University of Tennessee, Knoxville
Trace: Tennessee Research and Creative Exchange

The Harlan D. Mills Collection

Science Alliance

10-5-1988

A Case Study in Cleanroom Software Engineering: The IBM Cobol Structuring Facility

Richard C. Linger

Harlan D. Mills

Follow this and additional works at: http://trace.tennessee.edu/utk_harlan

Part of the Software Engineering Commons

Recommended Citation

Linger, Richard C. and Mills, Harlan D., "A Case Study in Cleanroom Software Engineering: The IBM Cobol Structuring Facility" (1988). *The Harlan D. Mills Collection.*
http://trace.tennessee.edu/utk_harlan/34

This Conference Proceeding is brought to you for free and open access by the Science Alliance at Trace: Tennessee Research and Creative Exchange. It has been accepted for inclusion in The Harlan D. Mills Collection by an authorized administrator of Trace: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

after the Norman conquest was recorded in Roman numerals, but never added up in spite of the obvious interest in such a result. Now the experts in arithmetic of the day would never have believed that with place notation and long division, school children of later centuries would be capable of arithmetic performance these experts deemed impossible. And so it will be that may current experts in heuristic, intuitive methods of software development will find the use of mathematical verification in place of trial and error unit debugging impossible to consider as a rational methodology.

The COBOL Structuring Facility

The COBOL Structuring Facility (COBOL/SF 1988a, 1988b) is comparable in function and complexity to a modern high-level language compiler. It embodies proprietary graph- and function-theoretic technology to automatically transform unstructured COBOL programs into hierarchies of structured procedures. COBOL/SF helps solve difficult software maintenance problems by reducing complexity and increasing understandability of program logic.

Table 1 summarizes the development history of COBOL/SF. This paper reports on development of Version 2; results for other versions, also developed with Cleanroom Software Engineering, were similar.

	Lines of Code (KLOC)			
	Reused	Changed	New	Total
Prototype	0	0	20	20
Version 1	18	2	15	35
Version 1A	30	5	11	46
Version 2	28	18	34	80

Table 1.
COBOL/SF Development Summary

The COBOL/SF prototype and versions 1 and 1A provided structuring capability for VS COBOL II programs into VS COBOL II only. Version 2 incorporated the following major additional functions: structuring of OS/VS COBOL programs into either OS/VS COBOL or VS COBOL II, automation of optional manual steps to enhance the structuring process, complexity metrics analysis, program modularization analysis, and structure chart generation for the output program procedure hierarchy. Some 52,000 lines of PL/I source code, new and changed, were written to produce Version 2, with 28,000 lines reused from Version 1.

Version 2 was developed by a Cleanroom software team composed of a technical engineering manager, six software engineers, and a certification engineer.[1] Three summer supplemental college students also participated. Team members held BS or MS degrees in computer science or mathematics and had recently joined IBM. With the exception of the team manager and certification engineer, COBOL/SF was their first software development project.

The Version 2 development proceeded through formal specification, design, functional verification, implementation, and Cleanroom testing in five increments, beginning on April 15 and completing December 15, 1987. Seventy person-months (eight full-time people for eight months, plus three supplementals for two months each) of effort were expended during this development period, for an overall productivity of 740 lines of code per person/month, including all specification, design, implementation, testing, and management activities. The system entered field test at customer sites on January 6, 1988.

Version 2 development was a real-world project in every respect, with shifting requirements and an extremely short development schedule. All schedules and budgets were met, and all committed functions were delivered.

All versions of COBOL/SF consist of four major components as show in Figure 1. The System Control Program manages system software and user interfaces, and certain common services. The Source Language Parsing Subsystem parses the input program and creates a knowledge base of program structure. The input program is prepared for structuring by the Control Flow Analysis Subsystem, which deals

[1]Team members: K. Cannaday, M. Deck, P. Hausler, R. Linger, C. Loving, L. Pedowitz, S. Rosen, A. Spangler

A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility

Richard C. Linger
IBM Corporation
Bethesda, Maryland

Harlan D. Mills
University of Florida
Gainesville, Florida
and
Software Engineering Technology, Inc.
Vero Beach, Florida

Abstract

The IBM COBOL Structuring Facility Program Product was developed by a small programming team using Cleanroon Software Engineering technology in a pipeline of increments with very high quality and productivity. In the Cleanroom approach, programs are developed under statistical quality control, and mathematical verification is used in place of unit debugging. The formal methods of specification, design, functional verification, and testing are described, together with development and management practices required for maintaining intellectual control over the process.

A Cleanroom Software Case Study

The IBM COBOL Structuring Facility (COBOL/SF) Version 2 Program Product automatically transforms unstructured COBOL programs into structured form. It was developed by a small programming team using Cleanroom Software Engineering technology [Mills 1987]. COBOL/SF Version 2 consists of 80,000 lines (52,000 new and changed over Version 1) of high function source code that was developed under statistical quality control, being specified, then designed, mathematically verified, and coded with no unit debugging in a pipeline of increments at very high productivity. Each increment was placed under engineering change control before any execution and subjected to system test under a sound statistical design.

As a result, COBOL/SF passed its field test of structuring a half-million lines of COBOL code in over 300 application programs with only 10 errors detected. All errors were trivial, none requiring more than a few hours to find and fix, and most just a few minutes. In all testing, only one error resulted in a COBOL program failing to execute functionally equivalent before and after structuring. As confidence in quality grew, field test participants engaged in wholesale structuring of entire systems of COBOL programs, in effect, treating the field test version of COBOL/SF as a final product.

Since the common wisdom in software engineering is that mathematical verification of sizable software products is impractical and that unit debugging by programmers is necessary, these results may appear incredible. As far as we know, the axiomatic verification (Hoare 1969, Gries 1981) of software as widely taught in university computer science courses today is indeed impractical for products of this size. However, functional verification (Linger 1979) was used for COBOL/SF. And even functional verification for products of this size is impractical, except for teams whose members are well educated in formal methods of specification, design and functional verification. Team members must be provided further internships in team operations, for scaling up such formal methods into work products and processes that permit day-to-day work to accumulate into mathematical verifications of software products of any size.

It is understandable that the common wisdom of such a new subject as software engineering can underestimate the potentials of human achievement in various ways. Centuries ago, the common wisdom in arithmetic with Roman numerals was that large scale arithmetic was impractical, so that the great inventory with structural problems caused by complex performed procedure logic, ALTER statements, etc. The Structured Program Generation Subsystem transforms the input program into structured form and generates code. Finally, an off-line Parser Generator compiles COBOL grammars into parse tables for use by the system.

COBOL/SF Version 2 was developed top down in five increments as depicted in Table 2. With no unit debugging permitted, the error rates shown are measured from first execution through the completion of Cleanroom testing. They range from 1.4 to 5.7 errors / KLOC of source code, with an average of 3.4 errors / KLOC. Table 2 suggests a possible correlation between increment size and error rate, however, no such relation appeared in the earlier versions, whose larger increments often exhibited the lowest error rates.

Published reports on software productivity and quality are highly variable, however, averages of 150 LOC / person-month and 70 errors / KLOC (including unit debugging) are representative of industrial experience for complex products [Boehm 1981, Jones 1986]. Table 2 shows anticipated errors for each increment at a rate of 70 errors / KLOC. Using the Cleanroom approach, a small team of software engineers produced code of compiler complexity at a rate of 3.4 errors / KLOC, roughly one-twentieth the industry average, and a productivity rate of 740 lines / person-month, roughly five times the industry average, all within schedule and budget.

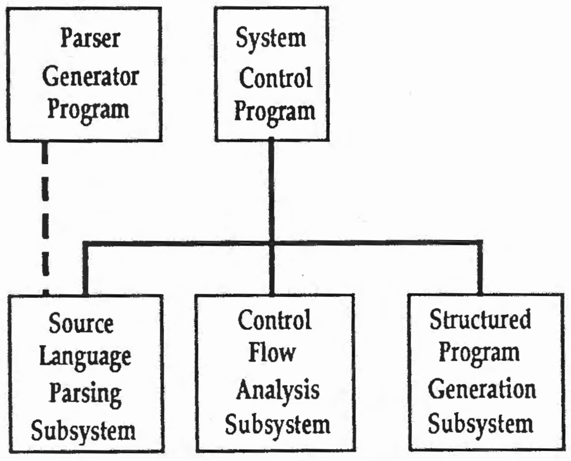
Parser Generator Program
System Control Program
Source Language Parsing Subsystem
Control Flow Analysis Subsystem
Structured Program Generation Subsystem

Figure 1
COBOL Structuring Facility Components

Cleanroom Software Engineering

Traditional software development proceeds through steps of specification, design, and code, then unit, component, and system testing. Selective tests are invented with knowledge of programmed internals, often by the developers themselves, typically to exercise primary functions, then secondary functions, error cases, etc. On completion of testing, the software is known to work as tested, but can still fail in circumstances not tested. As a result, the reliability evidence of selective testing is entirely anecdotal; it is known only that the software passed certain tests, with no inference possible of future failure rates. Worse, selective testing provides no rational basis for managing development. If few errors are found, is the code of high quality or is the test process faulty? If many errors are found, has the quality of the code been sufficiently improved or are there many more errors left to be found?

The objective of Cleanroom Software Engineering is to provide scientific evidence of reliability by embedding the entire development process in a statistical design [Mills 1987]. In the Cleanroom approach, a statistical property of software under test, namely successive times between execution failures, is used to estimate reliability directly using a new certification model [Curritt 1984]. In the statistical design, all testing is randomized over projected user input distributions, to rehearse eventual use of the software in arriving at reliability estimates. To keep the estimates valid, programs are placed under engineering change control from first execution on, with no unit debugging or developer testing permitted.

Cleanroom Software Engineering requires the best possible mathematics-based development methodologies. The objective is to develop such high quality software with no unit debugging that statistical testing will reveal a reliability growth, as lower and lower frequency errors are found and fixed, and not simply thrash from one high-frequency error to the next with no reliability growth possible, in effect debugging and not certifying the code.

Successful Cleanroom software development depends critically on the ability of team members to apply formal methods of software engineering in the following areas.

Increment	Lines of Code	Anticipated Errors at 70/ KLOC	Errors Found in Cleanroom Testing	Errors / KLOC	Errors Found in Field Testing
1	4150	291	6	1.4	1
2	11125	779	24	2.2	2
3	10080	706	23	2.3	2
4	19543	1368	111	5.7	4
5	7117	498	15	2.1	1
Totals	52015	3642	179		10

179 errors / 52.015 KLOC = 3.4 errors/KLOC

Table 2.
Error Rates in Cleanroom Testing Measured From First Execution for COBOL/SF Version 2 Development

Formal Specification

A cleanroom software specification defines required function and performance, the statistical distribution of user input, and the content of successive development increments.

A fundamental principle of Cleanroom Software Engineering is to identify formal mathematical structures for specifying the problem at hand, whether it be an entire system, a subsystem, or a component. Formal structures include the box structures of data abstractions [Mills 1988], formal grammars, regular expressions, propositional logic, predicate calculus, etc., in short, any appropriate mathematical structures at all.

Different parts of a system typically require different specification techniques. Box structures are a natural means to specify behavior of a system and its subsystems. Within box structure specifications, formal grammars and then semantics in conditional rules can provide the level of precision required. Much of COBOL/SF was specified with extensive formal grammars, which are closely related to the problem domain. Grammars were written both for the COBOL languages processed, and for internal string substitution operations in terms of recognition and transformation grammars.

A crucial mathematical property required of the formal structures is referential transparency in hierarchies [Mills 1988], that is, fully specified behavioral equivalence across levels of decomposition. This requirement precludes popularized specification techniques which lack referential transparency, such as structure charts and data flow hierarchies.

Natural language is used not to carry the burden of specification, for which it is not well suited, but rather to explain the formal specification structures. Where ambiguities arise, it is the formal structures that must be correct, no matter what the natural language says.

Specification structures are developed incrementally, with formal team review for correctness and simplicity at each step, and often undergo substantial revision to correct errors or take advantage of better ideas. No design work on an increment is undertaken until its specification is agreed by all team members to be correct. This level of formality is well suited to dealing with inevitable changes in requirements. The intellectual control provided by formal structures permits the precise impact of changes to be quickly assessed and accommodated.

No unnecessary work for the sake of formality was undertaken in specifying COBOL/SF; the specifications were written to a level of formality sufficient to guarantee completeness and correctness in team reviews.

Cleanroom testing of COBOL/SF required specifying a statistical user input distribution of COBOL programs with realistic statement frequencies and coding patterns, in order to generate test cases randomized against the distribution. Published papers analyzing COBOL program inventories provided statement frequencies, which were used by a PC-based test case generator to produce non-executable, random COBOL programs for testing.

Formal Design

The design of COBOL/SF was carried out using function-theoretic methodology [Linger 1979]. In the function-theoretic approach, program designs are regarded as mathematical objects, namely, rules for functions, and designs are treated as expressions in an algebra of functions, with keywords if, while, etc., as function operators.

The syntactic forms required for function-theoretic design are embodied in a Process Design Language [Linger 1979] whose principal components are function (subspecification) definitions, delimited by square brackets, and their decompositions into control structures, containing new function definitions, as illustrated in Figure 2. Great effort is expended in developing concise and correct intermediate function definitions, since these serve as specifications in the functional verification. Well over half the COBOL/SF design text is devoted to function definitions.

```
Sequence:     Ifthenelse:     Whiledo:

do [f]        [f]             [f]
  [g]         if              while
  [h]           p               p
od            then            do
                [g]             [g]
              else            od
                [h]
              fi
```

Figure 2.
Syntactic Forms for Function-Theoretic Design

Designs are constructed by repeatedly decomposing specified functions into control structures and subspecifications, as illustrated in Figure 3 for a miniature design fragment, and not by assembling control structures into designs through acts of heuristic invention. The difference is crucial, even though both processes end up with a structured program, because only the former provides the referential transparency at each decomposition step required for correctness verification.

```
. . .
[for queue q and stack s, append to q all
members of s (if any) in order followed by
eoq, set s to empty]
. . .
expands to:
. . .

do [for queue q and stack s, append to q all
     members of s (if any) in order followed
     by eoq, set s to empty]
     [for queue q and stack s, append to q all
     members of s (if any) in order, set s to
     empty]
     back [q] := eoq
od
. . .
expands to:
. . .
do  [for queue q and stack s, append to q all
     members of s (if any) in order followed
     by an eoq, set to empty]
     [for queue q and stack s, append to q  all
     members of s, (if any)  in order, set s to
     empty]
     while
         not empty (s)
     do  [move next member of stack s to
             queue q]
             x :=top (s)
             back (q) := x
     od
     back (q) := eoq
od
. . .
```

Figure 3.
Stepwise Decomposition of a Miniature Design Fragment

The entire design, not just its most interesting parts, is embodied with full precision in each decomposition step at increasing levels of detail. Because statistically generated tests can exercise exceptional cases as well as mainline processing, each increment must address the entire user input distribution, not just its principal components. In Cleanroom there is no protected testing of mainline functions.

The objective of formal correctness verification in team reviews requires designs that are as small and simple as possible, to help promote effective reasoning by team members.

Properly educated and motivated humans have substantial latent capability for logical precision in correctness verification, but only if program complexity can be held below a critical threshold. Dijkstra's original motivation for structured programming was to reduce the size of correctness proofs, but two additional factors contribute to complexity as well, namely, proliferation of state space data objects, forced by insufficient abstraction in the design, and sheer growth in design size, likewise forced by insufficient abstraction of case analyses into more general forms with simpler designs (the first idea is rarely the best idea!).

Data structured programming [Mills 1986a] was used to reduce the number of state space objects and simplify correctness verification. In this approach, data objects with disciplined access to data, such as stacks and queues, are employed, rather than objects with random access to data, such as arrays and pointers. The result is a sharp reduction in the number of objects and their references. Disciplined data access designs are more difficult to invent, but easier to verify, with less state information required in the mind at each verification step.

To help reduce the size of designs and the quantity of logical material to be verified, simpler design approaches were actively sought in review, and redesigning for simplicity was made an explicit objective. This activity produced astonishing results, with factors of up to five in size reductions achieved. For example, the prototype of COBOL/SF, estimated at 100 KLOC of PL/I by an independent IBM group, required just 20 KLOC as a result of data structured programming and design simplification.

Formal Verification

Formal verification begins with specifications, which are checked line-by-line in team reviews for correctness against requirements. For example, formal grammars for OS/VS COBOL and VS COBOL II, comprised of some 1500 productions each, were verified for correctness in intensive team reviews. As a result, no grammar errors whatsoever were encountered in field testing.

At the design level, traditional inspection methodology is aimed at finding errors through mental execution of program paths in group reviews. Such a process places demands on long term memory, to recall path histories and branches, and non-local reasoning, to integrate the effects of operations encountered. Worse, it is a non-finite activity, since programs of any size contain a virtually infinite number of possible paths.

In contrast, function-theoretic design verification is aimed at verifying the correctness of successive function decompositions [Mills 1986b]. This process is a reduction to practice of the Correctness Theorem [Linger 1979], which defines the correctness conditions that must hold for every control structure, as illustrated in Figure 4 in terms of correctness questions to apply in team reviews. Every design is a finite structure of function decompositions, and hence is verified in a finite, and large, number of mental function comparisons based on the correctness questions. Most of the function comparisons are made in seconds in team reviews through highly structured group dynamics, with more time taken if an error is suspected. Literally hundreds of such verifications can be made in a day's work, with astonishing savings possible in testing later on. In illustration, the 3300-line COBOL/SF Parser Generator program contained some 700 control structures, representing around 1200 correctness questions to be asked and answered in team review, easily accomplished in a few days work.

It is common wisdom today that all software errors are the result of inevitable human fallibility; however, function theoretic design and verification processes prove otherwise. It turns out that nearly all software errors result from heuristic development processes, and not from human fallibility itself. Heuristic development processes lack crucial mathematical properties such as referential transparency for decomposition and verification, and so embody errors of process that cannot be distinguished from human errors. Rigorous processes such as the function theoretic approach provide full referential transparency, and do not carry errors of process in their application. Like doing long division, one may make errors in computation, but they are readily identified through verification as errors of human fallibility in following a rigorous process.

The COBOL/SF experience demonstrates an upper bound on human fallibility on the order of three to four errors / KLOC remaining after a rigorous development and verification process and before first execution. Cleanroom testing then finds and fixes these errors to arrive at a near zero defect product. We believe that well over 90% of the 70 errors / KLOC in current industrial experience are in fact due to the processes in use and not the people.

```
Sequence:
    For all inputs,
      does [g] followed by [h] do [f]?

Ifthenelse:
    For all inputs,
      whenever p is true, does [g] do [f]
    and
      whenever p is false, does [h] do [f]?

Whiledo:
    For all inputs,
      does the whiledo terminate
    and
      whenever p is true, does [g] followed by
        [f] do [f]
    and
      whenever p is false, does doing nothing
        do [f]?
```

Figure 4.
Correctness Questions for the Control Structures of Figure 2

Cleanroom Implementation

Once correctness verification is complete for each increment, the designs are translated into the target language, in this case, PL/I. No acts of invention are permitted in the translation; hard-won design correctness must be maintained across the language representations. PDL designs are carried to a level of detail sufficient to ensure statement-to-statement mappings into PL/I. In addition, a PC-based translator was written to automate the implementation process.

It is worth noting that all development work, from specification through design and verification was carried out on Personal Computers, with a simple text editor as the only development tool. That is, the specifications and designs were treated strictly as accumulating logical objects in text form, in a development process aimed at ensuring their completeness and correctness at each step. Once translation to PL/I was completed, the programs were shipped to a mainframe to begin compilation and testing under full engineering change control. In the Cleanroom approach, only the certification engineers who execute the Cleanroom tests have access to the compilers. With no unit debugging, compilation during development is simply unnecessary. As a result, PC-based development with no compilation or execution capability is practical, and economical as well.

Formal tools to support mathematical specification, design, and verification will be welcome when they become available, but we believe that tools for heuristic specification, design, and trial and error coding, testing, and debugging are counter productive.

Cleanroom Testing

Cleanroom testing proceeds for successive code increments by executing test cases randomized against projected user input distributions and recording the resulting inter-fail time intervals. The accumulating time intervals are used by a PC-based certification model to compute current mean time to failure (MTTF) [Curritt 1986]. Failures are reported by certification engineers back to the software engineers.

Errors are fixed as they are found, and the code returned to testing. For high quality code, error frequency drops quickly in the testing and inter-fail times increase dramatically. In these cases, the certified MTTF rapidly exceeds total test time.

The MTTF values for early increments provide a scientific basis for managing development of later increments, say by allocating more effort to verification if MTTF values are too low, or even compressing schedules if the values are higher than required.

The types of errors present in Cleanroom code are very different from current industrial experience. The errors left behind after formal correctness verification are invariably "simple blunders," requiring little effort to find and fix. For example, an incorrect conjunction (say, an "and" where an "or" was intended), or a missing parameter on a call statement are typical errors.

The errors tend to show up quickly in the early testing; it is often the case that all the errors that will ever be found occur in the first few test cases. For example, the COBOL/SF Parser Generator was brought up in four increments subjected to 120 statistically generated test cases. Twelve minor errors were found, all in the first five cases, with error-free execution from then on, now passing three year's use.

A Cleanroom project is scheduled on the basis of code increments running defect free within a day or two of first execution. Under ten percent of project time is devoted to implementation and testing.

This experience is in sharp contrast to the deeper structural and interface errors commonly encountered with heuristic development processes. This difference reflects a synergism between mathematical verification and statistical testing. The former leaves behind simple errors that are easily found by test cases that cover the entire input distribution. Of course it is impossible to give a foolproof proof that a program is zero-defect, but that conclusion is increasingly justified as error-free executions accumulate over months and years of use.

Cleanroom Management

Cleanroom team management is technical engineering management, not administrative management. A team manager must ensure proper engineering methodologies in team operation, and must be an active participant in high level specification and design. Team management requires a deep understanding of formal methods, but also a deep conviction in their effectiveness. Without courage of convictions, it is easy to cut corners when the going gets rough, just when the best methods are needed most.

In illustration, our Cleanroom team understands that any code that exhibits high error rates (say 7 or 8 errors / KLOC) in early Cleanroom testing will come off the machine and back into design and review. Such action is rarely required, but occurs typically at a time of stress, say from a tight schedule which itself contributed to the high error rates. To an observer accustomed to heuristic methods, taking code off the machine may seem foolhardy, but time spent in rethinking the formal structures will save far more time in testing later on. In Cleanroom, the primary function of testing is to certify code, not debug it.

Cleanroom team management is carried out primarily through education in software engineering methodology, day-by-day, in group and individual interaction. Every design decision, every review, every execution failure is an opportunity to discuss, evaluate, and improve the use of formal methods.

Evolution of Cleanroom work products through iterations of design and review is an egoless process. All errors are team errors, the result of human fallibility in formal verification. Any error that survives review was missed by every team member. However, Cleanroom team success is a source of pride and accomplishment that is difficult to understand without firsthand experience. When a code increment runs right the first time on a machine and every time thereafter (as has occurred many times in developing COBOL/SF). the team satisfaction and motivation is remarkable indeed. Such performances become the "personal best" of the team, and anything less only strengthens the resolve to improve.

Cleanroom team performance requires both depth of knowledge in formal methods and the convictions to apply them. The formal methods are based on the flexibility and precision of mathematics, not on the latest buzz words of the moment. For example, the buzz word view of structured programming is syntactic and superficial, namely, programming with no gotos, but the mathematical view is semantic and powerful, namely, programming in an algebra of functions with verification at each decomposition step. In fact, this mathematical view defines the only known process that produces programs that are correct by construction.

One way to begin Cleanroom operations is to first form a single team whose members have been introduced to the formal methods, and can educate and reinforce each other in pilot projects to develop the required depth of understanding and convictions.

The firsthand experience of team membership is a critical step in developing future team leaders, in creating a whole new set of expectations and performance capabilities in software development. Once a team has demonstrated Cleanroom Software Engineering capability, new teams can be formed by cloning, as team members become leaders of new teams to continue the education process.

Cleanroom performance is demanding, but exhilarating too, in extending the human frontier in computing to new levels of excellence.

Acknowledgments

It is a pleasure to acknowledge the excellent comments of Kathy Cannaday and the referees in the preparation of this paper.

References

[Boehm 1981] Boehm, B. W. *Software Engineering Economics.* Englewood Cliffs, N.J.: Prentice-Hall, 1981.

[COBOL/SF 1988a] *COBOL Structuring Facility Re-Engineering Concepts.* IBM Publication SC34-4079, 1988.

[COBOL/SF 1988b] COBOL *Structuring Facility User's Guide and Reference.* IBM Publication SC34-4080, 1988.

[Curritt 1986] Curritt, P. A.; Dyer, M.; & Mills, H. D. "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering.* Vol. SE-12, No. 1. (Jan. 1986).

[Gries 1981] Gries, D. *The Science of Programming.* Springer Verlag, 1981.

[Hoare 1969] Hoare, C. A. R. "An Axiomatic Basis for Computer Programming." *Communications of the ACM.* Vol 12, (October 1969): pp. 576-83.

[Jones 1986] Jones, C. *Programming Productivity.* New York, N. Y.: McGraw-Hill, 1986.

[Linger 1979] Linger, R. C.; Mills, H. D.; & Witt, B. I. *Structured Programming: Theory and Practice.* Reading, Mass.: Addison-Wesley, 1979.

[Mills 1986a] Mills, H. D.; & Linger, R. C. "Program Design Without Arrays and Pointers." *IEEE Transactions on Software Engineering.* Vol. SE-12, No. 2 (Feb. 1986).

[Mills, 1986b] Mills, H. D. "Structured Programming: Retrospect and Prospect." *IEEE Software.* (Nov. 1986).

[Mills 1987] Mills, H. D.; Dyer, M.; & Linger, R. C.. "Cleanroom Software Engineering." *IEEE Software.* (Sept. 1987)

[Mills 1988] Mills, H.D.; Linger, R. C.; & Hevner, A. R. "Box Structured Information Systems." *IBM Systems Journal.* Vol. 26, No. 4 (1987) pp. 395-413..