1993

Zero-Defect Software - Cleanroom Engineering

Harlan D. Mills

Contents

Zero Defect Software: Cleanroom Engineering

HARLAN D. MILLS

Florida Institute of Technology
and
Software Engineering Technology, Inc.
Vero Beach, Florida

ISBN 0-12-012136-0

1. Background and Introduction

Software is either *correct* or *incorrect* in *design* to a *specification*, in contrast with hardware, which is reliable to a certain level in performing to a correct design. Certifying the correctness of such software requires two conditions, namely:

1. *Statistical testing* with inputs characteristic of actual usage, and
2. No failures in the testing.

If any failures arise in testing or subsequent usage, the software is incorrect, and the certification is invalid. If such failures are corrected, the certification process can be restarted, with no use of previous testing results. Such corrections may or may not lead to additional failures. So, certifying the correctness of software is an empirical process that is bound to succeed if the software is indeed correct and may appear to succeed for some time if the software is incorrect.

Cleanroom Engineering introduces new levels of practical precision for achieving correct software, using three engineering teams. First, one team of *specification engineers* creates formal specifications and breaks them into increments for development and certification. Next, another team of *development engineers* creates software to the specifications of these increments with formal verification, but without testing or debugging. Finally, another team of *certification engineers* tests and certifies the correctness of growing numbers of increments by stratified statistical testing. Any failures are returned for fixing to the development engineers and for retesting by the certification engineers for a new certification of correctness of the software. A new level of human capability is required in specification engineering, development engineering, and certification engineering, but it is a level that software engineers find possible.

In order to carry out effective software testing and to achieve high reliability, one needs to start with well-specified and well-developed software. Highly reliable performance cannot be tested into poorly developed software. So we will be concerned with the entire software engineering process that culminates in the certification of well-specified and well-developed software.

Software can be developed and certified as correct under statistical quality control to well-formed specifications of user requirements. To be humanly practical in sizable software systems, the specifications must be structured and defined in construction increments that accumulate into the final systems. This ability requires a sound development methodology to create software that is easily testable by engineering design and mathematical verification, in particular with no unit testing at all by the developers. Unit

testing and fixing of informally developed code is the most error-prone activity in software development today, leading to deeper failures in 15% or more of the fixes.

This ability also requires a test methodology based not only on the function and performance specifications, but also on the usage specifications, namely how critical each test case is to assessing the practical correctness of system behavior. Such a test methodology must be based on a stratified statistical strategy derived from the statistics of usage and the importance of the usage expected for the software. For an important case, a stratus may even consist of a single case (with probability 1), or may consist of a small subset of cases, on out to strata containing large sections of the software. A test design defines each stratus (possibly hundreds or thousands) and the number of tests in each one. Testing without any failures found leads to certification of correctness of the software or software segment.

If failures are later found, the certification is negated. If failures are fixed, the certification process can be started again. In any case, certification continues with software release to users, moving with confidence from a level of some three sigma at release up to and beyond six sigma with sufficient usage without failures.

Software is either correct or incorrect in design to a specification, in contrast with hardware that is reliable to a certain level in performing to a correct design. For small and regular software, it may be possible to test exhaustively the software to determine its correctness. But software of any size or complexity can only be tested partially, and typically a very small fraction of possible inputs are actually tested. While possibly frustrating at first glance, this is all humans can assert about the correctness of software. But on second glance, the sequential history of certification efforts provides a human basis for assessing the quality of the software and some expectations for achieving future correctness.

1.1 History in Statistical Quality Control

Computer software is little over a human generation old, and software development as it is practiced today has been worked out in just that short time. Think of accounting when it was just a human generation old, whenever and wherever that may have been. It certainly did not have double entry bookkeeping, and not even sound arithmetic methods. Civil engineering did not have right triangles or methods of calculating areas at that stage. Software has many more people than accounting and civil engineering at that time, but fundamental ideas still take time to develop, even though people in the field are making do with what is available.

In another direction, *statistical quality control* (SQC) came into being about a human generation ago, with the work of Dr. Edward Deming and others in manufacturing in the 1950s. However, American industry largely ignored the new ideas of SQC in that period, getting along with however they were dealing (or not dealing) with quality. Statistics seemed like odd and extraneous effort in the industry, and hardly seemed worth doing in manufacturing. Of course, the rest of the story is well known, with Dr. Deming and others taking SQC to Japan with dramatic successes in Japanese industry, creating products with entirely new levels of both quality and productivity. By now American industry has largely caught up with Japanese industry in manufacturing SQC, but it has taken quite a while.

It is now known how to develop software also by using statistical quality control. IBM and the US DOD DARPA STARS Program have supported this basis of SQC in software development. There is a considerable difference in SQC between manufacturing and software. But manufactuing SQC has been very informative and helpful in going to software.

In manufacturing, the design is considered correct and the SQC applies to creating physical products to the design specifications. The design may be wrong for the product, but the job of manufacturing is to meet the design, right or wrong. The physical parts may be slightly incorrect but the product must still meet the design on a physical basis. For example, a wire cannot be cut to a 10 mm length exactly, but say within 0.001 mm, and still meet specifications in the product performance.

Manufacturing under SQC is very different from that under previous controls. For example, in a 1950 assembly line of 20 stations, each station generating parts and adding to the product was producing products at a rapid rate, but many such products might then be found to be defective in the testing that followed. The attempted solution to such problems was to improve the part-making stations, because if each station was producing perfect parts the product would be satisfactory. But while some improvements were indeed made, new products had similar problems no matter how hard people tried.

Manufacturing under SQC used ideas that first seemed strange and of no use. In the assembly line of 20 stations, first work out how each intermediate assembly at each point should perform; in many cases the stations must be redesigned to make this possible. Next; provide statistical measurements for the performances of the intermediate assemblies at each station, and make these measurements right there as each partial product comes down the line. Now, shocking as it may seem, stop the entire assembly line if any partial product fails its performance test. Fix the reason for the failure in whatever preceding stations necessary. All the workers are idle now! What a dumb thing that seems. In the old assembly line everybody worked hard all the

time. But forcing all the parts to be right during assembly created a dramatic improvement in both quality and productivity. The idle workers were a clear motivation for getting the work stations accurate to levels previously unimagined.

So, in retrospect, SQC seemed very strange for manufacturing assembly lines in American industry. Who would think such ideas would be practical? No wonder American industry turned it down in the 1950s. And the objective is not statistics, it is quality control. The reason for statistics is that it is the only way to achieve real quality control. The improvement in productivity is a pleasant surprise, but it becomes understandable when the amount of rework becomes known. It is now understood as unnecessary with better parts work and good management.

1.2 Application of SQC to Software Development

With this background, it is time to apply SQC to software development. However, it is the design that must be produced correctly to meet a software specification. Just as in American manufacturing in the 1950s, American software in the 1990s is created in well-intended ways without SQC. Its performance is low in both quality and productivity compared with what is possible. In a 1990 European conference in Oslo, a Japanese group stated that Japanese companies were moving into SQC as described in this chapter.

But American companies need not bring up the rear this time around in software. Just as in manufacturing SQC 40 years ago, it is not easy for managers and workers of today to move into software SQC. Everyone is so busy, how do they find time to learn the new ideas? It requires new capabilities, but capabilities present in educated and disciplined people. For example, manufacturing workers discovered they could create parts that were orders of magnitude more accurate than previously imagined, with increased productivity. Right now, well-intentioned and experienced programmers imagine that software must have a few failures—say one to five per thousand lines of code—on release, and they cannot imagine a serious objective of creating software with no failures and higher productivity. It is not right to ask programmers to work faster, but to work smarter with real engineering discipline under SQC.

Zero failure software is not possible with heuristic methods of programming used in this first human generation of software development. It is possible with mathematics-based design discipline and statistics-based test discipline as discussed in Mills (1986). Despite the experiences of the first human generation in software development, zero defect software is possible with the use of formal methods of program design and verification. Correctness verification and statistical testing reinforce and complement each other

in surprising ways in achieving zero defects. Design discipline is made possible by the work of Dijkstra, Parnas and Wang (1989), and others. Test discipline is made possible by the work of Poore *et al.* (1990), Whittaker (1992), and others. Such a design discipline was taught in a six-course curriculum in Software Engineering (Linger *et al.*, 1979; Mills *et al.*, 1986) across IBM in the 1980s with a faculty of over 60 well-prepared teachers and over 10,000 students. SEI (Software Engineering Institute at Carnegie Mellon University) can teach and help others teach good design discipline.

Software development has certainly improved in many ways over the past 40 years. It has become better managed, here and there, in dealing with larger and more complex system development and software product problems. Basic technology has improved dramatically, with high-level languages, structured programming, and modular design for uniprogramming. It has not improved as dramatically for uniprogram testing or multiprogram design or testing. But the most deficient activity in software development today is the use of, and dependence on, private unit testing and debugging of software.

It seems unbelievable from the outside that debugging software should be so difficult. But such debugging with a fix for a discovered fault will lead to a new fault at least 15% of the time (Adams, 1980). This number of new faults resulting from fixes has been a major surprise. Many large software systems or products cannot be successfully debugged because of such new faults. For example, the first optimized PL/I compiler, with more than 50 programmers for more than two years, was never released because it could not be debugged. An airline passenger reservation system that involved even more effort and time was never released and resulted in a major loss by the developer. At the moment, there seems no other way to create software than to code, unit test, and debug it the way it has always been done. But major and minor software development failures continue, and there is another way to create software, namely to outlaw private unit testing and debugging, as discussed next.

2. Cleanroom Engineering

As noted, two major properties of Cleanroom Engineering are:

1. No debugging by the developers before the software goes to independent testers, and
2. Statistical testing taking into account both the usage and the criticalness of software parts.

As we discuss next, there are more properties, but these two both seem critical or impossible at first glance compared with how software is developed

today. They are both related to the short history of software of just a single human generation. For example, it took a human generation to discover touch typing for typewriters and it was not easy to make that happen. In another direction, farming today is entirely different than it was a human generation ago. It has become mechanized even more and moved from small one-family farms to larger corporate farms. In a similar way, serious software development will become a large-scale engineering operation rather than an intuitive programming operation.

Cleanroom Engineering develops software of *certified correctness* under *statistical quality control* in a *pipeline of increments* that *accumulate* into the specified software product. In the cleanroom process no *program debugging* is permitted before *independent statistical usage testing* of the increments as they accumulate into the final product (Cobb and Mills, 1990; Dyer, 1992a). The Cleanroom process provides rigorous methods of software specification, *development*, and *certification*, through which disciplined software engineering teams are capable of producing *zero defect* software of arbitrary size and complexity (Whittaker and Poore, 1992). Such engineering discipline is capable not only of producing correct software, but also of the certification of the correctness of the software as specified.

Software is either correct or incorrect in design to a well-defined specification, in contrast to hardware which is reliable to a certain level in performing to a design that is assumed to be correct. For small and regular software, it may be possible to test exhaustively the software to determine its correctness. Even then, failures from human fallibility can be overlooked. But software of any size or complexity can only be tested partially, and typically only a very small fraction of possible inputs can actually be tested. At first glance, the fractions are so small for systems of ordinary size that the task of testing looks impossible. But when combined with mathematical verification, correct software is indeed possible.

For interactive software, the statistical correlation of successive inputs must be treated as well. If any failures arise in testing or subsequent usage, the software is incorrect, and the certification is invalid. If such failures are corrected, the certification process can be restarted, with no use of previous testing results. Such corrections may lead to additional failures, or they may not. So certifying the correctness of software is an empirical process that is bound to succeed if the software is indeed correct and may succeed for some time if the software is incorrect.

2.1 Cleanroom Statistical Quality Control

Cleanroom software engineering achieves statistical quality control over software development by strictly separating the design process from the

testing process in a pipeline of incremental software development. There are three major engineering activities in this process (Linger and Mills, 1988; Mills *et al.*, 1987b):

Software Specification: First, structured architecture and precise specification of a pipeline of software increments that accumulate into the final software product, which includes the statistics of its use as well as its function and performance requirements.

Software Development: Second, box structured design and functional verification of each increment, delivery for testing and certification without debugging beforehand, and subsequent correction of any failures that may be uncovered during certification.

Software Certification: Third, statistical testing and certification of the software reliability for the usage specification, notification to developers of any failures discovered during certification, and subsequent recertification as failures are corrected.

These three activities are defined and discussed in later sections.

As noted, there is an explicit feedback process between certification and development on any failures found in statistical usage testing. This feedback process provides an objective measure of the reliability of the software as it matures in the development pipeline. It does, indeed, provide a statistical quality control process for software development that has not been available in this first human generation of trial-and-error programming.

Humans are fallible, even in using sound mathematical processes in functional verification, so software failures are possible and almost certain during the certification process. But there is a surprising power and synergism between functional verification and statistical usage testing (Dyer, 1992b). First, as already noted, functional verification can be scaled up for high productivity and still leave no more errors than heuristic programming often leaves after unit and system testing combined. Second, it turns out that the mathematical errors left are much easier to find and fix during testing than errors left behind in debugging, by a factor of five as measured in practice (Mills *et al.*, 1987b). Mathematical errors usually turn out to be simple blunders in the software, whereas errors left behind or introduced in debugging are usually deeper in logic or wider in system scope than those fixed. As a result, statistical usage testing not only provides a formal, objective basis for the certification of reliability under use, but also uncovers the errors of mathematical fallibility with remarkable efficiency.

2.2 Zero Defect Software Is Really Possible

In spite of the experiences of this first human generation in software development, zero defect software is really possible. However, there is no foolproof logical way to know that software is zero defect. The proof is in using the product without ever finding any failures. Mathematics is very helpful in creating software that executes with no defects, but it is insufficient by itself to guarantee it. Statistics is also very helpful in creating software that executes zero defect, but is also insufficient by itself. Even so, mathematics and statistics foundations combined are very powerful bases for software engineering with zero defects. Three illustrations of zero defect software are noted next.

First, the U.S. 1980 Census was acquired by a nationwide network system of 20 miniprocessors. The system was controlled by a 25 KLOC program, which operated its entire 10 months in field use with no failure observed. It was developed by Mr. Paul Friday, of the U.S. Census Bureau, using stepwise refinement and functional verification in Pascal. Mr. Friday used formal university courses in software engineering to achieve this feat. He was given the highest technical award of the U.S. Department of Commerce for that achievement.

Second, the IBM wheelwriter typewriter products released in 1984 are controlled by three microprocessors with a 65 KLOC program. It has had millions of users since, with no failures ever detected. The IBM team creating this software also used functional verification and extensive testing in a well-managed software engineering environment to achieve this result. The team had completed pass/fail courses in formal software engineering before entering this project.

Third, the NASA space shuttle software of some 500 KLOC, while not completely zero defect, has been zero defect in all flights. The first space shuttle flight initialization failed because of five computers, three initialized on one time frame, the other two on another time frame. That fault was fixed and did not reappear. The IBM team also used functional stepwise refinement and verification and extensive testing to achieve this result. The space shuttle software is such a large, complex, and visible product that there are real lessons in it. All managers and programmers were required to complete a basic curriculum of six pass/fail courses in understanding programs as rules for mathematical functions, and functional verification of programs and modules (Linger *et al.*, 1979). The team received the highest NASA award for this achievement.

Looking ahead, Hevner and Becker (1992) introduce an integrated development environment based on repository data models that support Cleanroom specification, verification, and certification, as well as incremental

development. Appropriate data models help identify tool requirements for this environment.

3. Statistical Quality Control in Software Engineering

The fundamental concepts of SQC in software versus other fields are significantly different. First, SQC in manufacturing or services assumes a correct design and deals with efforts to realize such a design. In this case the statistics deals with departures of the product from the design. But software is design whose manufacture is practically perfect in program compiling, linkage editing, etc. The design itself is under question and the statistics deals with departures of the design from the specifications that are discovered during testing and use.

Second, software development is a creative human activity only a human generation old, largely carried out today by heuristic, trial-and-error methods. In this first human generation, testing during development has been primarily coverage and ad hoc, with no scientific basis, rather than based on statistical usage that can provide certified correctness. The objective generation of testing on the basis of statistical usage specifications, and the certification of software correctness from such objective statistical testing leads to new human understandings of software development as a rigorous and repeatable engineering process.

Third, we need to work more on certification of correctness by statistical quality control because the other two areas of specification and development are better thought out with many fundamental problems well addressed. Certification of software is in its infancy, so there is much to learn, which will result in many substantial improvements over the present state of software testing. The technological goals are to create an entirely new human capability for developing zero defect software using methods of SQC.

Many people presently believe that software has nothing to do with SQC because software is deterministic, not statistical, with results that are either right or wrong. In these people's view software is bound to have periodic failures, and good software just exhibits fewer failures than poor software. The innovation is to make usage statistics into the foundation for SQC as applied to an imperfect design process rather than to a manufacturing process where the design is considered perfect. The elements of the Cleanroom Engineering process have been shown to be very practical and successful in creating software with zero defects and increased productivity.

In this connection, Dyer (1992b) describes the merger of functional correctness verification in cleanroom engineering with formal inspections often used today. The systematic, stepwise process of functional verification

provides a framework for effective team inspections of software designs. Procedures for conducting inspection-based verification are given in Dyer (1992b).

3.1 Statistical Test Design

In order to certify the correctness of software, it is necessary to define statistical testing that yields meaningful results. Measurements are needed to serve as a basis for the estimation of how well the software as designed and implemented will satisfy its requirements in actual use. In the past few years we have made significant progress in recognizing what measurements need to be made and in developing a program for making and utilizing the measurements.

This testing program must recognize both the statistics of use and the importance of this use. First, failures in execution may be of many degrees of seriousness. Some failures may produce correct data in incorrect formats, but otherwise not affect continued execution. Other failures may produce incorrect data but continue execution correctly. Still other failures may continue execution incorrectly, possibly losing data. The most serious of failures may terminate execution unintentionally and/or lose data. The more serious the failure, the more important it is to find it. For this reason, the criticalness as well as the probabilities of failures must be considered. Some conditions are so critical that they should be tested for sure, with probability 1. At first glance this may not appear to be statistical testing, but it is at the endpoint of a statistical domain.

The usage statistics of a software system or product is a new idea for testing. Usage statistics is often used informally to evaluate designs against performance requirements. For example, relative efficiency required for different responses, say for entering and accessing data, will help define data structures and algorithms required. If accessing data is much more frequent than entering it, perhaps the data should be stored alphabetically and accessed by binary search, while if entering is more frequent than accessing, perhaps the data should be stored in sequence of entry and accessed by linear search.

But usage statistics can be used to make testing more realistic. Well-intentioned testing by ad hoc invention can miss critical areas, for example incorrect entries that are expected occasionally and require recovery operations. Testing by simple uniform probability can be entirely unrealistic if usage statistics is far from uniform. For example, a programming language compiler accepts a remarkably small fraction of all text sequences. So the usage statistics of programs or near programs is important for realistic testing of compiler software. Any one user may not use exactly the usage statis-

tics defined for testing, but a class of users can be used to define a realistic usage statistics expected for them as a class. So the definition of realistic usage statistics is a substantial job that goes beyond functional and performance specifications into the likely usage of the software.

3.2 Markov Chain Techniques for Software Certification

Markov chains have exceptional value in software testing (Whittaker, 1992; Whittaker and Poore, 1992). First, there is a question of how to use a specification as a guide in constructing a Markov chain to model the usage of correct software. Second, the testing process itself is modeled by another Markov chain which includes any possible failures in execution. If the software is correct, these two chains are identical. But if incorrect, the chains are slightly different. While testing proceeds, the reliability and mean time between failures in the second Markov chain can be estimated, both to measure progress and to evaluate the reality for reaching zero defect operations.

As already noted, the certification team must account for both the statistical concerns in testing and the relative importance of various inputs and operations to users. A specific input under a specific condition may happen rarely, but it can be important and therefore critical for testing. As a result, a test design involves a hierarchical structure of tests, each requiring a specification Markov chain, ranging all the way from the entire system to a single important operation. Just as with the software design, this test design requires long-term, systematic research into usage statistics and usage importance.

While unlikely, if all inputs under all conditions are equally important, a single usage, statistics defined–set of tests is sufficient. But more likely, some inputs under some conditions will be more important than others. Today, much of testing is identified with specific cases of known importance, independent of their statistics. These specific cases can be brought into the statistics test designs as special cases. If an important test is defined with probability 1, it is now part of the hierarchical structure that defines the entire test. Usually, inputs and conditions will be partitioned naturally into subsets of the entire input space. Such subsets will themselves be partitioned into smaller subsets, and so on, clear down to single inputs and conditions.

The test analyses will involve successive inputs for which the Markov process provides a good model. It not only matters at what frequencies various inputs and conditions will arise, but also how these frequencies depend on previous inputs (and thereby outputs) with input-to-input frequencies, as well. This requirement for estimating usage frequencies is new, and brings a new level of design to the testing process. Because of the

test design work, the certification team must begin its analysis and design in parallel with the development team.

4. Software Testing in This First Human Generation

4.1 Unit Testing as a Private Activity

In this first generation of software development and maintenance, the primary methods of software specification and design have been heuristic, going from informal and imprecise natural language to formal programming language by trial and error. When programs don't do the right things, was it the specification or the design at fault? If the specification was informal, how can one tell? The specifier may have had the right idea, but never written it down completely. Or the programmer may have met all requirements that were written down and extrapolated differently than the specifier intended.

As already noted, private unit testing and fixing is used without question. What other way makes any sense? It seems so simple and so natural just to get the small errors out. How could that be harmful? It hasn't been suspected that so many new failures are introduced by unit testing and fixing. Unit testing is regarded as a private activity for getting defects out of the small parts of programs before assembling and integrating them into larger parts. Subsequently, more defects are discovered in the larger parts as smaller parts are tied together. The original programmers are often gone by this time. Finally, entire systems are frequently (almost always) delivered with more defects yet to be found. Users frequently find many more defects than the developers believed remained. As a result, many organizations have learned to expose new software products to a select few users for initial shakedown before distributing the products widely.

This first generation of heuristic methods and experiences seem to work; after all, products are built. Advanced software teams do better than others in using the best technical and managerial methods that can be found. And yet, even the advanced software teams stub their toes now and then. In fact, more systems and products than casual observers might imagine are seriously delayed or even abandoned with all the programs written, because they are too error-prone to release. Hundreds of person years may be involved, but the software still cannot be made to work correctly. In more recent times, several major PC upgraded products for word processing and financial analysis have been released more than a year behind their announcements at a major cost to their producing companies.

The strange thing about most such software disasters is that they were not looked on as dangerous undertakings in their beginnings. Of course,

any software effort is a bit risky because computer code is so detailed and programmers are a little unpredictable. But "nobody said it would be so hard at the outset," is a typical comment. New and better heuristics, especially supported by CASE tools, are becoming available in object-oriented methods. With object-oriented methods, better approaches to high-level designs and specifications seem possible. But the software rubber meets the hardware road with the program code, and unit debugging is seldom questioned. At that stage, larger parts and entire products are tested, often with good records kept on the coverage testing, to ensure that every branch is exercised both ways and that all code is tested at least once. Yet in spite of the testing coverage, users often find unexpected levels of failures in operations, making the product marginal or unacceptable.

The barely recognized fact is that unit debugging is the most error-prone activity in software development today. Fixing any failure found is usually successful. But creating a new and deeper failure occurs 15% of the time or more. This occurs because debugging strives to ensure correct outputs, which leads to developers modifying code to produce a correct output, and not modifying the code to produce the correct function. As a result, large failure-filled software systems may never be debugged sufficiently to be released, even though extensive efforts are made.

4.2 A Historical Lesson in Typewriting

These experiences are not surprising in this first human generation of software development. They just seem part of the problem facing people in the field. Or are they? A hundred years ago people faced another set of problems in using the new typewriters, whose practical invention occurred late in the 19th century. How to type text and tabular material without errors at reasonable rates? Typewriters were special machines for special purposes. Executives, even the president of the United States, hand wrote their own letters by and large, and assistants or secretaries did likewise. Typewriters were used to write reports and documents with relatively poor quality reprints compared with printing. They certainly did not replace assembling print for printing machines. Typewriting was error-prone. One had to look at the keys, of course, while typing, so a reasonable way was to memorize the text a sentence at a time. But in going back and forth between the text and the typewriter, small mistakes or lapses were very possible from time to time. Correcting a character, even a word, might not be so bad. Correcting a missed sentence early on a typed page was better fixed by starting the page over.

With this background for almost a human generation of using typewriters, the new idea of *touch typing*, typing without looking at the keys, was a

very strange one. "That's silly. Who could possibly do that?" In teaching typewriting, people who look at the keys can get useful work done in the very first day. In fact they learn practically all there is to know about a typewriter in the very first day, and just need to get more practice and skill by typewriting. In teaching touch typing, people get no useful work done in the first day or the first week. "Why would anyone spend time in such a useless activity?" Of course, we know that touch typing turned out to be an internationally useful method that put typewriters into business offices on a mass basis. Typewriter makers improved their products in many ways, but the reason typewriters were eventually made in such quantities was due to people knowing how to use them well rather than to companies knowing how to make them well.

There is a lesson in touch typing for software development. In software, teaching a programming language and how to compile and execute programs allows people to write programs right away. Very likely, such programs will require considerable debugging, and many text books say just that. With more and more experience in programming alone or in teams, errors and unit debugging are just an accepted and integral part of programming.

But people with the right education and training do not need to unit debug their software any more than people need to look at the keys when they type. Yet, just as in teaching touch typing, much less trial and error programming is done at the beginning in good software education, with much greater emphasis on formal methods in program specification, design, and verification. When serious programming begins only after formal methods, very little debugging will be required, because of more explicit design and verification from good specifications. But "why not let the computers find the errors, why make so much of a simple program?" For simple programs, that may be a perfectly good question. But as programs get larger and more complex, computers don't find the errors, and much more time will be spent debugging than writing the code originally.

4.3 Two Sacred Cows in Software

Software engineering and computer science are relatively new subjects, only a human generation old. In this first generation, two major sacred cows have emerged from the heuristic, error-prone software development of this entirely new human activity—namely program debugging and coverage testing. As noted previously, program debugging before independent usage testing is unnecessary and creates deeper errors in software than are generally found and fixed. It is also a surprise to discover that coverage testing is a very inefficient way of getting reliable software and provides no capability for scientific certification of reliability.

As a first generation effort, it has only seemed natural to debug programs as they are written, and even to establish technical and managerial standards for such debugging. For example, in the first generation in typing, it only seemed natural to look at the keys. Touch typing without looking at the keys must have looked very strange to the first generation of hunt and peck typists. Similarly, software development without debugging before independent, certification testing of user function looks very strange to the first generation of trial and error programmers. It is quite usual for human performance to be surprising in new areas, and software development will prove to be no exception.

Just as debugging programs has seemed natural, coverage testing has also seemed to be a natural and powerful process. Although 100% coverage testing is known to still leave errors behind, coverage testing seems to provide a systematic process for developing tests and recording results in well-managed development. So it comes as a major surprise to discover that statistical usage testing is more than an order of magnitude more effective than coverage testing in increasing the time between failures in use. Coverage testing may, indeed, discover more errors in error-prone software than usage testing, but it discovers errors of all failure rates, while usage testing discovers the high failure rate errors more critical to users.

4.4 The Power of Usage Testing over Coverage Testing

The writings and data of Adams (1980) in the analysis of software testing, and the differences between software errors and failures, give entirely new insights in software testing. Since Adams has discovered an amazingly wide spectrum in failure rates for software errors, it is no longer sensible to treat errors as homogeneous objects to find and fix. Finding and fixing errors with high failure rates produces much more reliable software than finding and fixing just any errors, which may have average or low failure rates.

The major surprise in Adams' data is the relative power of finding and fixing errors in usage testing over coverage testing, a factor of 30 in increasing mean time to failure (MTTF). That factor of 30 seems incredible until the facts are worked out from Adams' data. But it explains many anecdotes about experiences in testing. In one such experience, an operating systems development group used coverage testing systematically in a major revision and for weeks measured mean time to crashes in seconds. It reluctantly allowed user tapes in one weekend, but on fixing those errors, found that the mean time to abends jumped literally from seconds to minutes.

The Adams data is given in Table I (from Adams, 1980). It describes distributions of failure rates for errors in nine major IBM products, including the major operating systems, language compilers, and database systems.

TABLE I

DISTRIBUTIONS OF ERRORS (IN %) AMONG MEAN TIME TO FAILURE (MTTF) CLASSES

	MTTF in K months							
	60	19	6	1.9	0.6	0.19	0.06	0.019
Product								
1	34.2	28.8	17.8	10.3	5.0	2.1	1.2	0.7
2	34.2	28.0	18.2	9.7	4.5	3.2	1.5	0.7
3	33.7	28.5	18.0	8.7	6.5	2.8	1.4	0.4
4	34.2	28.5	18.7	11.9	4.4	2.0	0.3	0.1
5	34.2	28.5	18.4	9.4	4.4	2.9	1.4	0.7
6	32.0	28.2	20.1	11.5	5.0	2.1	0.8	0.3
7	34.0	28.5	18.5	9.9	4.5	2.7	1.4	0.6
8	31.9	27.1	18.4	11.1	6.5	2.7	1.4	1.1
9	31.2	27.6	20.4	12.8	5.6	1.9	0.5	0.0

The uniformity of the failure rate distributions among these very different products is truly amazing. But even more amazing is a spread in failure rates over four orders of magnitude, from 19 months to 5,000 years (60K months) calendar time in MTTF, with about 33% of the errors having an MTTF of 5,000 years, and 1% having an MTTF of 19 months.

With such a range in failure rates, it is easy to see that coverage testing will find the very low failure rate errors a third of the time with practically no effect on the MTTF by the fix, whereas usage testing will find many more of the high failure rate errors with much greater effect. Table II develops the data, using Table I, that shows the relative effectiveness of fixes in usage testing and coverage testing, in terms of increased MTTF. Table II develops the change in failure rates for each MTTF class of Table I, because it is the failure rates of the MTTF classes that add up to the failure rate of the product.

Line 1, Table II, denoted M (MTTF), is repeated directly from Table I, namely the mean time between failures of the MTTF class. Line 2, denoted ED (Error Density), is the average of the error densities of the nine products of Table I, column by column, which represents a typical software product.

TABLE II

ERROR DENSITIES AND FAILURE DENSITIES IN THE MTTF CLASSES OF TABLE I

Property								
M	60	19	6	1.9	0.6	0.19	0.06	0.019
ED	33.2	28.2	18.7	10.6	5.2	2.5	1.1	0.5
ED/M	0.6	1.5	3.1	5.6	8.7	13.2	18.3	26.3
FD	0.8	2.0	3.9	7.3	11.1	17.1	23.6	34.2
FD/M	0	0	1	4	18	90	393	1,800

Line 3, denoted ED/M, is the contribution of each class, on average, in reducing the failure rate by fixing the next error found by coverage testing (1/M is the failure rate of the class, ED is the probability that a member of this class will be found next in coverage testing, so their product, ED/M, is the expected reduction in the total failure rate from that class). Now ED/M is also proportional to the usage failure rate in each class, since failures of that rate will be distributed by just that amount. Therefore, line 3 is normalized to add to 100% in line 4, denoted FD (Failure Density). It is interesting to note that Error Density (ED) and Failure Density (FD) are almost reverse distributions, Error Density being about a third at the high end of MTTFs and Failure Density being about a third at the low end of MTTFs. Finally, line 5, denoted FD/M, is the contribution of each class, on average, in reducing the failure rate by fixing the next error found by usage testing.

The sums of the two lines ED/M and FD/M turn out to be proportional to the decrease in failure rate from the respective fixes of errors found by coverage testing and usage testing, respectively. Their sums are 77.3 and 2,306, with a ratio of about 30 between them. That is the basis for the statement of their relative worth in increasing MTTF. It seems incredible at first glance, but that is the number!

To see this in more detail, consider first the relative decreases in failure rate R in the two cases:

Fix next error from coverage testing

$$R \rightarrow R - (\text{sum of ED/M values})/(\text{errors remaining})$$
$$= R - 77.3/E.$$

Fix next error from usage testing

$$R \rightarrow R - (\text{sum of FD/M values})/(\text{errors remaining})$$
$$= R - 2{,}306/E.$$

Next, the increase in MTTF in each case will be

$$1/(R - 77.3/E) - 1/R = 77.3/[R * (E * R - 77.3)]$$

and

$$1/(R - 2{,}306/E) - 1/R = 2{,}306/[R * (E * R - 2{,}306)].$$

In these expressions, the numerator values 77.3 and 2,306 dominate, and the denominators are nearly equal when $E * R$ is much larger than 77.3 or 2,306 (either $77.3/(E * R)$ or $2{,}306/(E * R)$ is the fraction of R reduced by the next fix and is supposed to be small in this analysis). As noted previously, the ratio of these numerators is about 30 to 1, in favor of the fix with usage testing.

5. What Is Cleanroom Engineering of Software?

5.1 Cleanroom Engineering Process

The *Cleanroom Engineering* process develops software of *certified correctness* under statistical quality control in a pipeline of increments, with *box structured design* and *functional verification* but no program debugging permitted before independent statistical usage testing of the increments. It provides rigorous methods for software specification, development, and certification that are capable of producing low or zero defect software of arbitrary size and complexity. Box structured design is based on a Parnas *usage hierarchy* of modules. Such modules, also known as data abstractions or objects, are described by a set of operations that may define and access internally stored data. Functional verification is based on the fact that any program or program part is a *rule* for a *mathematical function.* It may not be the function desired, but it is a function.

The term Cleanroom is taken from the hardware industry to mean an emphasis on preventing errors, rather than allowing errors to appear and removing them later (of course any errors introduced should be removed). Cleanroom Engineering of software involves rigorous methods that enable greater control over both product and process. The Cleanroom process not only produces software of high correctness and high performance, but does so while yielding high productivity and meeting schedules. The intellectual control provided by the rigorous Cleanroom process allows both technical and management control.

Cleanroom Engineering achieves statistical quality control over software development by strictly separating the design process from the testing process in a pipeline of incremental software development. There are three major engineering activities in the process (Linger and Mills, 1988; Mills *et al.*, 1987b):

Specification: First, a specification team creates an incremental specification that defines a pipeline of software increments that accumulate into the final software product, which includes the statistics of its use as well as its function and performance requirements.

Development: Second, a development team designs and codes increments specified using box structured design and functional verification of each increment, with delivery to certification with no debugging beforehand, and provides subsequent correction for any failures that may be uncovered during certification.

Certification: Third, a certification team uses statistical testing and analysis for the certification of the software correctness to the usage specification,

notification to designers of any failures discovered during certification, and subsequent recertification as failures are corrected.

As noted, there is an explicit feedback process between certification and development on any failures found in statistical usage testing. This feedback process provides an objective measure of the correctness of the software as it matures in the development pipeline. It does, indeed, provide a statistical quality control process for software development that has not been available in this first human generation of trial and error programming.

5.2 Cleanroom Engineering Methods

Cleanroom Engineering provides a set of rigorous methods for software development under statistical quality control, based on sound mathematical and statistical principles. While millions of people are involved in software, most of them regard software development as an intuitive, heuristic activity. They do not imagine software engineering as a mathematics-based subject with complete rigor being possible. But software engineering should be and can be a mathematics-based activity. When mathematical rigor is applied, both quality and productivity increase. Nor can they imagine software engineering based on statistics since computers are completely deterministic in behavior. And yet the usage of software is statistical in nature.

For software engineering, being mathematics-based does not mean being numbers-based. Numbers are part of mathematics, but the finite basis of computers adds complexity to dealing with numbers. With integers, computers face overflow possibilities that need to be assured against. With real numbers, computers face roundoff problems, so arithmetic becomes approximate, not exact. In these cases, software must deal with computer operations, not with ideal numerical operations. But mathematics deals with any operations performed by computers, not simply approximate numerical operations. Fortunately, the nonnumerical operations are typically exact in computers, for example logic operations, even text processing operations, so their mathematical basis is very solid. At first glance, nonnumerical operations may not look mathematical, but they are. Logic, set theory, and function theory are clearly nonnumerical mathematics, but sorting theory, text processing theory, and graph theory can also be framed as mathematics as well.

Software is a human generation old, while mathematics is many human generations old. Although not understood early or widely, software has direct mathematical foundations because of the very deterministic behavior of computers. A *computer program* is a *rule* for a *mathematical function*, mapping all possible initial states into final states. Such functions are very

complex compared with functions in physical science and engineering, and traditional mathematical notation is very insufficient. But sufficient mathematical notation is emerging in computer science and software engineering for dealing with the syntax and semantics of programs and their functions.

As an example of deep and useful mathematics, place notation and long division moved arithmetic from error-prone operations on whole numbers to rigorous methods a numerical place at a time a thousand years ago in the Western world. As a result, school children today can out perform Archimedes and Euclid in arithmetic. Similar movement is possible in software today. Place notation and long division look pretty simple today, but it has taken hundreds of years to arrive at this simple form. For example, it took the Italian business world several hundred years to move from Roman numerals to arabic numbers in practice.

Statistics is another subject of longer professional development than software. But only a hundred years ago, statistics was intuitive and heuristic, even though rigorous arithmetic was used in creating sums and averages. Yet in this time, statistics has become a rigorous, mathematics-based subject, often finding counterintuitive results using statistics in specific topics. For example, the agriculture industry of the Western world has been greatly improved by the effective use of statistics in both plant selection and treatment. The application of statistics now makes it possible for software developers to predict with confidence the quality level of the software when it is fully developed quite early in the development life cycle.

Cleanroom Engineering not only puts software development under statistical quality control, but takes out debugging from the list of developer activities, instead using mathematical reasoning before independent testing and certification. Just as typists looked at the keys when typewriters first came out, programmers have felt the need to debug programs in this first human generation of programming. But while counterintuitive at the time, typists went to touch typing with both higher productivity and fewer errors. In the same way, well-educated software engineers can create software with no execution or debugging before it is tested by independent test and certification engineers with the product having higher productivity and much greater quality than previously.

5.3 Dealing with Human Fallibility

Humans are fallible, even in using sound mathematical processes in functional verification, so finding software failures is possible during the certification process. But there is a surprising *power* and *synergism* between functional verification and *statistical usage testing* (Mills *et al.*, 1987b). First, as already noted, functional verification can be scaled up for high produc-

tivity and still leave no more errors than heuristic programming often leaves after unit and system testing combined. Second, it turns out that the mathematical errors left are much easier to find and fix during testing than errors left behind in debugging, measured at a factor of five in practice (Mills *et al.*, 1987b). Mathematical errors usually turn out to be simple oversights in the software, whereas errors left behind or introduced in debugging are usually deeper in logic or wider in system scope than those fixed. As a result, statistical usage testing not only provides a formal, objective basis for the certification of correctness under use, but also uncovers the errors of mathematical fallibility with remarkable efficiency.

In Cleanroom Engineering a major discovery is the ability of well-educated and motivated people to create nearly defect-free software before any execution or debugging, with many fewer than five defects per thousand lines of code. Such code is ready for usage testing and certification with no unit debugging by the designers. In this first human generation of software development it has been counterintuitive to expect software with so few defects at the outset. Typical heuristic programming leaves 50 defects per thousand lines of code, then reduces that number to five or fewer by debugging. The problem is that for programmers with good capabilities and intentions, it seems on the surface that unit debugging makes complete correctness on first coding unnecessary. But the unknown result is the number of faults, over 15%, created in even the simple seeming fixes.

The mathematical foundations for Cleanroom Engineering come from the deterministic nature of computers themselves. As noted, a computer program is no more and no less than a rule for a mathematical function (Linger *et al.*, 1979; Mills, 1975). Such a function need not be numerical, of course, and most programs do not define numerical functions. But for every legal input, a program directs the computer to produce a unique output, whether correct as specified or not. And the set of all such input–output pairs is a mathematical function. A more intuitive way to view a program in this first generation is as a set of instructions for specific executions with specific input data. While correct, this view misses a point of reusing well-known and tested mathematical ideas, regarding computer programming as new and private art rather than more mature and public engineering.

With these mathematical foundations, software development becomes a process of constructing rules for functions that meet required specifications, which need not be a trial and error programming process. The functional semantics of a structured programming language can be expressed in an algebra of functions with function operations corresponding to program sequence, alternation, and iteration (Linger *et al.*, 1979). The systematic top down development of programs is mirrored in describing function rules in terms of algebraic operations among simpler functions, and their rules in

terms of still simpler functions until the rules of the programming language are reached. It is a new mental base for most programmers to consider the complete functions needed, top down, rather than computer executions for specific data.

Trammel *et al.* (1992) discuss the practical realities in adopting the Cleanroom process in software organizations. They define a three-phase process for introducing Cleanroom in a software development organization, including management commitment and team ownership as critical success factors.

5.4 Cleanroom Experiences

The IBM COBOL Structuring Facility (IBM COBOL/SF), a complex product of some 80K lines of PL/I source code, was developed in the Cleanroom discipline, with box-structured design and functional verification but no debugging before usage testing and certification of its correctness. A version of the U.S. A.F. HH60 (helicopter) flight control program of over 30 KLOC was also developed using Cleanroom. The Coarse/Fine Attitude Determination Subsystems (CFADS) of the UARS Attitude Ground Support System (AGSS) of some 30 KLOC has been developed with Cleanroom at NASA.

The IBM COBOL/SF converts an unstructured COBOL program into a structured one of identical function. It uses considerable artificial intelligence to transform a flat structured program into one with a deeper hierarchy that is much easier to understand and modify. The product line was prototyped with Cleanroom discipline at the outset, then individual products were generated in Cleanroom extensions. In this development, several challenging schedules were defined for competitive reasons, but every schedule was met.

The COBOL/SF products have high function per line of code. The prototype was estimated at 100 KLOC by an experienced language processing group, but the Cleanroom developed prototype was 20 KLOC. The software was designed not only in structured programming, but also in structured data access. No arrays or pointers were used in the design; instead, sets, queues, and stacks were used as primitive data structures (Mills and Linger, 1986). Such data-structured programs are more reliably verified and inspected, and also more readily optimized with respect to size or performance, as required.

COBOL/SF, Version 2, consists of 80 KLOC, 28 KLOC reused from previous products, 52 KLOC new or changed, designed and tested in a pipeline of five increments (Linger and Mills, 1988), the largest over 19 KLOC. A total of 179 corrections were required during certification, fewer than 3.5 corrections per KLOC for new code with no developer execution, fewer than 2 corrections per KLOC for all code. The productivity of

the development was 740 LOC per staff month, including all specification, design, implementation, and management, in meeting a very short deadline.

The HH60 flight control program was developed on schedule. Programmers' morale went from quite low at the outset ("why us?") to very high on discovering their unexpected capability in accurate software design without debugging. The 12 programmers involved had all passed the pass/fail coursework in mathematical (functional) verification of the IBM Software Engineering Institute, but were provided a week's review as a team for the project. The testers had much more to learn about certification by objective statistics (Currit *et al.*, 1986).

The subsystem Coarse/Fine Attitude Determination System (CFADS) of the NASA Attitude Ground Support System (AGSS) of some 30 KLOC was developed in Fortran. Sixty-two percent of the subroutines, which averaged 258 source lines each, compiled correctly the first time, with but one of the rest compiled correctly on the second attempt. Compared with well-measured related systems, the failure rate was down by a factor of five while the productivity was up by 70% (Kouchakdjian *et al.*, 1989).

V. R. Basili and F. T. Baker introduced Cleanroom ideas in an undergraduate software engineering course at the University of Maryland, assisted by R. W. Selby. As a result, a controlled experiment in a small software project was carried out over two academic years, using 15 teams with both traditional and Cleanroom methods. The result, even on first exposure to Cleanroom, was positive in the production of reliable software, compared with traditional results (Selby *et al.*, 1987).

Cleanroom projects have been carried out at the University of Tennessee, under the leadership of J. H. Poore (Mills and Poore, 1988) and at the University of Florida under H. D. Mills. At Florida, seven teams of undergraduates produced uniformly successful systems for a common structured specification of three increments. It is a surprise for undergraduates to consider software development as a serious engineering activity using mathematical verification instead of debugging, since software development is typically introduced primarily as a trial-and-error activity with no real technical standards.

6. Box Structured Software System Design

Box structured design is based on a *Parnas* usage hierarchy of modules (Parnas, 1972, 1979). Such modules, also known as data abstractions or objects, are described by a set of operations that may define and access internally stored data. In Ada, such modules are defined as *packages*, with operations defined by the calls of the procedures and functions of the packages, and internal data declared in the package.

Stacks, queues, and sequential or random access files provide simple examples of such modules or packages. Part of their discipline is that internally stored data cannot be accessed or altered in any way except through the explicit operations of the package. It is critical in box structured design to recognize that packages exist at every level from complete systems to individual program variables. It is also critical to recognize that a verifiable design must deal with a usage hierarchy rather than a parts hierarchy in its structure. A program that stores no data between invocations can be described in terms of a parts hierarchy of its smaller and smaller parts, because any use depends only on data supplied to it on its call with no dependence on previous calls. But each call to a specific realization of a package, say a queue, will depend not only on the present call and data supplied to it, but also on previous calls and data supplied then.

The parts hierarchy of a structured program identifies every sequence, alternation, and iteration (say every begin-end, if-then-else, while-loop) at every level. It turns out that the usage hierarchy of a system of packages (say an object-oriented design with all objects identified) also identifies every call (use) of every operation of every package. The semantics of the structured program are defined by a mathematical function for each sequence, alternation, and iteration in the parts hierarchy. That doesn't quite work for the operations of packages because of usage history dependencies. But there is a simple extension for packages that does work. It is to model the behavior of a package as a *state machine*, with its calls of its several operations as inputs to the common state machine. Then the semantics of such a package is defined by the *transition function* of its state machine (with an initial state). When the operations are defined by structured programs, the semantics of packages becomes a simple extension of the semantics of structured programs.

Deck *et al.* (1992) introduce a taxonomy of black box semantics based on interactive properties of the system to be specified. They define three classes of semantics to specify systems of increasing complexity in their interactions with other systems in the execution environment. The semantics extend to interactive and concurrent system specifications.

6.1 The Basis for Box Structured Design

While theoretically straightforward, the practical design of systems of Parnas modules (object-oriented systems) in usage hierarchies can seem quite complex on first exposure. It seems much simpler to outline such designs in parts hierarchies and structures, for example in data flow diagrams, without distinguishing between separate usages of the same module. While that may seem simpler at the moment, such design outlines are incomplete and often

lead to faulty completions at the detailed programming levels. In spite of their common use in this first human generation of system design, data flow diagrams should only be used within rigorous design methods rather than leaving critical requirements to details with incomplete specifications.

In order to create and control such designs based on usage hierarchies in more practical ways, their box structures provide standard, finer grained subdescriptions for any package of three forms, namely as *black boxes*, as *state boxes*, and as *clear boxes*, defined as follows (Mills, 1988; Mills *et al.*, 1986, 1987).

Black Box: External view of a Parnas package, describing its behavior as a mathematical function from historical sequences of stimuli to its next response.

State Box: Intermediate view of a Parnas package, describing its behavior by use of an internal state and internal black box with a mathematical function from historical sequences of stimuli and states to its next response and state, and an initial internal state.

Clear Box: Internal view of a Parnas package, describing the internal black box of its state box in a usage control structure of other Parnas packages; such a control structure may define sequential or concurrent use of the other packages.

Box structures enforce completeness and precision in design of software systems as usage hierarchies of Parnas packages. Such completeness and precision lead to pleasant surprises in human capabilities in software engineering and development. The surprises are in capabilities to move from system specifications to design in programs without the need for unit/package testing and debugging before delivery to system usage testing. In this first generation of software development, it has been widely assumed that trial-and-error programming, unit testing, and debugging were necessary. But well-educated, well-motivated software professionals are indeed capable of developing software systems of arbitrary size and complexity without program debugging before system usage testing (Anderson and Goodman, 1957).

Fetzer and Poore (1992) introduce techniques for using the Z notation in defining box structures using the set theoretic and predicate calculus constructs defined in Z. Z provides a rigorous, formal language for the inner syntax of black box and state box forms. They introduce the integration of box structures and Z notation in a miniature specification.

In Rosen *et al.* (1992), Rosen introduces general design language selection criteria based on the design and verification requirements of cleanroom

software development. Syntactic and semantic language requirements are described for disciplined control and data structures, for well-defined intended functions, and for function theoretic proof rules for verification as described. In the definition of the Design C language, a specialization of Z is given in terms of these requirements.

Fuhrer *et al.* (1990) describe some cleanroom tools, including the Development Assistant, Certification Assistant, and Management Assistant CASE tools for supporting cleanroom operations. A summary is given of the cleanroom development of these tools themselves through seven code increments, including metrics from design, verification, and statistical quality certification.

6.2 Stepwise Refinement and Verification of Software

Once the design is complete, the clear box at each level is expanded to code to implement fully the defined function rule for the black box function at that level by stepwise refinement, as introduced by Wirth (1971). Following each expansion, functional verification is used to help structure a proof that the expansion correctly implements the specification. The nature of the proof revolves around the fact that a program is a rule for a function and the specification for the program is a relation or function. What must be shown in the proof is that the rule (the program) correctly implements the relation or function (the specification) for the full range of the specification and no more. Linger, Mills and Witt (1979) have developed a correctness theorem that defines what must be shown to prove that a program is equivalent to its specification for each of the structured programming language constructs. The proof strategy is subdivided into small parts which easily accumulate into a proof for a large program. Experience indicates that people are able to master these ideas and construct proof arguments for very large software systems.

The development team expands each clear box in the usage hierarchy into the selected target code using stepwise refinement and functional verification. As the development team designs and implements the software, it is held collectively responsible for the quality of the software.

In describing the activities of software development, no mention is made of testing or even of compilation. The cleanroom development team does not test or even compile. They use mathematical proofs (functional verification) to demonstrate the correctness of programming units. Testing and measuring failures by program execution is the responsibility of the certification team.

6.3 The Mathematical Basis for Functional Verification

As noted, any program or program part is a rule for a mathematical function. It may not be the function desired, but it is a function. In structured programs, the rules are direct in form, building program rules out of just two function building operations: first, *function composition*, which corresponds to sequential execution of program parts, and second, *disjoint function union*, which corresponds to alternative execution of one program part or another, as in if/or case structures. Program iteration uses no more than these two operations together, and function recursion provides a useful view of an iteration process.

Any program part or total program defines a single, possibly complex function. The function is seldom a numerical function in classical terms. Even numerical programs must deal with finite sets of numbers in which overflow and roundoffs depart from classical number systems. Given the text or name of a program or program part in whatever language, say a program called Alpha in Ada defined by a set of external packages Gamma and an internal procedure called Beta

```
Alpha = with Gamma;
        procedure Beta
        is
        . . .
        begin
        . . .
        end Beta;
```

the *program function* will be denoted by brackets [ ] around the name or text, such as

```
[Alpha] = [with Gamma;
          procedure Beta
          is
          . . .
          begin
          . . .
          end Beta;]
```

In this case [Alpha] is a set of ordered pairs

$$[\text{Alpha}] = \{\langle X, Y\rangle \mid \text{Given initial state } X, \text{ Alpha will produce final state } Y\}$$

If Alpha loops indefinitely, or does not terminate for some other reason, for some entry state, that state is not part of [Alpha]. The function [Alpha] is

determined by Ada text, but is independent of the language Ada. In this case, Alpha is a rule for the function [Alpha], but there are many rules for a single function. The same function can be defined by a rule in Fortran text, COBOL text, etc., even machine code.

6.4 Functional Verification of Program Parts

From programs to program parts, starting with simple assignment statements, such as

$$x := y;$$

in Ada, the *program part function*

$$[x := y;]$$

takes its initial data state to its final data state. If legal, it will change the value of x in the final state to the value of y in the initial state and change no other values of variables in the initial state. If illegal, the final state may be quite different from the initial state, possibly with both x and y disappearing, as well as other variables, in terminating the entire program execution. So assignment statements have simple function parts when legal, but possibly more complex function parts when illegal. In summary, the function $[x := y]$ is a set of ordered pairs with second members determined uniquely by the first members

$$[x := y;] = \{\langle\langle x, y, \ldots\rangle, \langle y, y, \ldots\rangle\rangle \mid x := y; \text{ is legal}\}$$
$$\cup \{\langle\langle x, y, \ldots\rangle, \langle ???\rangle\rangle \mid x := y; \text{ is illegal}\}$$

where ??? will be determined by other aspects of the initial state. Illegal situations will be suppressed in what follows for the sake of time. In more direct function notation, dealing only with the legal situation,

$$[x := y;](\langle x, y, \ldots\rangle) = \langle y, y, \ldots\rangle$$

in which the function *argument* $\langle x, y, \ldots\rangle$ produces the function *value* $\langle y, y, \ldots\rangle$.

Next, for a sequence of statements, such as

$$x := y;\ y := z;\ z := x;$$

in Ada, the part function

$$[x := y;\ y := z;\ z := x;]$$

will alter values of x, y, and z as a composition of the three individual assignment functions

$$[x := y;] * [y := z;] * [z := x;].$$

That is, beginning with an initial state as argument, the first assignment function gives a new state as value

$$[x := y;](\langle x, y, z, \ldots \rangle) = \langle y, y, z, \ldots \rangle$$

the second assignment function uses this value as an argument

$$[y := z;](\langle y, y, z, \ldots \rangle) = \langle y, z, z, \ldots \rangle$$

and the third assignment function uses this last value as argument

$$[z := x;](\langle y, z, z, \ldots \rangle) = \langle y, z, y, \ldots \rangle$$

That is, the composition function is a nested set of simpler functions that evaluate as

$$\begin{aligned}
&([x := y;] * [y := z;] * [z := x;])(\langle x, y, z, \ldots \rangle) \\
&\quad = [z := x;]([y := z;]([x := y;](\langle x, y, z, \ldots \rangle))) \\
&\quad = [z := x;]([y := z;](\langle y, y, z, \ldots \rangle)) \\
&\quad = [z := x;](\langle y, z, z, \ldots \rangle) \\
&\quad = \langle y, z, y, \ldots \rangle
\end{aligned}$$

as worked out just before. In summary, this composition function will interchange the values of y and z and leave x with the initial value of y, not changing any other data in the initial state.

Finally, for an alternation statement, such as

if $x > y$ then $y := z$; else $x := z$ end if;

in Ada, the part function will execute either the then part or the else part, so that

$$\begin{aligned}
&[\text{if } x > y \text{ then } y := z; \text{ else } x := z; \text{ end if};] \\
&\quad = (x > y \rightarrow [y := z;] \,|\, x \Leftarrow y \rightarrow [x := z;]) \\
&\quad = [y := z; | x > y] \cup [x := z; | x \Leftarrow y]
\end{aligned}$$

where the expression $[y := z; | x > y]$ means the function $[y := z;]$ with its domain restricted to the condition $x > y$. That is, the part function is a union of disjoint functions.

7. Statistical Quality Control

Software is either correct or incorrect in design to a specification, in contrast to hardware which is reliable to a certain level in performing to a correct design. For small and regular software, it may be possible to test exhaustively the software to determine its correctness. But software of any size or complexity can only be tested partially, and typically a very small fraction of possible inputs are actually tested. Certifying the correctness of such software requires two conditions, namely

1. statistical testing with inputs characteristic of actual usage, and
2. no failures in the testing.

For interactive software, the statistical correlation of successive inputs must be treated as well. If any failures arise in testing or subsequent usage, the software is incorrect, and the certification is invalid. If such failures are corrected, the certification process can be restarted, with no use of previous testing results. Such corrections may lead to additional failures, or may not. So certifying the correctness of software is an empirical process that is bound to succeed if the software is indeed correct and may succeed for some time if the software is incorrect. While possibly frustrating at first glance, this is all humans can assert about the correctness of software. But on second glance, the sequential history of certification efforts provides a human basis for assessing the quality of the software and expectations for achieving future correctness.

The statistical foundations for cleanroom engineering come from adding usage statistics to software specifications, along with function and performance requirements (Cobb and Mills, 1990; Mills *et al.*, 1987b; Whittaker and Poore, 1992). Such usage statistics provide a basis for measuring the correctness of the software during its development, and thereby measuring the accuracy of the design in meeting functional and performance requirements. A more usual way to view development in this first generation is as a difficult-to-predict art form. Software with no known errors at delivery frequently experiences many failures in actual usage.

7.1 Precision Specifications

In this first human generation of software development, most of the progress and discipline has been discovered in the latter parts of the life cycle, first in coding machine programs in higher level languages, then in areas such as structured programming and object-oriented design. Problems

in requirements analyses and specifications are more difficult. Defining precisely what is needed and what should be provided by software is more general and difficult than simply producing working software in hopes that it will be satisfactory on trial by users. Even when specifications are required, they are frequently provided in informal, natural languages with considerable room for misunderstandings between designers and users, and with gaps in exact details in which programming misinterpretations are possible and likely.

Precision specifications require formal languages, just as programming does. In the case of programming the need is very obvious because computer machine languages are formal. But as systems become more complex and are used by more people with more critical impacts on business, industry, and government institutions, the need for formal languages for specifications becomes clearer. New programming languages have improved primarily in their abilities to provide explicit structure in data and procedure. For example, Ada has no more capability in defining machine operations than Fortran or COBOL. But it has more explicit design structures for people to use, for example in packages for data abstractions or objects. Specification languages also need explicit structures for the same reason, to allow people to express requirements as directly as possible.

Regardless of the language, formal or informal, a functional specification defines not only legal system inputs, but legal input histories, and for each legal input history, a set of one or more legal outputs. Such legal input histories may be defined in real time systems in which real time is a critical factor, and the outputs given real time requirements as well. Illegal inputs and histories may be treated in various ways, from ignoring them to attempts to decipher or correct them. Any definite treatments of illegal inputs or histories become part of the specification as well. The abstraction of any such functional specification, in any language, is a mathematical relation—a set of ordered pairs whose first members are input histories and whose second members are outputs. Then, there is a very direct and simple mathematical definition for a program meeting a specification. It is that the function defined by the program determines a value for every argument in the domain of the specification relation and that this value be associated with that argument in the relation (Mills, 1986; Mills *et al.*, 1987a).

In cleanroom software engineering, precision specifications are extended in two separate ways to create a structured architecture. First, the functional specifications are designed as a set of nested subspecifications, each a strict subset of the preceding subspecification. Then, beginning with the smallest subspecification, a pipeline of software increments is defined with each step going to the next larger subspecification (Mills *et al.*, 1987b). Second, the usage of the functional specifications is defined as a statistical distribution

over all possible input histories (Dyer, 1992a; Whittaker and Poore, 1990). The structured architecture makes statistical quality control possible in subsequent incremental software development to the functional specifications. The usage statistics provide a statistical basis for testing and certification of the reliability of the software in meeting its specifications.

The creation of a structured architecture defines not only what a software system is to be when it is finished, but also a construction plan to design and test the software in a pipeline of subsystems, step-by-step. The pipeline must define step sizes that the design group can complete without debugging prior to delivery to the certification group. Well-educated and disciplined design groups may handle step sizes up to 20,000 lines of high level code. But the structured architecture must also determine a satisfactory set of user executable increments for the development pipeline of overlapping design and test operations.

7.2 Statistical Certification

As each specified increment is completed by the designers, it is delivered to the certifiers, combined with preceding increments, for testing based on usage statistics. As noted, the cleanroom architecture must define a sequence of nested increments that are to be executed exclusively by user commands as they accumulate into the entire system required. Each subsequence represents a subsystem complete in itself, even though not all the user function may be provided in it. For each subsystem, a certified reliability is defined from the usage testing and failures discovered, if any.

The COBOL Structuring Facility consisted of 80 KLOC, 28 KLOC reused from previous products, 52 KLOC new or changed, designed and tested in a pipeline of five increments (Kouchakdjian, 1989), the largest over 19 KLOC. A total of 179 corrections were required during certification, under 3.5 corrections per KLOC for code with no previous execution. The productivity of the development was 740 LOC per person/month, including all specification, design, implementation, and management, in meeting a very short deadline.

Cleanroom statistical certification of software involves, first, the specification of *usage statistics* in addition to function and performance specifications. Such usage statistics provide a basis for assessing the correctness of the software being tested under expected use. As each specified increment is completed by the designers, it is delivered to the certifiers, who combine it with preceding increments, for testing based on usage statistics. As noted, the cleanroom architecture must define a sequence of nested increments that are to be executed exclusively by user commands as they accumulate into the entire system required. Each subsequence represents a subsystem complete in

itself, even though not all the user function may be provided in it. For each subsystem, a certified correctness is defined from the usage testing and failures discovered, if any.

It is characteristic that each increment goes through a maturation during the testing, becoming more reliable, from corrections required for failures found, serving thereby as a stable base as later increments are delivered and integrated to the developing system. For example, the HH60 flight control program had three increments (Cobb and Mills, 1990; Dyer, 1992a) of over 10 KLOC each. Increment 1 code required 27 corrections for failures discovered in its first appearance in increment 1 testing, but then only 1 correction during increment 1/2 testing, and 2 corrections during increment 1/2/3 testing. Code in increment 2 required 20 corrections during its first appearance in increment 1/2 testing, and 5 corrections during increment 1/2/3 testing. Increment 3 code required 21 corrections on its first appearance in increment 1/2/3 testing. In this case, 76 corrections were required in a system of over 30 KLOC, under 2.5 corrections per KLOC for verified and inspected code, with no previous execution or debugging.

In the certification process, it is not only important to observe failures in execution, but also the times between such failures in execution of usage-representative statistically generated inputs. Such test data must be developed to represent the sequential usage of the software by users, which, of course, will account for previous outputs seen by the users and what needs the users will have in various circumstances. The state of mind of a user and the current need can be represented by a stochastic process determined by a state machine whose present state is defined by previous inputs/outputs and a statistical model that provides the next input based on that present state (Mills *et al.*, 1987b).

7.3 Certification Tasks

In parallel with the cleanroom development team, the cleanroom certification team prepares to certify the software up to and including the increment being developed by the development team. The certification team uses the usage profile and the portion of the specification that is applicable to the increments to be verified to prepare test cases including proper outputs to tests.

When the development team has completed an increment, the certification team creates a version of the *accumulated system* up through this increment. For each version the certification team compiles the increment, combines it with previous increments, and certifies the accumulated system through this version. If failures are encountered in the certification of a version, they are returned to the development team for analysis and for engineering changes

to whatever increments are causing the failures. While failures are likely to be caused by the latest increment added, previous increments may be at fault and changed as well, as noted in the HH60 experience. Each redelivery of changed increments defines a new version of the accumulated system. If no failures are encountered in the certification of a version, no additional versions are required.

Within each version of the accumulating system, tests are constructed at random in accordance with the specified usage statistics profile and then exercised. Test results are compared with a standard and either a failure occurred or the result was correct.

7.4 Certification on a Scientific Basis

Certification of software on a scientific basis requires a statistical usage specification as well as functional and performance specifications. The testing must be carried out by statistical selection of tests from these specifications. Tests selected directly are ad hoc, and give no basis for statistical inference on the correctness of the software. Some uses of the software may be much more important than other uses, and the statistical selections can be given in various levels of stratified sampling. Thus, not only basic statistical usage is to be defined, but the relative importance of correctness for each usage. This is new information that is often not known until the software is put into actual use, but should be generated with functional and performance specifications beforehand.

Next, the actual statistical testing must be carried out when the software is available, possibly in stratified form. One extreme form of stratified form is an important case chosen with probability 1 in that stratus. Next, if a failure is found in testing, the software should be returned to the developers for correction before further testing. When the correction is made, a new start of testing is begun. The *Time to Failure* (TTF) is recorded for each failure discovered. The *Time without Failure* (TWF) is tracked when no failures have appeared. This TWF can be tracked after the software is distributed to users as part of the characterization of its correctness. If failures appear with users, the same rules of correction and restart of TWF should occur.

As already noted, there is a profound difference between the correctness of software and the reliability of hardware. When software has hundreds or thousands of errors, its behaviour may seem to approximate hardware reliability. But when software has under 0.1 failures per KLOC, possibly none, the statistics of hardware failures are not valid. In this first human generation, it has seemed impossible to create zero defect software, but it can be, and has been, done, as will be discussed further. Part of the issue is

discovering a new human possibility, with more engineering education and engineering management. Part of the issue is the economic feasibility. It requires less human effort to produce zero defect software with new methods than error prone software with older methods. The human effort required is both engineering verification and statistical testing, and they complement each other in unexpected ways.

7.5 Usage Testing

A user's specification for a substantial software system will identify various classes of user commands and data for various parts of the system. For example, bringing up an interactive system at the beginning of the day will require and accept certain kinds of user commands and data of which the ordinary interactive users may not even be aware. But bringing the system up is an integral part of the process for a certain class of users. During the day, several distinct classes of users may be interacting simultaneously and independently, such as users adding data to the system, or users making enquiries, or users monitoring the system use and performance. Within each such class, several or many users may be interacting simultaneously and independently, as well.

However, as simultaneously and concurrently as these various users seem to interact with the system, the individual computers in the system each operate strictly sequentially in real time, shifting from one user to another so rapidly that each user gets almost immediate response, even though ten, or a thousand, other users may have been serviced between the user's stimulus and the system's response. As a rule, users are separated from one another by operating in different, relatively protected, data spaces that represent the tasks they are doing. But users can interact, intentionally or not, as their tasks become more intertwined.

For example, in an airline reservation system, a ticket agent may inquire about availability of seats on a given flight and get the response that seats are available. Then when the seats are requested a moment later, the response is that no seats are available. Other users have interacted in picking up the seats in the previous moment. Such system behavior is designed. It would be conceivable to design an airline reservation system such that seats could be held from inquiry to request, but it would require entirely different levels of data storage and processing. In this way, it is clear that user independence is relative, with economic and technical issues involved with multiple users in systems.

This understanding that significant software systems have different kinds of uses applies whether there are single or multiple users. A single user may be using a system in different ways at different times, even within a single

session. The design of the software will typically reflect such different uses by packaging similar operations in common modules. For example, various kinds of data searching may be handled in a search module, but data retrievals may be handled in a different retrieval module. It also makes similar sense to identify similar stimuli response operations in specifications, entirely from the user point of view and state of mind. In particular, complex specifications need to be designed as carefully as programs to reflect the natural structure of the problem being solved and to find effective specification structures that reflect user activities and understandings.

7.6 Software Usage as a Markov Process

As noted, software specifications deal with functional behavior and performance. Functional behavior is ordinarily decomposed into various subfunctions in ways understandable by users, and often obtained from users as requirements. Performance will usually affect design in fundamental ways. But expected usage of the software will have critical impacts on performance issues. For example, a data base system with much more querying than data addition or deletion may call for a design with high performance queries at the expense of data addition and deletion performance. Such a design can be entirely unsatisfactory with different usage. Thus, expected usage statistics can play a key role in software system design.

However, there is another critical use for usage statistics as part of software specifications. It is to permit the certification of software. Software behavior depends not only on how correct the software is but also on how it is used. For every possible state of internally stored data, any command and input data is handled either correctly or incorrectly, denoted as a failure in the latter case at some level of seriousness.

Now, with a statistical usage specification for each possible internal state, the probability of each selection of commands and input data in such a state will be known. Next, the functional specification will define what the new internal state will become, as well as the response to the user. These two facts define a Markov process, namely the set of all internal data states and the probability of getting from each member of the state set to the next member. Of course, some members may be terminal when the process terminates.

In a Ph.D. thesis by Whittaker (1992), a sound approach to certification is given using the Markov processes to maintain the sequential integrity of testing. The first Markov process, called the usage Markov chain, describes usage of the software in terms of stimuli and state transitions. This chain is used as a test sequence generator for the statistical test. Furthermore, a

comprehensive analysis of the usage chain is developed that characterizes the stochastic properties of the sequence used in the statistical test. The second Markov process, called the testing Markov chain, describes the history of the statistical test including failure data. A method for constructing the testing chain is given and an analysis is performed that results in a discrete, data-driven software reliability model. Derived from this model are estimates of the reliability, the mean time between failure, and an analytical stopping criterion based on the stochastic properties of both Markov chains.

8. Conclusions

Software is either correct or incorrect in design to a well-defined specification, in contrast to hardware which is reliable to a certain level in performing to a design assumed to be correct. For small and regular software, it may be possible to test exhaustively the software to determine its correctness. Even then, failures can be overlooked from human fallibility. But software of any size or complexity can only be tested partially, and typically a very small fraction of possible inputs are actually tested. At first glance, the fractions are so small for systems of ordinary size that the task of testing looks impossible. But when combined with mathematical verification, getting correct software is indeed possible.

Certifying the correctness of such software requires two conditions, namely:

1. Statistical testing with inputs characteristic of actual usage, and
2. No failures in the testing.

For interactive software, the statistical correlation of successive inputs must be treated as well. If any failures arise in testing or subsequent usage, the software is incorrect, and the certification is invalid. If such failures are corrected, the certification process can be restarted, with no use of previous testing results. Such corrections may lead to additional failures, or may not. So certifying the correctness of software is an empirical process that is bound to succeed if the software is indeed correct and may succeed for some time if the software is incorrect.

While possibly frustrating at first glance, this is all humans can assert about the correctness of software. In both verification and testing, human fallibility is present. But on second glance, the sequential history of certification efforts provides a human basis for assessing the quality of the software and expectations for achieving future correctness.

References

Adams, E. N. (1980). Minimizing Cost Impact of Software Defects. IBM Research Report RC 8228 (#35669).

Anderson, T. W., and Goodman, L. A. (1957). Statistical Inference About Markov Chains. *Annals Math Stat.* **28**, 89–109.

Bogott, R. P., and Franklin, M. A. (1975). Evaluation of Markov Program Models in Virtual Memory Systems. *Software Practice Exp.* **5**, 337–346.

Cheung, R. C., (1980). A User-Oriented Software Reliability Model. *IEEE Trans Software Eng.* **SE-6**(1), 118–125.

Chow, T. S. (1978). Testing Software Design Modeled by Finite-State Machines. *IEEE Trans Software Eng.* **SE-4**(1), 178–187.

Cobb, R. H., and Mills, H. D. (1990). Engineering Software Under Statistical Quality Control. *IEEE Software*, 44–54.

Currit, P. A., Dyer, M., and Mills, H. D. (1986). Certifying the Reliability of Software. *IEEE Trans Software Eng.* **SE-12**(1), 3–11.

Curtis, B. (1980). Measurement and Experimentation in Software Engineering. *Proc. IEEE* **68**(9), 1144–1157.

Dalal, S. R., and Mallows, C. L. (1988). *J. Am. Stat. Assoc.* **83**(403), 872–879.

Deck, M. D., Pleszkoch, M. G., Linger, R. C., and Mills, H. D. (1992). Extended Semantics for Box Structures. Proc. Hawaii Int. Conf. System Sciences, Vol. II, IEEE, pp. 382–393.

Duran, J. W., and Ntafos, S. C. (1984). An Evaluation of Random Testing. *IEEE Trans Software Eng.* **SE-10**(4), 438–444.

Duran, J. W., and Wiorkowski, J. J. (1980). Quantifying Software Validity by Sampling. *IEEE Trans Reliability* **R-29**(2), 141–144.

Dyer, M. (1992a). "The Cleanroom Approach to Quality Software Development." Wiley, New York.

Dyer, M. (1992b). Verification-Based Inspection. Proc. Hawaii Int. Conf. System Sciences, Vol. II, IEEE, pp. 418–427.

Feller, W. (1950). "An Introduction to Probability Theory and Its Application." Vol. 1. Wiley, New York.

Fetzer, D. T., and Poore, J. H. (1992). Using Box Structures with the Z Notation. Proc. Hawaii Int. Conf. System Sciences, Vol. II, IEEE, pp. 394–405.

Fuhrer, D., Mao, H., and Poore, J. H. (1992). OS/2 Cleanroom Environment: A Progress Report on a Cleanroom Tools Development Project. Proc. Hawaii Int. Conf. System Sciences, Vol. II, IEEE, pp. 449–458.

Hamlet, D., and Taylor, R. (1990). Partition Testing Does Not Inspire Confidence. *IEEE Trans Software Eng.* **SE-16**(12), 1402–1411.

Hetzel, W. (Ed.) (1972). "Program Test Methods." Prentice-Hall, New York.

Hevner, A. R., and Becker, S. A. (1992). Central Repository Data models for Cleanroom Systems Development. Proc. Hawaii Int. Conf. System Sciences, Vol. II, IEEE, pp. 459–469.

Howden, W. E. (1976). Reliability of the Path Analysis Testing Strategy. *IEEE Trans Software Eng.* **SE-2**(3), 208–215.

Ianinio, A., Littlewood, B., Musa, J. D., and Okumoto, K. (1984). Criteria for Software Reliability Model Comparisons. *IEEE Trans Software Eng.* **SE-10**, 687–691.

Jelinski, Z., and Moranda, P. B. (1972). Software Reliability Research. *In* "Statistical Computer Performance Evaluation," (W. Friedberger, ed.), Academic Press, Boston.

Juang, B. H., and Rabiner, L. R. (1985). A Probabilistic Distance Mmeasure for Hidden Markov models. *AT&T Tech. J.* **64**(2), 391–408.

Kemeny, J. G., and Snell, J. L. (1976). "Finite Markov Chains." Springer-Verlag, New York.

Kouchakdjian, A., Green, S. E., and Basili, V. R. (1989). The Cleanroom Case Study in the Software Engineering Laboratory: An Experiment in Formal Methods. SEL, University of Maryland.

Leveson, N. G. (1986). Software Safety: Why, What, and How. *Computing Surveys* **18**(2), 125–163.

Linger, R. C., and Mills, H. D. (1988). A Case Study in Cleanroom Software Engineering: the IBM COBOL Structuring Facility. Proceedings of COMPSAC '88, IEEE.

Linger, R. C., Mills, H. D., and Witt, B. I. (1979). "Structured Programming: Theory and Practice." Addison-Wesley, Reading, Massachusetts.

Littlewood, B. (1978). How To Measure Software Reliability and How Not To. Proc. 3rd Inter. Conf. Software Eng.

Mills, H. D. (1975). The New Math of Computer Programming. *Comm ACM* **18**(1).

Mills, H. D. (1983). "Software Productivity." Little, Brown and Company.

Mills, H. D. (1986). Structured Programming: Retrospect and Prospect. *IEEE Software*, 58–66.

Mills, H. D. (1988). Stepwise Refinement and Verification in Box-Structured Systems. *IEEE Computer*, 23–36.

Mills, H. D. (1992). Certifying the Correctness of Software. Proc. Hawaii Int. Conf. System Sciences, Vol. II, IEEE, pp. 373–381.

Mills, H. D., and Linger, R. C. (1986). Data Structured Programming: Program Design without Arrays and Pointers. *IEEE Trans Software Eng*. **SE-12**(2), 192–197.

Mills, H. D., and Poore, J. H. (1988). Bringing Software Under Statistical Quality Control. *Quality Progress*, November, 52–55.

Mills, H. D., Linger R. C., and Hevner, A. R. (1986). "Principles of Information Systems Analysis and Design." Academic Press, Boston.

Mills, H. D., Basili, V. R., Gannon, J. D., and Hamlet, R. G. (1987a). "Principles of Computer Programming: A Mathematical Approach." William C. Brown.

Mills, H. D., Dyer, M., and Linger, R. C. (1987b). Cleanroom Software Engineering. *IEEE Software*, September, 19–24.

Mills, H. D., Linger R. C., and Hevner, A. R. (1987c). Box Structured Information Systems. *IBM Systems J.* **26**(4), 395–413.

Musa, J. D. (1975). A Theory of Software Reliability and Its Application. *IEEE Trans Software Eng*. **SE-1**, 312–321.

Musa, J. D. (1980). The Measurement and Management of Software Reliability. Proc. IEEE **68**(9), 1131–1143.

Parnas, D. L. (1972). A Technique for Software Module Specification with Examples. CACM **15**, 330–336.

Parnas, D. L. (1979). Designing Software for Ease of Extension and Contraction. *IEEE Trans Software Eng*. **SE-5**, 128–138.

Parnas, D. L. (1990). An Evaluation of Safety-Critical Software. *Comm ACM* **23**(6), 636–648.

Parnas, D. L., and Wang, Y. (1989). The Trace Assertion Method of Module-Interface Specification. Technical Report 89-261, Queen's University, TRIO.

Poore, J. H., Mills, H. D., Hopkins, S. L., and Whittaker, J. A. (1990). Cleanroom Reliability Mmanager: A Case Study Using Cleanroom with Box Structures ADL. Technical Report, Software Engineering Technology, IBM-SID, STARS CDRL 1880.

Poore, J. H., Duitz, M., Fuhrer, D., Mao, H., Trammel, C., and Whittaker, J. A. (1991a). A Preprocessor for a Box Structured Design Language for C (Cleanroom Case Study). Technical Report, Dept. Computer Science, Univ. of Tennessee.

Poore, J. H., Dodson, J., Duitz, M., Fuhrer, D., Mao, H., and Whittaker, J. A. (1991b). Certification Model (Cleanroom case Study). Technical Report, Dept. Computer Science, Univ. of Tennessee.

Poore, J. H., Dodson, J., Duitz, M., Fuhrer, D., Mao, H., and Whittaker, J. A. (1991c). OS/2 Cleanroom Environment (Cleanroom Case Study). Technical Report, Dept. Computer Science, Univ. of Tennessee.

Rosen, S. J. (1992). Design Languages for Cleanroom Software Engineering. Proc. Hawaii Int. Conf. System Sciences, Vol. II, IEEE, pp. 406–417.

Selby, R. W., Basili, V. R., and Baker, F. T. (1987). Cleanroom Software Development: an Empirical Evaluation. *IEEE Trans Software Eng.* **SE-13**(9).

Siegrist, K. (1988). Reliability of Systems with Markov Transfer of Control. *IEEE Trans Software Eng.* **14**(7), 1049–1053.

Trammell, C. J., Hausler, P. A., and Galbraith, C. E. (1992). Incremental Implementation of Cleanroom Practices. Proc. Hawaii Int. Conf. System Sciences, Vol. II, IEEE, pp. 437–448.

Whittaker, J. A. (1992). Markov Chain Techniques for Software Testing and Reliability Analysis, Ph.D. dissertation, Univ. of Tennessee.

Whittaker, J. A., and Poore, J. H. (1992). Statistical Testing for Cleanroom Software Engineering. Proc. Hawaii Int. Conf. System Sciences, Vol. II, IEEE, pp. 428–436.

Wirth, N. (1971). Program Development by Stepwise Refinement. *Comm. ACM* **14**(4), 221–227.