.

One final build 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 in which modifications to a commonly used routine can result in recompilation of almost the entire code, even though the subroutine interface hasn't 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. Furthermore, the interface files only get updated if the subroutine interface is changed, so that no unnecessary compilation is triggered.

The Extract System

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

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

Scientists at the Met Office often want to be able to pull together changes from different branches containing proposed scientific changes that 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 don't 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 that the build system requires, which means that users only ever need to deal with a single configuration file. Once this single file is set up appropriately, 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 must be able to run the extraction on desktop systems to include locally modified code before mirroring it to their supercomputer for compilation and execution.

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

Figure 3 gives an example of a simple extract configuration file for some Fortran code. Running the `fcm extract` command against this file would result in the code tree being extracted from

```
# 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
```

Figure 3. Example extract configuration file.

the Subversion repository and a build configuration file being generated. Running the `fcm build` command would then result in any program units found in the source code being compiled into executables. We should note some points about the configuration files:

- 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 doesn't follow the standard, FCM will need further information defined in the configuration file to help it do the right thing.
- No other Makefiles, or anything like them, clutter 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, and so forth) would require further entries in this file. However, Figure 3 illustrates the points that we can use a single file to configure the entire extract and build process and that the FCM system does a lot of the hard work.

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 the FCM team was trying to evolve existing practices rather than introducing entirely new concepts. In some ways, this made the task harder: the team wasn't just saying, "This is going to make your life so much easier," but "Please throw away the tools that you've been using happily for the last 10 years or more and start using these new unfamiliar ones instead." FCM's creators knew that persuading potential users to work their way through that period would be crucial to its success.

Therefore, the FCM team put a lot of effort into supporting the migration process—almost as much, in fact, as it put into building FCM. For example, the team wrote an import script that allowed all 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. The team also spent a lot of time documenting the system. Finally, it prepared an extensive tutorial that became the basis of training workshops that 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 brought code into FCM, the team took the opportunity to refactor it, sometimes extensively. Fixed-format Fortran was converted to free format, code headers were updated and standardized, and directory structures were redesigned. These kinds 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 FCM provided and were keen to migrate. In the remaining cases, however, migration to FCM was effectively forced because members of the new FCM team had been supporting the old tools. Judicious use of both carrot and stick allowed FCM to quickly reach critical mass within the Met Office's research areas; the first official deployment was November 2005 with most systems migrated by March 2006 and migration of the main NWP and climate model completed in November 2006.

FCM as a Service

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

One of the reasons this is manageable is that the team has automated many maintenance proce-

dures; it isn't much harder to manage 50 systems than it would be to manage five. The maintenance side of the service requires much less than one full-time staff member.

Evaluation

All the Met Office's key modeling systems have now used FCM for at least a year. The system is clearly a success, in part because most users are fairly indifferent to it; they can 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 that have been in place for many years. The FCM team has tried to listen to these users and 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 haven't yet attempted to assess the impact of FCM in quantitative terms, such as "X percent improvement in productivity," but there's plenty of anecdotal evidence of its success:

- Staff no longer must deal with lots of different tools; all key systems use the same tools and follow similar processes.
- Lots of smaller systems, some of which previously had little or no version control, have chosen to adopt FCM.
- Several systems with tens of developers now have the tools to support parallel development; this was very painful with previous tools.
- The Met Office Unified Model (MetUM), used for climate and weather prediction, has been able to introduce a new, improved directory structure and to make use of more Fortran 9X features. The previous tools prevented this.
- The MetUM has been able to introduce a three-month release cycle since it adopted FCM. The previous code management and build tools made it much harder to create releases, so they took up to 18 months.
- The use of Trac means that code changes are now much better documented and, just as importantly, the documentation is much more accessible.
- FCM now supports 50 different systems with more than 230 users, and this 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've grown accustomed to its simple interface, they've 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 several collaborators (including the Australian Bureau of Meteorology, the Commonwealth Scientific and Industrial Research Organization, the South African Weather Service, and the Natural Environment Research Council) 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, such as the NEMO ocean model, and it's possible that FCM will be adopted more widely in the future by other users of these models.

We regard FCM as a major success at the UK Met Office. We attribute its success to numerous 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, which reduce the learning curve and help users 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 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 the 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 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, however, 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.

FCM is available for general use (www.metoffice.gov.uk/research/nwp/external/fcm) under the Flexible Configuration Management License at www.metoffice.gov.uk/research/nwp/external/fcm/LICENSE.html. Although the Met Office isn't 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.

Acknowledgments

Our thanks to all the developers who contributed to 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 while migrating to FCM, which helped ensure the project's success, and to the other members of the FCM team at the Met Office: Matt Shin, who wrote most of the FCM code (currently approximately 15,000 lines of Perl), and Jim Bolton, who managed the project and dealt with a lot of the migration issues.

References

1. P.F. Dubois, "Maintaining Correctness in Scientific Programs," *Computing in Science & Eng.*, vol. 7, no. 3, 2005, pp. 80–85.
2. G.V. Wilson, "Where's the Real Bottleneck in Scientific Computing?" *Am. Scientist*, vol. 94, no. 1, 2006, pp. 5–6.
3. J.C. Carver et al., "Software Development Environments for Scientific and Engineering Software: A Series of Case Studies," *Proc. 29th ACM/IEEE Int'l Conf. Software Eng.*, IEEE CS Press, 2007, pp. 550–559.
4. C. Spinellis and D. Szyperski, "How Is Open Source Affecting Software Development?" *IEEE Software*, vol. 21, no. 1, 2004, pp. 28–33.
5. R.L. Glass, *Facts and Fallacies of Software Engineering*, Addison-Wesley, 2002.
6. C.K. Riemenschneider, B.C. Hardgrave, and F.D. Davis, "Explaining Software Developer Acceptance of Methodologies: A Comparison of Five Theoretical Models," *IEEE Trans. Software Eng.*, vol. 28, no. 12, 2002, pp. 1135–1145.
7. C.M. Pilato, B. Collins-Sussman, and B.W. Fitzpatrick, *Version Control with Subversion*, O'Reilly Media, 2004.
8. P.A. Miller, "Recursive Make Considered Harmful," *AU-UGN J. AUUG*, vol. 19, no. 1, 1998, pp. 14–25.

David Matthews *leads a small team within the Numerical Modelling group at the Met Office. His main interest is in providing researchers at the Met Office with the IT tools and systems they need to deliver world-class science. Matthews has a BSc in mathematics from Southampton University. Contact him at david.matthews@metoffice.gov.uk.*

Greg Wilson *is an assistant professor in Computer Science at the University of Toronto. His research interests include high-performance scientific computing, software engineering education, and lightweight software engineering tools. Wilson has a PhD in computer science from the University of Edinburgh. Contact him at gvwilson@cs.toronto.edu.*

Steve Easterbrook *is a professor of computer science at the University of Toronto. His research interests range from formal software systems modeling to the socio-cognitive aspects of team interaction, including topics such as multi-stakeholder requirements negotiation, model management, and reasoning with inconsistent information. Easterbrook has a PhD in computing from Imperial College in London. He is a member of the ACM. Contact him at sme@cs.toronto.edu.*