Geosci. Model Dev., 5, 1009–1022, 2012
www.geosci-model-dev.net/5/1009/2012/
doi:10.5194/gmd-5-1009-2012

Geoscientific Model Development

Assessing climate model software quality: a defect density analysis of three models

J. Pipitone and S. Easterbrook

Department of Computer Science, University of Toronto, Canada

Correspondence to: J. Pipitone (jon.pipitone@utoronto.ca)

Received: 1 January 2012 – Published in Geosci. Model Dev. Discuss.: 15 February 2012
Revised: 11 June 2012 – Accepted: 27 June 2012 – Published: 9 August 2012

Abstract. A climate model is an executable theory of the climate; the model encapsulates climatological theories in software so that they can be simulated and their implications investigated. Thus, in order to trust a climate model, one must trust that the software it is built from is built correctly. Our study explores the nature of software quality in the context of climate modelling. We performed an analysis of defect reports and defect fixes in several versions of leading global climate models by collecting defect data from bug tracking systems and version control repository comments. We found that the climate models all have very low defect densities compared to well-known, similarly sized open-source projects. We discuss the implications of our findings for the assessment of climate model software trustworthiness.

1 Introduction

In this paper we report on our investigation into the software quality of climate models. A study by Easterbrook and Johns (2009) of the software development practices at the UK Met Office Hadley Centre estimates an extremely low defect density for the climate model produced there, which suggests an extraordinary level of software quality. Our purpose in this study is to conduct a rigorous defect density analysis across several climate models to confirm whether this high level of quality holds, and whether it is true of other models.

Defect density measures the number of problems fixed by the developers of the software, normalised by the size of the body of code. We chose defect density as our indicator of quality because it is well-known and widely used across the software industry as a rough measure of quality, and because of its ease of comparison with published statistics. Additionally, the measure is general and does not rely on many assumptions about how software quality should be measured, other than the notion that fewer defects indicate greater software quality.

2 Background

2.1 Measuring software quality

In software engineering research, *software quality* is not a simple, well-defined concept. Kitchenham and Pfleeger (1996) suggest that software quality can be viewed through five different lenses:

– The *transcendental view* sees software quality as something that can be recognised and worked towards, but never precisely defined nor perfectly achieved. This view holds that quality is inherently unmeasurable.

– The *user view* describes software quality by how well the software suits the needs of its users. This view does not consider the construction of the software unless it has a bearing on the user experience.

– The *manufacturing view* considers quality as conformance to specifications and development processes. Measuring manufacturing quality is done through measuring defect counts and rework costs.

– The *product view* sees quality as indicated by measurable internal characteristics of the software itself without regard to its use or usability. Software metrics like code coverage, cyclomatic complexity, and program size are some ways of measuring software quality from the product view.

– The *value-based view* takes an economic perspective by equating software quality with what the customer is willing to pay for the software.

The product and manufacturing views are the dominant views adopted by software researchers (van Vliet, 2000). Software is seen as a product, produced by a manufacturing process, i.e. software development. This view enables the quality of a *product* to be measured independently of the manufacturing *process*. Quality is then either the extent to which the product or process *conforms* to predetermined quality requirements, or the extent to which the product or process *improves* over time with respect to those requirements. Quality requirements are then made measurable by decomposing them into quality factors and subfactors. Each factor is then associated with specific metrics taken as indicating the degree to which that factor is present, and so indicating the degree of overall quality. Software quality is variously defined as "the degree to which a system, component or process meets specified requirements," (IEEE, 1990), or more broadly as "the degree to which software possesses a desired combination of attributes" (IEEE, 1998).

These two perspectives on software quality have been formalised in software engineering standards. ISO Std 9126 (ISO, 2001) and IEEE Std 1061 (IEEE, 1998) are both aimed at managing product conformance. The Capability Maturity Model (CMM/CMMI)[1] is a framework for measuring and carrying out software development process improvement. ISO 9001 and related ISO 900x standards[2] define how to manage and measure (software development) process conformance. Whilst these standards reflect the product and manufacturing views in what aspects of software development they consider relevant to software quality, the standards do not prescribe specific quality measurements nor hold any specific measures as necessarily better at indicating quality than others. Those choices and judgements are left as tasks for individual projects.

2.2 Scientific software development

There is a long history of software research focused on industrial and commercial software development but it is only recently that *scientific software* development has been seen as an important area of research (Kelly, 2007). There is evidence to show that scientific software development has significant differences from other types of software development.

Segal and Morris (2008) and Segal (2008) point to two major differences. Experimentation and trial-and-error work is an essential part of the development process because the software is built to explore the unknown. It is often impossible to provide complete requirements for the software upfront, and in fact, the requirements are expected to emerge and change over the lifetime of the project as the understanding of the science evolves. Partly because of this, the scientists must develop the software themselves, or be intimately involved, since it would be impossible to build the software correctly without their guidance and knowledge.

In a study of high-performance computing (HPC) communities, Basili et al. (2008) find that scientists value scientific output as the highest priority and make decisions on program attributes accordingly. For instance, an increase in machine performance is often seen as the opportunity to add scientific complexity to their programs, not as an opportunity to save on execution time (since that may not serve as great a scientific purpose). They report that scientists recognised software quality as both very important and extremely challenging. They note that the techniques used are "qualitatively different for HPC than for traditional software development", and that many software engineering techniques and tools, such as interactive debuggers, are simply not usable in their environment.

In summary, scientific software development (under which climate modelling falls) is markedly different from the traditional domains studied by software researchers. It works with few upfront requirements and a design that never truly settles since it must adapt to constant experimentation.

This raises the question: what does software quality mean in the climate modelling context?

2.3 The problem of software quality in scientific software

Stevenson (1999) discusses a divide between the software research community and the scientific community as it applies to scientists building large-scale computer simulations as their primary research apparatus. Stevenson raises the concern that because the primary job of a scientist is to do science, software engineering notions of quality do not apply to software constructed as part of a scientific effort. This is because of fundamentally incompatible paradigms: scientists are concerned with the production of scientific insight, while software engineers are concerned with the manufacturing process that produces software. Stevenson argues that for the term *software quality* to have meaning in the scientific domain, our notions of quality must be informed by our understanding of the requirement for *insight* and all that it entails.

When considering the use of computational simulations for science, insights come by way of gaining knowledge about the natural system that is being modelled. Stevenson offers specific terminology to understand this point clearly. There are three kinds of systems involved: the *observational* (i.e. the world itself; in our case, the climate), the *theoretical* (i.e. our theory or model of the workings of the observational system; in our case, the equations and concepts that describe climate processes), and the *calculational* (i.e. the

[1]See http://www.sei.cmu.edu/cmmi/

[2]See http://www.iso.org/iso/iso_catalogue/management_standards/quality_management.htm

executable implementation of the theoretical model; in our case, climate model code)[3]. Computational scientists study the behaviour of the calculational system to gain insight into the workings of the theoretical system, and ultimately the observational system.

Two basic kinds of activity ensure that the systems correspond to one another. *Validation* is the process of checking that the theoretical system properly *explains* the observational system, and *verification* is the process of checking that the calculational system correctly implements the theoretical system. The distinction between validation and verification is expressed in the questions, "Are we building the right thing?" (validation) and, "Are we building the thing right?" (verification). Stevenson also uses the term *complete validation* to refer to checking all three systems – that is, to mean that "we compute the right numbers for the right reasons."

Stevenson describes two types of quality with respect to the above model of computational science. *Intrinsic quality* is "the sum total of our faith in the system of models and machines." It is an epistemological notion of a good modelling endeavour; it is what we are asking about when we ask what needs to be present in any theoretical system and any implementation for us to gain insight and knowledge. *Internal quality* applies to a particular theoretical and calculational system, and asks how good our model and implementation is in its own right. For a mathematician, internal quality may relate to the simplicity or elegance of the model. For a computer scientist or engineer, internal quality may relate to the simplicity or extensibility of the code.

We have seen that, from one perspective, scientific insight is the ultimate measure of the overall quality of a scientific modelling endeavour. Meaningful insight depends upon theoretical and calculational systems corresponding in sensible ways to each other, and ultimately to the observational system under study. So, the "correctness" of our models is bound up with our notion of quality: what are the "right numbers"? How do we know when we see them? The conceptual machinery for approaching these questions is discussed succinctly by Hook (2009) and Hook and Kelly (2009).

Hook divides *error*, the "difference between measured or calculated value of a quantity and actual value", into *acknowledged error* and *unacknowledged error*. Acknowledged errors "are unavoidable or intentionally introduced to make a problem tractable" whereas unacknowledged errors "result from blunders or mistakes". Defining a theoretical model and refining it into a calculational model necessarily introduces acknowledged error. This error may come in the form of uncertainties in experimental observations, approximations and assumptions made to create a theory of the observational system, truncation and round-off errors that come from algorithmic approximations and discretizations of continuous expressions, the implementation – i.e. programming – of those algorithms, or even from compiler optimizations made during translation to machine code. Unacknowledged errors may appear at any step along the way because of mistakes in reasoning or misuse of equipment.

There are two fundamental problems that make impossible the traditional notion of testing by way of directly comparing a program's output to an expected value. The first is what Hook terms the *tolerance problem*: it is impossible, or very difficult, to tell if errors in output are completely free of unacknowledged error since it may be difficult to bound acknowledged error, and even with a bound on acknowledged error it is impossible to detect unacknowledged errors that fall within those bounds. In short, because there is a range of acknowledged error in the output, some unacknowledged error cannot reliably be detected.

The second problem is the *oracle problem*: "available oracles are problematically imprecise and limited". That is, for certain inputs there may not exist a source of precise expected outputs with which to compare a program's output. For a computational scientist, many of the important outputs of scientific software are the *results* of an experiment. If the output was always known beforehand, then the scientists would not be engaging in science. As a result of the oracle problem, scientists may have to rely on educated guesses, intuition, and comparison to available data in order to judge the "correctness" of their software.

In summary, for any given input there may be no accurate expected output values (the oracle problem); and because of inherent error in the output, unacknowledged errors may be undetectable (the tolerance problem). These problems do not suggest that building correct models is impossible, but that in the scientific software domain we must redefine correctness so as to take into account these problems. That is, we cannot accept that an evaluation of a model's correctness consists only of comparing output to expected values.

How, then, should climate model quality be judged? This is the problem of quality in scientific software which the present work explores, albeit only partially since we concern ourselves with the question of *software* quality and not theoretical quality.

Kelly and Sanders (2008) discuss the core questions that ought to guide a research program to understand and improve the quality of scientific software. They motivate their discussion by noting that in all software domains, testing is the most widely used quality assessment technique, yet scientists typically run tests only to assess their theories and not their software. From a scientist's perspective, Kelly and Sanders observe, "the software is invisible" – that is, scientists conflate the theoretical and calculational systems – unless the software is suspected of not working correctly (Segal, 2008). Kelly and Sanders point to this conflation, as well as a variety of other factors (such as the oracle problem)

[3]There are alternatives to these terms which Stevenson does not mention. The term *model* is used both to refer to the theoretical system at times, and at other times to refer to the calculational system. The term *simulation* is only used to refer to the *calculational system*.

that prevent the study of scientific software quality from being a straightforward matter of applying existing software engineering knowledge to a new domain. Instead, they suggest that software researchers work with scientists to learn more about their development context, and establish which software development techniques can be used directly and what has to be adapted or created. With respect to software correctness, they ask:

"*At this point, we don't have a full list of factors that contribute to correctness of scientific software, particularly factors in areas that a software engineer could address. What activities can contribute to factors of importance to correctness? How effective are these activities?" (Kelly and Sanders, 2008)*

We will revisit these questions in Sect. 5.2.

Assessing the quality of scientific software may be tricky, but is it needed? Hatton (1997b) performed a study analysing scientific software from many different application areas in order to shed light on the answer to this question. Hatton's study involved two types of quality tests. The first test, T1, involved static analysis of over 100 pieces of scientific software. This type of analysis results in a listing of "weaknesses", or static code faults – i.e., known "misuse[s] of the language which will very likely cause the program to fail in some context". The second test, T2, involved comparing the output of nine different seismic data processing programs, each one supposedly designed to do the same thing, on the same input data. Hatton found that the scientific software analysed had plenty of statically detectable faults, that the number of faults varied widely across the different programs analysed, and that there was significant and unexpected uncertainty in the output of this software: agreement amongst the seismic processing packages was only to one significant digit. Hatton concludes that, "taken with other evidence, the T experiments suggest that the results of scientific calculations carried out by many software packages should be treated with the same measure of disbelief researchers have traditionally attached to the results of unconfirmed physical experiments." Thus, if Hatton's findings are any indication of quality of scientific software in general, then improvements in software quality assessment of scientific software is dearly needed.

2.4 Climate model development

The *climate* is "all of the statistics describing the atmosphere and ocean determined over an agreed time interval." *Weather*, on the other hand, is the description of the atmosphere at a single point in time. Climate modellers are climate scientists who investigate the workings of the climate by way of computer simulations:

"*Any climate model is an attempt to represent the many processes that produce climate. The objective is to understand these processes and to predict the effects of changes and interactions. This characterization is accomplished by describing the climate system in terms of basic physical, chemical and biological principles. Hence, a numerical model can be considered as being comprised of a series of equations expressing these laws." (McGuffie and Henderson-Sellers, 2005)*

Climate modelling has also become a way of answering questions about the nature of climate change and about predicting the future climate and, to a lesser extent, the prediction of societal and economic impacts of climate change.

Climate models come in varying flavours based on the level of complexity with which they capture various physical processes or physical extents. GCMs ("global climate models" or "general circulation models") are the most sophisticated of climate models. They are numerical simulations that attempt to capture as many climate processes as possible with as much detailed output as possible. Model output consists of data for points on a global 3D grid as well as other diagnostic data for each time-step of the simulation. Whilst GCMs aspire to be the most physically accurate of models, this does not mean they are always the most used or useful; simpler models are used for specific problems or to "provide insight that might otherwise be hidden by the complexity of the larger models" (McGuffie and Henderson-Sellers, 2005; Shackley et al., 1998). In this paper we focus on the development of GCMs for two reasons: they are the most complex from a software point of view; and, to the extent that they provide the detailed projections of future climate change used to inform policy making, they are perhaps the models for which software quality matters the most.

GCMs are typically constructed by coupling together several components, each of which is responsible for simulating the various subsystems of the climate: atmosphere, ocean, ice, land, and biological systems. Each component can often be run independently to study the subsystem in isolation. A special model component, the coupler, manages the transfer of physical quantities (energy, momentum, air, etc.) between components during the simulation. As GCMs originally included only atmosphere and ocean components, models that include additional Earth system processes are often referred to as Earth system models (ESMs). For simplicity, hereafter, we will use the phrase *climate model* to mean both GCMs and ESMs.

In order to facilitate experimentation, climate models are highly configurable. Entire climate subsystem components can be included or excluded, starting conditions and physical parameterizations specified, individual diagnostics turned on or off, as well as specific features or alternative implementations of those features selected.

We are only aware of one study, Easterbrook and Johns (2009), that specifically examines the software development practices of climate modellers. The authors performed an ethnographic study of a major climate modelling centre in order to explore how scientists "think about software correctness, how they prioritize requirements, and how they develop a shared understanding of their models." The results confirm what we have already summarised above about general scientific software development in a high-performance computing environment.

Easterbrook and Johns find that evolution of the software and structure of the development team resemble those found in an open source community even though the centre's code is not open nor is development geographically distributed. Specifically, the domain experts and primary users of the software (the scientists) are also the developers. As well, there are a small number of code owners who act as gatekeepers over their component of the model. They are surrounded by a large community of developers who contribute code changes that must pass through an established code review process in order to be included in the model.

Easterbrook and Johns also describe the verification and validation (V&V) practices used by climate modellers. They note that these practices are "dominated by the understanding that the models are imperfect representations of very complex physical phenomena." Specific practices include the use of *validation notes*: standardized visualisations of model outputs for visually assessing the scientific integrity of the run or as a way to compare it with previous model runs. Another V&V technique is the use of bit-level comparisons between the output of two different versions of the model configured in the same way. These provide a good indicator of reproducibility on longer runs, and strong support that the changes to the calculational model have not changed the theoretical model. Finally, results from several different models are compared. Organized model intercomparisons are conducted with models from several organisations run on similar scenarios[4]. Additionally, the results from several different runs of the same model with perturbed physical parameters are compared in model *ensemble runs*. This is done so as to compare the model's response to different parameterizations, implementations, or to quantify output probabilities. Easterbrook and Johns conclude that "overall code quality is hard to assess". They describe two sources of problems: configuration issues (e.g. conflicting configuration options), and modelling approximations which lead to acknowledged error. Neither of these are problems with the code per se.

3 Approach

In this study we analysed defect density for three different models: two fully coupled general circulation models (GCMs) and an ocean model. For comparison, we also analyzed three unrelated open-source projects. We repeated our analysis for multiple versions of each piece of software, and we calculated defect density using several different methods. There are a variety of methods for deciding on what constitutes a defect and how to measure the size of a software product. This section makes explicit our definition of a defect and product size, and explains in detail how we conducted our study.

[4]See http://cmip-pcmdi.llnl.gov/ for more information.

Table 1. Post-delivery problem rates as reported by Pfleeger and Hatton (1997)

Source	Language	Failures per KLOC
IBM normal development	Various	30
Satellite planning study	Fortran	6 to16
Siemens operating system	Assembly	6 to 15
Unisys communications software	Ada	2 to 9
IBM Cleanroom development	Various	3.4
NAG scientific libraries	Fortran	3.0
Lloyd's language parser	C	1.4
CDIS air-traffic-control support	C	0.8

We also compare our results with defect density rates reported in the literature, typically calculated as the number of failures encountered (or defects discovered) after delivery of software to the customer, per thousand lines of source code (KLOC). For example, Pfleeger and Hatton (1997) list a number of published post-delivery defect rates, which we reproduce in Table 1. Hatton (1997a) states: "three to six defects per KLOC represent high-quality software." Li et al. (1998) state that "leading edge software development organizations typically achieve a defect density of about 2.0 defects/KLOC". The COQUALMO quality model (Chulani and Boehm, 1999), which bases its interpretation of defect density on the advice of industry experts, suggests that high software quality is achieved at a post-release defect density of 7.5 defects/KLOC or lower.

3.1 Selection process

Convenience sampling and snowballing were used to find climate modelling centres willing to participate (Fink, 2008). We began with our contacts from a previous study (Easterbrook and Johns, 2009), and were referred to other contacts at other centres. In addition, we were able to access the code and version control repositories for some centres anonymously from publicly available internet sites.

We only considered modelling centres with large enough modelling efforts to warrant a submission to the IPCC Fourth Assessment Report (Solomon et al., 2007). We used this criteria because the modelling centres were well-known, and we had access to the code, project management systems, and developers. In the interests of privacy, the modelling centres remain anonymous in this report. We use the identifiers C1, C2, and C3 to refer to the three models we studied.

To provide a comparison to other kinds of software, we also performed a defect density analysis on three projects unrelated to climate modelling:

– the *Apache* HTTPD[5] webserver, which has been widely studied as an example of high quality open source software;

– the *Visualization Toolkit (*VTK*)* [6], a widely used open source package for scientific visualization;

– the *Eclipse* project, an open source Integrated Development Environment, for which Zimmermann et al. (2007) provide a detailed defect density analysis.

3.2 Terminology

For the remainder of this paper, we adopt the following terminology:

– *Error* is the difference between a measured or computed quantity and the value of the quantity considered to be correct.

– A *code fault* is a mistake made when programming; it is "a misuse of the language which will very likely cause the program to fail in some context" (Hatton, 1997b).

– A *failure* occurs when a code fault is executed (Hook, 2009).

– The terms *defect* and *bug* are commonly used to refer to failures or faults, or both. We use these terms to mean both failures and faults, unless specified otherwise.

– *Defect reports* are reports about faults or failures, typically recorded in a bug tracking system, along with documentation on the resolution of the problem.

– A *defect fix* is any change made to the code to repair a defect, whether or not a defect report was documented.

3.2.1 Identifying defects

One approach to measuring software defects is to count each documented report of a problem filed against a software product. This approach has the drawback of ignoring those defects that are not formally reported, but which are found and fixed nonetheless. Since we did not have information on the bug reporting practices for every model we studied, we broadened our characterization of a defect from *reported and fixed problems* to *any problem fixed*. Hence, in addition to examining defect reports, we examined the changes made to the code over a given period to identify those that were intended to repair a defect. This broader analysis reflects an operational definition of a software defect as "any problem that is worth fixing".

Defect reports are usually easy to identify, since they are labeled and stored in a project database which can be queried directly. We consider only those reports specifically labeled as defects (as opposed to enhancements or work items) as well as being labeled as fixed (as opposed to unresolved or invalid).

Identifying defects fixes is more problematic. Although all code changes are recorded in a version control repository, the only form of labeling is the use of free-form revision log messages associated with each change. We used an informal technique for identifying defect fixes by searching the revision log messages for specific keywords or textual patterns (Zimmermann et al., 2007). We began by manually inspecting a sample of the log messages and code changes from each project. We identified which revisions appeared to be defect fixes based on our understanding of the log message and details of the code change. We then proposed patterns (as regular expressions) for automatically identifying those log messages. We refined these patterns by sampling the matching log messages and modifying the patterns to improve recall and precision. The pattern we settled on matches messages that contain the strings "bug", "fix", "correction", or "ticket"; or contain the "#" symbol followed by digits (this typically indicates a reference to a report ticket). Figure 1 shows a sample of log messages that match this pattern.

Some centres were able to provide us with a snapshot of their version control repository, as well as access to their bug tracking system (e.g., Bugzilla or Trac). In the cases where

```
 1. CT : BUGFIX083 :  add the initialisation
    of the prd 2D array in the xxxxx
    subroutine
 2. xxxx_bugfix_041 :  SM : Remove unused
    variables tauxg and tauyg
 3. Correct a bug in ice rheology, see ticket
    #78
 4. Correct a bug and clean comments in
    xxxxx, see ticket #79
 5. Ouput xxxx additional diagnostics at the
    right frequency, see ticket:404
 6. Initialization of passive tracer trends
    module at the right place, see ticket:314
 7. additional bug fix associated with
    changeset:1485, see ticket:468
 8. CT : BUGFIX122 :  improve restart case
    when changing the time steps between 2
    simulations
 9. Fix a stupid bug for time splitting and
    ensure restartability for dynspg_ts in
    addition, see tickets #280 and #292
10. dev_004_VVL:sync:  synchro with trunk
    (r1415), see ticket #423
```

Fig. 1. A sample of version control log messages indicating a defect fix. Redacted to preserve anonymity.

[5]http://httpd.apache.org/

[6]http://www.vtk.org/

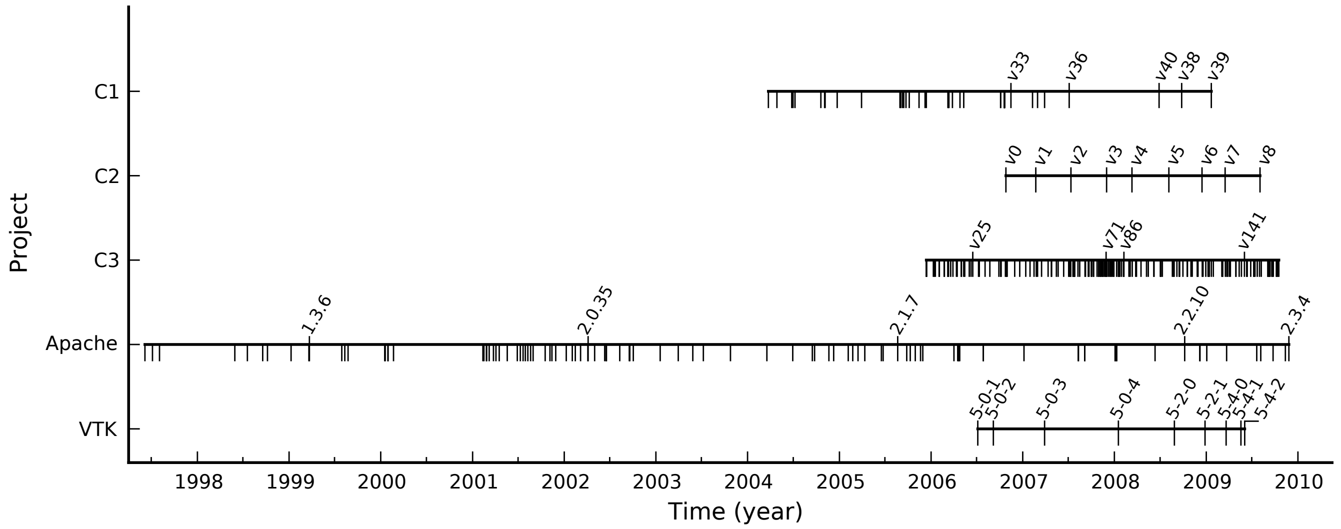

Fig. 2. Project repository time lines. Candidate versions are marked on the timelines with downward ticks, and analysed versions are labelled.

we only had access to the version control repository we used the tool CVSANALY[7] to build an SQLITE[8] database of the repository items. This database includes tables for: the log messages, tags, and all of the files and folders. One centre provided us with a snapshot of their *trac*[9] installation and repository. We used the database powering the *trac* installation (also based on SQLITE) as it stores the repository data in a similar way to CVSANALY.

3.2.2 Measuring product size

In order to normalize defect counts, it is necessary to select a method for calculating product size. The size of a software product is typically measured in terms of code volume (e.g., source lines of code) or function points (a measure of the functionality provided by the source code). Source lines of code (SLOC) can be measured automatically. In contrast, function points, which are considered to be a more accurate measurement of the essential properties of a piece of software (Jones, 2008), rely on subjective judgment, and are time-consuming to assess for large software systems. Therefore, we chose to use the source lines of code metric for its ease of measurement and repeatability (Jones, 2008; Park, 1992).

There are two major types of source lines of code measures: physical, and logical. The *Physical* SLOC measure views each line of text in a source file as a potential line of code to be counted. The physical SLOC measure we report counts all lines except blank lines and lines with only comments. The *Logical SLOC* measure ignores the textual formatting of the source code and considers each statement to be a line of code. In this study we report both of these measures but we use the physical SLOC measure in our calculation of defect density since we feel it as a more reproducible and language-neutral measure.

We used the CODECOUNT[10] tool to count source lines of code for all of our projects. We determined which source files to include in the count based on their extension: `.F, .f, .f90` for Fortran files and `.c, .cpp`, `.h`, and `.hpp` for C/C++ projects). We included other files if we knew from conversations with the developers that they contained code (for example, model C2 contained Fortran code in certain `.h` files). Additionally, we analysed the source files without performing any C preprocessing and so our line counts include C preprocessing directives and sections of code that might not appear in any specific model configuration.

3.2.3 Calculating defect density

Defect density is loosely defined as the number of defects found in a product divided by the size of the product. Defects are discovered continuously throughout the development and use of a software product. However, product size changes discretely as modifications are made to the source code. Thus, in order to calculate the defect density of a product we must be able to associate defects to a particular version of the product. We use the term *version* to refer to any snapshot of model source code whose size we can measure and assign defects to. A version in this sense does not necessarily refer to a public release of the product since defects can be both reported and fixed on unreleased or internally released versions.

In general, we attempted to limit the versions we analysed to major product releases only. We began with a pool of

[7]http://tools.libresoft.es/cvsanaly
[8]http://sqlite.org/
[9]http://trac.edgewall.org/
[10]http://csse.usc.edu/research/CODECOUNT/

candidate versions populated from the source code revisions in the version control repository. Where possible, we used only those versions indicated as significant by the developers through personal communication. Otherwise, we narrowed the pool of candidate versions to only those revisions that were tagged in the repository (models C1, C3, and comparators HTTPD and VTK) under the assumption that tags indicated significance. We further narrowed our candidate versions by selecting only those tagged revisions that had associated defect reports. We assumed that reports are typically logged against major versions of the product. We attempted to match repository tag names to the version numbers listed in the issue report database for the project. Where there was ambiguity over which tag version to choose we chose the oldest one[11]. We will refer to the remaining candidate versions – those that were included in our analysis – as *selected versions*. Figure 2 shows a time line for each project marking the selected versions, as well as the other candidate versions we considered. To maintain the anonymity of the models, we have used artificial version names rather than the repository tags or actual model version numbers.

Assigning a defect to a product version can be done in several ways. In a simple project, development proceeds sequentially, one release at a time. Simplifying, we can make the assumption that the defects found and fixed leading up to or following the release date of a version are likely defects in that version. Defects which occur before the release date are called *pre-release defects* and those which occur afterwards are called *post-release defects*. One method for assigning defects to a product version is to assign all of the pre- and post-release defects that occur within a certain time interval of a version's release date to that version. We call this method *interval assignment*. We used an interval duration of six months to match that used by Zimmermann et al. (2007).

An alternative method is to assign to a version all of the defects that occur in the time span between its release date and the release date of the following version. We call this method *span assignment*.

A third and more sophisticated method is used in Zimmermann et al. (2007), whereby defect identifiers are extracted from the log messages of fixes, and the version label from the ticket is used to indicate which version to assign the defect to. We call this method *report assignment*.

We used all three assignment methods to calculate defect density.

[11]For instance, in one project there were repository tags of the form <release_number>_beta_<id>, and a report version name of the form <release_number>_beta. Our assumption is that development on a major version progresses with minor versions being tagged in the repository up until the final release.

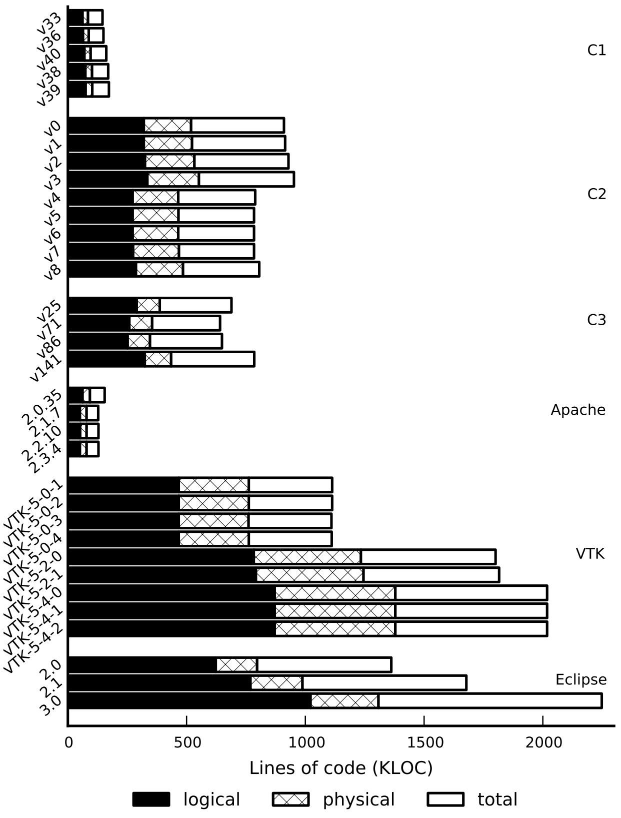

Fig. 3. Lines of code measurements for each project.

4 Results

Figure 3 displays the physical, logical and total line count for each project, and Table 2 lists the median defect densities of each project version using the physical product size measurement. For Eclipse, we extracted defect counts for each version by totalling the defects found across all of the plug-ins that compose the Eclipse JDT product using the data published by Zimmermann et al. (2007).

Figure 4 displays the post-release defect densities of the projects we analysed, with several of the listed projects from Table 1 marked down the right hand side of the chart for comparison. Both the fix- and report-defect densities are included for each assignment method.

Regardless of whether we count fixes or reported defects, and regardless of the assignment method used, the median defect density of each of the climate models is lower, often significantly, than the projects listed in Table 1. Similarly, the median model defect density is lower, often significantly, than the comparator projects.

Version defect densities are generally larger under span-assignment, and smaller under report-assignment. This is most likely because fewer defects are reported than those that are actually fixed. For instance, only suitably important

Table 2. Median project defect density (interquartile range in parenthesis) of analysed versions under different defect assignment methods.

Project	Interval-assignment Fixes	Interval-assignment Tickets	Span-assignment Fixes	Span-assignment Tickets	Report-assignment Fixes	Report-assignment Tickets
C1	0.540 (0.277)	0.475 (0.341)	0.752 (0.476)	0.284 (0.803)	0.124 (0.325)	0.241 (0.296)
C2	1.169 (0.549)	0.073 (0.029)	0.773 (0.357)	0.060 (0.019)	0.202 (0.106)	0.058 (0.025)
C3	0.838 (0.206)	0.191 (0.006)	0.522 (2.191)	0.124 (0.415)	0.006 (0.023)	0.039 (0.034)
Apache	3.586 (2.793)	0.755 (1.489)	12.503 (15.901)	3.436 (6.851)	0.283 (0.824)	0.270 (2.780)
VTK	1.217 (0.304)	0.010 (0.024)	0.776 (0.957)	0.009 (0.023)	0.000 (0.000)	0.000 (0.000)

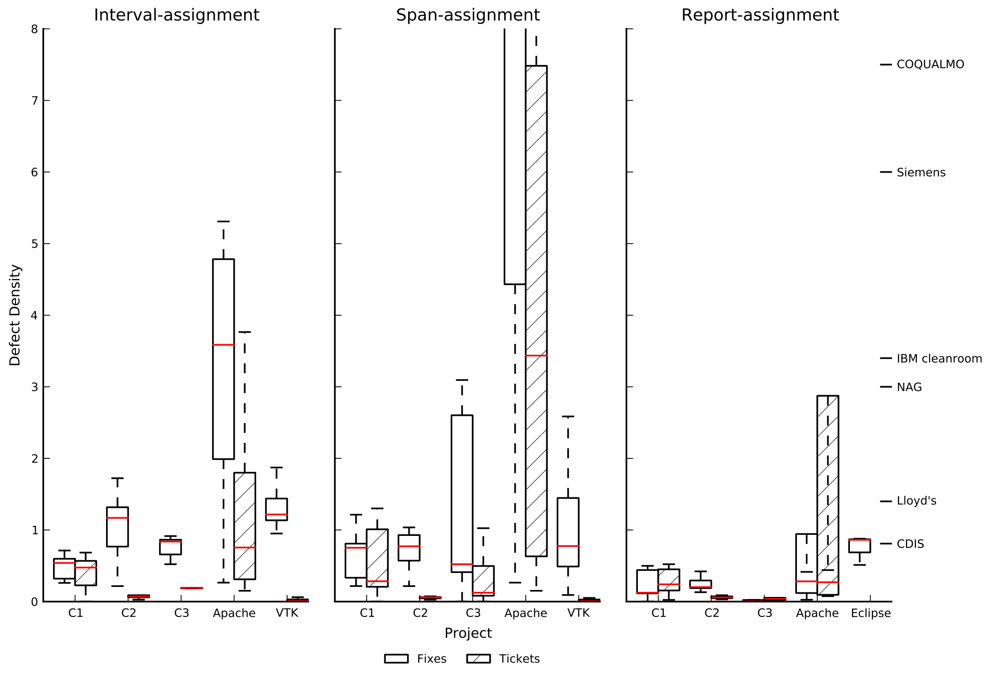

Fig. 4. Defect density of projects by defect assignment method. Previously published defect densities from Table 1 are shown on the right.

defects may be reported whereas minor defects are simply found and fixed.

5 Discussion

Each of the comparator projects we chose is a long-lived, well-known, open-source software package. We have good reason to believe that they are each of high quality and rigorously field-tested. Thus, our results suggest that the software quality of the climate models investigated is as good as, or better than, the comparator open source projects and defect density statistics reported in the literature. In addition, to the best of our knowledge, the climate modelling centres that produced the models we studied are representative of major modelling centres. This suggests that climate models from other centres may have similarly low defect densities.

Our results are surprising in light of previous studies of scientific software development, which show how volatile and vague their requirements are (Kelly, 2007; Segal and Morris, 2008; Segal, 2008; Carver et al., 2007). Our results suggest that the climate modellers have produced very high quality software under uncertain conditions with little in the way of guidance from the software engineering community.

Notwithstanding issues of construct validity that we discuss in Sect. 5.1.4, there are a number of possible explanations for low defect densities in the models. We offer the following hypotheses:

1. *Domain expertise.* Climate modellers are at once the scientific experts, primary users, and primary developers of climate models. This hypothesis asserts that because of their deep familiarity with the project, climate modellers make fewer requirements errors and introduce fewer logical defects. We would also expect that modellers are better able to recognise, find, and fix defects when they do arise, but that the increase in defect density this leads to is overwhelmed by former effect.

2. *Rigorous development process.* As we have discussed, scientific correctness is paramount for climate modellers. This concern is reflected in an extremely rigorous change management process where each code change undergoes intense scrutiny by other modellers (Easterbrook and Johns, 2009). Under this hypothesis, the relative effort put into inspecting code changes leads to fewer introduced defects.

3. *Domination by caution.* Fear of introducing defects may cause modellers to proceed with such caution as to slow down model development significantly, providing more time to consider the correctness of each code change. This hypothesis suggests we would also expect to see a lower code churn per developer per unit time than in commercial software practices[12]. If true, it might also mean that modellers are sacrificing some scientific productivity in return for higher quality code.

4. *Narrow usage profile.* Our comparators are general purpose tools (i.e. a numerical library, an IDE, and a webserver) whereas, this hypothesis holds, even though climate models are built to be extremely flexible, they are most often used and developed for a much smaller set of scenarios than they are capable of performing in. That is, only a limited number of the possible model configurations are regularly used in experiments. Development effort is concentrated on the code paths supporting these configurations, resulting in well-designed, well-tested and consequently, high quality code. However, this hypothesis would suggest the models may be relatively brittle, in that the less frequently used configurations of the models may include many more unnoticed code faults (unacknowledged errors). If code routines that are rarely used make up a significant proportion of the code size, then the defect density count will be skewed downwards.

5. *Intrinsic sensitivity/tolerance.* This hypothesis posits that there are intrinsic properties of climate models that lead to the production of high quality software independent of the skill of the development team. For instance, climate models may be sensitive to certain types of defects (those that change climate dynamics or numerical stability, for example). These defects appear as obvious failures (e.g. a crash, or numerical blowup) or improbable climate behaviours, and are therefore fixed at the time of development, resulting in fewer defect reports and fixes. At the same time, we have evidence that climate model behaviour is robust. One climate modeller we interviewed explained that the climate is a "heterogeneous system with many ways of moving energy around from system to system" which makes the theoretical system being modelled "tolerant to the inclusion of bugs." The combination of both factors means that code defects are either made obvious (and so immediately fixed) or made irrelevant by the nature of climate models themselves and therefore never reported as defects.

6. *Successful disregard.* Compared to other domains, climate modellers may be less likely to consider certain defects important enough to report or even be seen *as* defects. The culture of emphasizing scientific correctness may lead modellers to ignore defects which do not cause errors in correctness (e.g. problems with usability, readability or modifiability of the code), and defects for which there are ready workarounds (e.g output format errors). In other words, modellers have "learned to live with a lower standard" of code and development processes simply because they are good enough to produce valid scientific results. A net result may be that climate modellers incur higher levels of *technical debt* (Brown et al., 2010) – problems in the code that do not affect correctness, but which make the code harder to work with over time.

Several of these hypotheses call into question the use of standard measures of defect density to compare software quality across domains, which we will consider in depth in the following section.

5.1 Threats to validity

5.1.1 Overall study design

We do not yet understand enough about the kinds of climate modelling organisations to make any principled sampling of climate models that would have any power to generalize to all climate models. Nevertheless, since we used convenience and snowball sampling to find modelling centres to participate in our study, we are particularly open to several biases (Fink, 2008):

- Modelling centres willing to participate in a study on software quality may be more concerned with software quality themselves.

- Modelling centres which openly publish their climate model code and project artifacts may be also be more concerned with software quality.

One of the models used in the study (C1) is an ocean model rather than a full climate model. Even though this particular model component is developed as an independent project, it is not clear to what extent it is comparable to a full GCM.

Our selection of comparator projects was equally undisciplined: we chose projects that were open-source, and that

[12]although this may be masked by certain coding practices, (e.g. cut-and-paste, lack of granularity) where conceptually small changes result in disproportionate source code changes.

were large enough and well-known enough to provide an intuitive comparison to the climate models.

Our choice to use defect density as a quality indicator was made largely because of its place as a *de facto* rough measure of quality, and because of existing publications to compare to. Gauging software quality is known to be tricky and subjective and most sources suggest that it can only accurately be done by considering a wide range of quality indicators (Jones, 2008; IEEE, 1998; ISO, 2001; Hatton, 1995). Thus, at best, our study can only hope to present a very limited view of software quality.

5.1.2 Internal validity

The internal validity of the defect assignment methods (i.e. interval- and span-assignment) is threatened by the fact that we chose to view software development as proceeding in a linear fashion, from one major version to the next. This view assumes a defect found immediately before and after a release date is a defect in that release. However, when several parallel branches of a project are developed simultaneously, as some projects in our study were, this flattened view of development is not able to distinguish amongst the branches. We may have incorrectly assigned a defect to a version in a different branch if the defect's date was closer to the version's release date than to the version the defect rightfully is associated with.

In addition, we assumed a 1:1 mapping between defect indicators and defects. We did not account for reports or version control check-ins that each refer to multiple defects, nor for multiple reports or check-ins that, together, indicate only one defect.

Finally, we did not perform any rigorous analysis of recall and precision of our fix identification method. This means we cannot say whether our counts are over- or under-estimates of the true number of check-ins that contain defect fixes.

5.1.3 External validity

The external validity of our assignment methods depends on correctly picking repository versions that correspond to actual releases. If a version is used that is not a release, then it is not clear what is meant by pre-release and post-release defects, and whether they can be compared. For several of the projects we made educated guesses as to the versions to select (as described in Sect. 3), and so we may have classified some defects as post-release defect that may more rightly be classified as pre-release defects had we chosen the correct version. Similarly, if there were no releases of the project made in our repository snapshot, we used intermediate versions. This makes it difficult to justify comparing defect rates since pre- and post-release are not clearly defined.

Our definition of a defect as "anything worth fixing" was a convenient definition for our purposes but it has not been validated in the field, and it is even unclear that it corresponds to our own intuitions. What about defects that are found but not worth fixing right then and there? We confront this question in Sect. 5.2.

Finally, there are many small differences between the way we carried out our identification of code fixes and that of Zimmermann et al. (2007). In their study, they did not rigorously specify the means by which check-in comments were identified as fixes; they only gave a few examples of common phrases they looked for. We were forced to invent our own approximation. Furthermore, for report-assignment, Zimmermann et al. (2007) used the first product version listed in a report's history as the release date to associate defects with. Since we did not have access to the report history for every project we analysed, we only considered the product version as of the date we extracted the report information. As well, Zimmermann et al. (2007) only counted defects that occurred within 6 months of the release date whereas we counted all defects associated with a report version. Thus, it is not clear to what extent we can rightly compare our results.

5.1.4 Construct validity

As we have mentioned, defect density is the *de facto* informal measure of software quality but it is by no means considered a complete or entirely accurate measure. Hatton (1997a) says:

"We can measure the quality of a software system by its defect density – the number of defects found per KLOC over a period of time representing reasonable system use. Although this method has numerous deficiencies, it provides a reasonable though rough guide."

The question we explore in this section is: to what extent can we consider defect density even a rough indicator of quality?

We suggest the following aspects which make the defect density measure open to inconsistent interpretation:

– **Finding, fixing, and reporting behaviour.** In order to be counted, defects must be discovered and reported. This means that the defect density measure depends on the testing effort of the development team, as well as the number of users, and the culture of reporting defects. An untested, unused, or abandoned project may have a low defect density but an equally low level of quality.

– **Accuracy and completeness of repository comments or defect reports are accurate.** There is good reason to believe that these data sources contain many omissions and inaccuracies (Aranda and Venolia, 2009).

– **Product use.** The period of time over which to collect defects (e.g. "reasonable system use") is unclear and possibly varies from release to release.

– **Release cycle.** How do we decide which defects to consider post-release and which ones pre-release? Do we consider beta releases or only major releases? Does a project even make major releases or does it have continuous incremental releases?

– **Product size.** There are many ways of evaluating the product size, which one should we use and is it replicable? Can it account for the expressiveness of different languages, formatting styles, etc?

– **Criticality and severity.** Are all defects counted equally, or certain severity levels ignored?

When we use the defect density measure to compare software quality between projects, we are implicitly making the assumption that these factors are similar in each project. If they are not – and without any other information we have no way of knowing – then we suggest the defect density measure is effectively meaningless as a method of comparing the software quality, even roughly, between products. There is too much variability in the project conditions for a single interval measure to account for or express.

Even if all our concerns above are taken into account, we cannot rightly conclude that a product with low defect density is, even roughly, of better quality than one with a higher defect density. Jones (2008) states that whilst software defect levels and user satisfaction are correlated, this relationship disappears when defect levels are low: having fewer defects does not tell us anything about the presence of favourable quality attributes.

Our focus on defect density emphasizes code correctness over and above other aspects of software quality. Of particular concern to the climate modelling community is the extent to which poorly written or poorly structured code may slow down subsequent model development and hence may reduce scientific productivity, even it if works perfectly. Our study did not attempt to measure these aspects of software quality.

In Sect. 5.2 we will discuss ideas for future studies to help discover quality factors relevant in the climate modelling domain.

5.2 Future work

Many of the limitations to the present study could be overcome with more detailed and controlled replications. Mostly significantly, a larger sample size both of climate models and comparator projects would lend to the credibility of our defect density and fault analysis results.

As we have mentioned elsewhere, assessing software quality is not a simple matter of measuring one or two quality indicators, but neither is it clear how *any* collection of measurements we could make could give us an assessment of software quality with confidence. Hatton (1995) remarks:

“There is no shortage of things to measure, but there is a dire shortage of case histories which provide useful correlations. What is reasonably well established, however, is that there is no single metric which is continuously and monotonically related to various useful measures of software quality..."

Later on, he states that “individual metric measurements are of little use and [instead] combinations of metrics and some way of comparing their values against each other or against other populations is vital”. His proposal is to perform a *demographic analysis* – a comparison over a large population of codes – of software metrics in order to learn about the discriminating power of the measure in a real-world context.

While an important future step, mining our arsenal of metrics for strong correlations with our implicit notions of software quality, which we believe this approach boils down to, cannot define the entire research program. There is a deeper problem which must be addressed first: our notion of software quality with respect to climate models is theoretically and conceptually vague. It is not clear to us what differentiates high from low quality software, nor is it clear which aspects of the models or modelling processes we might reliably look to make to that assessment. If we do not get clear on what we mean by software quality first, then we have no way to assess what any empirical test is measuring, and so we will have no way to accept or reject measures as truly indicative of quality. We will not be doing science.

To tackle this conceptual vagueness, we suggest a research program of theory building. We need a theory of scientific software quality that describes the aspects of the climate models and modelling process which are relevant to the software quality under all of the quality views outlined by Kitchenham and Pfleeger, 1996 (except perhaps the transcendental view, which by definition excludes explanation), as well as the ways in which those aspects are interrelated. To achieve this, we propose in-depth empirical studies of the climate modelling community from which to ground a theory.

We suggest further qualitative studies to investigate the quality perceptions and concerns of the climate modellers, as well as documenting the practices and processes that impact model software quality. A more in-depth study of defect histories will give us insights into the kinds of defects climate modellers have difficulty with, and how the defects are hidden and found. As well, we suggest detailed case studies of the climate modelling development done in a similar manner to Carver et al. (2007), or Basili et al. (2008).

We also see a role for more participatory action research whereby software researchers work directly with climate modellers to implement a quality assessment program. Our interviews have shown us than software quality *is* a recognised concern for climate modellers but it is not one that is widely discussed outside of each climate modelling centre. Software researchers may be able to play a role in fostering the development of community-wide software quality benchmarks or assessment programs by providing climate modellers with a level-headed interpretation existing assessment methodologies, as well as helping with their implementation and studying their effectiveness.

6 Conclusions

The results of our defect density analysis of three leading climate models show that they each have a very low defect density nnnnnnnn across several releases. A low defect density suggests that the models are of high software quality, but we have only looked at one of many possible quality metrics. Knowing which metrics are relevant to climate modelling software quality, and understanding precisely how they correspond the climate modellers notions of software quality (as well as our own) is the next challenge to take on in order to achieve a more thorough assessment of climate model software quality.

Acknowledgements. We would like to thank the modelling centres who participated in this study and provided access to their code repositories. We also thank Tom Clune and Luis Kornblueh for their comments on an earlier version of this paper. Funding was provided by NSERC.

Edited by: R. Redler

References

Aranda, J. and Venolia, G.: The secret life of bugs: Going past the errors and omissions in software repositories, in: ICSE '09: Proceedings of the 31st International Conference on Software Engineering, 298–308, IEEE Computer Society, Washington, DC, USA, doi:10.1109/ICSE.2009.5070530, 2009.

Basili, V. R., Carver, J. C., Cruzes, D., Hochstein, L. M., Hollingsworth, J. K., Shull, F., and Zelkowitz, M. V.: Understanding the High-Performance-Computing Community: A Software Engineer's Perspective, Software, IEEE, 25, 29–36, doi:10.1109/MS.2008.103, 2008.

Brown, N., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., and Nord, R.: Managing technical debt in software-reliant systems, Proceedings of the FSE/SDP workshop on Future of software engineering research – FoSER '10, 47 pp., doi:10.1145/1882362.1882373, 2010.

Carver, J. C., Kendall, R. P., Squires, S. E., and Post, D. E.: Software Development Environments for Scientific and Engineering Software: A Series of Case Studies, in: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, 550–559, IEEE Computer Society, Washington, DC, USA, doi:10.1109/ICSE.2007.77, 2007.

Chulani, S. and Boehm, B.: Modeling Software Defect Introduction and Removal: COQUALMO (COnstructive QUALity MOdel), Tech. rep., Center for Software Engineering, USC-CSE-99-510, 1999.

Easterbrook, S. M. and Johns, T. C.: Engineering the Software for Understanding Climate Change, Comput. Sci. Eng., 11, 65–74, doi:10.1109/MCSE.2009.193, 2009.

Fink, A.: How to Conduct Surveys: A Step-by-Step Guide, Sage Publications, Inc, fourth edition edn., 2008.

Hatton, L.: Safer C: Developing Software for in High-Integrity and Safety-Critical Systems, McGraw-Hill, Inc., New York, NY, USA, 1995.

Hatton, L.: N-version design versus one good version, IEEE Software, 14, 71–76, doi:10.1109/52.636672, 1997a.

Hatton, L.: The T experiments: errors in scientific software, IEEE Comput. Sci. Eng., 4, 27–38, doi:10.1109/99.609829, 1997b.

Hook, D.: Using Code Mutation to Study Code Faults in Scientific Software, Master's thesis, Queen's University, 2009.

Hook, D. and Kelly, D.: Testing for trustworthiness in scientific software, in: SECSE '09: Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering, pp. 59–64, IEEE Computer Society, Washington, DC, USA, doi:10.1109/SECSE.2009.5069163, 2009.

IEEE: IEEE Standard Glossary of Software Engineering Terminology, Tech. rep., doi:10.1109/IEEESTD.1990.101064, 1990.

IEEE: IEEE standard for a software quality metrics methodology, Tech. rep., doi:10.1109/IEEESTD.1998.243394, 1998.

ISO: ISO/IEC, 9126-1:2001(E) Software engineering – Product quality, Tech. rep., http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749, [Last accessed 10 Feb, 2012], 2001.

Jones, C.: Applied Software Measurement: Global Analysis of Productivity and Quality, McGraw-Hill Osborne Media, 3 edn., 2008.

Kelly, D. and Sanders, R.: Assessing the quality of scientific software, in: First International Workshop on Software Engineering for Computational Science & Engineering, http://cs.ua.edu/~{}SECSE08/Papers/Kelly.pdf, [Last accessed 10 Feb, 2012], 2008.

Kelly, D. F.: A Software Chasm: Software Engineering and Scientific Computing, Software, IEEE, 24, 120–119, doi:10.1109/MS.2007.155, 2007.

Kitchenham, B. and Pfleeger, S. L.: Software quality: the elusive target [special issues section], Software, IEEE, 13, 12–21, doi:10.1109/52.476281, 1996.

Li, M. N., Malaiya, Y. K., and Denton, J.: Estimating the Number of Defects: A Simple and Intuitive Approach, in: Proc. 7th Int'l Symposium on Software Reliability Engineering, ISSRE, 307–315, available at: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.5000 (last access: 27 July 2012), 1998.

McGuffie, K. and Henderson-Sellers, A.: A Climate Modelling Primer (Research & Developments in Climate & Climatology), John Wiley & Sons, 2nd Edn., 2005.

Park, R. E.: Software Size Measurement: A Framework for Counting Source Statements, Tech. rep., Software Engineering Institute, 1992.

Pfleeger, S. L. and Hatton, L.: Investigating the influence of formal methods, Computer, 30, 33–43, doi:10.1109/2.566148, 1997.

Segal, J.: Models of Scientific Software Development, in: Proc. 2008 Workshop Software Eng. in Computational Science and Eng. (SECSE 08), http://oro.open.ac.uk/17673/, 2008.

Segal, J. and Morris, C.: Developing Scientific Software, IEEE Software, 25, 18–20, doi:10.1109/MS.2008.85, 2008.

Shackley, S., Young, P., Parkinson, S., and Wynne, B.: Uncertainty, Complexity and Concepts of Good Science in Climate Change Modelling: Are GCMs the Best Tools?, Climatic Change, 38, 159–205, doi:10.1023/A:1005310109968, 1998.

Solomon, S., Qin, D., Manning, M., Chen, Z., Marquis, M., Averyt, K. B., Tignor, M., and Miller, H. L. (Eds.): Climate Change 2007 – The Physical Science Basis: Working Group I Contribution to the Fourth Assessment Report of the IPCC, Cambridge University Press, Cambridge, UK and New York, NY, USA, 2007.

Stevenson, D. E.: A critical look at quality in large-scale simulations, Comput. Science Eng., 1, 53–63, doi:10.1109/5992.764216, 1999.

van Vliet, H.: Software Engineering: Principles and Practice, 2nd Edn., Wiley, 2000.

Zimmermann, T., Premraj, R., and Zeller, A.: Predicting Defects for Eclipse, in: PROMISE '07: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, pp. 9+, IEEE Computer Society, Washington, DC, USA, doi:10.1109/PROMISE.2007.10, 2007.