Chapter 6
Code Design and Quality Control

Steve Easterbrook

ESM codes have mostly been written directly by scientists, as experts in the various geoscientific domains relevant for these model. This contrasts with commercial forms of software development, where there is usually a clear division of labour between systems analysts, who determine what the system should do, and programmers, who are responsible for writing the code. Having scientists write their own code avoids many of the communication errors and misunderstandings common in commercial software practices, and fits well with exploratory and incremental nature of model development (Easterbrook and Johns 2009). Decisions about what to add to the model are tightly bound with exploration of scientific research questions, as it is hard to know what will be needed ahead of time. Furthermore, model development requires deep knowledge of the physical processes and the impact of different implementation choices, so that it is usually easier to write the code directly, than to explain to a programmer what is needed.

While this approach has been remarkably successful in building the first generations of general circulation models, it does not scale well. Understandably, scientists tend to focus on producing working code, and testing it for scientific validity, while postponing consideration of code quality issues, such as structure and readability of the code, portability, flexibility, modifiability and reusability. As ESM have become more complex, with more component models, more complex interactions between components, and more diverse user communities, these code quality issues become vital.

The current generation of ESM are significantly larger and more complex than their predecessors. For example the National Center for Atmospheric Research (NCAR) Community Earth System Model (CESM) and the UK Met Office's Unified

S. Easterbrook(✉)
University of Toronto,
40 St George Street,
Toronto, ON M5S 2E4 Canada
e-mail: sme@cs.toronto.edu

L. Bonaventura et al., *Earth System Modelling – Volume 2*,
SpringerBriefs in Earth System Sciences, DOI: 10.1007/978-3-642-23831-4_6,

Model (UM) are each close to million lines of code, a tenfold increase in size over about a fifteen year period.

Accordingly, in the past few years, ESM groups have created new specialist roles for *scientific programmers* or *software engineers* who take on more responsibility for maintaining the software infrastructure and ensuring code quality, while the scientists remain responsible for developing new science code. This approach does not obviate the need for domain expertise, and most of the people recruited to these engineering roles have advanced training (typically PhDs) in earth sciences.

Along with the specialization of roles, there is increasingly a distinction between different types of code in the models, as *infrastructure code* (shared libraries, I/O, couplers, configuration scripts, etc.) is increasingly separated from the *science code*.

This trend represents a move away from ESM development as a craft, and towards a more disciplined engineering approach. However, it does not change the fact that model development is fundamentally an exploratory, incremental process, with very uncertain outcomes. This exploratory nature means that many standard software engineering methodologies are a poor fit. However, it also means that care is needed to ensure that the code is designed to be flexible and to support frequent change.

In this section we explore some of the principles of code design and quality control, and the tools that have been adopted by the ESM community to help apply them.

6.1 Design for Sustainability

Design for sustainability is crucial because ESM have very long useful lives (spanning decades of research), and will undergo many ports to new hardware and new compilers during that time. In addition, the code can be expected to grow steadily over the life of the model, as new physical processes and parameterizations are added. The code will also be used to support many different model configurations, and the submodels (e.g. ocean, atmosphere, ice,...) may each have an independent existence from the coupled model; they may be used in several different coupled models built at different research labs, as well as in different stand-alone configurations. The need to support this variety of uses of the code, with multiple sources of code change, means the code needs to be designed with this flexibility built in.

Writing code is relatively easy; writing code that can be read, understood, validated and modified by a diverse group of people in a variety of contexts is much harder. Different coders tend to have very different coding styles, due to choice of code layout, choice of naming schemes, preferences for certain types of programming construct, and so on. Where programmers have contributed code to a model without adhering to a standard code style, it's often easy to see whose code is whose, because of huge differences in style. But this then leads to inconsistencies and clashes of style when people then modify each other's code, causing a loss of clarity and reduced readability.

In ESM, this problem is compounded by the complexity of the coupling. While the conceptual architecture of a climate model leads to standard organizations of

the code base (for example, different physical processes in different modules), the impact of code changes in one module often requires corresponding code changes throughout the model. This means code contributed from different communities of scientists cannot easily be isolated. For this reason, code contributed from diverse participants often needs to be re-written to conform to a standard program style, before accepting it into a reference version of the model.

Clean code also minimizes the need for other types of documentation—the code should be self-documenting enough that another scientist armed with a description of the physics (e.g. a set of equations in a published paper) and the commented code should be able to understand how the code works. Documentation that is separate from the code (e.g. design manuals) can be useful for explaining the overall conceptual design of a model, but is very poor for capturing detailed programming decisions, as such documents get out of date quickly. This means that stand-alone documentation rarely matches what is in the code base. Again, to avoid this problem, clear self-documenting code is essential.

Designing for code sustainability also supports the processes of determining whether the science is implemented correctly (and this can be hard to tell). For example, a common task is the need to diagnose biases in the simulations and to understand surprising results. When these are due to coding errors, inspection of the code is the easiest way to track them down, but only if the code is readable. When they are due to incorrect understanding of the science, the challenge is to make sense of why the model behaves as it does. Indeed, model validation is often more a process of trying to understand what the model is doing in various runs, rather than testing that it produces a particular result.

Finally, design for sustainability should help to identify opportunities for optimisation. In Chap. 7, a number of example code optimisations are shown. Many of these can be thought of as design patterns Markus (2006); Decyk and Gardner (2008); applying them in a model depends on the ability to read the code, recognize when one of these patterns occurs, and whether the code is already optimized for the target architecture. As these optimisation decisions have to be re-visited each time the code is ported to another platform, clear, well-commented code is vital to facilitate optimisation.

6.2 Software Coding Standards

The simplest step in supporting the development of sustainable code is to establish a project-wide style guide, and adhere to it. It is tempting to assume that the code can be cleaned up later, especially when experimenting with new ideas that might never get widely adopted. However, in practice such clean up is never attended to. It is better to write clear code in the first place than to expect to improve it later, at which point it will be harder to figure out how it works.

Many modeling teams have developed their own style guides (see for example the Nucleus for European Modelling of the Ocean (NEMO) Style Guide NEMO

System Team (2002), the Community Climate System Model (CCSM) developers' guide Kauffman et al. (2001)) and Andews et al. (1996). A project style guide would typically give recommendations for:

- *Language version*. For example, the standard might call for strict adherence to the Fortran F90 standard, to facilitate use of multiple compilers and avoid obsolete constructs.
- *Language features that should be avoided*, because they reduce readability, performance, portability, modifiability, etc. For example, a style guide might prohibit the use of dynamic memory allocation.
- *Use of pre-processor statements*, including how to make them clearly distinct from program code.
- *Naming conventions* for variables, parameters, modules, and so on.
- *Use of standard scientific units*, and a guide to definition and use of constants.
- *Code layout*, with recommendations for indentation, line length, and use of continuation lines. Such rules make the code easier to read.
- *Use of comments*, with recommendations for how to lay out comments, and when comments are needed.
- *Rules for allocation of routines to files*. Including closely related routines in the same file improve readability, while separating unrelated routines into different files makes it easier to manage subsequent changes to these routines.
- *Rules for defining packages* to structure the code base, setting out principles for deciding what to put into a package.

Note that most style guides are *recommendations* for good practice, rather than rigid rules, as occasionally a case can be made for deviating from the guide. The key point is that such exceptions should be rare: if they are not, then either the modelers haven't understood the point of a style guide, or the style guide might need to be re-designed.

In addition to style guides, various tools exist to automatically check that the code conforms to good practice, and to look for potential programming errors. The can be classed as *style checkers* or *static analysis tools*, although the main difference is a matter of degree—style checkers focus on syntactic features of the code, while static analysis tools dig deeper into the structure of the code, for example analyzing the static call structure of the program. Both depend on a set of heuristics for detecting symptoms of typically coding mistakes. Tools for Fortran code tend to lag behind those available for other languages, because the dominant market for such tools are the commercial and open source communities, rather than scientific programming. Notable examples of such tools for Fortran include Flint FLINT Manual (1994) and FTNchek Tool (2004). Flint for example, identifies four types of error: syntax issues, interface issues, data usage issues, and problems involving use of implicit typing. Flint also classifies each issue as either: error, warning, or FYI. The disadvantage of such tools is that they tend to report many false positives, which then places a burden on the programmer to interpret the reports carefully.

Static checkers are not currently used widely in the ESM community, although some have argued that static analysis and error removal is a precondition for

developing correct scientific code (Hatton 1997). However, a recent study, described below in Sect. 6.6, indicates that other quality control practices used for ESM are relatively successful at removing coding errors (Pipitone 2010), which suggests that the question remains open on whether better static analysis tools could detect errors more efficiently.

6.3 Version Control

Most of the challenges in supporting code development come from the need to coordinate the changes to the code made by a large community of modelers. Version control tools (e.g. CVS Free Software Foundation (1998), Subversion Collins-Sussman et al. (2008)) provide the basic support for this, allowing different team members to check out the most current version of the entire model code from a shared repository, edit it locally, and then check their modifications back into the repository when complete. Each new code check-in creates a new version, and all past versions are archived, so that it's possible to roll back to an earlier version if necessary.

Version control is essential when multiple developers are working on the code in parallel. Coordination is managed at the check-in point, by merging the new changes with any other changes that were checked by other developers. However, conflicts (where the same lines of code were edited) still have to be handled manually, which means some coordination between team members is needed to avoid frequent conflicts. An important weakness of these tools is that conflicts are detected only by the static location of the change within the code—i.e. where multiple edits modify the same lines of code. In ESM, complex dependencies exist between code changes in different modules, which means other coordination mechanisms are needed to identify potential interactions between code changes.

Some version management tools (e.g. Subversion) support a variety of code development workflows, by allowing developers to create branches from the main trunk in the version history. Creating a branch allows developers to keep checking in experimental modifications to the code they are working on, without affecting people who are working with code from other branches. Typically, the *trunk* is reserved for a stable, fully tested version of the model, while *branches* from the trunk are created for each new experimental modification of the model. Once all the changes on a branch have been thoroughly tested and are ready to share with the full community, the branch is folded back into the trunk. Each user can choose whether to check out code from an existing branch (because they want access to an experimental version of the model), create their own branch (to start a new line of development), or check out from the trunk (which is usually the stable, reference model). They can also check any older version of the model, which is useful when older model runs need to be repeated. The NEMO Developers Guide gives one set of recommended practices for deciding when to create new branches and when to merge them back into the trunk (NEMO System Team 2010).

Branching version control provides an important foundation for managing community contributions to ESM. Code contributions from scientists working at other labs can be isolated on a branch, until they have been reviewed for conformance to project standards, and tested across different model configurations. Meanwhile, the broader community can still access such experimental versions of the code if they need them. Furthermore, different experimental configurations of the entire model can be created and disseminated to the community through the use of tag releases, by tagging the appropriate version of each code module with a unique label. Carter and Matthews (2011, Vol. 5 of this series) suggests a set of specific project team-member roles for effective code management.

Good version control tools are an important weapon in preventing unnecessary code forking. Code forks occur when two different versions of a model diverge so much that it becomes to difficult to recombine them. This happens occasionally in open source projects, where a community working on a project disagree on future directions, and end up fragmenting, each with their own modified copy of the original codebase. Such forks are nearly always a bad idea, as they divide the community and dilute the effort (Fogel 2006).

Code forking in ESM has the same drawbacks as in open source software. As an example, the UK Met Office originally adopted the modular ocean model MOM from the Geophysical Fluid Dynamics Laboratory (GFDL) to use in their coupled climate system model. A number of platform specific changes were made to MOM in adapting it to work in the UK Met Office system, in effect creating a code fork. The effect was that the Met Office gained a new ocean model, but did not gain access to the subsequent development of that model, so eventually this ocean model fell behind the state-of-the-art.

Version control tools cannot remove the tensions between the needs of different subcommunities that often lead to the desire to fork. Such tensions occur especially when a particular component model (for example the ocean model, NEMO) is used in several different ESM managed at different research labs. Preventing a code fork requires negotiation and compromises between the communities that share the models. However, working to prevent a fork from occurring helps to maintain the long term utility of a model, and to ensure it continues to receive contributions from diverse groups of experts.

6.4 Other Tools to Support Coding

Several other types of tools are useful to support other aspects of code development:

Bug tracking tools provide a central database to record details of error reports and steps taken to resolve them (see for example, Trac Project (2003)). Bug trackers keep track not just of the problem reports, but information on who has worked on the problem, whether it has been resolved, and which version(s) of the code the fixes were included in. They also can become useful as an electronic discussion forum

for comments about how to resolve a problem, and the impacts it might have. While many ESM teams have adopted tools such as Trac, few have fully integrated it into their practices, typically using them only to record major bugs. This misses many of the benefits of these tools, in that they act as a long term memory for all changes to the code, no matter how minor, and hence provide a valuable form of documentation for understanding past design decisions. In commercial software practices, these tools are being used increasingly as project-wide TO-DO lists, by entering not just bugs, but all planned changes (e.g. new features, design changes, optimisations, etc). As the tool provides facilities to view the list of to-do items in various ways, it becomes a central mechanism for coordinating the entire project and monitoring progress.

Test automation tools provide a framework for repeated test execution, making it easier to run common tests more often, and to keep track of test failures. Example off-the-shelf testing tools include Buildbot (Warner 2010) and Cruisecontrol (The CruiseControl project 2010); however, for ESM these tools generally need additional custom scripts to handle the complexities of code extraction, model configuration and execution (see for example, the UK Met Office's Flexible Configuration Management (FCM) tool (Matthews et al. 2008)). Test automation tools can be configured to run a standard suite of tests on a regular schedule, for example every time code is checked into the repository. These tools also introduce the idea of a *project dashboard*, which provides a summary of successful and failed tests, and can improve awareness across a team of what other people are doing. Test automation tools support the process of *continuous integration*, the practice of integrating code changes early and often, by building and re-testing the entire system frequently, for example, at the end of each day. The idea is that the sooner errors are detected the better, and doing it in small increments makes it easier to pinpoint errors. This style of continuous integration is standard practice for ESM, since most code changes can only be properly evaluated by running coupled model simulations anyway; however the use of automated tools to support continuous integration is not yet widespread.

Symbolic debuggers support the identification of bugs by executing the code under user control, with the ability to monitor the contents of variables and data structures, execute the program line-by-line or to a specific breakpoint, and to visualize the execution path. For example, TotalView Technologies (2007) provides features specifically needed for ESM, including tight integration with a variety of Fortran compilers and supercomputer architectures, the ability to handle MPI calls, and massively parallel computations. However, use of symbolic debuggers is sporadic in the ESM community, although those who do use them regularly often report them to be the single most valuable tool they use.

Documentation tools support the automated creation of program documentation directly from the code itself, using especially structured comments. Examples include Doxygen (van Heesch 2007). Such tools close the gap that often occurs between the program code and its documentation, by embedding the documentation directly into the code via comments. Doxygen then automatically extracts these comments, along with various views of the structure of the code, so that documentation can be regenerated automatically whenever the code changes.

Code Review Tools, such as ReviewBoard (Hammond and Trowbridge 2010), support a process of peer-review for code, providing a platform to collect comments sections of code (see Sect. 6.5).

6.5 Code Reviews

Software engineering techniques to check for code correctness can generally be divided into three groups: (1) Testing; (2) Static analysis (e.g. the style checkers and static analyzers discussed in Sect. 6.2; and (3) Systematic Code Review. Of these, systematic code review has been shown in a number of studies to be more effective at finding bugs than any other technique. None of these techniques is 100% effective, but data from the software industry indicates that code reviews are typically 50–60% effective at detecting errors, while unit and module testing are around 20–40% effective, and system testing is 30–50% effective (McConnell 2004). However, each technique tends to find different kinds of error, so when a variety of techniques are used together in a systematic way, software development teams can produce code with extremely low defect densities.

Code reviews work because they reinforce the developers' familiarity with the code. Code reviews are often conducted in small teams (typically 5–6 reviewers), reviewing a section of code in detail by walking through it line-by-line, or by working through a checklist of common error types. Such team-based reviews have some intangible benefits, including reinforcing code style and code quality practices, exposing junior team members to the expertise of their senior colleagues, and fostering a sense of team responsibility for the code. Code review also reduces incidence of repeated mistakes because programmers are more likely to learn from their errors. Finally, code review is often faster: while it can take many hours of work to pinpoint the cause of a failed test, in code review, errors are discovered directly.

Team-based code reviews can be hard to apply for ESM, however, because the coding effort is often dispersed: scientists who contribute code may do so infrequently, or in bursts of work between other activities, and are often distributed across multiple research labs. This makes it challenging to bring together review teams when needed. Instead, ESM developers tend to rely more on informal code reviews, involving one or two experts. One approach, used at the UK Met Office, is to designate *code owners* for each section of code, usually the more senior scientists in the lab. Code owners are responsible for two separate review steps: (1) a science review, in which proposed changes are discussed for their impact on the model, and for potential interactions between different changes; and (2) a code review once the changes have been completed, to assess readiness for inclusion in an upcoming release of the model.

A related practice is *pair programming*. Pair programming was first made popular in agile software development methods such as Extreme Programming (Beck 1999). For this, two programmers work side-by-side on their code modifications, observing one another as they edit the code. In some cases, they share a single workstation so that only one is editing at any one time, and the other is acting as a "co-pilot". This practice

tends to slow down the initial production of code, but yields much higher quality code with fewer errors; this then leads to savings later on in reduced testing and debugging cycles. It also has many of the intangible benefits described above for code reviews. Advocates of extreme programming insist all coding must be done this way. However, in practice many organisations move back and forth between individual and pair programming, depending on the nature of each coding task. Some ESM teams report that pair programming is very effective for debugging, especially for complex numerical codes, although none (to our knowledge) have adopted it as routine practice.

6.6 Verification and Validation for ESM

Verification and Validation for ESM is hard, because running the models is an expensive proposition (a fully coupled simulation run can take weeks to complete), and because there is rarely a 'correct' result—expert judgment is needed to assess the model outputs (Carver et al. 2007).

However, it is helpful to distinguish between *verification* and *validation*, because the former can often be automated, while the latter cannot. Verification tests are objective tests of correctness. These include basic tests (usually applied after each code change) that the model will compile and run without crashing in each of its standard configurations, that a run can be stopped and restarted from the restart files without affecting the results, and that identical results are obtained when the model is run using different processor layouts. Verification would also include the built-in tests for conservation of mass and energy over the global system on very long simulation runs. In contrast, validation refers to science tests, where subjective judgment is needed. These include tests that the model simulates a realistic, stable climate, given stable forcings, that it matches the trends seen in observational data when subjected to historically accurate forcings, and that the means and variations (e.g. seasonal cycles) are realistic for the main climate variables (Phillips et al. 2004).

While there is an extensive literature on the philosophical status of model validation in computational sciences (see for example, Oreskes et al. (1994), Sterman (1994), Randall (1997), and Stehr (2001)), much of it bears very little relation to practical techniques for ESM validation and very little has been written on practical testing techniques for ESM. In practice, testing strategies rely on a hierarchy of standard tests, starting with the simpler ones, and building up to the most sophisticated.

Pope and Davies (2002) give one such sequence for testing atmosphere models:

- Simplified tests—e.g. reduce 3D equations of motion to 2D horizontal flow (e.g. a shallow water testbed). This is especially useful if the reduction has an analytical solution, or if a reference solution is available. It also permits assessment of relative accuracy and stability over a wide parameter space, and hence is especially useful when developing new numerical routines.
- Dynamical core tests—test for numerical convergence of the dynamics with physical parameterizations replaced by a simplified physics model (e.g. no topography, no seasonal or diurnal cycle, simplified radiation).

- Single-column tests—allows testing of individual physical parameterizations separately from the rest of the model. A single column of data is used, with horizontal forcing prescribed from observations or from idealized profiles. This is useful for understanding a new parameterization, and for comparing interaction between several parameterizations, but doesn't cover interaction with large-scale dynamics, nor interaction with adjacent grid points. This type of test also depends on availability of observational datasets.
- Idealized aquaplanet—test the fully coupled atmosphere–ocean model, but with idealized sea-surface temperatures at all grid points. This allows for testing of numerical convergence in the absence of complications of orography and coastal effects.
- Uncoupled model components tested against realistic climate regimes—test each model component in stand-alone mode, with a prescribed set of forcings. For example, test the atmosphere on its own, with prescribed sea surface temperatures, sea-ice boundary conditions, solar forcings, and ozone distribution. Statistical tests are then applied to check for realistic mean climate and variability.
- Double-call tests. Run the full coupled model, and test a new scheme by calling both the old and new scheme at each timestep, but with the new scheme's outputs not fed back into the model. This allows assessment of the performance of new scheme in comparison with older schemes.
- Spin-up tests. Run the full ESM for just a few days of simulation (typically between 1 and 5 days), starting from an observed state. Such tests are cheap enough to be run many times, sampling across the initial state uncertainty. Then the average of a large number of such tests can be analyzed (Pope and Davies (2002) suggest that 60 is enough for statistical significance). This allows the results from different schemes to be compared, to explore differences in short term tendencies.

Whenever a code change is made to an ESM, in principle, an extensive set of simulation runs is needed to assess whether the change has a noticeable impact on the climatology of the model. This in turn requires a subjective judgment for whether minor variations constitute acceptable variations, or whether they add up to a significantly different climatology.

Because this testing is so expensive, a standard shortcut is to require *exact reproducibility* for minor changes, which can then be tested quickly through the use of *bit comparison tests*. These are automated checks over a short run (e.g. a few days of simulation time) that the outputs or restart files of two different model configurations are identical down to the least significant bits. This is useful to check that a change did not break anything it should not, but requires that each change can be "turned off" (e.g. via run-time switches) to ensure that previous experiments can be reproduced. Bit comparison tests can also check that different configurations give identical results. In effect, bit reproducibility over a short run is a proxy for testing that two different versions of the model will give the same climate over a long run. It's much faster than testing the full simulations, and it catches most (but not all) errors that would affect the model climatology.

Bit comparison tests do have a number of drawbacks, however, in that they restrict the kinds of change that can be made to the model. Occasionally, bit reproducibility cannot be guaranteed from one version of the model to another, for example when there is a change of compiler, change of hardware, a code refactoring, or almost any kind of code optimisation. The decision about whether to insist on bit reproducibility, or whether to allow it to be broken from one version of the model to the next, is a difficult trade-off between flexibility and ease of testing.

A number of simple practices can be used to help improve code sustainability and remove coding errors. These include running the code through multiple compilers, which is effective because different compilers give warnings about different language features, and some allow poor or ambiguous code which others will report. It's better to identify and remove such problems when they are first inserted, rather than discover later on that it will takes months of work to port the code to a new compiler.

Building conservation tests directly into the code also helps. These would typically be part of the coupler, and can check the global mass balance for carbon, water, salt, atmospheric aerosols, and so on. For example, the coupler needs to check that water flowing from rivers enters the ocean; that the total mass of carbon is conserved as it cycles through atmosphere, oceans, ice, vegetation, and so on. Individual component models sometimes neglect such checks, as the balance isn't necessarily conserved in a single component. However, for long runs of coupled models, such conservation tests are important.

Another useful strategy is to develop a verification toolkit for each model component, and for the entire coupled system. These contain a series of standard tests which users of the model can run themselves, on their own platforms, to confirm that the model behaves in the way it should in the local computation environment. They also provide the users with a basic set of tests for local code modifications made for a specific experiment. This practice can help to overcome the tendency of model users to test only the specific physical process they are interested in, while assuming that the rest of the model is performing correctly.

6.7 Model Intercomparisons

During development of model components, informal comparisons with models developed by other research groups can often lead to insights in how to improve the model, and also as a method for confirming and identifying suspected coding errors. But more importantly, over the last two decades, model intercomparisons have come to play a critical role in improving the quality of ESM through a series of formally organised Model Intercomparison Projects (MIPs).

In the early days, these projects focussed on comparisons of the individual components of ESM, for example, the Atmosphere Model Intercomparison Project (AMIP), which began in 1990 (Gates 1992). But by the time of the Intergovernmental Panel on Climate Change (IPCC) second assessment report, there was a widespread recognition that a more systematic comparison of coupled models was needed, which led

to the establishment of the Coupled Model Intercomparison Projects (CMIP), which now play a central role in the IPCC assessment process (Meehl et al. 2000). For example, CMIP3, which was organized for the fourth IPCC assessment, involved a massive effort by 17 modeling groups from 12 countries with 24 models (Meehl et al. 2007). As of September 2010, the list of Model Intercomparison Projects (MIP) maintained by the World Climate Research Program included 44 different model intercomparison projects (Pirani 2010).

Model Intercomparison Projects bring a number of important benefits to the modeling community. Most obviously, they bring the community together with a common purpose, and hence increase awareness and collaboration between different labs. More importantly, they require the participants to reach a consensus on a standard set of model scenarios, which often entails some deep thinking about what the models ought to be able to do. Likewise, they require the participants to define a set of standard evaluation criteria, which then act as benchmarks for comparing model skill. Finally, they also produce a consistent body of data representing a large ensemble of model runs, which is then available for the broader community to analyze.

The benefits of these MIPs are consistent with reports of software benchmarking efforts in other research areas. For example, Sim et al. (2003) report that when a research community that builds software tools come together to create benchmarks, they frequently experience a leap forward in research progress, arising largely from the insights gained from the process of reaching consensus on the scenarios and evaluation criteria to be used in the benchmark. However, the definition of precise evaluation criteria is an important part of the benchmark—without this, the intercomparison project can become unfocussed, with uncertain outcomes and without the huge leap forward in progress (Bueler 2008).

Another form of model intercomparison is the use of *model ensembles* (Collins 2007), which increasingly provide a more robust prediction system than single models runs, but which also play an important role in model validation:

- Multi-model ensembles—to compare models developed at different labs on a common scenario.
- Multi-model ensembles using variants of a single model—to compare different schemes for parts of the model, e.g. different radiation schemes.
- Perturbed physics ensembles—to explore probabilities of different outcomes, in response to systematically varying physical parameters in a single model.
- Varied initial conditions within a single model—to test the robustness of the model, and to better quantify probabilities for predicted climate change signals.

6.8 Assessments of Model Quality

Experiments performed as part of these model intercomparison projects show that coupled climate system models have been steadily improving in their skill at reproducing observed climate states (Reichler and Kim 2008). However, the spread between models is not obviously reducing, which leads to suggestions that, in

assessment exercises, some of the poorer models ought to be downweighted, begging the question about what evaluation criteria might be used to determine suitable weights (Knutti 2010).

In constrast to these studies of model skill, very few studies have been done of the *software* quality of these models. Informal discussions with modeling groups yield plenty of anecdotal evidence that, despite its scientific skill, the model code is usually poorly structured, hard to read, and hard to modify.

One recent study Pipitone (2010) attempted to assess the software quality of five leading ESM, including a detailed measurement of software defect density over multiple versions of three of the models. The results indicated that the models had relatively low defect densities by software industry standards, with results below 3.5 post-release defects per thousand lines of code, a level regarded as 'very high quality' by industry standards. However, such metrics might not apply very well to ESM. For these studies, a 'defect' is usually taken to be an error reported in the project bug tracking database and subsequently fixed in the code. In recognition that ESM projects don't always make systematic use of their bug tracking databases, Pipitone also applied an alternative measure, based on assessing the nature of all subsequent changes to the code after a model release. This showed slightly higher defect densities, but still within the same range of 'very high quality'.

This study offers both good news and bad news for the ESM community. The good news is that the existing testing and model validation processes used for the major ESM projects appear to be effective at eliminating coding errors relatively quickly, so that stable release versions of the models are relatively free of software defects. However, the bad news is that it is equally plausible that remaining latent errors in the software are particularly hard to find, and are discovered only rarely. Such errors allow the model to produce a realistic climate, but mean the code is not doing quite what the modellers think it is. In some cases, such errors are detected, and may be left unaddressed on the basis that they are indistinguishable from the known approximations in the model's algorithms. However, no systematic method exists for determining how often such errors remain undetected.

References

Andrews P, Cats G, Dent D, Gertz M, Ricard JL (1996) European standards for writing and documenting exchangeable Fortran 90 code. http://nsipp.gsfc.nasa.gov/infra/eurorules.html

Beck K (1999) Extreme programming explained: embrace change. Addison-Wesley, Boston

Bueler E (2008) Lessons from the short history of ice sheet model intercomparison. The Cryosphere Discussions 2:399–412. http://www.the-cryosphere-discuss.net/2/399/2008/

Carter M, Matthews D (2011) Configuration management and version control in earth system modelling. In: Ford R, Riley G, Budich R, Redler R (eds) Earth system modelling workflow putting it all together, vol 5. Springer, Heidelberg, pp 11–22 (in preparation)

Carver J, Kendall R, Squires S, Post D (2007) Software development environments for scientific and engineering software: a series of case studies. In: 29th international conference on software engineering (ICSE'07), pp 550–559

Collins M (2007) Ensembles and probabilities: a new era in the prediction of climate change. Philos Trans R Soc 365(1857):1957–1970
Collins-Sussman B, Fitzpatrick BW, Pilato CM (2008) Version control with subversion. O'Reilly Media, Cambridge
Decyk VK, Gardner HJ (2008) Object-oriented design patterns in Fortran 90/95. Comput Phys Commun 178(8):611–620
Easterbrook SM, Johns TC (2009) Engineering the software for understanding climate change. Comput Sci Eng 11:65–74
FLINT Manual (1994) User's manual, FORTRAN-lint source code analyzer. http://www.fnal.gov/docs/products/flint/manual.txt
Fogel K (2006) Producing open source software: how to run a successful free software project. O'Reilly Media, Cambridge
Free Software Foundation (1998) CVS—Concurrent Versions System. http://www.nongnu.org/cvs/
FTNchek Tool (2004) ftnchek static analyzer for Fortran. http://www.dsm.fordham.edu/ftnchek/
Gates WL (1992) AMIP: The Atmospheric Model Intercomparison Project. Bull Am Meteorol Soc 73(12):1962–1970
Hammond C, Trowbridge D (2010) ReviewBoard: take the pain out of code review. http://www.reviewboard.org/
Hatton L (1997) The T experiments: errors in scientific software. IEEE Comput Sci Eng 4(2):27–38
Kauffman B, Bettge T, Buja L, Craig T, DeLuca C, Eaton B, Hecht M, Kluzek E, Rosinski J, Vertenstein M (2001) Chapter 6, coding conventions. In: Community climate system model software developer's guide, NCAR. http://www.ccsm.ucar.edu/working_groups/Software/dev_guide/dev_guide/node7.html
Knutti R (2010) The end of model democracy? Climatic Change :1–10
Markus A (2006) Design patterns and Fortran 90/95. SIGPLAN Fortran Forum 25(1):13–29
Matthews D, Wilson GV, Easterbrook SM (2008) Configuration management for large-scale scientific computing at the UK met office. Comput Sci Eng 10(6):56–65
McConnell S (2004) Code complete. Microsoft Press, Redmond
Meehl GA, Boer GJ, Covey C, Latif M, Stouffer RJ (2000) The Coupled Model Intercomparison Project (CMIP). Bull Am Meteorol Soc 81(2):313–318
Meehl GA, Covey C, Taylor KE, Delworth T, Stouffer RJ, Latif M, McAvaney B, Mitchell JFB (2007) The WCRP CMIP3 multimodel dataset: a new era in climate change research. Bull Am Meteorol Soc 88(9):1383–1394
NEMO System Team (2002) FORTRAN coding standard in OPA system. http://www.nemo-ocean.eu/content/download/250/1629/file/coding_rules_OPA9.pdf
NEMO System Team (2010) NEMO: good practices, version 2.1. http://www.nemo-ocean.eu/content/download/11081/55387/file/NEMO.good-practicesV2.1.pdf
Oreskes N, Shrader-Frechette K, Belitz K (1994) Verification, validation, and confirmation of numerical models in the earth sciences. Science 263(5147):641–646
Phillips T, Potter G, Williamson D, Cederwall R, Boyle JS, Fiorino M, Hnilo J, Olson J, Xie S, Yio J (2004) Evaluating parameterizations in general circulation models—climate simulation meets weather prediction. Bull Am Meteorol Soc 85:1903–1947
Pipitone J (2010) On the software quality of climate models. Master's thesis, Department of Computer Science, University of Toronto
Pirani A (2010) Catalogue of model intercomparison projects. http://www.clivar.org/organization/wgcm/projects.php
Pope V, Davies T (2002) Testing and evaluating atmospheric climate models. Comput Sci Eng 4(5):64–69
Randall D, Wielicki B (1997) Measurements, models, and hypotheses in the atmospheric sciences. Bull Am Meteorol Soc 78(3):399–406
Reichler T, Kim J (2008) How well do coupled models simulate today's climate?. Bull Am Meteorol Soc 89(3):303–311

Sim S, Easterbrook S, Holt R (2003) Using benchmarking to advance research: a challenge to software engineering. In: 25th IEEE international conference on software engineering (ICSE'03), pp 74–83

Stehr N Models as focusing tools: linking nature and the social world. In: Storch H, Flöser G (eds) Models in environmental research. Springer, New York (2001)

Sterman J (1994) The meaning of models. Science 264(5157):329–330

The CruiseControl Project (2010) The CruiseControl continuous integration tool. http://cruisecontrol.sourceforge.net/

The Trac Project (2003) The Trac user and administration guide. http://trac.edgewall.org/wiki/TracGuide

TotalView Technologies (2007) Acomprehensive debugging solution for demanding multi-core applications. http://www.totalviewtech.com/pdf/TotalViewDebug.pdf

van Heesch D (2007) Generate documentation from source code. http://www.stack.nl/dimitri/doxygen/

Warner B (2010) BuildBot manual 0.8.1. http://buildbot.net/trac