11-1988

Bringing Software Under Statistical Quality Control

Harlan D. Mills

J. H. Poore

Bringing Software Under Statistical Quality Control

The clean room method of development offers several benefits.

by Harlan D. Mills and J.H. Poore

As with every other area of business, management must question old methods and past performance in software. The time has come to hold software to exacting standards. As long as management is willing to have sympathy for shortcomings and failures, the software industry has little incentive to reform. Software development and procurement must be brought under the same scrutiny as other aspects of the business. Software deserves serious management attention.

Statistical quality control is one of the tools being used by aggressive companies seeking to improve quality and cut costs. Bringing software under SQC starts with management expectations about the software's performance and how the performance will be measured.

There are many dimensions of software quality in which local concepts and standards are important.[1] These riches notwithstanding, if the buyer can specify performance, function, and features, and if a scientifically auditable measure can be agreed upon, then the essentials exist for bringing software under SQC. All that is needed is a process for software development that can be modified and that responds to variable standards of performance. The clean room process of software development has several parts. This method offers the ability to:

- cast the design process in increments.
- test each increment statistically.
- generate data that can prove that the process is producing the expected product.

Statistical testing of software

Statistical testing, as contrasted with anecdotal testing, supports scientific statements concerning software. Exhaustive testing constitutes proof of correctness or population testing rather than testing samples. But because exhaustive testing is usually impossible, extensive and ingenious software testing methods have evolved that emphasize debugging. Such testing is called anecdotal testing because, at most, one can say that such-and-such cases work, or that all paths were tested at least once, or that certain boundary cases worked. The anecdote can be quite extensive and impressive, but when the number of possible inputs is astronomical, even the most extensive anecdotes are not compelling. Anecdotal testing has something of the flavor of UFO sightings. Statistical testing shifts the emphasis away from testimony and toward scientific measurement.

Bringing software under SQC depends on both a controllable, auditable process of software development and the ability to statistically analyze performance. Two forms of statistical testing are used, one of which is generally applicable to any software however developed, and one that is modeled specifically for clean room. Both forms have the same basis and can be driven by the same testing process, but differ in the statistical model to which the testing outcomes are passed for calculation. The basics of statistical testing include:

- a characterization of the intended use of the software, and the ability to sample test cases randomly from this usage environment.
- the ability to know whether each input test case produced an output that was right or wrong.
- a statistical model of interest.

In addition, there are other necessities that include the ability to automate the process of generating test cases, the availability of a test harness to conduct the tests, and the availability of an automated "oracle" to judge each test case right or wrong. Statistical testing of software is not well-developed, either in theory or in practice. However, recent research[2] is showing it to be a powerful technique and an emerging rival to conventional testing.

Hypotheses that are simple to state, simple to test, and quite revealing characterize statistical testing in its most basic form. For example, one might wish to test the accuracy of a subroutine that calculates the sine function for a new personal computer mathematical software package, and to issue a warranty for the subroutine. In this example the input space is understood, random samples can be easily generated, and a trusted algorithm running on a trusted scientific computer can be used as the oracle.

Both the fundamentals and the practical necessities are easy in this example. The hypothesis might be: "Seventy-five percent of the output of the sine function will be correct to the eighth decimal place." One sets the confidence level,

determines the sample size, generates the sample, runs the test recording the number of outputs right and the number wrong, performs calculations of the statistical model, and, let's say, is willing to make the following warranty: "ABC, Inc. warrants with 95% confidence that the sine subroutine will produce output that is correct to eight decimal places for at least 75% of its inputs." (Don't judge this a weak warranty without first statistically testing your favorite sine routine.)

It can be hard to test complex software systems because it is hard to ensure the assumptions to the statistical models. Furthermore, if one seeks high confidence that the number of failures is very low, the sample size required will become large, which increases the cost of making and testing cases. Indeed, clean room strives for zero defects, which means that errors become rare events in the statistical sense.[3] Still, the effort and cost are worthwhile because the result is an auditable scientific statement about how the software will perform. While such tests are of great value, the goal goes beyond testing the efficacy of existing software.

If software is to be brought under SQC, the process by which it is made must itself be revisable in light of statistical evidence. In the clean room method, there are various points at which it can be determined whether the process is in control within the development of an increment (i.e., to ensure that clean room standards are being met at intermediate points). However, research and experience to date have dealt only with the final certification of the increment. Results of random tests become data to the certification model of reliability.

The clean room development method

The clean room software development method has three main attributes: a set of attitudes, a series of carefully prescribed processes, and a rigorous mathematical basis.

Cultivating attitudes is management's job. Meaningful management control must prevail throughout the process steps. Clean room developments are done by teams and the manager must be a full participant. Team size varies from five to eight members, and some members of the team might specialize in certain skills used in the steps of the process. However, the manager must point out that all successes are team successes and all failures are team failures. Clean room teams strive for zero defects, so every member of the team must have first-hand successes and know that such a goal is not only possible, but that its achievement is expected.

Clean room teams maintain total intellectual control over the development. That is, each member of the team must confidently understand each step the team takes and attest to its correctness. In this day of sophisticated software engineering tools, the main clean room tool is the wastebasket. The team leader must see that this tool is used without hesitation whenever the team lacks full confidence and intellectual control over the work.

Finally, the manager must protect the team attitudes, the series of processes, and the clean room standard within the organization. In clean room, code is not written until very late. There is no debugging, no integration, no overtime, no panic, and the workers don't behave like typical programmers when the end is near. These differences can be disturbing to the surrounding organization.

Clean room processes are quite specific and lend themselves to process control and statistical measures. The Box Structure analysis and design process[4] allows the contract between customer and developer to be made precise. Further analysis and design identify the increments that are crucial to the final statistical certification. At this juncture the process forks and usage analysis, followed by test case preparation, can proceed concurrently with efforts leading to code production. Still further Box Structure work sorts out common services, settles the correct organization of the system, and breaks out independently verifiable modules. Ultimately, pseudo code is derived from the specification.

Functional verification[5] of the pseudo code is a critical step in the process. The extent to which functional verification succeeds or fails indicates whether the process is in control. Functionally verified pseudo code is then transliterated into the target language for the application. Code walk-throughs at this point again produce evidence of whether the process is in control.

The test data development and the code now come together at the computer for certification. With no debugging, clean room teams expect their codes to pass the statistical tests and to be delivered in full compliance with the contract.

Clean room software engineering uses mathematical verification to replace program debugging before release to statistical testing. This mathematical verification is done by people, based on rigorous software engineering practices.[5] We have found that:

> The main clean room tool is the wastebasket.

- human verification is surprisingly synergistic with statistical testing.
- mathematical fallibility is very different from debugging fallibility.
- errors of mathematical fallibility are much easier to find in statistical testing than are errors of debugging fallibility.

The method of human mathematical verification used in clean room is called functional verification. This method is quite different from the method of axiomatic verification usually taught in theoretical computer science. It reduces software verification to ordinary mathematical reasoning about sets and functions as directly as possible. The motive for doing so is the problem of scaling up. Producing a product in a high-volume factory is very different from making a single item by hand in laboratory. The key difference is that of scale. A similar phenomenon exists with software. Systems of thousands and millions of lines of code are intellectually different from small programs of 40 or 60 lines of code. Success with large systems depends on behaviors and techniques that are provably correct in small programs and that scale up to very large systems. Techniques that do not scale up reliably are very harmful in large systems.

Introducing verification in terms of sets and functions establishes a basis for reasoning that scales up. Large programs have many variables, but only one function. Functional verification works well for both million-line systems and 60-line programs.[6]

The feasibility of combining human verification with statistical testing was the motive for defining a new software engineering process under SQC.[7] For that purpose, it was necessary to define a new software development life cycle of several incremental releases according to a structured specification of function and statistical usage. A structured specification is a formal specification, a relation, with a decomposition into a hierarchy of subsets, that provides a specification for each release that

includes those of all previous releases. That is, a structured specification defines not only the final software, but also a release plan for the implementation and statistical testing of each increment of the specification. As each release becomes available, statistical testing provides statistical estimates of its reliability.

Software process analysis and feedback can be used to meet prescribed reliability goals, e.g., by increased verification or more intermediate specification formality for later releases. As errors are found and fixed during certification, the growth in the reliability of the accumulating system can also be estimated. Thus, a certified reliability estimate of the system-tested final release can be provided.

What makes this work? The mathematical character of the entire process. It begins with the attitudes that are mathematically sound; namely, that the task is undertaken with the aim of understanding it thoroughly, doing it correctly, and verifying the result. Box Structure analysis and design is in the best mathematical tradition of taking small creative steps and then verifying them. It involves repeated analysis and synthesis until the basics are clear and the correct definitions, lemmas, and theorems are articulated. Functional verification is constructive set theoretical proof. Finally, statistical analysis allows one to make scientific, rather than anecdotal, statements about the software and the process.

A key process in the clean room methodology is the information system analysis and design. At the highest level, this process rationalizes the specification of any ambiguities or deficiencies. At the lowest level, it produces pseudo code meeting the strictest standards for structured programming.[5] At intermediate levels, the intricate technical details that make the difference between a robust system and a trouble-ridden system must be settled.

Certifying statistical quality in software

Software under clean room development requires a mathematical model that mirrors two key aspects of the process:[8]

1. With each clean room increment, results of statistical testing might indicate software changes to correct errors.
2. With the release of each clean room increment, new untested software will be added to software already under test.

Each set of changes to correct failures within a release creates a new software product much like its predecessor, but with a mathematically different reliability measure. Each of these increments will be subject to a measure of statistical testing before it is superseded by its successor. Statistical estimates of reliability will be of a certain confidence. Therefore, to aggregate the testing experience for an increment release, a model of reliability change with parameters M,R, was defined in reference 7 for the mean time to failure (MTTF) after a number c of software changes, of the form

$$MTTF = MR^c$$

where

M is the initial MTTF of the release.

R is the ratio of change in MTTF for one software change.

Although various technical rationales are given for this "certification" model in reference 7, it should be considered a contractual basis for the eventual certification of the finally released software by the developer to the customer.

The data from statistical testing can be organized as a sequence of pairs:

$$((t_1,c_1), (t_2,c_2), \ldots, (t_n,c_n))$$

in which $t_1, t_2, \ldots, t_n$ are the consecutive fail-free intervals (i.e., failures were detected at time t_1, $t_1 + t_2$, . . ., and interval t_n might or might not have terminated with a failure), and c_i is the cumulative number of changes made to the software by the end of the interval t_i. If the model were absolutely correct for parameters M,R, and there were no statistical variations, these intervals and changes would satisfy the equations

$$t_1 = MR^{c_1}, \; t_2 = MR^{c_2}, \ldots,$$

and M and R could be solved for these equations with the test data.

Of course, there is no way to know that the model is absolutely correct, and there will be statistical variation. Thus, statistical estimators were defined for M,R in terms of the test data. The choice of these estimators is based on statistical analysis, but should also be a contractual basis for certification.

The results of these two contractual bases—a reliability change model and statistical estimators for its parameters—give buyer and seller an objective way to certify the reliability of the delivered software. The certification is a scientific statistical inference obtained by a prescribed computation on test data warranted by the developer to be factual and auditable. Such a certification is little different from the certification of the net worth of a business, defined by a prescribed computation of financial data warranted to be factual and auditable. Of course, not all software aspects are covered by this certification (e.g., maintainability, transportability, etc.), any more than all aspects of a business (e.g., goodwill, growth potential, etc.) are covered by its financial audit.

However, this contractual basis for certifying the reliability of software provides a foundation for SQC. This is little different from SQC for producing items to any prescribed measurements, in that both buyer and seller must agree to a common set of measurements, and to statistical estimators based on test data.

The estimators given in reference 7 for software reliability are, in principle, no more than a sophisticated way of averaging the interfail times, taking into account the change activity called for during statistical testing. As test data materialize, the reliability can be estimated, even change by change. With successful corrections, the reliability estimates will improve with further testing, as objective quantitative evidence of the achievement (or not) of prescribed reliability goals.

This evidence is itself a basis for management control of the software development process to meet reliability goals. For example, process analysis might reveal unexpected sources of errors, such as poor understanding of the underlying hardware, too much fallibility in verification, etc., with appropriate corrections in the process itself for later increments. That is, intermediate rehearsals of the final certification provide a feedback basis for management to meet final goals.

The treatment of separate increment releases should also be part of the contractual basis between developer and user. Perhaps the simplest treatment is to treat separate increments independently. However, more statistical confidence in the final certification will result by aggregating testing experience across increments. The reliability change model has the property that any software change can be used as a new point of departure, since for (c+d) changes,

$$MR^{c+d} = (MR^c)R^d$$

thus, MR^c serves as a new "initial" MTTF. A simple aggregation could be used to complement separately treated increments with management judgment.

There are other reliability models besides the certification model. However, a recent comparative study[9] shows the certification model to have distinct advantages when used with high-quality software. Among the advantages are that the model uses least squares estimators rather than the more complex maximum likelihood estimators. This in turn avoids differential equations that lack analytical solutions and avoids numerical methods that fail to converge at points of interest. In short, the least squares estimators are easily understood and computed.

Also, the software development process more nearly satisfies the assumptions of the model and testing readily provides the data needed to drive the model. Finally, the certification model is better behaved in predictive qualities than competing models. Again, the model assumes high-quality software, which means that interfail times will grow geometrically, that errors are corrected with great assurance that more good than harm will be done, and that statistical testing will expose early those errors that will be most common in usage.

Clean room software experience

The clean room methodology is an evolutionary step in the development of software engineering. It is evolutionary in eliminating debugging because over the past 20 years program design has been emphasized in languages that must be verified rather than executed. So the relative effort in debugging, compared to verifying, among advanced developers is now quite small, even in nonclean room development. It is evolutionary in statistical testing because with higher-quality programs at the outset, user representative testing is correspondingly a greater fraction of the total testing effort. And, as already noted, a strong synergism exists between human verification and statistical testing. People are fallible with human verification, but the errors they leave behind for system testing are much easier to find and fix than those left behind for debugging.

Experience to date

Clean room experience includes three commercial projects, an IBM language product of 80,000 lines of code (80 KLOC), an Air Force contract helicopter flight program (35 KLOC), and a NASA contract space transport planning system (45 KLOC). The major finding in these projects is that human verification can replace debugging in software development. Human verification can produce software robust enough to go to system test without debugging. Typical increments are 5 to 15 KLOC; with experience and confidence, such increments can be expected to greatly increase in size. All three projects showed productivity equal to or better than expected for ordinary software development.

In a controlled experiment at the University of Maryland, students developed a project in message processing (1 to 2 KLOC). The results indicate better productivity and quality with clean room than with interactive debugging and integration, even on first experience.[10]

A team at the University of Tennessee has achieved the clean room level of performance. Students continue to participate in clean room projects to learn the methodology and to be able to form and lead clean room teams. Thorough training of clean room teams is an arduous process, but the resulting productivity justifies the effort.

Several leading corporations are sufficiently intrigued with the potential of clean room that they are establishing clean room teams. These teams will compete with other groups and other methodologies within these organizations. As this experience base broadens, research will continue to refine the process and the statistical measures.

Human verification can replace debugging in software development.

Although the collective experience with clean room projects is not yet broad enough to be itself statistically analyzed, the anecdotal evidence is compelling. Compilation without errors the first time the code is taken to the computer is common. Jobs can be done with one-third the number of lines of code required by other efforts. Where comparable data have been available, errors in code prior to first execution have been reduced by a factor of 25. Errors in released code have been reduced by nearly two orders of magnitude. Project turnaround has been halved by experienced teams. As a rule of thumb, clean room quality is achieved if the error rate in statistical testing is less than five errors per 1,000 lines of code the first time the code goes to the machine.

Such performance gives management a powerful tool. Business decisions that depend upon significant software development are among the most distressing decisions faced by business leaders. To be able to schedule a project in useful increments, to contract for a measurable and auditable level of reliability, and to have each increment available on schedule and certified will surely make software-dependent decisions much sharper.

References

1. J.H. Poore, "Derivation of Local Software Quality Metrics (Software Quality Circles)," *Software Practice and Experience* (to be published).
2. J.W. Duran and S.C. Ntafos, "An Evaluation of Random Testing," *IEEE Transactions on Software Engineering*, January 1986, pp. 3-11.
3. R.W. Madsen and J.E. Holstein, "Determining Sample Size When Searching for Rare Items," *IEEE Transactions on Reliability*, December 1982, pp. 451-454.
4. H.D. Mills, R.C. Linger, and A. Hevner, *Principles of Information Systems Analysis and Design* (New York: Academic Press, 1986).
5. R.C. Linger, H.D. Mills, and B.I. Witt, *Structured Programming: Theory and Practice* (Reading, MA: Addison-Wesley, 1979).
6. Some evidence that such reasoning is effective in very large systems designed top-down with functional verification is given in A.J. Jordano, "DSM Software Architecture and Development," *IBM Technical Directions*, Vol. 10, No. 3 (1984), pp. 17-28.
7. A. Currit, M. Dyer, and H.D. Mills, "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering*, SE-12, 1, January 1986, pp. 3-11.
8. H.D. Mills, M. Dyer, and R.C. Linger, "Cleanroom Software Engineering," *IEEE Software*, September 1987, pp. 19-25.
9. C.K. Cobb, *Selecting a Software Reliability Model Based on Failure Data Characteristics*, master's thesis, University of Tennessee, Knoxville, 1988.
10. R.W. Selby, V.R. Basili, and F.T. Baker, "Cleanroom Software Development: An Empirical Evaluation," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 9, September 1987.

Harlan D. Mills is the director of the Information Systems Institute, Vero Beach, FL. He holds a PhD in mathematics from Iowa State University.

J.H. Poore heads the department of computer science at the University of Tennessee, Knoxville. A member of ASQC, Poore holds a PhD in computer science from Georgia Institute of Technology.