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

The Harlan D. Mills Collection

Science Alliance

11-1986

Structured Programming - Retrospect and Prospect

Harlan D. Mills

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

Part of the Software Engineering Commons

Recommended Citation
Mills, Harlan D., "Structured Programming - Retrospect and Prospect" (1986). *The Harlan D. Mills Collection.*
http://trace.tennessee.edu/utk_harlan/20

This Article 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.

FUNDAMENTAL CONCEPTS
IN SOFTWARE ENGINEERING

Structured Programming: Retrospect and Prospect

Harlan D. Mills, IBM Corp.

Structured programming has changed how programs are written since its introduction two decades ago. However, it still has a lot of potential for more change.

Edsger W. Dijkstra's 1969 "Structured Programming" article[1] precipitated a decade of intense focus on programming techniques that has fundamentally altered human expectations and achievements in software development.

Before this decade of intense focus, programming was regarded as a private, puzzle-solving activity of writing computer instructions to work as a program. After this decade, programming could be regarded as a public, mathematics-based activity of restructuring specifications into programs.

Before, the challenge was in getting programs to run at all, and then in getting them further debugged to do the right things. After, programs could be expected to both run and do the right things with little or no debugging. Before, it was common wisdom that no sizable program could be error-free. After, many sizable programs have run a year or more with no errors detected.

Impact of structured programming. These expectations and achievements are not universal because of the inertia of industrial practices. But they are well-enough established to herald fundamental change in software development.

Even though Dijkstra's original argument for structured programming centered on shortening correctness proofs by simplifying control logic, many people still regard program verification as academic until automatic verification systems can be made fast and flexible enough for practical use.

By contrast, there is empirical evidence[2] to support Dijkstra's argument that infor-

Introducing the fundamental concepts series

A group of leading software engineers met in Columbia, Maryland, in September 1982 to provide recommendations for advancing the software engineering field. The participants were concerned about the rapid changes in the software development environment and about the field's ability to effectively deal with the changes.

The result was a report issued six months later and printed in the January 1985 issue of *IEEE Software* ("Software Engineering: The Future of a Profession" by John Musa) and in the April 1983 *ACM Software Engineering Notes* ("Stimulating Software Engineering Progress — A Report of the Software Engineering Planning Group").

The group's members were members of the IEEE Technical Committee on Software Engineering's executive board, the ACM Special Interest Group on Software Engineering's executive committee, and the IEEE Technical Committee on VLSI.

In the area of software engineering technology creation, the highest priority recommendation was to "commission a 'best idea' monograph series. In each monograph, an idea from two to four years ago, adjudged a 'best idea' by a panel of experts, would be explored from the standpoint of how it was conceived, how it has matured over the years, and how it has been applied. A key objective here is to both stimulate further development and application of the idea and encourage creation of new ideas from the divergent views of the subject."

Another way to state the objectives of the series is to (1) explain the genesis and development of the research idea so it will help other researchers in the field and (2) transfer the idea to the practicing software engineer.

After the report was published in this magazine, an editorial board was created to implement the series. John Musa, then chairman of the IEEE Technical Committee on Software Engineering, and Bill Riddle, then chairman of ACM SIGSE, appointed the following board members:

- Bruce Barnes, of the National Science Foundation,
- Meir Lehman, of Imperial College, as adviser,

0740-7459/86/1100/0058/$01.00 ©1986 IEEE

KEVIN REAGAN

mal, human verification can be reliable enough to replace traditional program debugging before system testing. In fact, structured programming that includes human verification can be used as the basis for software development under statistical quality control.[3]

It seems that the limitations of human fallibility in software development have been greatly exaggerated. Structured programming has reduced much of the unnecessary complexity of programming

- Peter Neumann, of SRI International (and no longer with the board),
- Norman Schneidewind, of the Naval Postgraduate School, as editor-in-chief, and
- Marv Zelkowitz, of the University of Maryland.

Rather than produce a monograph series, the board decided that *IEEE Software* would be a better medium for the series, since it reaches a large readership. Furthermore, the magazine's editor-in-chief, Bruce Shriver of IBM, strongly supported the series' objectives.

I am delighted that Harlan Mills, an IBM fellow, agreed to write the first article, "Structured Programming: Retrospect and Prospect," in this series. I am also grateful for Bruce Shriver's enthusiastic support and for agreeing to publish the series in *IEEE Software*. Future articles in this series will appear in this magazine. I also thank the *IEEE Software* reviewers for the excellent job they did of refereeing Mills's article.

In presenting this series, the editorial board is not advocating the idea of this or any article published. Rather, our purpose is to be an agent for the transfer of technology to the software engineering community. We believe it is the readers who should evaluate the significance to software engineering of the ideas we present.

The board is very interested in your opinions on this article and on the general concept of the series. Do you think it is a good idea? Has the article helped you to better understand the origins, concepts, and application of structured programming? What topics would you like covered? Please send your thoughts and opinions to Norman Schneidewind, Naval Postgraduate School, Dept. AS, Code 54Ss, Monterey, CA 93943.

Norman Schneidewind

Norman Schneidewind
Series Editor-in-Chief

and can increase human expectations and achievements accordingly.

Early controversies. Dijkstra's article proposed restricting program control logic to three forms — sequence, selection, and iteration — which in languages such as Algol and PL/I left no need for the goto instruction. Until then, the goto statement had seemingly been the foundation of stored-program computing. The ability to branch arbitrarily, based on the state of data, was at the heart of programming ingenuity and creativity. The selection and iteration statements had conditional branching built in implicitly, but they seemed a pale imitation of the possibilities inherent in the goto.

As a result, Dijkstra's proposal to prohibit the goto was greeted with controversy: "You must be kidding!" In response to complex problems, programs were being produced with complex control structures — figurative bowls of spaghetti, in which simple sequence, selection, and iteration statements seemed entirely inadequate to express the required logic. No wonder the general practitioners were skeptical: "Simple problems, maybe. Complex problems, not a chance!"

In fact, Dijkstra's proposal was far broader than the restriction of control structures. In "Notes on Structured Programming"[4] (published in 1972 but privately circulated in 1970 or before), he discussed a comprehensive programming process that anticipated stepwise refinement, top-down development, and program verification.

However, Dijkstra's proposal could, indeed, be shown to be theoretically sound by previous results from Corrado Boehm and Giuseppe Jacopini[5] who had showed that the control logic of any flowchartable program — any bowl of spaghetti — could be expressed without gotos, using sequence, selection, and iteration statements.

So the combination of these three basic statements turned out to be more powerful than expected, as powerful as any flowchartable program. That was a big surprise to rank and file programmers.

Even so, Dijkstra's proposal was still greeted with controversy: "It can't be practical." How could the complex bowls of spaghetti written at that time otherwise be explained? Formal debates were held at conferences about practicality, originality, creativity, and other emotional issues in programming, which produced more heat than light.

Early industrial experience

The *New York Times* project. An early published result in the use of structured programming in a sizable project helped calibrate the practicality issue. F. Terry Baker reported on a two-year project carried out by IBM for the *New York Times*, delivered in mid-1971, that used structured programming to build a system of some 85,000 lines of code.[6] Structured programming worked!

The project used several new techniques simultaneously: chief-programmer team organization, top-down development by stepwise refinement, hierarchical modularity, and functional verification of programs. All were enabled by structured programming.

The *New York Times* system was an on-line storage and retrieval system for newspaper reference material accessed through more than a hundred terminals — an advanced project in its day. The *Times* system met impressive performance goals — in fact, it achieved throughputs expected in an IBM 360/Model 50 using an interim hardware configuration of a Model 40. The IBM team also achieved an impressive level of productivity — a comprehensive internal study concluded that productivity, compared to other projects of similar size and complexity, was a factor of five better.

Unlike a spaghetti program, a structured program defines a natural hierarchy among its instructions.

In this case, since the *New York Times* had little experience in operating and maintaining a complex, on-line system, IBM agreed to maintain the system for the newspaper over the first year of operation. As a result, the exact operational experience of the system was also known and published by Baker.[7]

The reliability of the system was also a pleasant surprise. In a time when on-line software systems typically crashed several times a day, the *Times* software system crashed only once that year.

The number of changes required, for any reason, was 25 during that year, most of them in a data editing subsystem that was conceived and added to the system after the start of the project. Of these, about a third were external specification changes, a third were definite errors, and a third interpretable either way.

The rate of definite errors was only 0.1 per thousand lines of code. The highest quality system of its complexity and size produced to that time by IBM, the *Times* project had a major effect on IBM software development practices.

The structure theorem and its top-down corollary. Even though structured programming has been shown to be possible and practical, there is still a long way to go to achieve widespread use and benefits in a large organization. In such cases, education and increased expectations are more effective than exhortations, beginning with the management itself.

The results of Boehm and Jacopini were especially valuable to management when recast into a so-called structure theorem,[8] which established the existence of a structured program for any problem that permitted a flowchartable solution.

As an illustration, hardware engineering management implicitly uses and benefits from the discipline of Boolean algebra and logic, for example, in the result that any combinational circuit can be designed with Not, And, and Or building blocks. If an engineer were to insist that these building blocks were not enough, his credibility as an engineer would be questioned.

The structure theorem permits management by exception in program design standards. A programmer cannot claim the problem is too difficult to be solved with a structured program. To claim that a structured program would be too inefficient, a program must be produced as proof. Usually, by the time a structured program is produced, the problem is understood much better than before, and a good solution has been found. In certain cases, the final solution may not be structured — but it should be well-documented and verified as an exceptional case.

The lines of text in a structured program can be written in any order. The history of which lines were written first and how they were assembled into the final structured program are immaterial to its execution. However, because of human abilities and fallibilities, the order in which lines of a structured program are written can greatly affect the correctness and completeness of the program.

For example, lines to open a file should be written before lines to read and write the file. This lets the condition of the file be

checked when coding a file read or write statement.

The key management benefit from top-down programming was described in the top-down corollary[8] to the structure theorem. The lines of a structured program can be written chronologically so that every line can be verified by reference only to lines already written, and not to lines yet to be written.

Unlike a spaghetti program, a structured program defines a natural hierarchy among its instructions, which are repeatedly nested into larger and larger parts of the program by sequence, selection, and iteration structures. Each part defines a subhierarchy executed independently of its surroundings in the hierarchy. Any such part can be called a program stub and given a name — but, even more importantly, it can be described in a specification that has no control properties, only the effect of the program stub on the program's data.

The concept of top-down programming, described in 1971,[9] uses this hierarchy of a structured program and uses program stubs and their specifications to decompose program design into a hierarchy of smaller, independent design problems. Niklaus Wirth discussed a similar concept of stepwise refinement at the same time.[10]

Using the top-down corollary. The top-down corollary was counterintuitive in the early 1970's because programming was widely regarded as a synthesis process of assembling instructions into a program rather than as an analytic process of restructuring specifications into a program. Furthermore, the time sequence in which lines of text were to be written was counter to common programming practice.

For example, the corollary required that the JCL (job-control language) be written first, the LEL (linkage-editor language) next, and ordinary programs in programming languages last. The custom then was to write them in just the reverse order. Further, the hard inner loops, usually worked out first, had to be written last under the top-down corollary. In fact, the top-down corollary forced the realization that the linkage editor is better regarded as a language processor than a utility program.

It is easy to misunderstand the top-down corollary. It does not claim that the thinking should be done top-down. Its benefit is in the later phases of program design, after the bottom-up thinking and perhaps some trial coding has been accomplished. Then, knowing where the top-down development is going, the lines of the structured program can be checked one by one as they are produced, with no need to write later lines to make them correct. In large designs, the top-down process should look ahead several levels in the hierarchy, but not necessarily to the bottom.

The *New York Times* team used both the structure theorem and its top-down corollary. While the proof of the structure theorem (based on that of Boehm and Jacopini) seemed more difficult to understand, the team felt the application of the top-down corollary was more challenging in program design, but correspondingly more rewarding in results.

For example, with no special effort or prestated objectives, about half of the *Times* modules turned out to be correct after their first clean compile. Other techniques contributed to this result, including chief-programmer team organization, highly visible program development library procedures, and intensive program reading. However, these techniques were permitted to a great extent by top-down structured programming, particularly in the ability to defer and delegate design tasks through specifications of program stubs.

Dijkstra's proposal to prohibit the goto was greeted with controversy: "You must be kidding!"

NASA's Skylab project. In 1971-74, a much larger but less publicized project demonstrated similar benefits of top-down structured programming in software development by IBM for the NASA Skylab space laboratory's system. In comparison, the NASA Apollo system (which carried men to the Moon several times) had been developed in 1968-71, starting before structured programming was proposed publicly.

While the *New York Times* project involved a small team (originally four but enlarged to 11) over two years, Apollo and Skylab each involved some 400 programmers over consecutive three years of development. In each system, the software was divided into two major parts, of similar complexity: (1) a simulation system for flight controller and astronaut training and (2) a mission system for spacecraft control during flight.

In fact, these subsystems are mirror images in many ways. For example, the simulation system estimates spacecraft behavior from a rocket engine burn called for by an astronaut in training, while the mission system will observe spacecraft behavior from a rocket engine burn called for by an astronaut in flight.

Although less spectacular than Apollo, the Skylab project of manned space study of near-Earth space was in many ways more challenging. The software for the Skylab simulation system was about double the size of Apollo's, and the complexity was even greater.

The Skylab software project was initiated shortly after the original proposals for structured programming, and a major opportunity for methodology comparison arose. The Skylab mission system was developed by the same successful methods used for both subsystems in Apollo. But the Skylab simulation system was developed with the then-new method of top-down structured programming under the initiative of Sam E. James.

The Skylab results were decisive. In Apollo, the productivity of the programmers in both simulation and mission systems was very similar, as to be expected. The Skylab mission system was developed with about the same productivity and integration difficulty as experienced on both Apollo subsystems.

But the Skylab simulation system, using top-down structured programming, showed a productivity increase by a factor of three and a dramatic reduction in integration difficulty.

Perhaps most revealing was the use of computer time during integration. In most projects of the day, computer time would increase significantly during integration to deal with unexpected systems problems. In the Skylab simulation system, computer time stayed level throughout integration.

Language problems. By this time (the mid-1970's), there was not much debate about the practicality of structured programming. Doubtless, some diehards were not convinced, but the public arguments disappeared.

Even so, only the Algol-related languages permitted direct structured programming with sequence, selection and iteration statements in the languages. Assembly languages, Fortran, and Cobol were conspicuous problems for structured programming.

One approach with these languages is to design in structured forms, then hand-translate to the source language in a final

coding step. Another approach is to create a language preprocessor to permit final coding in an extended language to be mechanically translated to the source language. Both approaches have drawbacks.

The first approach requires more discipline and dedication than many programming groups can muster. It is tempting to use language features that are counter to structured programming.

The second approach imposes a discipline, but the programs actually compiled in the target language will be the result of mechanical translation themselves, with artificial labels and variables that make reading difficult. The preprocessing step can also be cumbersome and expensive, so the temptation in debugging is to alter the mechanically generated target code directly, much like patching assembly programs, with subsequent loss of intellectual control.

As a result of these two poor choices of approach, much programming in assembly languages, Fortran, and Cobol has been slow to benefit from structured programming.

Paradoxically, assembly language programming is probably the easiest to adapt to structured programming through the use of macroassemblers. For example, the Skylab simulation and mission systems were both programmed in assembly language, with the simulation system using structured programming through a macroassembler.

Both Fortran and Cobol have had their language definitions modified to permit direct structured programming, but the bulk of programming in both languages — even today — probably does not benefit fully from structured programming.

Current theory and practice

Mathematical correctness of structured programs. With the debate over and the doubters underground, what was left to learn about structured programming? It turned out that there was a great deal to learn, much of it anticipated by Dijkstra in his first article.[1]

The principal early discussions about structured programming in industry focused on the absence of gotos, the theoretical power of programs with restricted control logic, and the syntactic and typographic aspects of structured programs (indentation conventions and pretty printing, stepwise refinement a page at a time).

These syntactic and typographic aspects permitted programmers to read each other's programs daily, permitted them to conduct structured walk-throughs and program inspections, and permitted managers to understand the progress of software development as a process of stepwise refinement that allowed progressively more accurate estimates of project completion.

When a project was claimed to be 90-percent done with solid top-down structured programming, it would take only 10 percent more effort to complete it (instead of possibly another 90 percent!).

However, Dijkstra's first article on structured programming did not mention syntax, typography, readability, stepwise refinement, or top-down development. Instead, his main argument for structured programming was to shorten the mathematical proofs of correctness of programs! That may seem a strange argument when almost no one then (and few now) bothered to prove their programs correct anyway. But it was an inspired piece of prophecy that is still unfolding.

The ideas of structured programming, mathematical correctness, and high-level languages are mutually independent.

The popularizations of structured programming have emphasized its syntactic and superficial aspects because they are easiest to explain. But that is only half the story — and less than half the benefit — because there is a remarkable synergy between structured programming and the mathematical correctness of programs. And there have been many disappointments for people and organizations who have taken the structured-programming-made-easy approach without mathematical rigor.

Two reasons that Dijkstra's argument about the size of proofs of correctness for structured programs seems to be inspired prophecy are

- The proof of program's correctness is a singularly appropriate definition for its necessary and sufficient documentation. No gratuitous or unnecessary ideas are needed and the proof is sufficient evidence that the program satisfies its specification.
- The size of a correctness proof seems at least a partial measure of the complexity of a program. For example, a long program with few branches may be simpler to prove than a shorter one with many loops — and it may be less complex, as well. Or, tricky use of variables and operations may reduce the number of branches but will make the proof longer.

However, unless programmers understand what proofs of correctness are, these insights will not be realized. That was the motivation of the article "How to Write Correct Programs and Know It."[9] Then, whether structured programs are proved correct or not, this understanding will implicitly reduce complexity and permit better documentation.

In fact, Dijkstra's argument shows that the mathematical correctness of programs was an independent and prior idea to structured programming (even anticipated by writings of von Neumann and Turing). Yet it was strange and unknown to most programmers at the time. It is curious, although the earliest computers were motivated and justified by the solution of numerical problems of mathematics (such as computing ballistic tables), that the programming of such computers was not widely viewed as a mathematical activity.

Indeed, when it was discovered that computers could be used in business data processing, dealing with mostly character data and elementary arithmetic, the relation between programming and mathematics seemed even more tenuous.

As the Skylab project showed, structured programming is also independent of high-level languages. As treated syntactically and superficially, structured programming may have seemed dependent on high-level languages. But this is not true. Of course, high-level languages have improved programmer productivity as well, but that is a separate matter.

The ideas of structured programming, mathematical correctness, and high-level languages are mutually independent.

Program functions and correctness. A terminating program can be regarded as a rule for a mathematical function that converts an initial state of data into a final state, whether the problem being solved is considered mathematical or not.

For example, a payroll program defines a mathematical function just as a matrix inversion program does. Even nonterminating programs, such as operating systems and communication systems, can be expressed as a single nonterminating loop that executes terminating subprograms endlessly.

The function defined by any such ter-

minating program is simply a set of ordered pairs: the initial and final states of data that can arise in its execution. That matrix inversion seems more mathematical than payroll processing is a human cultural illusion, an illusion not known to or shared by computers.

Since programs define mathematical functions, which thereby abstract out all details of execution — including even which language or which computer is used — it is possible to discuss the correctness of a program with respect to its specification as a purely mathematical question. Such a specification is a relation. If the specification admits no ambiguity of the correct final state for a given initial state, the specification will be a function.

For example, a square root specification that requires an answer correct to eight decimal places (so any more places can be arbitrary) is a relation. But a sort specification permits only one final ordering of any initial set of values, and is thus a function.

A program will be correct with respect to a specification if and only if, for every initial value permissible by the specification, the program will produce a final value that corresponds to that initial value in the specification.

A little notation will be helpful. Let function f be defined by program P, and relation r be a specification (r is possibly a function). Then program P is correct with respect to relation r if and only if a certain correctness equation between f and r holds, as follows: $\text{domain}(f \cap r) = \text{domain}(r)$.

To see this, note that $f \cap r$ consists of just those pairs of r correctly computed by P, so $\text{domain}(f \cap r)$ consists of all initial values for which P computes correct final values. But $\text{domain}(r)$ is just the set of initial values for which r specifies acceptable final values, so it should equal $\text{domain}(f \cap r)$.

Such an equation applies equally to a payroll program or a matrix inversion program. Both can be mathematically correct, regardless of human interpretations of whether the computation is mathematical or not.

To picture this correctness equation, we can diagram f and r in a Venn diagram with projections of these sets of ordered pairs into their domain sets (see Figure 1). The correctness equation requires that the two domain sets $D(f \cap r)$ and $D(r)$ must coincide.

Mathematical correctness proofs. In principle, a direct way to prove the mathematical correctness of a program is clear. Start with a program P and the specification r. Determine from P its function f and whether the correctness equation between f and r holds.

In practice, given a spaghetti program, such a proof may be impractical — even impossible — because of the program's complexity. But a structured program with the same function f will be simpler to prove correct because of the discipline on its control structure. In retrospect, the reason lies in an algebra of functions that can be associated with structured programming.

It is easy to see in principle why a program is a rule for a function. For any initial state from which the program terminates normally (does not abort or loop endlessly), a unique final state is determined. But unlike classical mathematical function rules (such as given by polynomial expressions, trigonometric expressions, and the like), the function rules determined by programs can be quite arbitrary and complex. The final state, even though unique, may not be easily described because of complex dependencies among individual instructions.

For a spaghetti program, the only reasonable way to think of the program as a rule for a function is to imagine it being executed with actual data — by mental simulation. For small programs, a limited generic simulation may be possible (for example, "for negative values the program is executed in this section").

But for a structured program, there is a much more powerful way to think of it: as a function rule that uses simpler functions. For example, any sequence, selection, or iteration defines a rule for a function that uses the functions of its constituent parts.

Algebra of part functions. The remarkable thing about building these functions from the nested parts of a structured program is that the rules for constructing them are very simple and regular. They are simply described as operations in a certain algebra of functions.

The rules for individual instructions depend on the programming language. For example, the rule for an assignment statement $x := y + z$ is that the final state is exactly the same as the initial state except that the value attached to identifier x is changed to the value attached to identifier y plus the value attached to identifier z.

The rule for sequence is function composition. For example, if statements $s1$, $s2$ have functions $f1$, $f2$, the function for the sequence $s1;\ s2$ will be the composition $f1 \circ f2 = \{<x,y>\} : y = f2(f1(x))\}$.

It is important to note that the rules at each level use the functions at the next lower level, and not the rules at the next lower level. That is, a specific program part determines the rule of a function, but the rule itself is not used at higher levels. This means that any program part can be safely changed at will to another with the same function, even though it represents a different rule.

For example, the program parts $x := y$ and If $x \neq y$ Then $x := y$ define different rules for the same function and can be exchanged at will.

Axiomatic and functional verification. There is a curious paradox today between university and industry. While program correctness proofs are widely taught in universities for toy programs, most academics not deeply involved in the subject

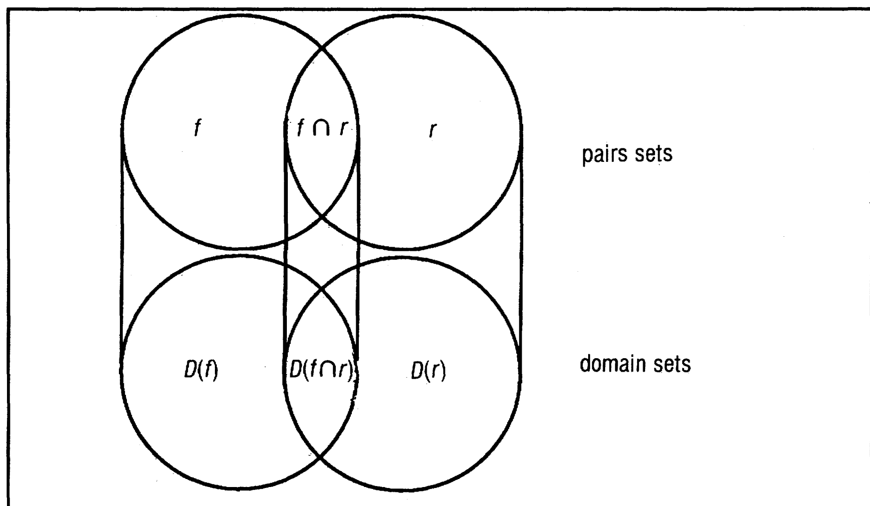

Figure 1. Correctness equation diagram with projections of the ordered pair sets into their domain sets.

regard program correctness as academic. Their motivation is cultural: "You'd never want to do this in practice, but it is good for you to know how to do it."

On the other hand, the IBM Software Engineering Institute curriculum is centered on the idea of program correctness exactly because it is not academic. Rather, it provides a practical method of reasoning about large programs that leads to much improved quality and productivity in software development.

There is also a simple answer to this paradox. Academics primarily teach a form of program correctness, called axiomatic verification, applied directly to toy programs, while the IBM Software Engineering Institute teaches a different form called functional verification in a way intended to scale up to large programs.

Axiomatic verification proves correctness by reasoning about the effect of programs on data. This reasoning takes the form of predicates on data at various places in the program that are invariant during execution. The relations between these predicates are given by axioms of the programming language (hence the name), and the entry/exit predicates together define the program function in an alternative form. Tony Hoare has given a beautiful explanation for this reasoning as a form of natural deduction, now called Hoare logic.[11]

Functional verification is based on function theory from the outset. For example, a simple assignment statement $x := y + z$ defines a function that can be denoted by $[x := y + z]$ and then used as a function in the algebra of structured-program part functions. In practice, functional verification is harder to teach but easier to scale up to large programs because of the presence of algebraic structure in an explicit form.

The most critical difference in practice between axiomatic and functional verification arises in the treatment of loops. In axiomatic verification, a loop invariant must be invented for every loop. In functional verification, during stepwise refinement, no such loop invariants are required because they are already embodied in the loop specification function or relation.[8]

Axiomatic verification can be explained directly in terms of program variables and the effects of statements on them, concretely in any given programming language. But when programs get large, the number of program variables get large, too — while the number of functions remains just one. The variable-free theory scales up to a more complex function rather than to many more variables.

Such a function may be defined in two lines of mathematical notation or a hundred pages of English. But its mathematical form is the same: a set of ordered pairs. There are many more opportunities for ambiguity and fallibility in a hundred pages of English, but increased individual fallibility can be countered by checks and balances of well-managed teams, rather than abandoning the methodology.

As a result, the functional verification of a top-level design of a 100,000 lines has the same form as for a low-level design of 10 lines: There is one function rule to be verified by using a small number of functions at the next level. The function defines a mapping from initial states to final states. These states will eventually be represented as collections of values of variables, but can be reasoned about as abstract objects directly in high-level design.

Eliminate the use of arrays in structured programs, and use instead data abstractions without arbitrary access.

While most of this reasoning is in the natural language of the application, its rules are defined by the algebra of functions, which is mathematically well-defined and can be commonly understood among designers and inspectors. There is considerable evidence that this informal kind of reasoning in mathematical forms can be effective and reliable in large software systems (exceeding a million lines) that are designed and developed top-down with very little design backtracking.[12]

There is yet another way to describe the reasoning required to prove the correctness of structured programs. The predicates in program variables of axiomatic verification admit an algebra of predicates whose operations are called predicate transformers in a classic book by Edsger Dijkstra,[13] and followed in a beautiful elaboration by David Gries.[14]

Looking to the future

Data-structured programming. The objective of reducing the size of formal correctness proofs can be reapplied to structured programs with a surprising and constructive result. In carrying out proofs of structured programs, the algebraic operations on the functions involved are the same at every level, but the functions become more complex in the upper parts of the hierarchy.

Two features in the data of the program have a large effect on the size of formal proofs: (1) The sheer number of program variables that define the data and (2) assignments to arrays.

Arrays represent arbitrary access to data just as gotos represent arbitrary access to instructions. The cost of this access shows up directly in the length and complexity of proofs that involve array assignments. For example, an array assignment, say $x[i] := y[j+k]$ refers to three previous assignments to i, j, and k. The values of i or $j+k$ may be out of range, and certainly must be accounted for if in range. Furthermore, array x will be altered at location i, and this fact must be accounted for the next time x is accessed again for the same value of i (which may be the value of another variable m).

Dijkstra's treatment of arrays[13] is very illuminating evidence of their complexity. Gries has also given the predicate transformers for array assignments,[14] which are much more complex than for simple assignments.

Happily, there is a way to address both of these proof expanders in one stroke: eliminate the use of arrays in structured programs, and use instead data abstractions without arbitrary access. Three simple such abstractions come to mind immediately: sets, stacks, and queues — the latter two data structures with LIFO and FIFO access disciplines. No pointers are required to assign data to or from stacks or queues, so fewer variables are involved in such assignments.

Furthermore, the proofs involving assignments to sets, stacks, and queues are much shorter than proofs involving arrays. It takes a good deal more thinking to design programs without arrays, just as it takes more thinking to do without gotos. But the resulting designs are better thought out, easier to prove, and have more function per instruction than array programs.

For example, the array-to-array assignment $x[i] := y[j+k]$ is but one of four instructions needed to move an item of data from y to x (assignments required for x, i, j, and k).

On the other hand, a stack to queue assignment, such as back(x) := top(y) moves the top of stack y to the back of queue x with no previous assignments. Of

course it takes more planning to have the right item at the top of stack y when it is needed for the back of queue x.

This discipline for data access, using stacks and queues instead of arrays, has been used in developing a complex language processing system of some 35,000 lines.[2] Independent estimates of its size indicates a factor of up to five more function per instruction than would be expected with array designs.

The design was fully verified, going to system test without the benefit of program debugging of any kind. System testing revealed errors of mathematical fallibility in the program at a rate of 2.5 per thousand instructions, all easily found and fixed. The kernel of the system (some 20,000 instructions) has been operating for two years since its system test with no errors detected.

Functional verification instead of unit debugging. The functional verification of structured programs permits the production of high-quality software without unit debugging. Just as gotos and arrays have seemed necessary, so unit debugging has also seemed necessary. However, practical experience with functional verification has demonstrated that software can be developed without debugging by the developers with some very beneficial results.

This latent ability in programmers using functional verification has a surprising synergy with statistical testing at the system level — that is, testing software against user-representative, statistically-generated input.[3] Statistical testing has not been used much as a development technique — and indeed for good reason in dealing with software that requires considerable defect removal just to make it work at all, let alone work reliably. However, statistical testing of functionally verified structured programs is indeed effective.

Cleanroom software development. The combined discipline of no unit debugging and statistical testing is called cleanroom software development. The term "cleanroom" refers to the emphasis on defect prevention instead of defect removal, as used in hardware fabrication, but applied now to the design process rather than the manufacturing process.

In fact, cleanroom software development permits the development of software under statistical quality control by iterating incremental development and testing. Early increments can be tested statistically for scientific estimates of their quality and for management feedback into the development process for later increments to achieve prescribed levels of quality.

At first glance, no unit debugging in software development seems strange, because unit debugging appears to be such an easy way to remove most of the defects that might be in the software. However, unit debugging is a good way to inadvertently trade simple blunders for deep system errors through the tunnel vision of debugging. And the very prospect of unit testing invites a dependence on debugging that undermines concentration and discipline otherwise possible.

More positively, eliminating unit testing and debugging leads to several benefits:

- more serious attention to design and verification as an integrated personal activity by each programmer,

The latent ability of people in new technologies is a source of continual amazement to experts.

- more serious attention to design and verification inspection by programming teams,
- preserving the design hypothesis for statistical testing and control (debugging compromises the design),
- selecting qualified personnel by their ability to produce satisfactory programs without unit debugging, and
- high morale of qualified personnel.

On the other hand, user-representative, statistical testing of software never before debugged provides several benefits:

- valid scientific estimates of the software's reliability and the rate of its growth in reliability when errors are discovered and fixed during system testing,
- forced recognition by programmers of the entire specification input space and program design by specification decomposition (instead of getting a main line running then adding exception logic later), and
- the most effective way to increase the reliability of software through testing and fixing.

The evidence is that industrial programming teams can produce software with unprecedented quality. Instead of coding in 50 errors per thousand lines of code and removing 90 percent by debugging to leave five errors per thousand lines, programmers using functional verification can produce code that has never been executed with less than five errors per thousand lines and remove nearly all of them in statistical system testing.

Furthermore, the errors found after functional verification are qualitatively different than errors left from debugging. The functional verification errors are due to mathematical fallibility and appear as simple blunders in code — blunders that statistical tests can effectively uncover.

Limits of human performance. The latent ability of people in new technologies is a source of continual amazement to experts. For example, 70 years ago, experts could confidently predict that production automobiles would one day go 70 miles an hour. But how many experts would have predicted that 70-year-old grandmothers would be driving them?!

Thirty years ago, experts were predicting that computers would be world chess champions, but not predicting much for programmers except more trial and error in writing the programs that would make chess champions out of the computers. As usual, it was easy to overestimate the future abilities of machines and underestimate the future abilities of people. Computers are not chess champions yet, but programmers are exceeding all expectations in logical precision.

From the beginning of computer programming, it has been axiomatic that errors are necessary in programs because people are fallible. That is indisputable, but is not very useful without quantification. Although it is the fashion to measure errors per thousand lines of code, a better measure is errors released per person-year of software development effort.

Such a measure compensates for the differences in complexity of programs — high complexity programs have more errors per thousand lines of code but also require more effort per thousand lines of code. It normalizes out complexity differences and has the further advantage of relating errors to effort rather than product, which is more fundamental.

For example, the *New York Times* released error rate was about one error per person-year of effort. That was considered an impossible goal before that time, but it is consistently bettered by advanced programming teams today.

An even better result was achieved by Paul Friday in the 1980 census software system for the distributed control of the national census data collection and communications network. The real-time software contained some 25,000 lines,

developed with structured programming and functional verification, and ran throughout the production of the census (almost a year) with no errors detected.

Friday was awarded a gold medal, the highest award of the Commerce Department (which manages the Census Bureau), for this achievement. Industrial software experts, looking at the function provided, regard the 25,000 lines as very economical, indeed. (It seems to be characteristic of high-quality, functionally verified software to have more function per line than is usual.)

At 2500 lines of code per person-year for software of moderate complexity, and one error per 10 person-years of effort, the result is one expected error for a 25,000-line software system. Conversely, a 25,000-line software system should prove to be error-free with appreciable probability.

These achievements already exist. With data structured programming, functional verification and cleanroom software development (and good management), we can expect another factor of 10 improvement in this dimension of performance in the next decade.

Structured programming has reduced much of the unnecessary complexity of programming and can increase human expectations and achievements accordingly. Even so, there is much yet to be done. It is not enough to teach university students how to verify the correctness of toy programs without teaching them how to scale up their reasoning to large and realistic programs. A new undergraduate textbook[15] seeks to address this issue.

Better software development tools are needed to reduce human fallibility. An interactive debugger is an outstanding example of what is not needed — it encourages trial-and-error hacking rather than systematic design, and also hides marginal people barely qualified for precision programming. A proof organizer and checker is a more promising direction.

It is not enough for industrial management to count lines of code to measure productivity any more than they count words spoken per day by salesmen. Better management understandings are needed for evaluating programming performance, as are increased investment in both education and tools for true productivity and quality.

But the principal challenge for management is to organize and focus well-educated software engineers in effective teams. The limitations of human fallibility, while indisputable, have been greatly exaggerated, especially with the checks and balances of well-organized teams. □

Acknowledgments

I appreciate the early help of the Fundamental Concepts in Software Engineering Series' editorial board, especially its editor-in-chief, Norman Schneidewind, in materially shaping this article. Key technical suggestions and improvements, due to David Gries and *IEEE Software*'s referees, are appreciated very much.

References

1. Edsger W. Dijkstra, "Structured Programming," in *Software Engineering Techniques*, J.N. Buxton and B. Randell, eds., NATO Science Committee, Rome, 1969, pp. 88-93.
2. Harlan D. Mills and Richard C. Linger, "Data Structured Programming: Program Design without Arrays and Pointers," *IEEE Trans. Software Eng.*, Vol. SE-12, No. 2, Feb. 1986, pp. 192-197.
3. Paul A. Currit, Michael Dyer, and Harlan D. Mills, "Certifying the Reliability of Software," *IEEE Trans. Software Eng.*, Vol. SE-12, No. 1, Jan. 1986, pp. 3-11.
4. O.J. Dahl, Edsger W. Dijkstra, and C.A.R. Hoare, *Structured Programming*, Academic Press, New York, 1972.
5. Corrado Boehm and Giuseppe Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Comm. ACM*, Vol. 9, No. 5, May 1966, pp. 366-371.
6. F. Terry Baker, "Chief-Programmer Team Management of Production Programming," *IBM Systems J.*, Vol. 1, No. 1, 1972, pp. 56-73.
7. F. Terry Baker, "System Quality Through Structured Programming," *AFIPS Conf. Proc. FJCC, Part 1*, 1972, pp. 339-343.
8. Richard C. Linger, Harlan D. Mills, and Bernard I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Mass., 1979.
9. Harlan D. Mills, *Software Productivity*, Little, Brown, and Co., Boston, 1983.
10. Niklaus Wirth, "Program Development by Stepwise Refinement," *Comm. ACM*, Vol. 14, No. 4, April 1971, pp. 221-227.
11. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, Vol. 12, No. 10, Oct. 1969, pp. 576-583.
12. Anthony J. Jordano, "DSM Software Architecture and Development," *IBM Technical Directions*, Vol. 10, No. 3, 1984, pp. 17-28.
13. Edsger W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976.
14. David Gries, *The Science of Programming*, Springer-Verlag, New York, 1981.
15. Harlan D. Mills et al., *Principles of Computer Programming: A Mathematical Approach*, Allyn and Bacon, Rockleigh, N.J., 1987.

Harlan D. Mills is an IBM fellow in the IBM Federal Systems Division and a professor of computer science at the University of Maryland. Before joining IBM in 1964, Mills worked at General Electric and RCA. He was president of Mathematica. Before joining the University of Maryland in 1975, he taught at Iowa State, Princeton, and Johns Hopkins universities.

Mills received a PhD in mathematics from Iowa State University and was named an honorary fellow by Wesleyan University. He has been a governor of the IEEE Computer Society and a regent of the DPMA Education Foundation. Mills received DPMA's 1985 Distinguished Information Science Award.

Mills can be reached at IBM Federal Systems Division, 6600 Rockledge Dr., Bethesda, MD 20817.